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 ni att arbeta i särskilda datortentakonton där ni har tillgång till en starkt begränsad miljö. Ni har t.ex. ingen koppling till internet. I praktiken kommer miljön bestå av ett terminalfönster och Emacs.

Starta tentaklienten från menyn som visas när man högerklickar på skrivbordet. Tentaklienten används för att ställa frågor till examinator under tentan, samt för att skicka in lösningar.

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

Onsdag 15 januari kl 14-19

Uppgift 1

Deluppgift 1a (3p)

Vi vill ha en funktion `ascending` som givet ett ord (en sträng) returnerar en strikt växande delsekvens av bokstäver. Det är mycket enklare än det låter, vilket förhoppningsvis framgår av följande exempel:

```
>>> ascending("Kalle")
'Kl'
>>> ascending("abakus")
'abku'
>>> ascending("dekor")
'dekor'
```

Första tecknet i originalsträngen måste alltid vara med i resultatet. Därefter kollar vi ett tecken i taget. Tag exemplet 'abakus' ovan. Första 'a' får vara med. Så även 'b', eftersom det är större än 'a'. Sedan kommer 'a' igen som inte får vara med, eftersom det är mindre än 'b'. Däremot får nästa 'k' vara med, eftersom det är större än 'b' som just nu ligger sist i resultatet. Samma gäller 'u', men inte 's'.

Originalordet kan innehålla både små och stora bokstäver, och jämförelsen ska likställa 'A' med 'a', men aldrig ändra det. Se exemplet 'Kalle' ovan. *OBS! Du kan utgå från att funktionen alltid får in en sträng med minst ett tecken.*

Deluppgift 1b (2p)

Funktionen `ascending` skulle kunna användas som en del i att skicka och ta emot hemliga meddelanden. Det är kanske inte så säkert, men alltid lurar det någon. Därför vill vi ha en funktion `decode` som kan ta en sträng med ord. Varje ord ska behandlas med funktionen `ascending` ovan, och resultaten ska slås ihop till en ny sträng, utan mellanslag. Detta resultat innehåller det hemliga meddelandet. Exempel:

```
>>> decode("apa banan")
'apbn'
>>> decode("Mack ide dada namn atoll toviga idag dakob nissan ubat migg")
'Midnattvidkonsum'
>>> decode("Skolledarens eventuella eder uttalat sirliga. Sablar? nej,
apkall edda.")
'SeverusSnape'
```

I resultatsträngen kan vi nu läsa det tidigare dolda meddelandet.

Uppgift 2

Ett vetenskapligt instrument producerar data i form av olika mätserier. Dessa ska vi behandla i Python som listor av listor med heltal. Följande utgör ett sådant exempel:

```
values = [[3, 2, 3, 3, 4, 3, 2, 3, 4], [], [6, 5, 6, 7, 8], [-2],
          [5, -2, 5, 3, 0, 4, 3], [], [4, 6, 0, 2, -3]]
```

Vi vill nu ha två olika varianter av en funktion som kan plocka bort de tomma (felaktiga) mätserierna. I exemplet ovan finns två tomma mätserier. Funktionerna ska heta `remove_empty_r` samt `remove_empty_i` och de ska använda en rekursiv respektive en iterativ lösningsmodell. Exempel:

```
>>> remove_empty_r(values)
[[3, 2, 3, 3, 4, 3, 2, 3, 4], [6, 5, 6, 7, 8], [-2],
 [5, -2, 5, 3, 0, 4, 3], [4, 6, 0, 2, -3]]
```

Båda funktionerna ska behandla varje element i taget, vilket innebär att du inte får använda inbyggda 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 indatat.

Uppgift 3

Deluppgift 3a (3p)

Vi ska nu bygga vidare på mätserierna från föregående uppgift. Nu vill vi ha en högre ordningens funktion `with_series` som tar en lista med mätserier samt två funktioner som vi kan kalla `fn_check` och `fn_series`. Den första funktionen, `fn_check`, ska givet en mätserie returnera ett sanningsvärde, som talar om huruvida denna mätserie ska behandlas eller ej. Den andra funktionen, `fn_series`, behandlar mätserier, d.v.s. den tar in en mätserie och returnerar en mätserie.

Följande exempel behandlar alla mätserier som innehåller en nolla och behandlingen består i att ta bort alla nollor:

```
>>> with_series(values, (lambda s: 0 in s),
                (lambda s: list(filter((lambda x: x != 0), s))))
[[5, -2, 5, 3, 4, 3], [4, 6, 2, -3]]
```

Även denna lösning ska vara funktionell i bemärkelsen att indata inte får modifieras.

Deluppgift 3b (2p)

Vi vill också ha en funktion `averages`. Förutom en lista med mätvärden tar den även ett tröskelvärde. Endast mätserier där alla mätvärden är större än eller lika med tröskelvärdet ska behandlas. För dessa serier ska medelvärdet beräknas. Helt tomma mätserier ska hoppas över. Med samma exempel på mätserie som ovan får vi följande exempel:

```
>>> averages(values, 2)
[3.0, 6.4]
```

Det är endast två mätserier i exemplet som inte är tomma och där samtliga mätvärden är minst 2.

OBS! Funktionen ska lösa problemet genom att anropa `with_series`.

Uppgift 4

När vi arbetar med listor i listor talar vi ibland om *nivå* eller *djup*. I en rak lista ligger alla element på den översta nivån 1. Om vi byter ut ett av dessa element mot en underlista kommer elementen i den underlistan att befinna sig på nivå 2, och så vidare. En liststrukturs djup är den högsta förekommande nivån.

Deluppgift 4a (2,5p)

Skriv en funktion `max_level` som går igenom en lista med listor och räknar ut nivån för det djupaste elementet. Om listan är tom är svaret 0. Om det rör sig om en rak lista är svaret 1. Om listan har underlistor blir svaret 2 eller högre, beroende på hur många listor i listor det finns. Exempel:

```
>>> max_level([])
0
>>> max_level([1, 2, 3])
1
>>> max_level([1, [2, [3], 4], 5])
3
```

Deluppgift 4b (2,5p)

Vi vill också ha en funktion `levels` som kan söka igenom en lista av listor efter ett visst element. Funktionen ska returnera en lista med tal som motsvarar nivåerna där detta element förekommer. Listan ska ha lika många element som antalet förekomster av det sökta elementet i listan. Toppnivån är 1 och djupare nivåer har högre nummer. Exempel:

```
>>> levels('x', ['a', ['b'], 'x'])
[1]
>>> levels('x', ['a', ['b'], 'c'])
[]
>>> levels('x', [['a', ['x']], 'x', ['b', 'c']])
[3, 1]
>>> levels('x', [['a', 'x'], 'x', ['b', 'x']])
[2, 1, 2]
```

Uppgift 5

Skriv en funktion `postfix` som tar ett aritmetiskt uttryck med *infix*-notation och genererar samma uttryck i *postfix*-notation, d.v.s. operatorerna ska stå efter argumenten. De aritmetiska uttrycken kan antingen vara atomära (tal eller variabler som strängar) eller sammansatta *binära* uttryck (d.v.s. de innehåller alltid exakt två argument) med något av de fyra enkla räknesätten. Infix-uttrycket representeras som listor i listor, men postfix-uttrycket skall representeras som en rak lista. Följande körexempel belyser hur funktionen ska fungera:

```
>>> postfix(3)
3
>>> postfix([2, '+', 'a'])
[2, 'a', '+']
>>> postfix([1, '+', [2, '*', [[3, '+', 4], '-', 5]]])
[1, 2, 3, 4, '+', 5, '-', '*', '+']
```

Uppgift 6

Vi har byggt en mönstermatchare som till viss del liknar den från laborationsomgång 7, men med skillnaden att vi försökt göra den mer abstrakt. Dessutom har vi utökat den med möjlighet att binda upp resultatet av enskilda matchningar i en tabell.

Ett mönster är här ett tal, en sträng eller en lista. Det kan dessutom innehålla mönstersymboler. För närvarande finns endast en sådan mönstersymbol och den uttrycker vi så här:

```
['?', variabelnamn]
```

Ett mönster kan matchas mot en lista som innehåller tal, strängar och underlistor. En matchning lyckas om elementen exakt stämmer överens, förutom för mönstersymbolen `['?', variabelnamn]` som matchar ett element vilket som helst. Exempel:

- Mönstret `['a', 'b', 'c']` matchar uttrycket `['a', 'b', 'c']` men inte `['a', ['b', 'c']]` eller `['a', 'b']`.
- Mönstret `['a', ['b', ['c']]]` matchar uttrycket `['a', ['b', ['c']]]` men inga andra.
- Mönstret `['a', ['?', 'x'], 'c']` matchar uttrycket `['a', 'b', 'c']` och producerar en variabeltabell där variabeln `'x'` är bunden till värdet `'b'`.

Vid matchningen är man intresserad av att veta mot vilket deluttryck mönstersymbolen `['?', variabelnamn]` matchade emot. Därför kommer algoritmen att binda upp motsvarande variabelnamn till det matchade värdet i en dictionary. Om mönstret `[['?', 'x'], 'b', ['?', 'y']]` matchas mot `['a', 'b', 'c']` får vi alltså en dictionary där `'x'` binds till `'a'` och `'y'` till `'c'`.

Vi har redan skrivit en huvudfunktion `matcher` som gör matchningen. Returvärdet av hela funktionen är en dictionary med bindningar (som kan vara tom), eller strängen `'fail'` om matchningen inte lyckades.

Lösningen är gjort någorlunda abstrakt och vi kan här skönja följande abstrakta datatyper:

- **element** - De grundläggande byggstenar som kan förekomma i vår tillämpning. I denna uppgift utgörs de av heltal och strängar i Python.
- **expression** - **Element** eller sekvenser av **element**, vilka implementeras som listor.
- **arbitrary** - En mönstersymbol som matchar vad som helst, och som också innehåller ett variabelnamn. Implementeras som en lista med ett frågetecken och ett variabelnamn.
- **pattern** - Ett objekt av typen **arbitrary**, ett **element** eller en sekvens bestående av dessa två, vilket implementeras som listor.

Du hittar koden till mönstermatcharen i filen `match_s.py`. Uppgifterna finns på nästa sida.

Deluppgift 6a (2p)

Inför ytterligare en mönstersymbol `['=', variabelnamn]` som endast matchar heltal. Mönstret `['a', ['=', 'i'], 'c']` matchar `['a', 10, 'c]` och ger en dictionary med `'i'` bundet till 10. Däremot matchas inte `['a', 'b', 'c']`.

Inför nya primitiva funktioner och förändra de funktioner som behöver utökas. Lösningen ska vara abstrakt, på samma nivå som resten av koden.

Deluppgift 6b (3p)

Om en mönstersymbol med samma variabelnamn förekommer mer än en gång vill man att mönstersymbolen ska matcha samma uttryck vid alla tillfällen. Detta sker inte med den lösning som finns. Förändra de funktioner som behöver utökas för att detta ska funka. Mönstret `[['?', 'x'], ['?', 'x']]` matchar `['a', 'a']` men inte `['a', 'b']`.