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 <u>en enda gång</u> 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

Tisdag 13 januari 2015 kl 14-19

Uppgift 1

Deluppgift 1a (2p)

Skriv en funktion `split` som delar upp en sträng i delsträngar av en given storlek. Returvärdet ska vara en lista av delsträngar, där den sista strängen får vara kortare än det givna värdet om det inte finns tillräckligt med innehåll. Funktionen ska fungera så här:

```
>>> split('abcdefghijklm', 5)
['abcde', 'fghij', 'klm']
>>> split('internet of things', 3)
['int', 'ern', 'et ', 'of ', 'thi', 'ngs']
>>> split('short', 100)
['short']
>>> split('', 14)
[]
```

Deluppgift 1b (3p)

Skriv ytterligare en funktion `column` som fungerar i princip som ovan, men är lite mer komplicerad. Funktionen `column` ska dela upp en sträng i delsträngar av maximalt en angiven längd, men försöka bryta raderna vid mellanslag eller andra separerande tecken. Funktionen ska fungera så här:

```
>>> story = 'Mary had a little lamb. His fleece was white as snow'
>>> column(story, 10)
['Mary had ', 'a little ', 'lamb. His ', 'fleece ', 'was white ',
 'as snow']
>>> column(story, 15)
['Mary had a ', 'little lamb. ', 'His fleece was ', 'white as snow']
>>> column(story, 20)
['Mary had a little ', 'lamb. His fleece ', 'was white as snow']
>>> column('1234567890abcdefghij', 5)
['12345', '67890', 'abcde', 'fghij']
```

De tecken som ska användas som separatorer är mellanslag samt våra sex vanligaste skiljetecken som återfinns i följande sträng: `' .,:;!?'`. Om det inte finns några separatorer bryts strängen ändå efter det angivna antalet tecken. De resulterande raderna ska vara så långa som möjligt, men maximalt så långa som den givna längden. Som framgår av exemplet kan de dock vara olika långa. Inga tecken försvinner, utan ursprungssträngen kan återskapas genom att slå ihop alla delsträngar igen.

Uppgift 2

Skriv två varianter av en funktion som kan lokalisera ett givet element i en rak lista och returnera dess index. Endast den första förekomsten är intressant. Om elementet inte finns i listan returneras värdet -1. Detta är ungefär samma beteende som den inbyggda metoden `index`, så denna får givetvis inte användas.

Funktionen ska finnas i två varianter: en som arbetar enligt en rekursiv modell (kallad `find_r`) och en som arbetar enligt en iterativ modell (kallad `find_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.

Funktionerna ska fungera exakt likadant, bortsett från att de ska ha olika lösningsmodeller. Här är några exempel:

```
>>> find_r(['a', 'b', 'a', 'c', 'd'], 'a')
0
>>> find_i(['a', 'b', 'a', 'c', 'd'], 'c')
3
>>> find_i(['a', 'b', 'a', 'c', 'd'], 'q')
-1
```

Uppgift 3

Deluppgift 3a (3p)

Skriv en högre ordningens funktion `insert_after` som tar en rak lista, en predikatsfunktion och ett element. Funktionen ska returnera en ny lista som är en kopia av originalet, men där det givna elementet har skjutits in efter alla element i originallistan som uppfyller predikatsfunktionen. Om man t.ex. vill skjuta in `'x'` efter varje etta kan man anropa funktionen så här:

```
>> insert_after([0, 1, 1, 2, 0, 2, 1], (lambda x: x == 1), 'x')
[0, 1, 'x', 1, 'x', 2, 0, 2, 1, 'x']
```

Deluppgift 3b (2p)

Använd därefter funktionen `insert_after` för att skriva ytterligare en funktion `mark_interval` som i en lista skjuter in ett givet element efter varje tal i ett givet intervall, inklusive ändpunkterna. Om man t.ex. vill markera alla tvåsiffriga tal med `'x'` kan man anropa funktionen så här:

```
>> mark_interval([7, 'ignore', 14, 9, 435, 'ignore'], 10, 99, 'x')
[7, 'ignore', 14, 'x', 9, 435, 'ignore']
```

Uppgift 4

En vanlig sysselsättning bland mindre uppfostrade ungdomar är att förstöra informationsskyltar. Särskilt roligt kan det bli om man stryker några utvalda bokstäver på en skylt så att de resterande bildar något betyligt mer kreativt budskap än originalet. Till exemel kan man genom att stryka några bokstäver på en skylt som säger "Endast nödutgång" förändra texten till "Endast nöt".

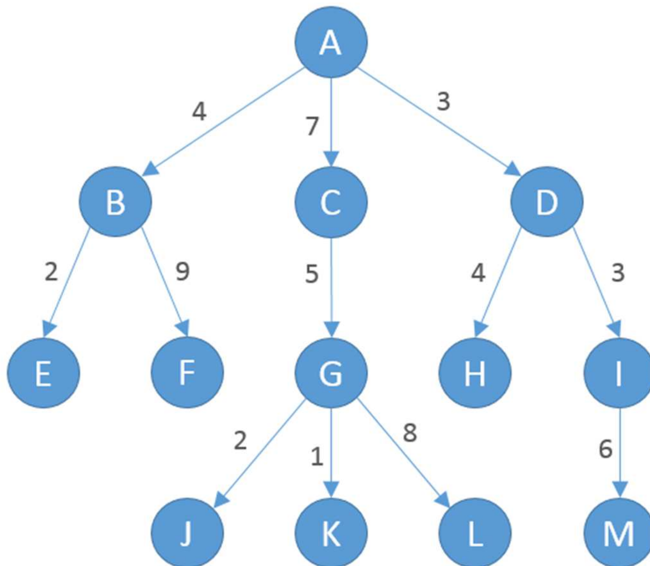
För att i någon mån hjälpa till med denna kreativa process vill vi ha en funktion som kan avgöra om man kan bilda en viss kortare sträng ur en längre sträng genom att stryka några tecken. Vi vill ha den här funktionen i två olika varianter: en som arbetar enligt en rekursiv modell (kallad `contain_r`) och en som arbetar enligt en iterativ modell (kallad `contain_i`). Funktionerna ska fungera så här:

```
>>> contain_r('Endast nöt', 'Endast nödutgång')
True
>>> contain_i('grov', 'grammofonskiva')
True
>>> contain_i('grov', 'grammofonskixa')
False
```

För säkerhets skull vill vi påpeka att detta inte ska ses som en uppmuntran till allmän skadegörelse.

Uppgift 5

Skriv en funktion `distance` som kan beräkna avståndet mellan två noder i ett träd. Varje båge i trädet har ett tal knutet till sig som anger avståndet. Ett exempel på hur ett träd för denna uppgift kan se ut är följande:



Detta träd representerar vi i Python på följande sätt:

```
t = {'a': (('b', 4), ('c', 7), ('d', 3)),
     'b': (('e', 2), ('f', 9)),
     'c': (('g', 5), ),
     'd': (('h', 4), ('i', 3)),
     'e': (),
     ... }
```

Hela trädet i denna representation återfinns i filen `graph.py`. Observera det extra kommatecknet som måste till efter tupler med endast ett element.

Funktionen `distance` ska, förutom trädet, ta namnen på två noder i trädet. Funktionen ska söka reda på vägen från start till mål och beräkna det totala avståndet genom att summera talen på bågarna i trädet. Om det inte finns en väg mellan noderna ska talet -1 returneras. Funktionen ska alltså fungera så här:

```
>> distance(t, 'a', 'b')
4
>> distance(t, 'b', 'a')
-1
>> distance(t, 'c', 'j')
7
>> distance(t, 'a', 'm')
12
```

Uppgift 6

Vi vill kunna hantera en grammatik för att kunna hantera meningar i naturligt språk samt lexikon där ord och deras egenskaper är lagrade. En grammatik består av grammatiska regler av typen

vänsterkategori ::= *en eller flera högerkategorier*

En kategori är ett grammatiskt begrepp av typen mening, substantivfras, verbfras, artikel etc. Som exempel kan vi beskriva en enkel grammatik så här:

mening ::= *substantivfras verbfras*
verbfras ::= *verb substantivfras*
substantivfras ::= *namn*
substantivfras ::= *artikel substantiv*

Här beskriver vi en grammatik där en *mening* skall bestå av en *substantivfras* och en *verbfras*. En *verbfras* skall bestå av ett *verb* och en *substantivfras*. Vi kan ha flera regler för en vänsterkategori. En *substantivfras* kan beskrivas på två sätt. En kategori som inte har en regel associerad med sig beskriver själva orden. Vi kallar denna en grundkategori. Dessa ord lagrar vi i ett lexikon tillsammans med en eller flera grundkategorier. Som exempel kan ett lexikon innehålla:

Karl – namn
Stina – namn
ser – verb
äter – verb
en – artikel, substantiv (både som artikeln 'en' och busken 'en')
gris – substantiv
kung – substantiv

Med hjälp av en grammatik och ett lexikon kan vi avgöra om en sekvens av ord uppfyller en kategori. I ovanstående exempel är vi intresserade av om en sekvens av ord bildar en *mening*. Exempel på korrekta meningar, enligt vår exempelgrammatik, är:

Karl ser en gris
en en äter Stina

I filen `grammatik_s.py` finns programkod för att kunna arbeta med lexikon, grammatik och meningar. I koden finns följande abstrakta datatyper:

word = en sträng	ett ord i ett lexikon eller en mening
sentence = [*word*, ...]	en sekvens av ord
category = en sträng	beskriver ett ord eller en fras
category sequence = [*category*, ...]	en sekvens av kategorier
lexicon entry = <*word*, *category sequence*>	en rad i ett lexikon
lexicon = [*lexicon entry*, ...]	ett lexikon, dvs en sekvens av rader
rule = <*category*, *category sequence*>	en grammatisk regel
grammar = [*rule*, ...]	en hel grammatik, dvs en sekvens av regler

För full poäng på följande deluppgifter ska lösningarna fullt ut använda sig av redan existerande primitiva funktioner, eller implementera nya. Abstraktionen ska inte brytas och lösningarna ska ansluta sig till de modeller som redan finns i den existerande koden.

Deluppgift 6a (2p)

Definiera funktionerna `find_rules` och `extract_grammar`. Funktionen `find_rules` ska givet en kategori och en grammatik finna alla regler med denna kategori som vänsterled. Returvärdet är en grammatik. Huvuddelen av arbetet ska skötas av funktionen `extract_grammar`. Givet att variabeln `g` innehåller exempelgrammatiken ovan ska funktionen fungera så här:

```
>>> find_rules('substantivfras', g)
[['substantivfras', ['artikel', 'substantiv']],
 ['substantivfras', ['namn']]]
```

Deluppgift 6b (3p)

Definiera funktionen `generate_sentence` som givet en startkategori, ett lexikon och en grammatik slumpmässigt genererar en korrekt fras. Med samma exempel som tidigare ska vi kunna generera slumpmässiga korrekta meningar så här:

```
>>> generate_sentence('mening', l, g)
['Karl', 'äter', 'en', 'gris']
>>> generate_sentence('mening', l, g)
['en', 'kung', 'ser', 'Stina']
```