Programmeerimise süvendatud algkursus ITI0140

2014

Klassid ja objektid

- Klassi võib vaadelda kui seotud muutujate ja funktsioonide kogumit
  - Näiteks **koer** ja **kass** on klassid
    - sätestavad, millised koer ja kass üldiselt võiks olla
    - kassil ja koeral võivad olla näiteks:
      - nimi
      - karvkatte värv
      - jalgade arv
    - ja omadustest:
      - häälitsemine
        - haukumine
        - näugumine
- Objekt on seevastu konkreetse klassi üks eksemplar (isend)
  - Pitsu ja Muri on konkreetsed koerad, kummalgi:
    - oma nimi
    - karvkatte värv
      - must
      - pruun

Lihtne klass Point2D

```
class Point2D:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def print_point(self):
        print("(%s, %s)" % (self.x, self.y))
```

- **__init__** on meetod, mille abil me algväärtustame ühe konkreetse antud tüüpi (**Point2D**) objekti
  - **self** argument on Pythoni klasside meetodite puhul esimesena kohustuslik, et meetodi sees saaks konkreetse objekti atribuutidele ligi
  - **x** ja **y** määravad, et ühe punkti algväärtustamiseks on vaja anda kaks väärtust, st x ja y koordinaat
- **print_point** on meetod, mis trükib välja punkti väärtused kasutades sisemist olekut, st meelde jäetud atribuutide väärtusi

Lihtne klass Point2D

```
class Point2D:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def print_point(self):
        print("(%s, %s)" % (self.x, self.y))

p1 = Point2D(1, 2)
p2 = Point2D(0, 7)
p1.print_point()
p2.print_point()
```

```
(1, 2)
(0, 7)
```

- uue objekti loomisel kasutame klassinime ja anname argumendid (konstruktor meetod)
  - **self** on vaikimisi "kaasas"
- antud näites tekivad kaks punkti eri koordinaatidega, mida salvestame objekti sisemistes muutujates (neid nimetame atribuutideks)
- objekti atribuutide ja meetodite poole pöördume läbi punkti
- kumbki punkt omab oma olekut, st antud juhul teab oma koordinaate, mida **print_point** väljakutsel kasutab

Objektide loomine ja kasutamine

```
p1 = Point2D(1, 2)
p2 = Point2D(0, 7)
p1.print_point()
p2.print_point()
print(p1.x)
print(p2.x)
p1.x += 3
p1.print_point()
p2.print_point()
```

```
(1, 2)
(0, 7)
1
0
(4, 2)
(0, 7)
```

- konkreetse objekti atribuute võime kasutada ja muuta
- muutes ühe objekti atribuute, teise atribuudid ei muutu
  - kumbki objekt säilitab oma oleku

Pärimine, keerulisem klass Point3D

```
class Point3D(Point2D):
    def __init__(self, x, y, z):
        pass

p3 = Point3D(1, 2, 3)
p3.print_point()
```

```
Traceback (most recent call last):
  File "---.py", line 24, in <module>
    p3.print_point()
  File "---.py", line 7, in print_point
    print("(%s, %s)" % (self.x, self.y))
AttributeError: 'Point3D' object has no
attribute 'x'
```

- Klasse on võimalik laiendada, seda nimetame klasside pärimiseks
  - Pärimisel määrame ära klassi, mida laiendame, st **Point2D**
- Täiendatud klass **Point3D** on antud näites "kõvem" punkt, omab ka kolmandat koordinaati
- Pärimisel omandab uus klass kõik atribuudid ja omadused, mis on ülemklassil
- Antud näites on aga midagi puudu, **print_point** meetod ei paista teadvat, x ja y koordinaatidest midagi

Pärimine, keerulisem klass Point3D

```
class Point3D(Point2D):
    def __init__(self, x, y, z):
        Point2D.__init__(self, x, y)

p3 = Point3D(1, 2, 3)
p3.print_point()
```

```
(1, 2)
```

- Selleks, et x ja y väärtused meelde jäetaks, kutsume välja ülemklassi **__init__** meetodi andes seekord kaasa ka **self** muutuja
  - Samas me ei defineeri seda, kuidas tegelikult need x ja y meelde jäetakse
- Nüüd **print_point** väljakutsel on meil väärtused olemas
- Tegelikult aga oleks vaja veel, et ka kolmas koordinaat meelde jäetaks ja välja trükitaks

Pärimine, keerulisem klass Point3D

```
class Point3D(Point2D):
    def __init__(self, x, y, z):
        Point2D.__init__(self, x, y)
        self.z = z
    def print_point(self):
        print("(%s, %s, %s)" % (self.x, self.y, self.z))

p3 = Point3D(1, 2, 3)
p3.print_point()
```

```
(1, 2, 3)
```

- Klasside laiendamisel on oluline ka omaduste täiendamine ja muutmine
- **print_point** täiendamisel võime sobival hetkel ka __init__ meetodile sarnaselt kutsuda välja ka **Point2D.print_point** meetodi
  - Antud juhul kirjutame **print_point** meetodi küll täiesti üle ning anname talle uue sisu, mis vastab **Point3D** klassile

Objektide loomine ja kasutamine

```
p2D = Point2D(3, 5)
p3D = Point3D(2, 3, 8)
p2D.print_point()
p3D.print_point()
p3D.x -= 1
p2D.print_point()
p3D.print_point()
p2D.x += 1
p2D.print_point()
p3D.print_point()
```

```
(3, 5)
(2, 3, 8)
(3, 5)
(1, 3, 8)
(4, 5)
(1, 3, 8)
```

- Nagu näha, säilitab kumbki objekt oma olek

Pärimine

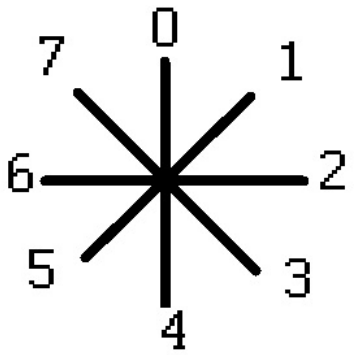
- Pärides võib luua sügavaid hierarhiaid, st
  - Loom
    - Koer
      - Taks
      - Hundikoer
    - Kass
      - Sfinks
    - Inimene
  - Lind
    - Kana
    - Tuvi
- Pärida ja laiendada on võimalik korraga ka mitut klassi, tuleb ettevaatlikult näpuga järge ajada, millised atribuudid ja meetodid kust pärinevad
  - Näiteks footon käitub nii laine kui osakesena
- Mõistlikku hierarhiat on tihti üpris keeruline teha

Ülesanne: robotite aardejaht

Mõned nädalad tagasi sõbralt laenatud metallidetektori abil leitud aarded tõid nii palju tulu, et ostsite portsu lihtsaid roboteid, kes hakkaks teie eest ise aardeid otsima. Paraku on ostetud kratid täiesti abitud ja te peate aarde leidmiseks nendele juhtimisalgoritmi arendama.

Aine kodulehel on link moodulile "**simulator.py**". Teie loodud lahendus kasutab seda moodulit kasutades importi. (**NB! Te ei tohi simulator.py failis midagi muuta (v.a. kui avastate vea)!**)

Teie esimeseks ülesandeks on luua klass ***Robot***, mis pärineb klassist ***Agent*** (*defineeritud moodulis simulator.py*). Klassis ***Agent*** on defineeritud kaks olulist funktsiooni:



1) ***detect(****direction****)*** – sensorite abil ümbruse tajumine (0 = põhi, 2 = ida, 4 = lõuna, 6 = lääs)

2) ***turn(****way****)*** – pööramine vasakule või paremale (-1 = vasakule, 1 = paremale)

... jätkub järgmisel slaidil ...

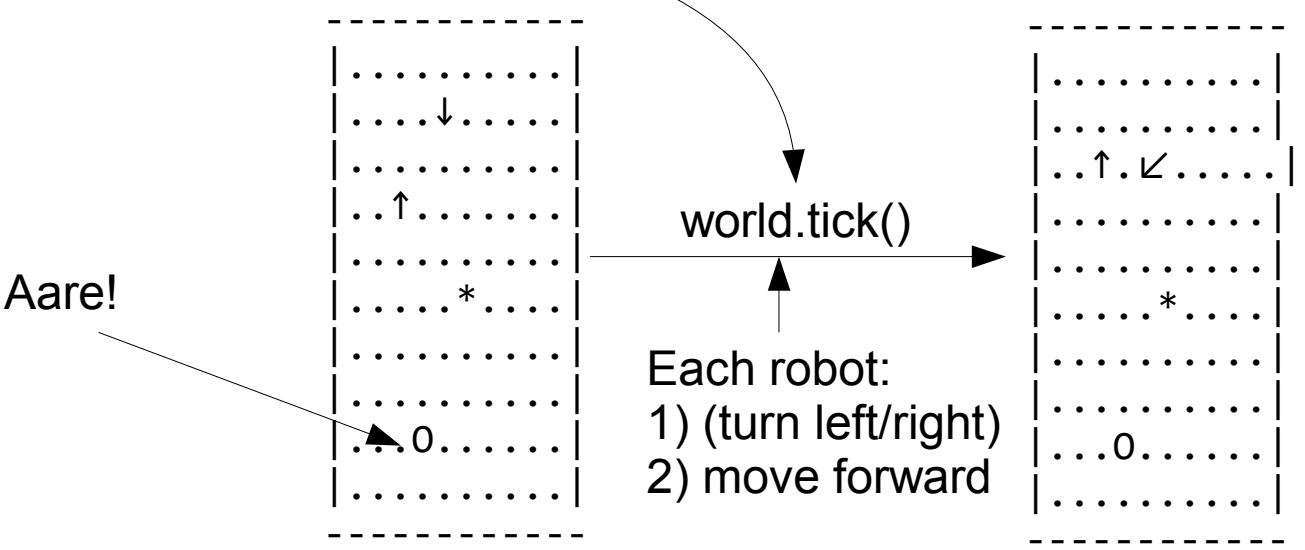
Simulaatori kasutamine

Iga sammu viiteaeg sekundites
(parem jälgida kui 1, aga võib ka 0)

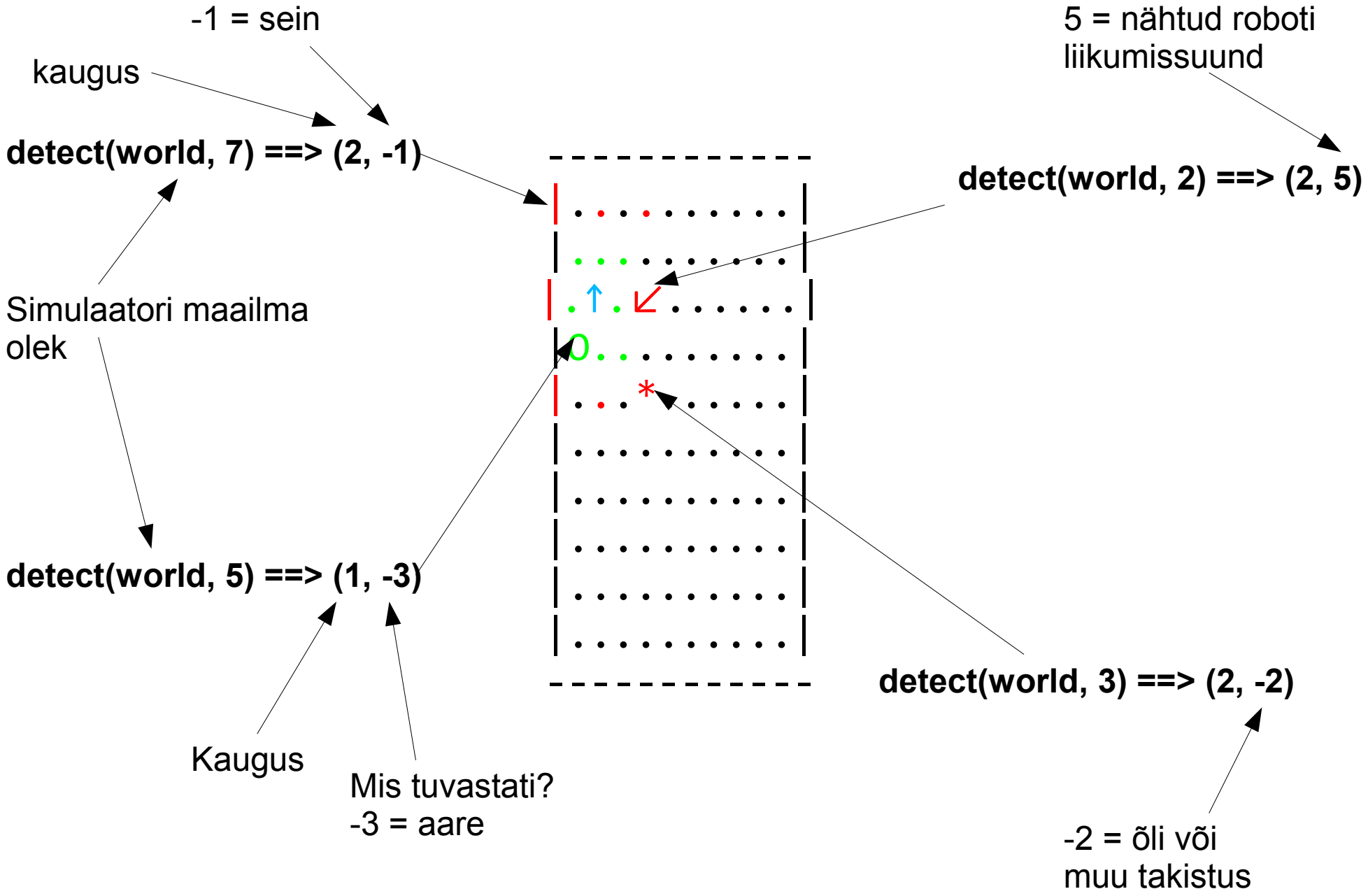
```
import simulator
import random
world = simulator.World(width = 10, height = 10, sleep_time = 1, treasure = None,
obstacles = [(5, 5)], reliability = 0.9)

robots = []
robots.append(Robot(world, 4, 1, 4))
robots.append(Robot(world, 2, 3, 0))
while True:
    for robot in robots:
        robot.turn(world, random.randint(-1, 1))
    world.print_state()
    world.tick()
```

Robotite töökindlus
= tõenäosus, et ei
leki õli põrandale

None =
random



detect() näitab maksimaalselt kauguseni 2 ruutu (märgitud punasega)



Teie põhiülesandeks on kirjutada teie loodud **Robot** klassile funktsioon ***reason()***, mis kasutades funktsiooni ***detect(***direction***)*** annab robotile juhtkäske rakendades ***turn(***way***)*** funktsiooni.

Eesmärk: leida aare kasutades ***detect()*** funktsiooni ja sõita robotiga sellele ruudule.

Piirangud:

- Robot liigub iga ajaühikuga alati ühe ruudu võrra edasi (ei ole võimalik peatuda).
- Robot saab pöörata ühe ajaühiku jooksul ainult ühe korra (turn(way) kasutamisel toimub liikumine kas vasakule->otse või paremale->otse, ei saa pöörata mitu korda (vasakule-vasakule->otse))
- Robotid ei tohi sõita vastu seina, takistuste pihta ega põrgata teineteisega kokku.
- Ei tohi "häkkida" simulaatorit ja kasutada simulaatoris olevaid olekumuutujaid robotite ja maailma kohta. Robot suhtleb simulatsioonikeskkonnaga ainult detect() ja turn() funktsioonide kaudu.

Ülesanne loetakse sooritatuks kui robotid jäävad terveks (ei sõida end katki) "mõistliku aja" vältel
(ideaalselt "igavesti" kui reliability=1.0).
(Näites toodud juhuslik pööramine ja ainuüksi lotoõnn ei piisa.)
Kehtivad tavapärased nõuded docstringile, erinditöötlusele ja ülesehitusele (if __name__; main() jne.)
Kui suudetakse järjepidevalt "mõistliku aja" jooksul aare leida ja korjata, teenitakse **+1 lisapunkt**.

Näpunäiteid:
Robot võib meeles pidada oma eelnevaid käske (puhvris).
Robot võib omaenda maailma kaardi luua, et teada kus ta juba käinud on ja mis seal oli. (NB! See tuleb implementeerida Robot klassis ainult detect() käsu baasil, mitte "häkkida" kuidagi simulaatori olekumuutujatest) .

print_state() prindib välja iga roboti suunad visuaalsete nooltega, mida ei ole tavalises ASCII kodeeringus defineeritud.

```
print(grid[i][j], end="")
  File "C:\Python33\lib\encodings\cp1257.py", line 19, in encode
    return codecs.charmap_encode(input,self.errors,encoding_table)[0]
UnicodeEncodeError: 'charmap' codec can't encode character '\u2199' in position 0:
character maps to <undefined>
```

Selle saab korda Eclipse'is kui muudate **Run -> Run configurations -> Common -> Encoding -> Other -> UTF-8**

Tutorial: http://decoding.wordpress.com/2010/03/18/eclipse-how-to-change-the-console-output-encoding/

Kui kasutate midagi muud, siis uurige kuidas seal käib googlest või kommenteerige simulator.py's ASCII printimine sisse noolte prindi asemel. (#icons = ["^", "/", ">", "\\", "|", "/", "<", "\\"] # ASCII, uncomment if problems)

Seletus kuidas roboti suunda alguses määrata.

```
class Robot(simulator.Agent):

    def __init__(self, world, x, y, direction):
        self.direction = direction
        # .... more stuff ....

    # ..... more Robot stuff ......



myrobot = Robot(world, 4, 4, 1)
print("Robot direction is", myrobot.direction)

>>> Robot direction is 1
```

(ver3 simulator.py's on ka funktsioon Agent.compass(), mis tagastab hetkesuuna)