Het relaas van de beginnende programmeur

Het hoe en waarom van de assistent

1. Help, mijn code doet niks…

Mogelijke oplossingen:

- *Heb je op run geduwd (groene pijltje)*? Zolang je niet op 'run' duwt, kent de python je code niet. Code is gewoon tekst die, door op run te drukken, vertaald wordt naar een voor de computer verstaanbare taal. De Wing editor zal de commando's die je in je .py bestand hebt geschreven van boven naar onder 'uitvoeren', net zoals je ze zelf één voor één in de shell zou
- *Bevat je code fouten?* Je hebt wel degelijk op run geduwd, maar je code bevat fouten. Stel je bestand heet 'test.py', en je drukt op run. Indien er geen fouten in je code staat, verschijnt het volgende in je shell:

```
>> [evaluate test.py]
```

Indien je fouten in je code hebt, zal er foutmelding komen. Volgende is eenvoorbeeldje wat je dan zou krijgen:

```
>> [evaluate test.py]
Traceback (most recent call last):
File "<string>", line 137, in <fragment>
invalid syntax: <string>, line 137, pos 16
```

Dit betekent dus dat je ergens verkeerde syntax hebt gebruikt. Hier op lijn137, ergens rond positie 16 bijvoorbeeld ':' vergeten achter een if- of whileconditie.

Geen enkele functie zal werken zolang de fouten niet zijn opgelost!

- *Heb je een import van je huidige bestand in de shell gedaan?* Stel je hebt netje code geschreven in het bestand 'test.py'. Je runt je code, en schrijft dan inde shell 'import test'. Gezien je huidige bestand 'test.py' is, kan je reeds aanalle functies van dit bestand door op run te duwen, een import moet dan nietgedaan worden.
- *Heb je de functie die je nodig hebt, opgeroepen?* Ken het verschil tussen een functie definitie, en een functie aanroep. Volgend is een functie definitie:

```
def test():
    print "hallo"
```

Zoals het woord “functie definitie” al zegt, definieert het gewoon wat de functie met de naam “test” moet doen (‘hallo’ afprinten). Het blijft een definitie, en de functie wordt pas geactiveerd als men ze aanroept:

```
>> test()

hallo
```

Let ook op: een functie definitie **en** aanroep gebeurt **altijd** met ronde haakjes! Volgend is dus niet goed:

```
>> test

<function test at 0x0110D770>
```

Alles in Python zijn objecten, functies dus ook. Als je de functie naam schrijft zonder haakjes, geeft python je het functie object terug: <type_object (een functie in dit geval) naam_object(test) at adres_in_geheugen>

2. Help, mijn functie geeft niets terug …

- Ken het verschil tussen de keywoorden **print** en **return**. Print doet letterlijk wat het zegt, iets afprinten in de shell. Return met een waarde erachter, zal die waarde terugkeren wanneer er uit de functie wordt gesprongen.

```
def test1():
    print “hallo”
```

```
def test2():
    print “hallo”
    return “hallo”
```

Als we een oproep doen naar beiden dan:

```
>> mytest1 = test1()
hallo
>> mytest2 = test2()
hallo
>> mytest1
None
>> mytest2
‘hallo’
```

De variabele mytest1 bevat dus niks, en mytest2 de string “hallo”. Let op: code na het return statement wordt nooit uitgevoerd: indien een functie een ‘return’ statement bereikt, zal deze return worden uitgevoerd waarna de functie wordt beëindigd. Meestal schrijven we de return dus als allerlaatste regel, of gebruiken we return om vroegtijdig uit een loopje te springen (en dat is dan altijd in combinatie met een if-conditie), bijvoorbeeld:

```
for i in range(len(myword)/2):
   if myword[i]!=myword[len(myword)-i-1]:
      return False
return True
```

Als de letter op plaats *i* en op plaats *len(myword)-i-1* niet gelijk is, komt hij een return tegen en zal hij vroegtijdig de functie gaan. Dit betekent dat hij nooit aan het statement 'return True' zal komen indien het woord geen palindroom is.

3. Parameters, hoezo parameters???

Stel volgende functies in het bestand 'test.py':

```
def printMyList1():
   mylist = [1,2,3,4]
   for i,v in enumerate(mylist):
      print i,':',v

def printMyList2():
   for i,v in enumerate(mylist):
      print i,':',v

def printMyList3(mylist):
   for i,v in enumerate(mylist):
      print i,':',v
```

Voorbeelden van oproep zijn dit:

```
>>> printMyList1()
0 : 1
1 : 2
2 : 3
3 : 4
>>> printMyList3([1,2,3,4])
0 : 1
1 : 2
2 : 3
3 : 4
>>> printMyList3([4,5,6,7])
0 : 4
1 : 5
2 : 6
3 : 7
>>> printMyList2()
Traceback (most recent call last):
  File "C:\Program Files\Wing IDE 101 4.0\src\debug\tserver\_sandbox.py", line 1, in <module>
    # Used internally for debug sandbox under external interpreter
  File "C:\Program Files\Wing IDE 101 4.0\src\debug\tserver\_sandbox.py", line 7, in
printMyList2
NameError: global name 'mylist' is not defined
>>>mylist = [8,9,10]
>>> printMyList2()
0 : 8
1 : 9
2 : 10
```

Geen probleem, hoor ik jullie denken. Hierbij horen dus wel volgende opmerkingen:

- printMyList1 zal nooit een andere lijst kunnen printendan [1,2,3,4]
  - Ja maar, dan verander ik de waarden in de lijst toch gewoon! → Stel je code wordt verwerkt in een bibliotheek van functies. De bedoeling is dat de gebruiker deze functies krijgt en niet in de code hoeft te gaan kijken. Hij wilt dus zeker niet elke keer de functie moeten aanpassen. Meestal worden zulke bibliotheken enkel in computertaal gedistribueerd en kan hij ze zelfs niet aanpassen. Dit bedoelen we dus met maak je code zo algemeen mogelijk. Je zou een functie moeten schrijven, zodat ze niet hoeft aangepast te worden indien gebruikte waardes veranderen.
- printMyList2 heeft een groot probleem: een variabele *mylist* wordt gebruikt maar deze is niet gedefinieerd binnen de functie.
  - Geen probleem, denk je. Ik definieer ze wel in de shell → Ok dit zal werken als je in je bestandje test.py blijft werken. (Let op: Het blijft een slechte programmeer gewoonte, ook in dit geval!)
  - Stel je wilt test.py importeren in een ander bestand, en de functie gebruiken. Volgend zal niet werken!!!

```
import test

mylist = [1,2,3,4]
test.printMyList2()
```

  - Alweer geen probleem, denk je. Ik voeg gewoon 'mylist=[1,2,3,4]' ergens toe aan mijn bestandje test.py! → Hier zit je weer met hetzelfde probleem als met de functie 'printMyList1', telkens je een nieuwe lijst wilt afprinten, moet je deze gaan aanpassen in het bestand test.py.
- De enige goede oplossing die ruimte voor verandering geeft zonder je functie aan te passen is dus de functie 'printMyList3', waarbij je de te printen lijst als argument met de functie meegeeft.

4. Hoe kan ik die waarde uit de functie krijgen?

Wanneer een functie een return statement bevat (dit is niet verplicht) dan geeft deze functie een bepaalde waarde terug. Deze waarde kan je dus gewoon toewijzen aan een variabele. Stel:

```
Mijnnaam = 'Isabel'

Mijnzin = test2() + ' ' + Mijnnaam
```

Python evalueert je code steeds van links naar rechts en van binnen naar buiten, steeds rekening houdend met de voorrangsregels. Zo denkt Python als hij bovenstaande code ziet (tweede regel) :

➔ Ik zie een variabele met naam 'Mijnzin' (aangezien het geen string is want er staan geen '' rond)

➔ Ik zie een toewijzing (=), ik moet dus alles na dit teken evalueren.

- ➔ Achter = staat een samengestelde operatie, de eerste operand die ik tegen kom is '+', ik mag dus het eerste deel evalueren.
- ➔ Het eerste deel is test2(), dit is een functie aanroep, ik doe deze dan ook (en we veronderstellen ook dat de functie gedefinieerd is). Stel even dat de functie test2()de string 'hallo' teruggeeft.
- ➔ Het tweede deel van de '+' bewerking is een string, ik plak die aan het eerste deel (voorlopig is onze string dus 'hallo ')
- ➔ Hetvolgende wat hij tegen komt is de '+' operatie, dus hij kan het deel erna weer evalueren
- ➔ Dit volgende deel is een variabele, dus hij haalt de waarde hiervan op
- ➔ Hij voert opnieuw de '+' bewerking tussen strings uit: hij plakt de waarde ('Isabel')achter de eerder berekende string. De gehele string is dus nu 'hallo Isabel'.
- ➔ Hij wijst deze string aan de variabele 'Mijnzin' toe

5. Als ik een lijst aan mijn functie meegeef dan doet hij iets vreemds…

Als je aan een functie een string of een getal als argument meegeeft dan zal de waarde van die variabele niet wijzigen na de uitvoering van de functie:

```
def f(a):
  a = a * 2

b = 10
f(b)
print b(=10)
```

Wanneer je wil dat je 'b' toch van waarde verandert dan zal je met een 'return' moeten werken:

```
def f(a):
   a = a * 2
   return a

b = 10
b = f(b)
print b(=20)
```

Let op: wanneer je een lijst aan een functie meegeeft dan kunnen bepaalde wijzigingen die in de functie gebeuren WEL worden doorgegeven naar de lijst die je als argument had meegegeven:

```
def f(a):
   a.append(40)

b = [10,20,30]
f(b)
print b(=[10,20,30,40])
```

Dit gebeurt ook wanneer we bijvoorbeeld nog een nieuwe lijst ‘c’ zouden definiëren:

```
def f(a):
    c = a
    c.append(40)

b = [10,20,30]
f(b)
print b(=[10,20,30,40])
```

Om dit gedrag te vermijden kunnen we de lijst ‘c’ beter aanmaken als *nieuwe* lijst waarin de waarden uit a worden gekopieerd:

```
def f(a):
    c = a[:]
    c.append(40)

b = [10,20,30]
f(b)
print b(=[10,20,30])
```

6. Hij wil niks tekenen!

Ben je zeker dat je VPython correct hebt geïnstalleerd? Ben je zeker dat de juiste import van VPython bovenaan je code staat?

Indien alles juist is en je VPython nog steeds niet werkt dan kan je eens proberen om Wing te sluiten en terug op te starten, dit blijkt soms te helpen!

Merk op dat de 64bit versie van Python niet werkt met VPython!

7. Mijn programma doet iets fout maar ik weet niet waar…?!?

Elke programmeur, beginnende of ervaren, maakt vaak mee dat zijn programma niet exact doet wat hij/zij verwacht dat het programma zou doen. Aangezien Wing in dit geval niet kan aangeven op welke regel er een fout zit, is het soms moeilijk om te vinden waar het in je codejuist mis loopt. Een goede reflex is in dit geval het afprinten van de waarden van een bepaalde variabele uit je programma. Door op één of meerdere plaatsen een ‘print …’ in je code te zetten kan je inspecteren of de waarden van je variabelen variëren zoals je zou verwachten. Bijvoorbeeld, indien je programma niets of iets compleet fout tekent, dan kan je eens enkele coördinatenuit je code afprinten om na te gaan welke waarden deze aannemen…

8. Hoe werkt een for-lus nu eigenlijk?

```
for i in range(aNumber):
   print 'index = ', i
```

Voordat de lus wordt uitgevoerd, zal Python eerst checken of er een lijst staat achter de '**in**'. Indien er een expressie staat, zal hij deze dus eerst uitvoeren. In een bovenstaand voorbeeld zal Python dus volgende stappen uitvoeren (stel aNumber=3):

1) **Evalueer** de expressie '*range(aNumber)*'
   ➔ **[0,1,2]**
2) **Begin** de for-lus
   1. **Neem** *i=0*
   2. **Evalueer '***print 'index= ', 0***'**
      ⇨ 'index = 0'
   3. **Neem** *i=1*
   4. **Evalueer '***print 'index= ', 1***'**
      ⇨ 'index = 1'
   5. **Neem** *i=2*
   6. **Evalueer '***print 'index= ', 2***'**
      ⇨ 'index = 2'
   7. **STOP**

'*aNumber*' moet dus vóór de for-lus zijn gedefinieerd. Hetzelfde geldt voor voor het rechtstreeks gebruik van lijsten, vb:

```
1) for v in mylist:
      print 'value = ', v
2) mylist = []
   for i in range(len(mylist)):
      mylist.append(aValue)
```

In 1) is er geen probleem indien '*mylist*' is gedefinieerd. 2) is een veel voorkomende fout. Gezien het allereerste wat wordt geëvalueerd de expressie '*range(len(mylist))*' is, zal '*len(mylist)*' de waarde 0 teruggeven, en '*range(0)*' zal vervolgend een lege lijst teruggeven. De lijst wordt opgebouwd tijdens de uitvoering van de for-loop, en zal pas volledig zijn wanneer er uit de for-lus wordt gesprongen.

9. Hoe kan ik een animatie programmeren?

Stel dat je een rechthoek wil laten bewegen over het scherm. Je zal deze rechthoek eerst moeten aanmaken:

```
mybox = box(pos=(0,0,0))
```

Let op: de rechthoek heeft nu dus de naam ‘mybox’ gekregen. Dit kan ik doen, omdat het box-commando (net als sphere(), arrow(), etc.) een object aanmaken. Telkens als je het box() commando uitvoert zal er dus een nieuw object worden aangemaakt en bijgevolg een nieuwe rechthoek worden getekend. Wanneer we nu de bestaande rechthoek willen laten bewegen, moeten we er uiteraard geen nieuwe aanmaken maar enkel de positie van de bestaande wijzigen. Je kan de positie van een tekenobject aanspreken door ‘.pos’. We kunnen dus een nieuwe positie toekennen als volgt:

```
mybox.pos = (1,2,3)
OF
mybox.pos.x = 1
mybox.pos.y = 2
mybox.pos.z = 3
```

Een tekenobject heeft nog een ander veldje, met de naam ‘.velocity’. Hierin kan je een vector opslaan. Je kan dus bijvoorbeeld:

```
mybox.veloctity = vector(5,5,0)
```

Deze vector kan je dan later gebruiken om stap voor stap de positie van je object aan te passen:

```
mybox.pos  = mybox.pos + mybox.velocity * dt
```

waarbij ‘dt’ je tijdsstap voorstelt. Dit zet je uiteraard in een for- of while-lus. Zet je stap niet te groot want anders merk je de afzonderlijke stapjes in de beweging. Verder zal je de lus waarin je de aanpassing van de positie doet niet te snel mogen laten uitvoeren. Dit doe je door het rate() commando in die lus te zetten.

Aan de studenten: Indien er nog belangrijke/interessante vragen zijn, gelieve deze per e-mail door te geven. Wij zorgen dat ze toegevoegd worden aan dit document.