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

Fredag 25 april 2014 kl 14-19

Uppgift 1

Deluppgift 1a (2,5p)

Python har en inbyggd strängmetod `replace` som returnerar en ny sträng där alla förekomster av ett givet tecken har bytts ut mot ett annat. Exempel:

```
>>> 'bananer'.replace('a', '*')
'b*n*ner'
```

Metoden `replace` har dock en begränsning. Om man har flera olika tecken som ska bytas ut får man anropa `replace` flera gånger. Vi vill istället att du skapar en *funktion* `replace` som kan byta ut flera olika tecken i ett svep. Alla de tecken som ska bytas ut anges i en sträng. Exempel:

```
>>> replace('bananer', 'a', '*')
'b*n*ner'
>>> replace('bananer', 'aeiouy', '*')
'b*n*n*r'
>>> replace('One, two, three!', ' .,!?', '')
'Onetwothree'
```

I det första exemplet byts alla *a* ut mot en asterisk. I det andra exemplet byts alla engelska vokaler mot en asterisk. I det tredje exemplet byts alla mellanslag och skiljetecken mot en tom sträng, dvs de försvinner helt.

Deluppgift 1b (2,5p)

Python har en inbyggd strängmetod `title` som returnerar en ny sträng där alla bokstäver i början av ord är versaler och alla andra bokstäver är gemener. Prova gärna metoden så att du får en känsla för hur den fungerar. Exempel:

```
>>> "my fair lady".title()
'My Fair Lady'
```

Vi vill att du ska återskapa samma beteende, men som en *funktion* istället. Det ska då funka så här:

```
>>> title('my fair lady')
'My Fair Lady'
>>> title('ONE, two, three!')
'One, Two, Three!'
```

Funktionen får givetvis inte använda den inbyggda metoden `title`, utan ska lösa problemet på annat sätt. Alla tecken som inte är bokstäver ska behållas som de är. Observera att funktionen ska returnera en sträng, inte göra några utskrifter!

Uppgift 2

I Python finns en inbyggd datatyp *set* som motsvarar mängder, i samma bemärkelse som mängder i diskret matematik. Mängder skrivs som listor, men använder tecknen { och } som avgränsare. Mängder har flera inbyggda metoder, t.ex. kan man beräkna snittet mellan två mängder så här:

```
>>> {1, 2, 3}.intersection({9, 1, 2})
{1, 2}
```

Vi vill nu ha en *funktion* som kan beräkna snittet mellan två listor istället. Det ska fungera så här:

```
>>> intersect_r([1, 2, 3], [9, 1, 2])
[1, 2]
```

Du får inte lösa problemet genom att omvandla listorna till mängder, alltså datatypen *set*, utan måste gå igenom åtminstone en av listorna element för element. Funktionen ska finnas i två varianter: en med rekursiv lösningsmodell kallad `intersect_r` och en med iterativ lösningsmodell kallad `intersect_i`. Ordningen mellan elementen i resultatlistan är inte viktig.

Båda funktionerna ska behandla ett 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 (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 en funktion `interleave` som kombinerar två raka listor genom att ta element växelvis från de olika listorna. Om den ena listan tar slut fyller vi på med elementen från den andra listan. Exempel:

```
>>> interleave_r(['a', 'b', 'c'], [1, 2, 3])
['a', 1, 'b', 2, 'c', 3]
>>> interleave_r(['a', 'b', 'c'], [1, 2, 3, 4])
['a', 1, 'b', 2, 'c', 3, 4]
>>> interleave_r([], [1, 2, 3, 4])
[1, 2, 3, 4]
```

Funktionen `interleave` ska finnas i två versioner: en som arbetar rekursivt kallad `interleave_r` och en som arbetar iterativt kallad `interleave_i`. Problemet ska lösas genom att gå igenom listorna element för element.

Uppgift 5

Det finns många olika sätt att sortera listor. På föreläsning tittade vi på *insertion sort* och *selection sort*, och under laborationerna övade ni på *quick sort*. En ytterligare variant, som inte är särskilt effektiv, är *bubble sort*.

Bubble sort kallas så eftersom sorterade element bubblar upp till ytan allt eftersom processen fortskrider. (Det hjälper alltså om man försöker se listan stående snarare än liggande.) Första varvet i *bubble sort* går till så att man jämför elementen parvis efter varandra. Om de ligger i fel ordning byter man plats på dem, annars lämnas de kvar. Om man gör detta nedifrån och upp (från index 0 upp till det högsta) så kommer det största elementet i listan garanterat att finnas längst upp (förutsatt att det är den ordning man sorterar efter, men det utgår vi från). Nästa varv gör man samma sak, men man behöver bara fortsätta upp till det näst sista elementet. Tredje varvet upprepar man samma sak igen, men bara upp till det näst näst sista element o.s.v.

5	21
4	14
3	5
2	29
1	32
0	17

Ta ovanstående lista som exempel. Första varvet jämförs plats 0 och 1, plats 1 och 2, plats 2 och 3, plats 3 och 4 och slutligen plats 4 och 5. Nästa varv behöver man inte ta med plats 5, eftersom det största talet (32) redan bubblat upp dit.

Funktionen ska i praktiken fungera så här:

```
>>> bubble_sort([17, 32, 29, 5, 14, 21])
[5, 14, 17, 21, 29, 32]
```

Uppgift 6

Påskharen står inför den stora utmaningen att dela ut påskägg med godis till alla små barn. Var och en av dem har skrivit en önskelista som innehåller deras favoritgodis. Påskharen vill ge varje barn så mycket godis så möjligt, men det totala priset för ett påskägg får inte överstiga ett visst tak. Påskharen har också regeln att varje sak som finns med i barnens önskelista bara får finnas med en gång.

För att lösa detta problem har påskharens fru konstruerat ett litet Python-program som kallas *Egginator™*. Programmet kan, givet en önskelista och ett maxpris, producera alla giltiga förslag på godislistor som inte överstiger maxpriset. Egginatorn innehåller ett antal olika datatyper. Primitiva datatyper är följande:

name (sträng)	namn på en sorts godis
price (heltal)	godisets pris i hela kronor
weight (heltal)	godisets vikt i gram

Följande sammansatta datatyper finns:

candy = <*name*, *price*, *weight*>

wishlist = {{*candy*}}*

candylist = <*price*, *wishlist*>

Datatyperna *candy* och *candylist* är tupler och datatypen *wishlist* är en sekvens som innehåller datatyen *candy*. Vi bearbetar också listor av objekt av datatypen *wishlist* och listor av objekt av datatypen *candylist*, men för dessa använder vi vanliga Python-listor.

I den bifogade koden (se filen `egginator_s.py`) finns större delen av Egginator™. Studera hur de olika datatyperna är implementerade. Huvudfunktionen `egginator` genererar först alla möjliga godislistor utifrån den givna önskelistan. Därefter filtreras alla godislistor som är för dyra bort. Sist filtreras alla godislistor som är en delmängd av någon annan godislista bort. Som exempel kan vi använda en lista som skapas av följande funktion:

```
def make_testlist():
  w = make_wishlist()
  w = extend_wishlist(w, make_candy('Aladdin', 79, 300))
  w = extend_wishlist(w, make_candy('Lyxchoklad', 139, 350))
  w = extend_wishlist(w, make_candy('Storpaket', 189, 500))
  return w
```

Vi bestämmer oss för ett tak på 220 kr och testar Egginator™ så här:

```
>>> print_candylists(egginator(make_testlist(), 220))
----------------------------------------
Lyxchoklad 139
Aladdin 79
*** TOTAL 218
----------------------------------------
Aladdin 79
*** TOTAL 79
```

```
---------------------------------------
Lyxchoklad 139
*** TOTAL 139
---------------------------------------
Storpaket 189
*** TOTAL 189
---------------------------------------
*** TOTAL 0
```

Större delen av Egginator™ återfinns i filen `egginator_s.py`, men några saker saknas. Du hittar dem lätt genom att söka på ERROR i filen. Dessa ska du implementera. Gör detta genom att lägga lösningarna i en ny fil och importera `egginator_s.py`.

Deluppgift 6a (2p)

Bestäm en lämplig representation för datatypen *wishlist* och definiera de primitiva funktioner vars funktionshuvuden finns i den bifogade programkoden.

Deluppgift 6b (3p)

Definiera funktionen *possible_wishlists* som från en given önskelista genererar alla möjliga önskelistor av godis.