Heuristieken – Python Code Review

Chi Chun Wan - 2525244
Shabaz Sultan - 2566703
Boudewijn Zwaal - 1897527

February 6, 2015

1 Code Stijl

1.1 Getters & Setters

De structuur van het Python programma lijkt erg sterk de Java code te volgen. Aangezien dit een port is van Java code en de Java code correct werkt is dit niet zo gek. Er zijn echter een aantal dingen die anders kunnen en tot leesbaardere en idiomatischere code zouden lijden. Het eerste voorbeeld zijn getters en setters. Neem de `City` klasse.

```
class City(object):
    def __init__(self, index, name, x, y, distances):
        self.index = index
        self.name = name
        self.x = x
        self.y = y
        self.distances = distances

    def getIndex(self): return self.index
    def getName(self): return self.name
    def getX(self): return self.x
    def getY(self): return self.y
    def getDistances(self): return self.distances
    def getDistanceTo(self, index): return self.distances[index]
```

Getters en setters in een taal als Java worden voor twee redenen gebruikt: om data toegankelijk te maken die m.b.v. access modifiers niet toegankelijk zijn gemaakt om redenen van encapsulatie en om extra gedrag te hangen aan de actie van lezen of schrijven van een stuk data. In Python heb je geen echte encapsulatie m.b.v. access modifiers (i.e. private, public, protected) en is het beter om attributen aan te spreken met `foo.bar` in plaats van `foo.getBar()`. Als je extra gedrag wilt binden aan het lezen of schrijven van variablen kan dit in Python met properties. Bijvoorbeeld de `Flight` klasse zou als volgt herschreven kunnen worden.

```
class Flight(object):
    DOCKING_TIME = 60
    REFUEL_TIME = 60
```

```
def __init__(self, passengers, departurTime, departureCity,\
                arrivalTime, arrivalCity, distance, refuel):
    self.passengers = passengers
    self.departurTime = departurTime
    self.departureCity = departureCity
    self.arrivalTime = arrivalTime
    self.arrivalCity = arrivalCity
    self.distance = distance
    self.processTime = self.DOCKING_TIME
    self.refuel = False
    if(refuel): self.setRefuel()
    self.waitingTime = 0


def wait(self, time): self.waitingtime += time

def removeWaitingTime(self): self.waitingTime = 0

@property
def travelTime(self): return self.arrivalTime - self.departurTime

@property
def groundTime(self): return self.processTime + self.waitingTime

@property
def totalTimeFlight(self): return self.groundTime + self.travelTime

@property
def flightPoints(self): return self.passengers * self.distance

def setRefuel(self):
    if(not self.refuel):
        self.refuel = True
        self.processTime += self.REFUEL_TIME

def removeRefuel(self):
    if(self.refuel):
        self.refuel = False
        self.processTime -= self.REFUEL_TIME
```

het aanspreken van properties is gelijk qua syntax als het direct aanspreken van een instantie variable. Dit is express; het direct aanspreken van variable in objecten is bedoeld als de standaard manier van het aanspreken van instantie variable en de gelijke syntax van aanspreken met properties is bedoelt om zonder te veel frictie tussen de twee te bewegen. Het valt te betwisten of de getter/setter stijl zelfs in Java een goed idee is, maar zeker in Python wil je dit vermijden.

1.2 Nadruk ontwerp in klasses

Een andere gewoonte die Java benadrukt is dat alles een klasse is en dat de architectuur van een stuk software vaak een grote klasse hiërarchie is. Vaak is het ok (of zelfs beter) om data in simpelere datastructuren te zetten en code in losse functies. Waarom bestaat `DistanceMap` bijvoorbeeld? Het bevat enkel een lijst aan `City` objecten, wat is er mis mee dit gewoon enkel als Python list in de code te zetten? De interface van de klasse bestaat uit een methode die genoemde lijst geeft en een methode die een element uit een lijst geeft, iets wat al tot de ingebouwde interface van een python list behoord. En als we kijken naar hoe de klasse wordt gebruikt zien we dat een object wordt geïnstantieerd en de lijst wordt opgehaald met een methode. Daarna wordt de klasse nooit meer gebruikt en wordt enkel de `cities` list direct aangesproken. In denk dat instinctief je al doorhad dat het omhulsel van een klasse om de list geen nut heeft, dus laat die dan ook maar weg.

1.3 For loops

Ik zie het volgende patroon af en toe in de code (bijv. de for loop in `isValid` van `Airplane`):

```
X = [member_one ,member_two , member_three]
for i in range(len(X)):
    do_stuff_with(X[i])
```

Als eerste kleine suggestie; raak in de gewoonte om `xrange` i.p.v. `range` in for loops te gebruiken. Het maakt in dit geval niet uit, maar voor een for loops die over grote aantallen loopt, bijv. een miljard iteraties zaal de `range` ongeveer een GB aan geheugen alloceren om alle mogelijke waarden die de loop kan aannemen op te slaan. Dit is natuurlijk niet nuttig, je hebt enkel een variable nodig die alle mogelijke waarden kan aannemen tijdens het draaien van de loop; met `xrange` gebeurd er wat je zou hopen dat er gebeurd.
De tweede is dat Python enkel een 'for each' loop heeft, een for loop die over een aantal elementen in een lijst wilt itereren. In oudere versies van Java en C++ moest je expliciet een iteratie counter aanmaken en deze ophogen om elementen in een lijst af te lopen. De bovenstaande code wordt vaak geschreven door nieuwe Python programmeurs die bekend zijn met één van deze talen en is een teken dat je de eigenschappen van Python's for loop nog niet compleet hebt geïnternaliseerd. De volgende code is iets simpler en meer idiomatisch:

```
X = [member_one ,member_two , member_three]
for i in X:
    do_stuff_with(i)
```

2 Code geldigheid vluchten

Specifiek aan de code voor deze opdracht zitten er wat haken en ogen aan de `isValid` code van `Airplane`. Deze ziet er als volgt uit:

```
1       def isValid(self, city):
2           distance = self.currentCity.getDistanceTo(city.getIndex())
```

```
3           flightTime = (distance / self.AIRPLANE_SPEED) * 60
4           time = 0
5           for i in range(self.numberOfFlights):
6               time += self.route[i].getTotalTimeFlight()
7           if(distance < (self.MAX_DISTANCE - self.distanceCovered)):
8               if(flightTime < (self.MINUTES_PER_DAY - time)):
9                   return True
10              else: print "Cannot fly to", city.getName(), ", \
11  ␣␣␣␣␣␣␣␣␣␣␣␣␣␣will not arrive before 02:00."
12          else: print "Cannot fly to", city.getName(), ", \
13  ␣␣␣␣␣␣␣␣␣␣needs refueling in", self.currentCity.getName()
14          return False
```

We hebben de derde regel herschreven naar:

```
flightTime = (distance / float(self.AIRPLANE_SPEED)) * 60.0
```

Het belangrijkste hier is de `float()`. In de originele versie zijn `distance` en `self.AIRPLANE_SPEED` beide een integer getal. In Python (specifiek versie 2) doet de '/' division operator een integer division, i.e. $1/3 = 0$ en niet $1/3 = 0.333$. Deze designfout in de taal is in Python 3 opgelost, maar het is duidelijk aan de rest van de code dat deze voor versie 2 is geschreven. Door `float()` om één van de variable te zetten wordt deze naar een floating point getal omgezet en wordt de floating point division aangeroepen. Dit is waarschijnlijk wat je verwacht, anders wordt je vlucht tijd ontzettend naar beneden afgerond en kan je enkel hele uren vliegen.
We hebben in dezelfde functie regel 7 als volgt herschreven:

```
if(distance <= (self.MAX_DISTANCE - self.distanceCovered)):
```

Als er `<` i.p.v. `<=` staat kan niet de hele tank leeg worden gevlogen. De regel erna is herschreven naar

```
if (flightTime + time - self.MINUTES_PER_DAY) <= 0.0001:
```

Dit is in twee stappen gedaan. Als eerst is de `<` herschreven naar `<=` zodat er ook nog in de laatste minuut gevlogen kan worden. Anders zijn er maar effectief 1199 minuten om te vliegen i.p.v. 1200. De tweede stap is om de `self.MINUTES_PER_DAY` naar de linkerkant van de ongelijkheid te halen en rechts i.p.v. een nul een kleine epsilon te zetten. Omdat `flightTime` nu een floating point getal is kan de ongelijkheid anders niet goed omgaan met vluchten die exact de minutes per day vullen.

3 PyPy

Een laatste ding dat mogelijk handig is om toekomstige gebruikers van de Python code op te wijzen is het bestaan van PyPy. Python is een taal die het programmeren een stuk simpler kan maken, maar de standaard 'cPython' interpreter is erg traag, o.a. omdat er geen modernere technieken zoals JIT compilatie in zitten. PyPy (een Python interpreter geschreven in een subset van Python) heeft deze technieken wel en is hierdoor veel sneller. Ons team heeft exclusief simulaties gedraaid met PyPy, omdat het anders niet doenlijk was simulaties in een redelijke tijd te draaien.