Instruktioner - Datortentamen TDDD73 Funktionell och imperativ programmering i Python

Hjälpmedel

Följande hjälpmedel är tillåtna:

- Exakt en valfri bok, t.ex. den rekommenderade kursboken. Boken får ha anteckningar, men inga lösa lappar.
- Exakt ett A4-papper med egna anteckningar om precis vad som helst. Det kan vara hand- eller maskinskrivet, på ena eller båda sidorna.
- Pennor, radergummin och liknande för att kunna skissa lösningar på papper.

Icke tillåtna hjälpmedel inkluderar bl.a. alla former av elektronisk utrustning samt böcker och anteckningar utöver det som listats ovan.

Lösa tomma papper för att göra anteckningar tillhandahålls av tentamensvakterna.

Genomförande

Under datortentan kommer du att arbeta i en begränsad och övervakad miljö. Med hjälp av en särskild tentaklient skickar du in dina svar. Du kan även använda tentaklienten för att ställa frågor till examinator. Besök i skrivsalen kommer endast ske vid allvarligare tekniska problem.

Varje uppgift kan skickas in en enda gång och kommer därefter poängsättas. Det finns alltså inga möjligheter till komplettering.

Betygsättning

Datortentan består av totalt sex uppgifter som vardera kan ge maximalt fem poäng. Uppgifterna är utvalda för att ha olika svårighetsgrad och är ordnade så att den lättaste oftast kommer först. Observera att vad som är lätt eller svårt naturligtvis skiljer sig från person till person, så det är alltid en bra idé att snabbt läsa igenom alla uppgifterna för att bättre kunna prioritera arbetet. Betygssättningen sker enligt följande tabell:

- För betyg 3 krävs minst 12 poäng.
- För betyg 4 krävs minst 19 poäng.
- För betyg 5 krävs minst 25 poäng.

Rättningskriterier

Följande allmänna kriterier används vid poängsättning av uppgifter. Lösningar som bryter mot dessa kommer att få poängavdrag.

- Funktioner ska ha exakt samma namn som i uppgiften. Detta underlättar rättningen.
- Namn på parametrar, variabler och hjälpfunktioner som inte specificerats explicit i uppgiften ska vara beskrivande och följa namnstandarden.
- Lösningen ska följa de regler som uppgiften har satt upp. Det kan t.ex. röra sig om att vissa funktioner eller metoder ej får användas, eller att lösningen ska göras enligt en särskild modell.
- Lösningen ska fungera exakt som i körexemplen i uppgiften, om inte texten indikerar något annat.
- Lösningen ska vara generell, d.v.s. den ska inte enbart fungera för de körexempel som finns i uppgiften, utan även för alla andra möjliga indata på samma form. Felkontroller av indata behöver dock normalt inte göras, om inte uppgiften särskilt uttrycker det.

Uppgifter - Datortentamen TDDD73 Funktionell och imperativ programmering i Python

Fredag 10 april 2015 kl 14-19

Uppgift 1

Skriv en funktion `print_calendar` som skriver ut en månadskalender på traditionellt sätt (se exempel nedan). Funktionen tar två parametrar. Den första anger antalet dagar i månaden och den andra anger vilken veckodag som är den första i månaden (0 = måndag, 1 = tisdag, o.s.v.). Följande exempel visar utskriften för en månad med 30 dagar där den första i månaden är en onsdag:

```
>>> print_calendar(30, 2)
  M  T  O  T  F  L  S
        1  2  3  4  5
  6  7  8  9 10 11 12
 13 14 15 16 17 18 19
 20 21 22 23 24 25 26
 27 28 29 30
```

Utskriften av kalenderna ska vara formatterad exakt som i exemplet, d.v.s. varje datum ska ta upp exakt tre tecken där mellanslag används som utfyllnad och talet är högerjusterat inom fältet. Första raden utgörs av korta dagrubriker. Kalendern ska avslutas med en ny rad, så att inte Python-prompten dyker upp omedelbart efter sista dagen.

Uppgift 2

Skriv två olika funktioner som kan samsortera två sorterade listor. Detta innebär att vi utgår från att de två listorna som kommer in till funktionen redan är sorterade i stigande ordning. Målet är att skapa en ny lista med alla elementen i båda originallistorna i sorterad ordning. Exempel:

```
>>> merge_r([1, 3, 6, 8], [2, 4, 5, 9])
[1, 2, 3, 4, 5, 6, 8, 9]
>>> merge_i(['a', 'j', 's'], ['f', 'm'])
['a', 'f', 'j', 'm', 's']
```

Om ett element finns med i båda listorna ska det endast finnas med en gång i resultatlistan.

Funktionen ska finnas i två varianter: `merge_r` som arbetar rekursivt och `merge_i` som arbetar iterativt.

Båda funktionerna ska behandla ett element i taget, vilket innebär att du inte får använda inbyggda funktioner eller metoder som behandlar hela listor. Du får inte heller använda listbyggare (eng. *list comprehensions*). Lösningarna ska vara funktionella i bemärkelsen att de inte får modifiera indata.

Uppgift 3

Deluppgift 3a (2p)

Skriv en högre ordningens funktion `each_pair` som applicerar en funktion av två argument på alla efterföljande par av element från en lista. Exempel:

```
>>> each_pair([3, 7, 9, 15, 23, 27, 33], (lambda x, y: y-x))
[4, 2, 6, 8, 4, 6]
```

Som framgår av exemplet appliceras inte funktionen på alla möjliga par, utan på par av element som står efter varandra. I exemplet ovan först 3 och 7, därefter 7 och 9, 9 och 15, osv. Resultatet samlas ihop i en lista som är ett element kortare än originallistan. Om originallistan innehåller ett element eller färre ska resultatet bli en tom lista. Exemplet ovan räknar alltså ut skillnaderna mellan talen i listan.

Deluppgift 3b (3p)

Skriv en högre ordningens funktion `combine_pairs` som är en lite mer avancerad version av ovanstående funktion. På samma sätt som ovan har vi en lista av element som ska behandlas parvis, men nu har vi två olika funktioner: en funktion `fn_pair` som appliceras på paren och en funktion `fn_combine` som används för att kombinera ihop resultaten av `fn_pair`. Startvärde för funktionen `fn_combine` anges som fjärde argument till `combine_pairs`. Exempel:

```
>>> combine_pairs([1, 2, 3, 4], (lambda x, y:x<y),
      (lambda a, b:a and b), True)
True
>>> combine_pairs([1, 2, 5, 4], (lambda x, y:x<y),
      (lambda a, b:a and b), True)
False
```

I det här exemplet är det en jämförelse som appliceras på varje efterföljande par. Funktionen som kombinerar ihop resultaten är i grund och botten operationen `and`, och som startvärde används `True`. Detta funktionsanrop testar alltså att listan är sorterad. Som synes är den första listan sorterad, men den andra inte.

Uppgift 4

Skriv två funktioner som testar om en rak lista är ett anagram av en annan rak lista, d.v.s. om de innehåller exakt samma element men inte nödvändigtvis i samma ordning. Funktionen ska finnas i två varianter: en som arbetar rekursivt (`is_anagram_r`) och en som arbetar iterativt (`is_anagram_i`).

Båda funktionerna ska behandla ett element i taget, vilket innebär att du inte får använda inbyggda funktioner eller metoder som behandlar hela listor. Du får inte heller använda listbyggare (eng. *list comprehensions*). Lösningarna ska vara funktionella i bemärkelsen att de inte får modifiera indata.

```
>>> is_anagram(['p', 'e', 't', 'e', 'r'], ['r', 'e', 'e', 'p', 't'])
True
>>> is_anagram(['p', 'e', 't', 'e', 'r'], ['r', 'e', 'e', 'p', 'q'])
False
>>> is_anagram([], [])
True
```

Uppgift 5

Vi tänker oss att vi har ett släktträd med barn, barnbarn o.s.v. som vi lagrar som listor i listor. Exempel:

```
svensson = ['Erik', ['Olle', ['Eva', 'Karin', 'Anna'],
                             ['Lars', 'Maria'],
                             ['Pär', 'Sofia']],
                    'Lisa',
                    ['Stina', ['Gunnar', 'Lasse'],
                              'Lennart']]
```

Detta släktträd utgår från Erik som har barnen Olle, Lisa och Stina. Olle har i sin tur barnen Eva, Lars och Pär. Lisa har inga barn, men Stina har barnen Gunnar och Lennart. I den här uppgiften antar vi för enkelhetens skull att alla personer har olika namn.

Vi vill ha en funktion `ancestors` som givet en person och ett släktträd enligt ovan ger släktledet till en person. För Maria är det alltså Erik → Olle → Lars → Maria, vilket ska returneras som en rak lista. Om personen i fråga inte finns med i trädet returneras en tom lista. Exempel:

```
>>> ancestors('Maria', svensson)
['Erik', 'Olle', 'Lars', 'Maria']
>>> ancestors('Erik', svensson)
['Erik']
>>> ancestors('Gunnar', svensson)
['Erik', 'Stina', 'Gunnar']
>>> ancestors('Barbro', svensson)
[]
```

Definiera funktionen `ancestors`.

Uppgift 6

Vi vill ha en abstrakt datatyp som vi kallar **ring**. Ett ringobjekt kan innehålla ett godtyckligt antal element som är cykliskt ordnade, där ett av dem benämns toppelement. Vi vill ha följande primitiva funktioner:

`make_ring : list of elements -> ring`

Skapar och returnerar en ny ring från en lista med element där första elementet i listan är toppelement i ringen.

`is_ring : any object -> truth value`

Kontrollerar om det givna objektet är en ring.

`top : ring -> element`

Returnerar toppelementet i ringen.

`left_rotate : ring -> ring`

Returnerar en ny ring genom att elementen roteras ett steg åt vänster.

`right_rotate : ring -> ring`

Returnerar en ny ring genom att elementen roteras ett steg åt höger.

left_rotate_in : ring ->

Roterar elementen i den aktuella ringen ett steg åt vänster. Inga nya objekt ska skapas eller returneras, utan ringen ska modifieras på plats.

right_rotate_in : ring ->

Roterar elementen i den aktuella ringen ett steg åt höger. Inga nya objekt ska skapas eller returneras, utan ringen ska modifieras på plats.

Här följer några exempel på hur primitiverna ska fungera:

```
>>> ring1 = make_ring([1, 2, 3])
>>> top(ring1)
1
>>> top(left_rotate(ring1))
2
>>> top(right_rotate(ring1))
3
>>> top(left_rotate(left_rotate(left_rotate(ring1))))
1
>>> ring2 = make_ring(['a', 'b', 'c'])
>>> top(ring2)
'a'
>>> left_rotate_in(ring2)
>>> top(ring2)
'b'
>>> right_rotate_in(ring2)
>>> right_rotate_in(ring2)
>>> top(ring2)
'c'
```

Välj en lämplig representation för objekt av typen **ring** och definiera primitiverna (enligt samma principer som i laborationsomgång 6).