Python voor Natuur- en Sterrenkundigen Week 2

Kristian Rietveld

http://liacs.leidenuniv.nl/~rietveldkfd/courses/pmpy2015/

Universiteit Leiden
The Netherlands

Vorige week

- `range()` voor karakters:

```
import string
for c in string.lowercase:
    print c,
```

Vorige week (verv.)

- Maken complexe getallen op basis van variabelen:

```
>>> a, b = 3, 4
>>> z = (a + bj)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
```

Vorige week (verv.)

- Maken complexe getallen op basis van variabelen:

```
>>> a, b = 3, 4
>>> z = (a + bj)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'bj' is not defined
>>> z = complex(a, b)
>>> z
(3+4j)
```

Deze week

- Meer over ingebouwde datastructuren
- Functies
- Files lezen/schrijven

Meer over lijsten

- We hebben tot nu toe alleen maar vooraf de lijsten gedefinieerd.
- We kunnen ook een lege lijst maken en dan elementen toevoegen en verwijderen.
- Let op dat je een lege lijst niet zomaar kunt indexeren!

Lege lijsten

```
>>> a = []
>>> a = list()
>>> a[4] = "test!"
Traceback (most recent call last):
File "<stdin>", line 1, in <module> IndexError: list
assignment index out of range
```

Lijsten manipuleren

- `append(obj)`: voeg `obj` toe achteraan de lijst.
- `insert(idx, obj)`: zet `obj` *voor* plek `idx`.
- `pop()`: geef en verwijder laatste element.
- `remove(obj)`: haal `obj` uit de lijst.
- `del lijst[i]`: verwijder `lijst[i]`.
- `del lijst[i:i+10]`: verwijder `lijst[i:i+10]`.

Lijsten manipuleren (verv.)

```
>>> a = []
>>> a.append("een")
>>> a.append("twee")
>>> a.append("drie")
>>> a.insert(0, "nul")
>>> a
['nul', 'een', 'twee', 'drie']
```

Lijsten manipuleren (verv.)

```
>>> a
['nul', 'een', 'twee', 'drie']
>>> a.remove("een")
>>> a.pop()
'drie'
>>> a
['nul', 'twee']
>>> a.pop()
'twee'
```

Lijsten manipuleren (verv.)

```
>>> b = range(5)
>>> b.pop(2)
2
>>> del b[1]
>>> del b[2]
>>> b
[0, 3]
```

Slicing

- Vorige week maakten we kennis met slicing. Slicing kent ook een stapgrootte.

```
start : eind : stap
```

- Eind-index telt niet mee.
- Elk van de delen mag worden weggelaten.
- Bij lijsten mag je ook toekenningen doen aan de slice.

Slicing (verv.)

```
>>> a = range(10)
>>> a[2:5]
[2, 3, 4]
>>> a[2:]
[2, 3, 4, 5, 6, 7, 8, 9]
>>> a[:5]
[0, 1, 2, 3, 4]
>>> a[2:8:2]
[2, 4, 6]
>>> a[::2]
[0, 2, 4, 6, 8]
>>> a[::3]
[0, 3, 6, 9]
```

Slicing (verv.)

```
>>> a = range(10)
>>> a[0:5] = ['a', 'b', 'c', 'd', 'e']
>>> a
['a', 'b', 'c', 'd', 'e', 5, 6, 7, 8, 9]
>>> a[5:5] = ['x', 'y', 'z']
>>> a
['a', 'b', 'c', 'd', 'e', 'x', 'y', 'z', 5, 6, 7, 8, 9]
>>> a[10:] = range(100, 110)
>>> a
['a', 'b', 'c', 'd', 'e', 'x', 'y', 'z', 5, 6, 100, 101,
102, 103, 104, 105, 106, 107, 108, 109]
>>> a[0:10] = []
>>> a
[100, 101, 102, 103, 104, 105, 106, 107, 108, 109]
>>> a[:] = []
>>> a
[]
```

Andere operaties op lijsten

- `x in lijst` operator: zit `x` in `lijst`?
- `index(obj)`: op welke index kunnen we `obj` vinden?
- `count(obj)`: hoe vaak komt `obj` in de lijst voor?

Lijsten nesten

- Je kunt lijsten toevoegen aan lijsten. Lijsten van lijsten!
- Je kunt dan over meerdere niveau's indexeren: `a[i][j][k]`.
- **Let op!** Geen multi-dimensionale arrays.

Lijsten nesten (verv.)

```
>>> a = [[1, 2, 3, 4, 5], ['a', 'b', 'c'], [], ['x']]
>>> for lijst in a:
...     print len(lijst),
...
5 3 0 1
>>> a[0][1]
2
>>> a[1][2]
'c'
```

Lijsten nesten (verv.)

```
>>> a = [[1, 2, 3, 4, 5], ['a', 'b', 'c'], [], ['x']]
>>> a[2][4]
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
IndexError: list index out of range
>>> b = [[1, 2, 3], 591243, ['a', 'b', 'c']]
>>> b[1][3]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'int' object has no attribute '__getitem__'
```

Tuples

- Een tuple is een geordende reeks.
- Tuples kunnen **niet** worden veranderd.
- Nesten, indexing, slicing mogelijk!

```
a = (1, 2, 3, 'a', 'b', 'c')
a[4]
a[4:8]
b = (a, 4, ('q', 'z'), 6)
```

Dictionaries (+)

- Soms is het handig om een 'lijst' te indexeren met iets anders dan een geheel getal.
- Dit kan in Python met een 'dictionary'.
- Vaak bekend als: associatieve array of hash table.
- We behandelen dictionaries niet uitgebreid, laten nu alleen enkele voorbeelden zien.

Dictionaries (verv.)

```
>>> d = dict()
>>> d["walter"] = "071-52700000"
>>> d["kris"] = "06-12345678"
>>> d["joop"] = "0123-524513"
# Value ophalen uit de dictionary aan de
hand van een key
>>> d["kris"]
’06-12345678’
```

Dictionaries (verv.)

```
>>> k = {}
>>> k[4,3] = "rood" # We maken hier gebruik van een tuple!
>>> k[1,2] = "blauw"
>>> k[9,4] = "zwart"
>>> len(k)
3
>>> k
{(1, 2): 'blauw', (9, 4): 'zwart', (4, 3): 'rood'}
>>> k[5,2]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: (5, 2)
>>> k[1,2]
'blauw'
```

Iteratietechnieken

```
for i in range(len(lijst)):
    print i, "-", lijst[i]
```

```
i = 0
for l in lijst:
    print i, "-", l
    i += 1
```

```
for i, l in enumerate(lijst):
    print i, "-", l
```

Iteratietechnieken (verv.)

```
lijst = [(1, 'a'), (2, 'b'), (3, 'c')]
for getal, letter in lijst:
    print getal, ",", letter
```

Iteratietechnieken (verv.)

```
lijst = [4, 13, 2, 8, 11, 5]
for l in reversed(lijst):
    print l,
# Geeft: 5 11 8 2 13 4
for l in sorted(lijst):
    print l,
# Geeft: 2 4 5 8 11 13
for l in reversed(sorted(lijst)):
    print l,
# Geeft: 13 11 8 5 4 2
```

Functies

- Functies worden gebruikt om code goed te structureren en duplicatie van code te vermijden.
- Een functie heeft een *naam*, *argumenten* (of parameters) en een *resultaat*.
- Types voor argumenten en resultaat worden **niet** expliciet opgegeven.

Functiedefinitie

```
def functienaam(arg1, arg2, ..., argn):
    blok van statements (met inspringen!)
    return iets
```

Functiedefinitie (verv.)

```
def hallo():
    print "hello world"

hallo()
```

Functiedefinitie (verv.)

```
def telop(a, b):
    c = a + b
    return c

q = telop(12354, 23451234)
```

Functiedefinitie (verv.)

```
def sommeer(lijst):
    som = 0
    for l in lijst:
        som += l
    return som

s = sommeer(range(10))
```

Functiedefinities (verv.)

```
def paar(a, b, c):
    # Gebruik tuple als returnwaarde
    return (a, a + b, a + b + c)

x, y, z = paar(1, 2, 3)
```

```
t = paar(1, 2, 3)
```
mag ook!

Opletten!

```
def telop(a, b):
    c = a + b
    return c

q = telop(12354, "hallo!!")
```

Actueel en formeel

Formele parameters

```
def telop(a, b):
    c = a + b
    return c

q = telop(12354, 7234)
```

Actuele parameters

Default arguments

- Het is mogelijk om een "standaardwaarde" op te geven voor elke **formele** parameter.
- In dat geval is het niet meer verplicht om het **actuele** argument op te nemen.

```
def teken_cirkel(x, y, straal, kleur="blauw"):
    # Code om een cirkel te tekenen.


teken_cirkel(0, 0, 10)
teken_cirkel(0, 0, 10, "rood")
teken_cirkel(0, 0)  # Mag *NIET*!
```

Keyword arguments

- In combinatie met default arguments wordt er veel gebruik gemaakt van keyword arguments (ook in bv. matplotlib).
- Handig in het geval van functies met veel parameters en veel standaardwaarden (denk bv. aan plot functies).
- Niet alle parameters hoeven dan expliciet te worden opgegeven.
- Actuele parameters nemen de vorm `naam=waarde` aan.
- Keyword arguments **moeten** aan het einde van de reeks van actuele parameters worden geplaatst.

Keyword arguments (verv.)

```
def test(x, y):
    return x + y

test(3, 4)
test(y=4, x=3)
test(y=4)    # FOUT!
```

Keyword arguments (verv.)

```
def teken_lijn(p1, p2, kleur="zwart", dikte=1.0,
pijl=None, stippel=False):
    # hier wordt de lijn getekend
    pass

teken_lijn( (10, 10), (100, 10), stippel=True,
dikte=2.0)
```

Docstrings

```
def mijn_functie(x, y, z):
    """Het is een goed gebruik om functies te documenteren, vooral wanneer
    je programma groter wordt of wanneer je functies schrijft die je in
    de toekomst kunt hergebruiken. In deze functie worden x en y bij elkaar
    opgeteld en dan vermenigvuldigd met z."""

    result = (x + y) * z
    return result
```

Voorbeeld: simpelsort

```
def simpelsort(l):
    for i in range(len(l)):
        # Zoek kleinste element in ongesorteerde stuk [i:]
        min_idx = i
        for j, el in enumerate(l[i:]):
            if el < l[min_idx]:
                min_idx = i + j

        # Wissel om
        if i != min_idx:
            l[i], l[min_idx] = l[min_idx], l[i]

# Test
l = [47, 10, 7, 3, 31, 75, 18, 21, 48, 79]
simpelsort(l)
print l
```

Globale structuur Python programma

- Geen main functie.
- Code hoeft niet verplicht in een functie te staan.
- Als we "globale" code en code in functies mengen, wat gebeurt er dan?
- Oppassen dat de code niet onoverzichtelijk wordt.
- Wat is nu een nette manier om een Python-programma te structuren?

Voorstel globale structuur

- `import' statements bovenaan het bestand.
- Zet alle code in functies.
- Maak ook een main functie.
- **Let op**: een functie **moet** zijn gedefinieerd *voordat* deze kan worden aangeroepen. Python kent **geen** functieprototypen.
- Later: functies verspreiden over meerdere bestanden.

Voorstel globale structuur (verv.)

```
# Eerst alle import statements
import sys

# Dan alle hulpfuncties
def hulpfunctie(a):
    print "Hello world, a=", a

# De main-functie
def main():
    q = 10354
    hulpfunctie(q)
    return 0

# En tenslotte de "globale" code die main
aanroept.
if __name__ == "__main__":
    sys.exit(main())
```

Files

- Het lezen en schrijven van bestanden van bestanden zal een belangrijk deel uitmaken van je "workflow".
- In C++ gebruikten we `ifstream` en `ofstream`. In Python hebben we het `file` object.

Maken file object

- Met de functie `open` maken we een `file` object:

```
f = open(bestandsnaam, modus)
```

- Modus is `"r"` voor lezen, `"w"` voor schrijven.

```
f = open("experiment.txt", "r")
f = open("resultaten.txt", "w")
```

- Sluit een `file` object met `f.close()`.

Bestand inlezen

- C++-achtige manier

```
f = open("test.txt", "r")
line = f.readline()
while line != "":
    print line,
    line = f.readline()
f.close()
```

Bestand inlezen (verv.)

- Dit is meer Python-achtig (en dus simpeler :)

```
f = open("test.txt", "r")
for line in f:
    print line,
f.close()
```

Bestand inlezen (verv.)

- Je kan ook een enkele (of elk gewenst aantal) bytes lezen

```
f.read(1)
```

- Maar heel vaak gaat het lezen van tekst-data in Python per regel.

Voorbeeld

```
f = open("getallen.txt", "r")
for line in f:
    line = line.rstrip("\n")
    a, b, c = line.split(" ")
    a, b, c = int(a), int(b), int(c)
    som = a + b + c
    print "Som:", som
f.close()
```

Schrijven naar bestanden

- Om te schrijven naar een `file` object kan je gebruik maken van de `write` methode of van `print`.
- Bij `write` **moet** de parameter een string zijn:

```
f.write("hello world")
f.write(42)              # NEE!
f.write(str(42))         # OK
```

Schrijven naar bestanden (verv.)

```
f = open("uitvoer.txt", "w")
print >>f, "Met print is het eenvoudiger"
print >>f, "Geheel getal: {0} Floating point
{1}.".format(51, 3.1412345)
f.close()
```

Als functieparameter

- Een `file` object kan als object gewoon als functieparameter worden gebruikt.

```
def mooi_formatteren(f, a, b, c):
    print >>f, "{0:4d} | {1:4d} | {2:4d}".format(a, b, c)


f = open("tabel.txt", "w")
mooi_formatteren(f, 12, 54, 50)
mooi_formatteren(f, 54, 34, 41)
# ...
f.close()
```

Begin eindopdracht

- Je kunt nu al een begin maken met de eindopdracht:
  - Inlezen van de bestanden.
  - Functies om een nette tabel af te drukken.
  - Functies om valversnelling te berekenen en km/h naar m/s om te zetten.
- Werk ook al buiten de werkcolleges aan de eindopdracht!

Volgende week

- NumPy
- matplotlib

Morgen werkcollege

- 11:15 - 13:00 uur
- Snelliusgebouw: zalen 303-308
- Aanwezigheidscontrole
- 1 verplichte opdracht voor 0.5 punt
- Beginnetje met de eindopdracht.