Tentamen för TDP002 Imperativ programmering - praktisk del
2009-10-22 08-12

Ola Leifler

Instruktioner:

- Uppge namn och personnummer på varje inlämnat papper.
- Uppgifterna löses enskilt.
- Inga hjälpmedel är tillåtna.
- Kommunikation med andra under tentamenstillfället är strikt förbjudet.

Fråga:	1	2	3	4	Totalt
Poäng:	3	4	2	3	12

För betyg 3 gäller 50% rätt, för betyg 4 65% och för betyg 5 80%, avrundat till närmaste hela poäng.
Både korrekta lösningar och välskriven kod krävs för full poäng.
Det totala betyget på tentamen är detsamma som det lägsta betyget av teoridel och problemdel.

1. (3 poäng) Förklara vad som händer när koden nedan körs med hjälp av begreppen *higher-order functions* (högre ordningens funktioner), *LEGB* och *shared references*.

```
l1 = [1,2,4]
def inc(index, n):
    def fn(lst):
        lst[index]+=n
    return fn
modifier=inc(2,3)
modifier(l1)
```

Lösningsförslag: `inc` är en högre ordningens funktion i och med att den *returnerar en funktion* `fn`. `fn` i sin tur modifierar ett värde i den lokala variabeln `lst` (Local) genom att slå upp `index` och `n` i den omgivande funktionen `inc` (Enclosing function locals). I körexemplet binds resultatet av att anropa `inc` med argumenten `2` och `3` (`inc(2,3)`) till den globala variabeln `modifier` (Global). När `modifier` anropas med så slår Python i den globala miljön upp `l1` och binder parametern `lst` till värdet `l1` pekar på. `l1` och `lst` kommer då peka på samma sak och ha dela referensen (share reference) till listan `[1, 2, 4]`. Resultatet blir att efter sista raden i körexemplet har listan som den globala variabeln l1 refererar till ändrats till `[1,2,7]`. LEGB är regeln som används av Python för att avgöra vad variabler refererar till, där man i ordningen *Local-Enclosing Function locals-Global-Builtin* letar efter värden på en variabel.

2. (4 poäng) Gör ett program `select_columns.py` som läser in data från en fil och väljer ut vissa delar att skriva till en annan fil. I indatafilen innehåller varje rad ett antal kolumner som separeras med semikolon (`;`) och `select_columns.py` får två argument som anger den första och sista kolumnen som ska väljas ut. Utfilen ska få som radslut både tecknen CR (\r) och LF (\n) så att filen tolkas rätt i Windows.

```
$ cat data.csv
123;hund;4567;ben,godis
-99;katt;55;sand
44.5;ko;333E10;gräs
$ ./select_columns.py 1 2 data.csv out.csv
$ cat out.csv
hund;4567
katt;55
ko;333E10
```

Lösningsförslag:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
import sys

if __name__ == "__main__":
    # We expect sys.argv[1:] to contain [start_col, end_col, in_file, out_file]
    start_col, end_col, infile_name, outfile_name = sys.argv[1:]
    in_file = file(infile_name,'r')
    out_file = file(outfile_name,'w')
    divider=';'
    def merge(s, result):
        return (result+divider+s if result == '' else s)
    for line in in_file:
        in_columns = line.split(divider)
        out_columns = in_columns[int(start_col):int(end_col)]
        out_file.write(reduce(merge, out_columns, '')+'\r\n')
```

```
        out_file.close()
```

3. (2 poäng) Skapa en funktion `make_filter_map` som tar två funktioner `filter` och `term` som argument och returnerar en funktion `fn` som tar en sekvens `seq` som argument. `fn` returnerar en ny sekvens genom att för varje element `x` i `seq` som godkänns av `filter` applicera `term` på `x`.

```
>>> process = make_filter_map(lambda x: x%2 == 1, lambda x: x*x)
>>> process(range(10))
[1, 9, 25, 49, 81]
```

Lösningsförslag:

```
def make_filter_map(filter_fn, term):
    return lambda seq: map(term, filter(filter_fn, seq))
```

4. (3 poäng) Skapa en abstrakt datatyp `repository` för versionshantering av text, specifikt programkod. Programkod lagras i *filer* (`files`) och ordnas med hjälp av *versioner* (`versions`). Datatypen ska användas för att bygga versionshanteringssystemet `Aluminum`.

   När man skapar ett `repository`-objekt så ska en fil `.project.au` skapas i katalogen man står. `Aluminum` ska låta en användare registrera filer (i katalogen man står i) i ett projekt (`add_file(file)`), märka upp en version för alla registrerade filer (`tag_version()`) och visa vilka filer som ändrats sedan senaste versionen skapades (`show_changes()`). Inställningsfilen `.project.au` ska innehålla all information som behövs för att hantera datatypen `repository`. *Information du tycker saknas i beskrivningen får du själv göra antaganden om för att lösa uppgiften att definiera den abstrakta datatypen.*

Lösningsförslag:

```
#!/usr/bin/env python

import os, sys

# As an example, we use ConfigParser for saving version information

# This is the shortest, simplest solution to implementing the ADT with the methods
    mentioned

from ConfigParser import ConfigParser

repository = ConfigParser()
repository_file = ".project.au"

def init():
    project_file = open(repository_file, "w")
    project_file.write("# Aluminum configuration file\n\n")
    project_file.close()

def add_file(file):
    repository.add_section(file)
    repository.write(open(repository_file, "a"))

# Mark the version with the current modification time for all files
def tag_version():
```

```
        repository.read(repository_file)
        for registered_file in repository.sections():
            repository.set(registered_file, 'tag', os.stat(registered_file).st_mtime)
        init()
        repository.write(open(repository_file,"w"))

    # Show files with changes since the last call to tag_version
    def show_changes():
        repository.read(repository_file)
        for registered_file in repository.sections():
            mtime = os.stat(registered_file).st_mtime
            tagtime = float(repository.get(registered_file, 'tag'))
            if mtime > tagtime:
                print "File %s changed since last version"%(registered_file)


    print sys.argv[1:-2]
```