Programmeermethoden NA

Week 8: NumPy

Kristian Rietveld

http://liacs.leidenuniv.nl/~rietveldkfd/courses/prna/

Universiteit
Leiden

Blok 3

Thema: Python inzetten voor wetenschappelijk rekenen.

- *Week 8*: NumPy
- *Week 9*: Matplotlib & NumPy
- *Week 10*: NumPy, iPython & Python module showcase
- *Week 11*: Uitloop, oefententamen.

Deadline opdracht 3: vrijdag 1 december.

Schriftelijk tentamen: vrijdag 8 december.

Datastructuren

- Een computerprogramma doet in feite niets meer dan het manipuleren van data.
- Data kan op vele verschillende manieren in computergeheugen worden opgeslagen.
- Structuren waarin data wordt opgeslagen noemen we datastructuren.
- Vaak is de keuze van datastructuur bepalend voor de code van het programma.
  - Code om data in te laden.
  - Code om de data te manipuleren.

Datastructuren (2)

We hebben al verschillende datastructuren gezien.

- **Scalaire waarden**: een variabele die een enkele waarde opslaat.
- **Lijsten**: een collectie van waarden, waarbij de individuele waarden te benaderen zijn via integer subscripts.
- **Dictionaries**: een collectie van waarden, waarbij de individuele waarden te benaderen zijn via "keys". Ofwel, waarden zijn geassocieerd met keys.

Datastructuren (3)

Hoe slaan we vectoren en matrices op? Als lijst?

- Maar lijsten zijn dynamisch.
- Lijsten kunnen verschillende typen waarden bevatten.
- En rekenen met lijsten is niet handig:
  - `[4, 5] * 3 = [4, 5, 4, 5, 4, 5]`.
  - `[4, 5] + 3 = Error?`.

Dit moet handiger kunnen?

Datastructuren (4)

Een belangrijke datastructuur is de *array*. Een *array* is een geordende rij variabelen van ***hetzelfde*** type.

Er zijn ook meer-dimensionale arrays. Zoals twee-dimensionale arrays: matrices.

In twee-dimensionale arrays adresseren we elementen door een rij en kolom op te geven.

NumPy introductie

- Het is nu zo dat vectoren en matrices nog al eens voorkomen in de natuur- en sterrenkunde.
- NumPy (Numerical Python) is een package om te werken met (onder andere) vectoren en matrices en wordt in veel takken van de wetenschap gebruikt voor numeriek rekenwerk.
- Belangrijkste onderdeel: multidimensionale array datastructuur.
- NumPy is een package en moeten we eerst importeren:

```
import numpy as np
```

Arrays

Waarin verschilt de NumPy array ten opzichte van Python lijsten?:

- Aantal elementen staat na aanmaken vast (niet dynamisch).
- Alle elementen zijn van hetzelfde type.
- Het gebruik van operatoren op NumPy arrays is wat je zou verwachten in tegenstelling tot Python lijsten (zie ook later).
- Veel eenvoudiger om te werken met meer-dimensionale arrays dan geneste lijsten.
- Operaties op NumPy arrays zijn significant sneller! Belangrijk wanneer jullie gaan werken met grotere datasets.

NumPy arrays maken

We beginnen met 1-dimensionale arrays.

Bij het maken geven we de gewenste grootte van het array op als het aantal elementen. Dit kan op verschillende manieren:

- Creëren aan de hand van een Python list.
- `np.zeros`: initialisatie met nullen.
- `np.ones`: initialisatie met enen.
- `np.tile`: initialisatie met gespecificeerde waarde.

NumPy arrays maken (2)

```
>>> np.array([1, 2, 3, 4, 5, 6])
array([1, 2, 3, 4, 5, 6])
>>> np.zeros(6)
array([ 0.,  0.,  0.,  0.,  0.,  0.])
>>> np.ones(6)
array([ 1.,  1.,  1.,  1.,  1.,  1.])
>>> np.tile(39., 6)
array([ 39.,  39.,  39.,  39.,  39.,  39.])
```

NumPy arrays maken (3)

- `np.arange(start, stop, stap)`: maak een getallen reeks. Mag ook floating-point gebruiken!
- `np.linspace(begin, eind, N)`: N getallen uit gesloten interval, gelijke afstand tussen de elementen.

```
>>> np.arange(0, 10, 2)
array([0, 2, 4, 6, 8])
>>> np.linspace(1, 5, 10)
array([ 1.        ,  1.44444444,  1.88888889,  2.33333333,  2.77777778,
        3.22222222,  3.66666667,  4.11111111,  4.55555556,  5.        ])
```

Eigenschappen van NumPy arrays

```
>>> A = np.zeros(6)   # 6 elementen, waarde nul.
>>> A.ndim            # Aantal dimensies.
1
>>> A.shape    # De grootte van elke dimensie
(6,)
>>> A.size     # Het aantal elementen in de array.
6
>>> A.dtype   # Het datatype van elk element
dtype('float64')
```

Datatypen in NumPy

- `float64`? Die hebben we nog niet eerder gezien.
- NumPy kent vele extra datatypen waaruit kan worden gekozen om de data zo efficient mogelijk op te slaan.
- De belangrijkste: `np.bool8`, `np.int32`, `np.float64`, `np.complex128`.

Floating-point getallen

- Reele getallen worden in computers opgeslagen als "floating-point" getal.
- De opslagcapaciteit is niet onbeperkt, dus we kunnen niet alle mogelijke getallen opslaan.
- Er vindt afronding plaats!
- Irrationele getallen als pi, sqrt(2) worden in ieder geval nooit exact gerepresenteerd!

Floating-point getallen (2)

- IEEE 754 standaard.
- Python: double precision, 64 bits.
  - 1 bit sign
  - 11 bits exponent
  - 53 bits significant (waarvan 1 bit niet expliciet opgeslagen)
- 1.11011_2 * 2^-3 => 0.23046875
  - (1/1 + ½ + ¼ + 1/16 + 1/32) * 2^-3
  - `0011111111001101100000000000000000000000000000000000000000000000`
- 53 bits => ~16 significante cijfers.
- Bereik: -10^308 tot 10^308.

Floating-point getallen (3)

Wanneer opletten?

- Er zullen afrondfouten optreden, voornamelijk bij iteratieve berekeningen.
- Verschillende volgorde in toepassen operatoren: verschillende resultaten.
- Operatie op een zeer klein en zeer groot getal: significantie kleine getal verdwijnt.
- Deling van integer getallen geeft standaard een integer.
- Pas op met vergelijken van floating-point getallen! 6.1 kan zijn opgeslagen als 6.099999999999996. Dan doet `x == 6.1` wellicht iets anders.
  - Oplossing: maak gebruik van `np.allclose(a, b)`.

Datatypen in NumPy

- Bij initialisatie probeert NumPy een geschikt datatype te kiezen.
- Soms is de gok niet wat je wilt, zelf opgeven met `dtype=`.

```
>>> np.ones(10)
array([ 1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.])
>>> np.ones(10, dtype=np.int32)
array([1, 1, 1, 1, 1, 1, 1, 1, 1, 1], dtype=int32)
>>> np.ones(5, dtype=np.bool8)
array([ True,  True,  True,  True,  True], dtype=bool)
```

Lijst vs. NumPy array

Laten we eens proberen te rekenen met een lijst van integers:

```
>>> l = [1, 2, 3, 4]
>>> l * 4
[1, 2, 3, 4, 1, 2, 3, 4, 1, 2, 3, 4, 1, 2, 3, 4]
>>> l + 4
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate list (not "int") to list
>>> l * l
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can't multiply sequence by non-int of type 'list'
```

Rekenen met NumPy arrays

Python lijsten geven ons niet de resultaten die we zouden verwachten. Lijsten zijn echt bedoeld als "container" en niet als vector of matrix.

Daarom: als je gaat rekenen, gebruik NumPy arrays! Alle NumPy operatoren werken elementgewijs.

```
>>> a = np.array([1, 2, 3, 4])
>>> a * 4
array([ 4,  8, 12, 16])
>>> a + 4
array([5, 6, 7, 8])
>>> a * a
array([ 1,  4,  9, 16])
```

Rekenen met NumPy arrays (2)

Een formule kan met 1 statement worden toegepast op een volledige getallenreeks:

```
>>> x = np.arange(0, 10)
>>> print x
[0 1 2 3 4 5 6 7 8 9]
>>> f1 = x ** 2
>>> f1
array([ 0,  1,  4,  9, 16, 25, 36, 49, 64, 81])
>>> f2 = x ** 3 + 2 * x**2 - 3
>>> f2
array([ -3,   0,  13,  42,  93, 172, 285, 438, 637, 888])
```

Reductieoperatoren

Een reductieoperator berekent 1 resultaat voor een gehele array. Voorbeelden (met `a` een array):

- Sommeren: `np.sum(a)`
- Gemiddelde: `np.mean(a)`
- Standaardeviatie: `np.std(a)`
- Minimum: `np.amin(a)`
- Maximum: `np.amax(a)`

Wiskundige functies

Alle belangrijke wiskundige functies vind je terug in NumPy.

Parameter `a` mag natuurlijk zowel een scalair als array zijn.

Voorbeelden:

- `np.log(a), np.log10(a), np.exp(a)`
- `np.sin(a), np.cos(a), np.tan(a)`
  - Let op: `np.deg2rad(a)`
- `np.sqrt(a), np.floor(a), np.ceil(a)`

Constanten: `np.pi`, `np.e`. (Natuurkundige constanten: zie Scipy).

Slicing & indexing

Indexing en slicing zoals je bent gewend in strings en lijsten.

Toekenning aan een slice:

- Toekenning van een scalair: elk element in de slice krijgt deze waarde.
- Toekenning van een array: arrays moeten evenveel elementen bevatten!

Slicing & indexing (2)

```
>>> A = np.arange(0, 10)  # 0 t/m 9
>>> A[1:4] = 10
>>> print A
[ 0 10 10 10  4  5  6  7  8  9]

# Reeks om toe te kennen groter dan slice
>>> A[8:] = [20, 21, 22, 23]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: cannot copy sequence with size 4 to array axis
with dimension 2


>>> A[8:] = [20, 21]
>>> print A
[ 0 10 10 10  4  5  6  7 20 21]
```

Meer-dimensionale arrays

- NumPy arrays kunnen een arbitrair aantal dimensies aan.
- Dimensies worden ook wel "assen" genoemd.
- Elke as heeft een bepaalde lengte.
- Elke NumPy array heeft een "vorm" (shape) waarin de lengte van elke as is vastgelegd.
  - `(3, )`: 1 dimensie lengte 3.
  - `(3, 4)`: 2 dimensies: 3 rijen, 4 kolommen.
  - `(10, 3, 4)`: 3 dimensies: 10 vlakken, 3 rijen, 4 kolommen. (volgende week).

Meer-dimensionale arrays (2)

Om te maken werken de gebruikelijke functies. In plaats van een aantal elementen vul je een shape tuple in.

```
>>> A = np.ones( (3, 4) )   # 3 rijen, 4 kolommen
>>> print A
[[ 1.  1.  1.  1.]
 [ 1.  1.  1.  1.]
 [ 1.  1.  1.  1.]]
>>> A = np.tile(6, (3, 4) )
>>> print A
[[6 6 6 6]
 [6 6 6 6]
 [6 6 6 6]]
```

Identiteitsmatrices

`np.eye(n)` maakt een n x n identiteitsmatrix.

```
>>> I = np.eye(3)     # Een 3x3 identiteitsmatrix
>>> print I
[[ 1.  0.  0.]
 [ 0.  1.  0.]
 [ 0.  0.  1.]]
```

Hoe zit dat met blokhaken?

- Het aantal blokhaken correspondeert met het aantal dimensies.
- `[0 1 0]` is iets anders dan `[[0 1 0]]`, zie ook volgende week.

Arrays kopieren

Pas op: een toekenning is geen kopieeractie, net als bij lijsten!!

```
>>> A = np.eye(3)
>>> B = A       # Kopieert niet, maar legt een extra referentie aan.
>>> B[0,2] = 9
>>> print A             # A is dus ook aangepast!
[[ 1.  0.  9.]
 [ 0.  1.  0.]
 [ 0.  0.  1.]]

# De correcte manier om een kopie te maken.
>>> B = np.copy(A)
```

Indexeren over meerdere dimensies

Om een element aan te duiden in een multidimensionale array: geef per as (dimensie) een index op, gescheiden door komma's.

- `B[0,2]` (rij 0, kolom 2)
- `B[2,3]`
- `B[1,2,3,4,5]`

```
>>> B = np.ones( ( 3, 4) )
>>> B[0,2] = 10
>>> B[2,3] = 20
>>> print B
[[  1.   1.  10.   1.]
 [  1.   1.   1.   1.]
 [  1.   1.   1.  20.]]
```

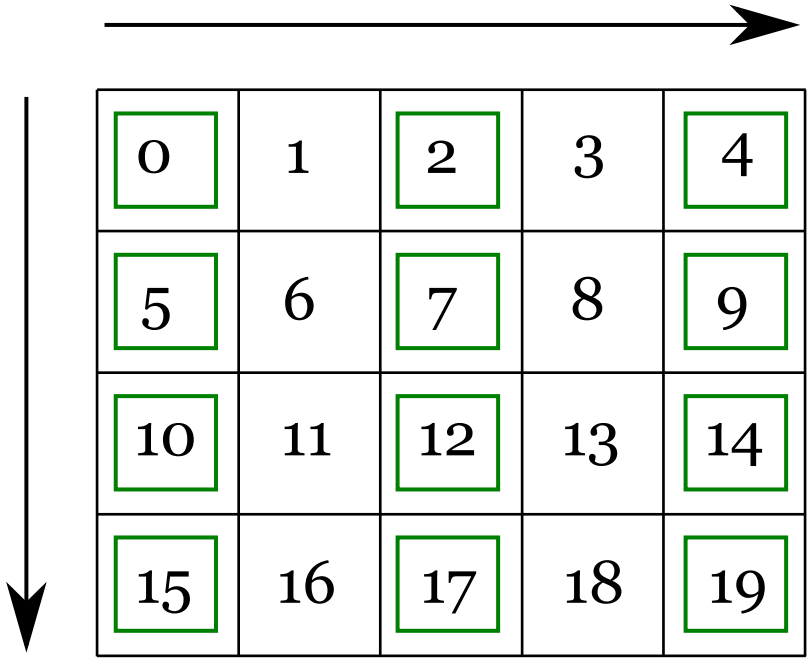
Slicing over meerdere dimensies

In plaats van een index mag je natuurlijk ook een slice opgeven.

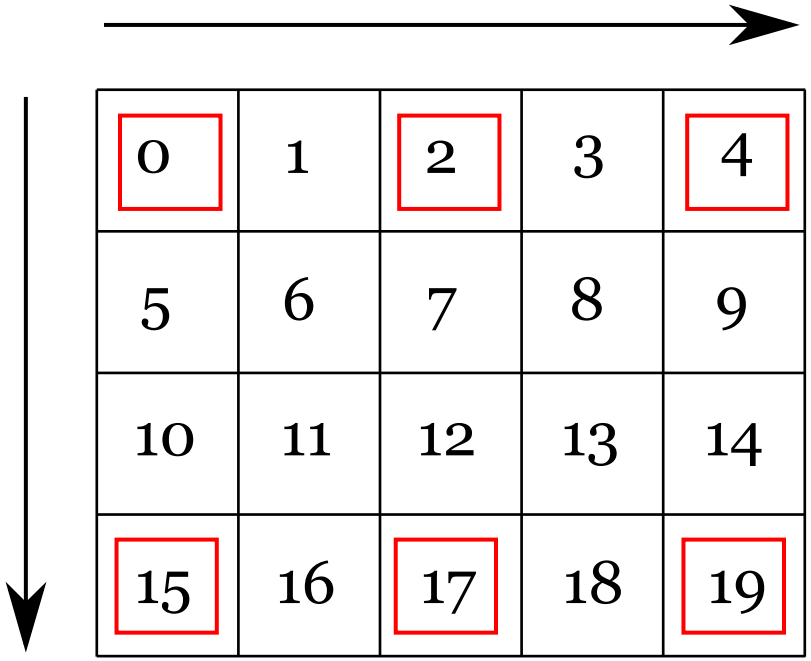
De lege slice `:` selecteert de gehele as.

```
>>> A[:,:]
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14],
       [15, 16, 17, 18, 19]])
>>> A[2,1]         # Selecteer een enkel element
11
>>> A[2,:]         # Selecteer de derde rij.
array([10, 11, 12, 13, 14])
>>> A[:,3]         # Selecteer de vierde kolom
array([ 3,  8, 13, 18])
```

Slicing (2)



`A[:,::2]`



`A[::3,::2]`

Rekenen in meerdere dimensies

```
>>> np.ones( (3, 3) ) + np.tile(10, (3, 3) )
array([[ 11., 11., 11.],
       [ 11., 11., 11.],
       [ 11., 11., 11.]])


>>> np.eye(4) * 9
array([[ 9., 0., 0., 0.],
       [ 0., 9., 0., 0.],
       [ 0., 0., 9., 0.],
       [ 0., 0., 0., 9.]])
```

* is geen dot product!

Wat gebeurt hier???

```
>>> A = np.eye(3)
>>> B = np.tile(4, (3, 3))
>>> A * B
array([[ 4.,  0.,  0.],
       [ 0.,  4.,  0.],
       [ 0.,  0.,  4.]])
```

Matrix/dot product

- Gebruik `np.dot()` als je een matrix/dot product wilt doen.

```
>>> np.dot(A, B)
array([[ 4.,  4.,  4.],
       [ 4.,  4.,  4.],
       [ 4.,  4.,  4.]])
```

- Of maak matrices met `np.matrix()` ipv `np.array()`:

```
>>> A = np.matrix(np.eye(3))
>>> B = np.matrix(np.tile(4, (3, 3) ))
>>> A * B
matrix([[ 4.,  4.,  4.],
        [ 4.,  4.,  4.],
        [ 4.,  4.,  4.]])
```

Random Numbers

Met `np.random.random()` kun je zowel random getallen als arrays maken. De elementen zullen zitten in het interval [0.0, 1.0).

```
>>> np.random.random()
0.9420733512975746
>>> np.random.random( (3, 3) )
array([[ 0.85083159,  0.28587965,  0.69833045],
       [ 0.98522151,  0.93762675,  0.29451167],
       [ 0.17332978,  0.87714118,  0.36772117]])
```

Random Numbers (2)

Voor een integer range, maak gebruik van `np.random.random_integers()`:

```
# Integers tussen 0 t/m 10, shape (3, 3)
>>> np.random.random_integers(0, 10, (3, 3) )
array([[6, 9, 4],
       [8, 0, 5],
       [8, 7, 1]])
```

Random Numbers (3)

- Wat ook vaak voorkomt is dat je een trekking wilt doen uit een reeks van getallen.
- Standaard gebeurt dit "met terugleggen". Voor zonder terugleggen voeg toe `replace=False`.

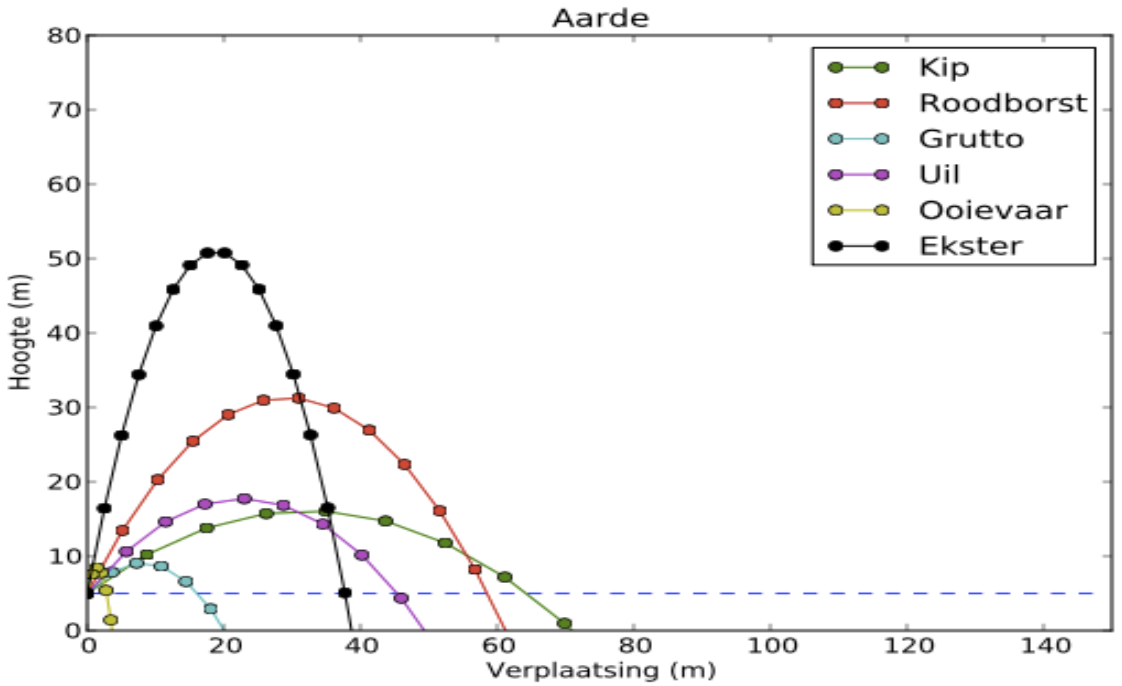
```
>>> np.random.choice(np.arange(100, 200))
117

# Trekking van 5 elementen
>>> np.random.choice(np.arange(100, 200), 5)
array([190, 135, 176, 112, 101])
```

Derde programmeeropdracht

De derde programmeeropdracht heet "2-in-1". We schrijven een menu-gestuurd programma waarin twee "spellen" kunnen worden gespeeld:

- (Mini) Game of Life
- “Angry Birds”



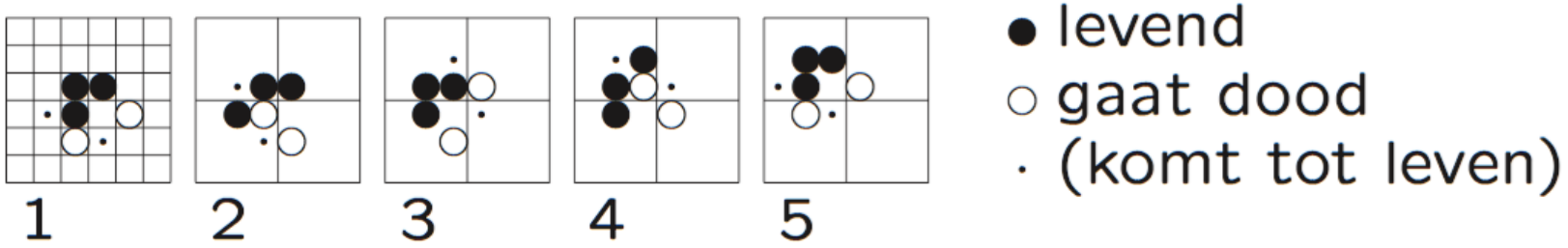
In beide gevallen: oefenen met NumPy arrays.

http://liacs.leidenuniv.nl/~rietveldkfd/courses/prna2017/opdr3.html

Life

Life is een cellulaire automaat, in 1970 bedacht door John Conway.

In een 2-dimensionaal oneidig groot rooster beginnen we met een eindig aantal levende vakjes (cellen). Een levend vakje met minder dan 2 of meer dan 3 buren (van de 8) gaat dood. Met precies 2 of 3 levende buren overleeft het. In een dood vakje met precies 3 levende buren ontstaat leven. Aan de hand van deze "regels" wordt de volgende generatie bepaald, dit gebeurt voor alle vakjes tegelijk.

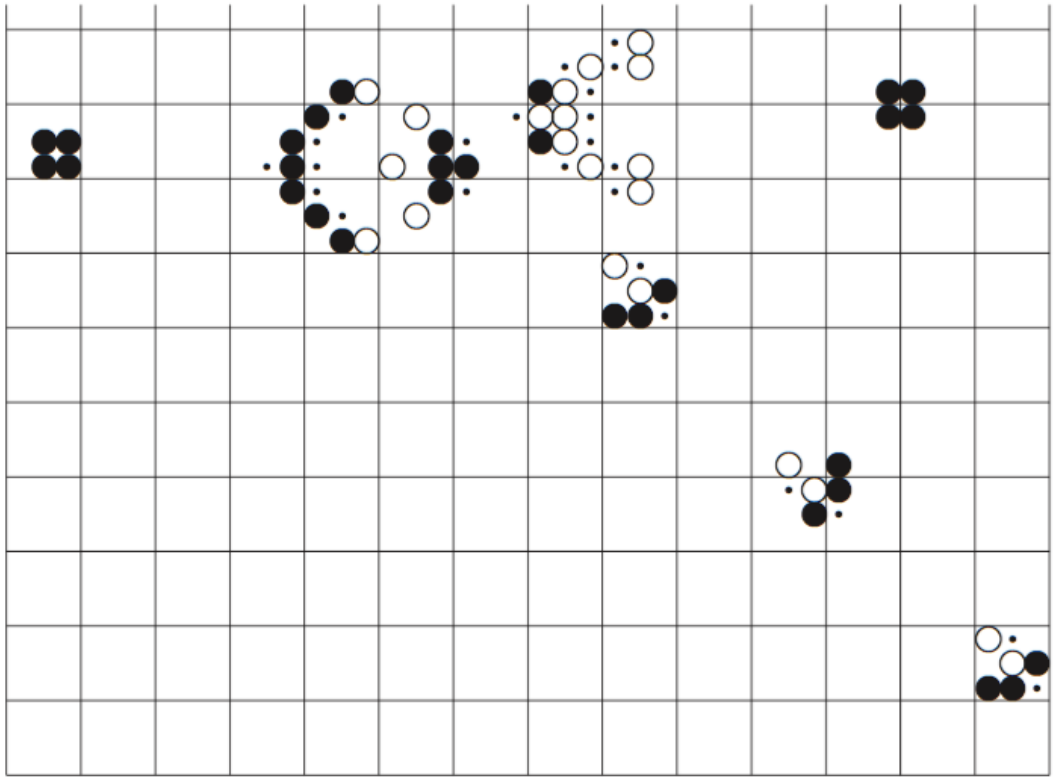


Dit patroon heet "glider".

Wiki: http://www.conwaylife.com/wiki/

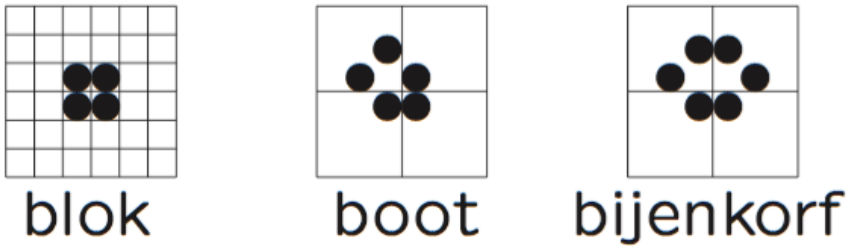
Life (2)

Een bijzondere beginconfiguratie is de glider gun, die elke dertigste generatie een nieuwe glider afvuurt:



Life (3)

Een stilleven is een Life-configuratie die niet verandert:



En een oscillator repeteert met een zekere periode (stilleven is een periode-0 oscillator):

Life (4)

Een beginnetje voor een klasse `LifeWereld` voor life zit er zo uit:

```
class LifeWereld(object):
    '''Bevat alle data van de huidige wereld en methoden om een nieuwe
    generatie uit te rekenen en de wereld aan te passen.'''
    def __init__(self, hoogte, breedte):
        self.hoogte = hoogte
        self.breedte = breedte
        self.generatie = 0

        self.wereld = np.zeros( (self.hoogte, self.breedte), dtype=np.bool)
        # enz ...
    def afdrukken(self):
        '''Drukt de wereld af naar standard output.'''
        pass
    def schoon(self):
        '''Verschoon de wereld: alle cellen worden gedood.'''
        pass
    def random(self):
        pass
    def plaats_glider(self, rij, kolom):
        '''Plaats een glider op positie (rij, kolom); dit is de linkerbovenhoek
        van de glider.'''
        pass
    # TODO: enzovoort
```

Tot slot

Werkcollege:

- Oefenen met NumPy.
- Begin aan de derde programmeeropgave: zie de werkcollege pagina.

Volgende week:

- Meer NumPy.
- Plotten met matplotlib.

Programmeermethoden NA Week 8

Universiteit Leiden