Pythoncursus

Opdrachtenserie 4

Tanja, Koen en Marein

september 2017

Opdracht 1 - Grafiekjes

Met wiskunde hebben jullie al grafiekjes moeten plotten op je rekenmachine. In deze opgave gaan we Python gebruiken om deze grafiekjes op een mooie manier op je computerscherm te toveren. Om grafieken te tekenen gebruiken we de library `matplotlib`. Je kunt `matplotlib` met de aanroep `plt.plot(xlijst, ylijst)` gebruiken. Hierbij is `xlijst` een lijstje met x-waarden en `ylijst` een lijstje met y-waarden waardoor de grafiek gaat lopen. Natuurlijk moeten de lijstjes even lang zijn: het eerste element uit de x-lijst hoort namelijk bij het eerste element uit y-lijst, enzovoort... Je kunt de grafiek, zodra je hem geplot hebt, tonen met `plt.show()`.

Hint: Zet eerst onderstaande imports in je programma:

```
import matplotlib.pyplot as plt
from math import pi, sin
from numpy import arange
```

a) Maak de functie `genSin(xlijst)` die een `xlijst` als argument krijgt en vervolgens een lijstje met y-waarden oplevert. Iedere y-waarde is dan de sinus van de bijbehorende x. Als je vervolgens `xlijst` en `ylijst` hebt berekend, kun je deze plotten met `plt` zoals eerder is uitgelegd.

Hint: Een lijstje met x-waarden tussen 0 en 2π met stappen van 0.1 kun je snel genereren met:

```
xlijst = arange(0,2*pi,0.1)
```

b) Nu de sinus gelukt is gaan we een parabool plotten. Maak hiervoor de functie `calcY(a,b,c,x)`. Hierin zijn a,b,c de coëfficienten in de formule $y = ax^2 + bx + c$. De functie levert vervolgens de y-waarde op die hoort bij de gegeven x-waarde.

Nu hebben we een lijstje met x-waarden nodig voor de plot-functie. Genereer nu deze lijst zoals bij de `sin`.

Nu we een `xlijst` hebben en een functie om de bijbehorende y-waarden te berkenen kunnen we dit samen voegen in een functie die een `ylijst` maakt. Maak nu de functie `genParabool(a,b,c,xlijst)`. Deze functie krijgt de a, b, c

en gemaakte `xlijst` en levert als resultaat een `ylijst` op. Deze functie moet gebruik maken van `calcY(...)`. Plot vervolgens het resultaat met `plt`.

c) Maak nu de functie `berekenTopX(a,b,c)` die de x-coördinaat van de top van de parabool berekent. De x-top kan berekend worden met de formule $\frac{-b}{2a}$. Maak vervolgens ook de functie `berekenTopY(a,b,c)`. De y-top kan natuurlijk berekend worden door x in de paraboolformule in te vullen.

d) Omdat de raaklijn van de top altijd een rechte lijn is, kun je hem nu met `plt` plotten: `ylijst` bevat hier voor iedere x-waarde het y-coördinaat van de top. Plot deze raaklijn in dezelfde grafiek als de parabool.

e) We kunnen nu parabolen plotten, maar het is natuurlijk veel leuker als we functies van iedere graad kunnen plotten (dus bijvoorbeeld $y = 3x^3 + 4x^2 + 8x + 2$). Om dit voor elkaar te krijgen gaan we een functie maken waarin de coëfficienten (dus de a,b,c,d enzovoort) als een lijst worden meegegeven. Maak daarom nu de functie `genPolynoom(coeffs, xlijst)`. `coeffs` is dan een lijstje van coëfficienten. Dit zou in het eerder genoemde voorbeeld dan zijn: $2, 8, 4, 3$. De index van de rij is dan de graad van de bijbehorende x. `xlijst` is dan weer een lijstje met x-waarden. De functie levert dan uiteindelijk een lijstje met y-waarden op, waarna je dit kunt gebruiken om de grafiek te plotten.

f) De afgeleide $f'(x)$ van een polynoom van bovenstaande vorm is vrij eenvoudig te berekenen. Maak de functie `genAfgeleide(coeffs)` die een lijstje coëfficienten van een polynoom krijgt, en vervolgens de coëfficienten van de afgeleide oplevert. De afgeleide van een polynoom is ook een polynoom, dus met `genPolynoom(..)` kun je die afgeleide vervolgens weer plotten.

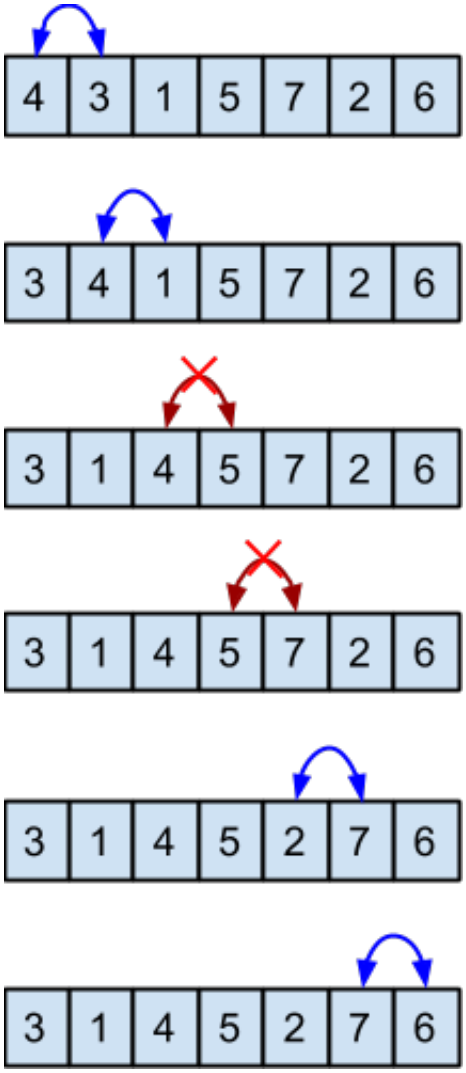
bonus We kunnen nu afgeleiden van polynomen plotten, maar nog niet van andere soorten wiskundige functies. In plaats dat we van elke soort functie een afgeleide uitrekenen, kunnen we ook een algemene functie maken die een afgeleide benadert. Je krijgt dan dus een rij van x- en y-waarden, en hiervan moet je functie een rijtje y-waarden van de afgeleide opleveren. De afgeleide ken je als het goed is al als $\frac{dy}{dx}$. Als dx kunnen we gewoon het verschil in twee opvolgende x-waarden nemen, en dy kan op dezelfde manier. Dit is slechts een benadering, hoe goed deze is, hangt af van hoe dicht de punten op elkaar liggen. Als de punten verder uit elkaar liggen wordt de afgeleide ook onnauwkeuriger.

Opdracht 2 - Sorteren

Zoals je misschien al gezien hebt, kan je in Python lijsten sorteren. Maar het is een goede oefening om dit zelf te kunnen programmeren. We gaan dus een sorteeralgoritme maken, die een lijst van getallen als invoer krijgt en een gesorteerde lijst teruggeeft. In deze opdracht gaan we uit van sorteren van klein naar groot (maar andersom mag ook, dit werkt precies hetzelfde). Er zijn verschillende algoritmes, waarvan de ene efficiënter is dan de andere.

We beginnen met *bubble sort*, een algoritme dat in kleine, begrijpelijke stapjes werkt. Hierbij ga je van voor naar achteren door de lijst, en je vergelijkt steeds twee (naast elkaar liggende) elementen met elkaar. Als het eerste element groter is dan het volgende, verwissel je ze. Het algoritme zorgt er zo voor dat het grootste getal achteraan

komt te staan. Zie de volgende afbeelding, waarbij twee elementen worden verwisseld bij een blauwe pijl, en ze onveranderd blijven bij een rode pijl:



a) Maak een loop dat deze eerste stap van bubble sort uitvoert, waarmee je het grootste element achteraan zet.

b) Als de stap in de vorige opdracht goed werkt, staat het grootste element op de goede plek. Het ongesorteerde deel van de lijst is nu de hele lijst behalve het laatste element. Voer dezelfde stap op dit ongesorteerde deel uit, om zo ook het op één na grootste getal op de juiste plek te zetten (de tweede plek van achteren).

c) Het niet-gesorteerde deel van de lijst wordt nu steeds korter, en het gesorteerde deel groeit steeds meer. Voer de code uit vraag *a* uit in een loop, zodat je de hele lijst sorteert. Misschien moet je de code uit opdracht *a* wat aanpassen, om het algoritme goed te laten werken (is bijvoorbeeld de `range` goed?).

d) Maak nu een functie die je een lijst als argument kan geven, en die de gesorteerde

lijst teruggeeft.

e) Test je code op willekeurig gevulde lijsten, van een zelfgekozen lengte. Hiervoor kun je de onderstaande code gebruiken. Wat gebeurt er bij een lege lijst?

```
import random

# genereert een willekeurig gevulde lijst van lengte n
def randomList(n):
    randoms = []
      for   in range(n):
          randoms.append(random.randrange(10000))
    return randoms
```

Bubble sort is niet zo'n efficiënt algoritme. Een efficiënter algoritme is *merge sort*. Hierin speelt *recursie* een belangrijke rol. Het algoritme werkt als volgt (lees door tot en met f voordat je begint met programmeren):

- Als je lijst een lengte heeft van 0 of 1, dan ben je meteen klaar. Een lijst met minder dan twee elementen kan namelijk nooit in de verkeerde volgorde staan en is dus altijd gesorteerd. Je kunt simpelweg de gegeven lijst weer returnen.
- Als de lijst langer is, knippen we hem op in twee helften. We sorteren de twee helften weer met ditzelfde algoritme. Dit is de magie van recursie: als we de rest van ons algoritme goed implementeren, kunnen we er hier op vertrouwen dat dit goed gaat!
- We hebben nu twee gesorteerde lijsten, waarvan we één gesorteerde lijst moeten maken. We kunnen een hulpfunctie maken die dit doet.

Opmerking: De recursie werkt, omdat de lijsten die je sorteert steeds kleiner worden. Een lijst van 100 wordt dus opgeknipt in lijsten van 50, die opgeknipt worden in lijsten van 25, etc. Uiteindelijk kom je altijd uit op lijsten van lengte 1. Deze korte lijsten worden apart afgehandeld (zoals hierboven beschreven) en hier stopt de recursie.

f) We hebben eerst een hulpfunctie nodig om merge-sort aan te kunnen. Deze hulpfunctie krijgt twee gesorteerde lijsten en maakt hiervan één gesorteerde lijst. Maak er gebruik van dat de lijsten al gesorteerd zijn, dat maakt het probleem gemakkelijker!

g) Maak nu een functie die merge-sort uitvoert. Doe niet meer stappen dan nodig, en vertrouw erop dat je de functie die je aan het maken bent, weer kunt gebruiken voor de kleinere lijsten.

h) Test het algoritme weer op willekeurig gevulde lijstjes.

Merge-sort is een stuk efficiënter dan bubble-sort. We gaan dit verschil nu visualiseren, door de tijd te meten die nodig is voor het sorteren, en dit in een grafiekje weer te geven. Hiervoor kun je het volgende stukje code gebruiken:

```
import time

# vraag de huidige tijd op
t = time.time()
```

i) Schrijf nu een stukje code dat een aantal (willekeurig gevulde) lijsten gebruikt, hierop beide sorteeralgoritmes uitvoert, en meet hoe lang beide algoritmes duren. Sla deze waarden op in een lijst zodat je met *pyplot* een grafiekje kunt tekenen. Teken het grafiekje zodra het meten klaar is, bijvoorbeeld voor de lengtes [100, 200, 400, 800, 1600].

j) Hoe verlopen de grafieken? Welk algoritme is sneller bij bijvoorbeeld een rij van 2.000 elementen?

Als het goed is, zie je dat de tijd om bubble-sort uit te voeren een parabool is. Als de lengte van de lijst n is, is de benodigde tijd dus ongeveer n^2. Merge-sort (en enkele andere efficiënte sorteer-algoritmes) kunnen dit in een tijd van ongeveer $n \cdot ^2\log(n)$. Dit neemt natuurlijk minder hard toe dan n^2. Hoe groter de lijst, des te belangrijker de snelheid — en des te groter het verschil!

Opdracht 3 - Een grafische gebruikersinterface

Deze opdracht is erg veel werk. Indien je geen zin hebt in het programmeren van een grafische gebruikersinterface mag je in plaats van deze opdracht ook Galgje van vorige week (af)maken indien je die nog niet af hebt.

In deze opdracht gaan we een simpel grafisch programma in elkaar zetten. Het maken van grafische programma's is erg veel werk. Daarom beperken we ons hier tot een simpele rekenmachine om je een beetje een idee te geven van hoe dit in zijn werk gaat, zodat je eventueel na afloop van deze cursus zelf uitgebreidere grafische programma's kunt maken, mocht je daar interesse in hebben.

Graphical User Interfaces (GUI's) maken zeer intensief gebruik van objecten, klassen en methoden. Voor grotere grafische programma's is het daarom van belang dat je goed weet hoe dit werkt (zie hoofdstuk 14 uit de Waterloo cursus). Voor deze opdracht proberen we het simpel te houden. Waarschijnlijk is deze opdracht goed te maken zonder dat je precies hoeft te weten hoe objecten werken.

Om een grafisch programma te maken heb je een GUI-library nodig (die je met `import` importeert). We gebruiken voor deze opdracht de library QT. Deze library bevat allerlei handige functies om een GUI in elkaar te zetten. QT ondersteunt naast Python ook andere talen/platformen, zoals C++, Android of Java. Alternatieve GUI-libraries zijn GTK, TCL/TK of wxWidgets. Voor een uitgebreide introductie van QT in Python kun je kijken op `http://zetcode.com/gui/pyqt4/`.

In het eerste deel van de deze opdracht gaan we een venster maken. Voor dit venster maken we een klasse IntroGUI. Dit object stoppen we in de variabele app en starten we met de code `gui = IntroGUI()`. Zodra je `gui`-code aanroept, wordt de code in de functie `__init__()` aangeroepen. Binnen de init-functie gebruiken we voor 'ingebouwde' GUI-methodes die over het venster zelf gaan het keyword `self`.

a) We gaan nu een voorbeeld bekijken. Kopieer en plak de code van `http://cs.ru.nl/pythoncursus/gui/intro.py` in een Python-bestand en voer het uit. Probeer te snappen wat er precies gebeurt. Indien je niet snapt wat er precies gebeurt, moet je even om hulp vragen omdat anders de rest van deze opgave erg lastig wordt. Het pictogram `aap.png` kun je vinden op `http:`

`//cs.ru.nl/pythoncursus/gui/aap.png` en moet je zetten in dezelfde map als het `intro.py` bestand.

Een leeg venster is natuurlijk een beetje saai. We gaan daarom nu de benodigde knoppen voor een rekenmachine toevoegen. Om de knoppen in het venster netjes te ordenen, hebben we een *layout* nodig. Er zijn verschillende layouts mogelijk, waar wij hier gebruik maken van een *gridlayout*[1]. Met een gridlayout kunnen we widgets in een 'grid' neerzetten: we kunnen een coördinatenstelsel gebruiken om de de widgets (in dit geval dus knoppen/buttons) op de goede plaats neer te zetten.

Buttons zelf worden gemaakt met een `QPushButton`. Als argument heeft dit widget een string nodig, dat de titel van je knopje wordt. Als we een button gemaakt hebben kunnen we hem aan de layout toevoegen.

Een voorbeeld van een enkele knop in een venster is te vinden op `http://cs.ru.nl/pythoncursus/gui/button.py`.

b) Breid de voorbeeldcode uit met knoppen voor de getallen 0 tot en met 9 en de +, -, /, * en de =. Doe dit zo slim mogelijk: maak gebruik van loops en functies om de buttons te maken en aan de gridlayout toe te voegen.

Nu we de buttons allemaal aangemaakt hebben, moeten we ze gaan 'verbinden' met een 'listener'. Een listener is een methode die wordt uitgevoerd zodra er op een knopje geklikt wordt. Hierdoor moet je in de `IntroGUI` klasse een methode aanmaken, en deze methode verbinden met je knop. In deze methode kun je bijvoorbeeld een `print(..)` stoppen, zodat er wat in de console geprint wordt.

Een voorbeeld van hoe je een knop met een listener verbindt is te vinden op `http://cs.ru.nl/pythoncursus/gui/button-listener.py`. Als je de voorbeeldcode begrepen hebt kun je nu listeners toevoegen aan al je knoppen. Je kunt voor iedere knop een aparte listener-functie maken, maar je kunt waarschijnlijk af met één listener en een slim if-statement.

c) Koppel nu listeners aan de knoppen in de rekenmachine. Zorg ervoor dat de gebruiker op willekeurige knoppen kan drukken, om zo een som uit kunnen rekenen. Print het resultaat van je berekening (nu nog in de console) zodra de gebruiker op de knop = drukt. Voordat hij op = drukt moet je dus bijhouden op welke andere knoppen er gedrukt is. De rekenvolgorde kan nog lastig zijn: $2 + 3 * 5 = 17$, en niet 25. Dit is een erg lastig probleem: het is mooi als je het opgelost krijgt, maar je mag het ook laten zitten en verder gaan.

De rekenmachine zou nu moeten werken, maar de uitvoer komt helaas nog steeds op de console te staan en niet in het venster met de rekenknoppen. Daar gaan we nu aan werken. Naast de standaard buttons, zijn er ook nog andere soorten widgets zoals tekstlabels en invoervelden.

Op dit moment maken we gebruik van een gridlayout om de knoppen netjes te rangschikken. Een gridlayout is hier ook zeer geschikt voor. Het uitvoerveldje van de rekenmachine past echter niet in de 'grid', maar hoort erboven te staan. We zullen daarom gebruik gaan maken van een *vboxlayout*, waarmee je widgets (of groepen van widgets — dus layouts) onder elkaar kunt plaatsen.

[1] Zie `http://zetcode.com/gui/pyqt4/layoutmanagement/` voor andere mogelijke layouts

Om tekst te tonen gebruiken we de widget `QLabel`. De tekst van dit widget kan met de methode `setText()` gewijzigd worden.

Een voorbeeld van een QLabel in combinatie met een grid en vboxlayout is te vinden op `http://cs.ru.nl/pythoncursus/gui/qlabel.py`.

d) Vervang de `print(..)` in je programma door een `QLabel` en voeg een vboxlayout aan je rekenmachine toe. Als het goed is moet je rekenmachine nu volledig werken zonder console.

In plaats van een `QLabel` kun je ook een `QLineEdit` gebruiken. Hiermee kun je de tekst ook aanpassen. Zodra de tekst aangepast wordt kun je daar weer een melding in een listener van krijgen. Een voorbeeld is te vinden op `http://cs.ru.nl/pythoncursus/gui/qlineedit.py`

e) Vervang de `QLabel` door een `QLineEdit`. Zorg er ook voor dat de gebruiker in plaats van op de knoppen te drukken een som in het QlineEdit tekstvakje in kan voeren en deze uit kan rekenen door op de knop = te drukken.

Naast knoppen kunnen we ook menubalken of toolbars toevoegen. Hiervoor moeten we een `QMainWindow` in plaats van een `QWidget` gebruiken. Dit zorgt ervoor dat een layout aan je venster toevoegen wat lastiger wordt. Een voorbeeld van een menubalk is te vinden op `http://cs.ru.nl/pythoncursus/gui/menubar.py`. Een voorbeeld van een toolbar is te vinden op `http://cs.ru.nl/pythoncursus/gui/toolbar.py`.

bonus Mocht je nog niet genoeg gekregen hebben van GUI's, dan kun je nu een menubalk en/of een toolbar aan je programma toevoegen met behulp van bovenstaande voorbeeldcode.