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 vara välstrukturerad och väldokumenterad på samma sätt som kursens laborationer.
- 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

Onsdag 16 augusti 2017 kl 14-19

Uppgift 1

Collatz problem är ett olöst problem inom talteorin. Lothar Collatz formulerade problemet under sin tid som student. Problemet utgår från följande räknelek:

1. Utgå från ett positivt heltal *n*.
2. Om *n* är jämnt, dividera det med två. Om det är udda, multiplicera det med tre och addera därefter ett.
3. Upprepa steg 2 tills du når talet ett.

Resultatet från räkneleken kan skrivas som en talföljd. Collatz problem är att avgöra om man, oavsett vilket tal man börjar med, kan nå talet ett. Din uppgift är att skriva en funktion `collatz` som tar ett heltal *n* och returnerar en lista med talföljden som skapas av att följa räkneleken ovan. Exempel:

```
>>> collatz(6)
[6, 3, 10, 5, 16, 8, 4, 2, 1]

>>> collatz(7)
[7, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1]

>>> collatz(13)
[13, 40, 20, 10, 5, 16, 8, 4, 2, 1]
```

Uppgift 2

Skriv en funktion som tar en lista och returnerar en ny lista med elementen i omvänd ordning. Funktionen ska finnas i två varianter: en som arbetar enligt en rekursiv modell (kallad `reverse_r`) och en som arbetar enligt en iterativ modell (kallad `reverse_i`). Du får inte 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. Funktionerna ska fungera exakt likadant, bortsett från att de ska ha olika lösningsmodeller. Här är några exempel:

```
>>> reverse_r([1, 3, 5])
[5, 3, 1]

>>> reverse_i([2, 8, 4, 6])
[6, 4, 8, 2]
```

De båda funktionerna `reverse_r` och `reverse_i` ger vardera 2.5 poäng, och det är möjligt att få poäng på dem oberoende av varandra.

Uppgift 3

Deluppgift 3a (3p)

Skriv en högre ordningens funktion `add_for_each` som tar en rak lista och en funktion. Funktionen `add_for_each` ska applicera den givna funktionen på varje element i listan och summera ihop de erhållna resultaten. I denna uppgift ställs inga särskilda krav på rekursiva eller iterativa lösningar, utan problemet får lösas på valfritt sätt. Dokumentera funktionen på lämpligt sätt. Exempel:

```
>>> add_for_each([1, 2, 3, 4], lambda x: x**2)
30
>>> add_for_each([[1, 2, 3], [1], [1, 2, 3, 4]], lambda x: len(x))
8
```

Deluppgift 3b (2p)

Vi har en lista med listor som innehåller temperaturmätningar under olika dagar. Varje underlista motsvarar en dag. Vi är nu intresserade av att veta genomsnittet av alla maxtemperaturer. Skriv en funktion `average_max` som tar en lista med listor av mätvärden och som med hjälp av `add_for_each` beräknar genomsnittet av maxvärdena. Dokumentera funktionen på lämpligt sätt. Exempel:

```
>>> temp = [[12,13,15,11], [8,9,10], [5,7,6], [8,9,11,10], [3,5,5,2]]

>>> average_max(temp)
9.6
```

Uppgift 4

Skriv en funktion `palindrom` som tar en lista som kan innehålla godtyckligt nästlade listor och returnerar `True` om den är palindrom, dvs . om listan ser likadan ut även om man vänder på den, annars `False`. Tänk på att ett listelement kan vara en godtycklig lista av listor. Funktionen ska vara icke-destruktiv. Det går bra att lösa uppgiften rekursivt eller iterativt. Tänk som vanligt på att dokumentera funktionen på lämpligt sätt.

```
>>> palindrom([1, 1, 1])
True

>>> palindrom([[1], [2], [1]])
True

>>> palindrom([[1], 1])
False

>>> palindrom([[1, 2, 3], 2, [7], 2, [3, 2, 1]])
True

>>> palindrom([[1, 2, 3]])
False
```

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.

Skriv en funktion `ancestors` som givet en person och ett släktträd enligt ovan ger släktledet till personen som en rak lisa. För Maria är det alltså Erik -> Olle -> Lars -> Maria, vilket ska returneras som en rak lista `['Erik', 'Olle', 'Lars', 'Maria']`. Om personen i fråga inte finns med i trädet returneras en tom lista. Dokumentera funktionen på lämpligt sätt. Exempel:

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

>>> ancestors('Erik', svensson)
['Erik']

>>> ancestors('Gunnar', svensson)
['Erik', 'Stina', 'Gunnar']

>>> ancestors('Barbro', svensson)
[]
```

Uppgift 6

Att skapa abstrakta datatyper är en vanlig uppgift för programmerare. Den här uppgiften handlar om att implementera en multimängd (*en. multiset* eller *bag*), dvs en mängd som kan innehålla flera likadana element. Din lösning ska stödja följande operationer och ska implementeras från grunden med listor eller dictionaries samt operationer på dessa. Lösningen ska vara funktionell och dokumentation är viktigt.

- `create_bag()` : returnera en ny tom bag.
- `add_element(b, x)` : lägg till elementet `x` till `b`.
- `remove_element(b, x)` : ta bort ett element `x` från `b`.
- `contains(b, x)` : returnera sant om och endast om `x` finns i `b`.
- `count(b, x)` : returnera antalet gånger som `x` förekommer i `b`.
- `is_sub_bag(b1, b2)` : returnera sant om alla element i `b1` också finns i `b2`.
- `bag_union(b1, b2)` : returnera en ny bag som innehåller alla element från `b1` och `b2`.
- `get_elements(b)` : returnera en sorterad lista med alla element i `b`.

De tre första operationerna ger tillsammans 1 poäng, de två följande tillsammans 1 poäng och de tre sista 1 poäng var. Körexempel:

```
>>> b1 = create_bag()
>>> b1 = add_element(b1, 1)
>>> b1 = add_element(b1, 1)
>>> get_elements(b1)
[1, 1]
>>> contains(b1, 1)
True
>>> contains(b1, 'hej')
False
>>> b2 = create_bag()
>>> b2 = add_element(b2, 1)
>>> is_sub_bag(b2, b1)
True
>>> b2 = add_element(b2, 2)
>>> is_sub_bag(b2, b1)
False
>>> b3 = bag_union(b1, b2)
>>> get_elements(b3)
[1, 1, 1, 2]
>>> b1 = remove_element(b1, 1)
>>> get_elements(b1)
[1]
>>> is_sub_bag(b1, b2)
True
>>> b4 = add_element(b1, 2)
>>> get_elements(b4) != get_elements(b1)
True
```