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

Tisdag 13 januari 2015 kl 8-13

Uppgift 1

Automatisk stavningskontroll är en favoritfunktion hos många, särskilt sådan som rättar felen själv. Vi ska nu göra några inledande försök att skriva enklare stavningskontroll i Python med några olika funktioner.

Observera att du kan skicka in svar på uppgiften endast en gång. De tre funktionerna nedan ger 2+2+1 poäng, i tur och ordning, och det är möjligt att få poäng på dem oberoende av varandra.

Först vill vi ha en funktion `count_similar`. Den ska ta två strängar och räkna hur många tecken i den första strängen som är samma som tecknet på motsvarande position i den andra. Vi utgår från att den andra stängen alltid är minst lika lång som den första.

```
>>> count_similar('tomat', 'tomap')
4
>>> count_similar('tomat', 'tomatjuice')
5
```

Sedan vill vi ha en funktion `check_alike` som tar in två strängar där den första är ett tecken kortare än den andra. Funktionen ska undersöka om den första strängen är samma som den andra, men med ett tecken borttaget. Vi utgår från att den andra strängen alltid är ett tecken längre, men tecknen måste komma i samma ordning i båda strängarna för att vi ska säga att de är likartade.

```
>>> check_alike('tomat', 'tommat')
True
>>> check_alike('tomat', 'tommap')
False
>>> check_alike('banan', 'aabbnn')
False
```

Till sist vill vi ha en funktion `close_enough` som tar in två strängar. Med hjälp av de två funktionerna ovan ska den svara på frågan om de två strängarna är tillräckligt lika. Detta är de om de antingen är lika långa och maximalt en bokstav skiljer dem åt, eller om den ena strängen är likadan som den andra, men med en extra bokstav inskjuten någonstans. Funktionen ska fungera så här:

```
>>> close_enough('skumpa', 'skummpa')
True
>>> close_enough('världen', 'värden')
True
>>> close_enough('owned', 'pwned')
True
>>> close_enough('abcde', 'abcxx')
False
```

Uppgift 2

Skriv två varianter av en funktion som kan gå igenom en rak lista och returnera en ny lista där alla upprepningar av samma element i en följd är kompakterade. Om det förekommer `[7, 7, 7]` i originallistan ska resultatlistan enbart innehålla `[7]`. Listan förutsätts inte vara sorterad och om elementet `7` uppträder senare i listan får det vara med igen.

Funktionen ska finnas i två varianter: en som arbetar enligt en rekursiv modell (kallad `uniq_r`) och en som arbetar enligt en iterativ modell (kallad `uniq_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:

```
>>> uniq_r([1, 1, 2, 2, 3, 3, 4, 5])
[1, 2, 3, 4, 5]
>>> uniq_i(['a', 'a', 'b', 'a', 'c', 'c'])
['a', 'b', 'a', 'c']
```

Uppgift 3

Deluppgift 3a (3p)

Skriv en högre ordningens funktion `gen_find` som givet en lista och en predikatsfunktion kan tala om huruvida något element i listan uppfyller predikatet eller ej. Funktionen ska dessutom ta en tredje parameter som avgör huruvida `gen_find` ska bearbeta underlistor eller inte. Funktionen ska returnera ett sanningsvärde.

Om man t.ex. vill söka efter `'b'` i en rak lista gör man så här:

```
>> gen_find([5, 'b', [7, 19], 'w'], (lambda x: x == 'b'), False)
True
```

Om man vill söka efter 7 i samma lista och behandla den som en rak lista gör man så här:

```
>> gen_find([5, 'b', [7, 19], 'w'], (lambda x: x == 7), False)
False
```

Om man även vill söka i underlistor anger man `True` som tredje argument:

```
>> gen_find([5, 'b', [7, 19], 'w'], (lambda x: x == 7), True)
True
```

Deluppgift 3b (2p)

Använd därefter funktionen `gen_find` för att skriva en ny funktion `where` som i grova drag testar var ett element finns i en lista. Funktionen `where` ska returnera en av tre möjliga strängar: `'no'` om elementet inte alls finns i listan, `'top'` om det finns på översta nivån och `'deep'` om det finns på någon djupare nivå. Som mest får två anrop till `gen_find` göras. Funktionen `where` ska fungera så här:

```
>> my_list = [1, 2, 3, [4, 5, 6, [7, 8], 9], 10]
>> where(my_list, 10)
'top'
>> where(my_list, 7)
'deep'
>> where(my_list, 42)
'no'
```

Uppgift 4

En av de svåraste problemen när man startar ett företag är att komma på ett vettigt namn. Om man är i en bransch där man kommer skriva firmanamnet på en skåpbil vill man dessutom gärna undvika onödig komik som uppstår när man öppnar skjutdörren och enbart delar av namnet syns. Skriver man t.ex. "Steffes mekaniska" på en skjutdörr finns det en risk att det står "Steka" eller "Sniska" när man öppnar dörren och gömmer en bit i mitten. Detta kanske inte är så farligt, men på internet kan man hitta andra mer vågade exempel som inte lämpar sig för tentor.

För att hjälpa folk som väljer firmanamn vill vi därför ha en funktion `combos` som kan räkna upp alla möjliga sätt att plocka bort en bit av mitten av en sträng. Funktionen ska funka så här:

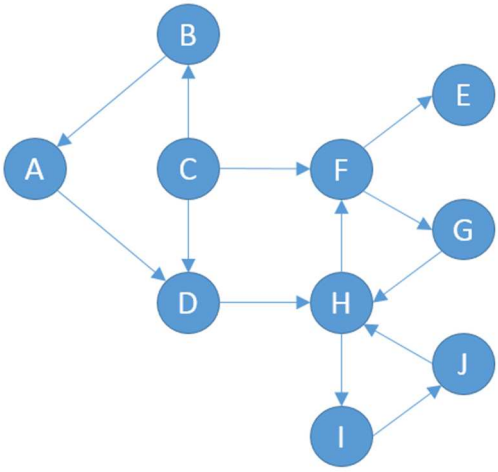
```
>>> combos('peter')
['pr', 'per', 'pter', 'per', 'peer', 'petr']
>>> combos('abcedefg')
['ag', 'afg', 'aefg', 'adefg', 'aedefg', 'acedefg', 'abg', 'abfg',
 'abefg', 'abdefg', 'abedefg', 'abcg', 'abcfg', 'abcefg', 'abcdefg',
 'abceg', 'abcefg', 'abceefg', 'abcedg', 'abcedfg', 'abcedeg']
```

I det första exemplet ser vi att `'per'` förekommer två gånger, eftersom man antingen kan ta bort `'te'` eller `'et'` från mitten av strängen. Det är helt i sin ordning. Vi förväntar oss inte heller att delsträngarna ska komma i någon speciell ordning, så länge alla möjliga alternativ finns med. Ett krav är dock att minst ett tecken från början och ett från slutet ska finnas med i varje alternativ.

(Den som är diskretmatematiskt lagd kan säkert härleda en formel för antalet möjliga delsträngar baserat på orginalsträngens längd. Det får man dock inga extrapoäng för.)

Uppgift 5

Skriv en funktion `locations` som givet en graf kan tala om till vilka noder man kan komma från en given startpunkt med ett givet antal steg. Nedanstående bild visar ett exempel på hur en graf kan se ut i den här uppgiften.



Denna graf representerar vi som en dictionary i Python där varje nod utgör en nyckel och värdena är tupler som talar om till vilken eller vilka andra noder man kan komma från denna nod.

```
g = {'a':('d'), 'b':('a'), 'c':('b', 'd', 'f'), 'd':('h'), 'e':(),
     'f':('e', 'g'), 'g':('h'), 'h':('f', 'i'), 'i':('j'), 'j':('h')}
```

Denna kod finns även tillgänglig i filen `graph.py`.

Om man t.ex. startar i noden `'a'` och går två steg i pilarnas riktning kan man bara komma till `'h'`. Vår funktion `locations` ska alltså fungera så här:

```
>>> locations(g, 'a', 2)
['h']
```

Några ytterligare exempel:

```
>>> locations(g, 'q', 5)
[]
>>> locations(g, 'c', 0)
['c']
>>> locations(g, 'c', 1)
['b', 'd', 'f']
>>> locations(g, 'c', 2)
['a', 'h', 'e', 'g']
```

Ordningen mellan noderna spelar ingen roll, så länge rätt noder finns med i svaret.

Uppgift 6

Vi har tagit fram ett paket för att hantera polynom av en variabel, t.ex:

```
5x^2+3x-2
```

För detta ändamål har vi definierat ett antal abstrakta datatyper:

- **polynom** består av en variabel (sträng) och en termlista
- **termlist** är en sekvens av termer lagrade i en array
- **term** består av ordning (heltal) och koefficient
- **coefficient** är i den här versionen enbart ett tal
- **variable** är en sträng som innehåller variabelns namn

I filen `poly.py` finns all kod som behövs. Där finns bl.a. funktionen `plus_poly` som kan addera två polynom och funktionen `print_poly` som skriver ut polynom. Det finns två testpolynom definierade i variablerna `p1` och `p2`. Prova först att använda dessa funktioner:

```
>>> print_poly(p1)
-2+3x+5x^2
>>> print_poly(plus_poly(p1, p2))
4+7x+3x^3+5x^2
```

Vi vill nu förbättra det här matematiska paketet på ett par olika punkter.

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 (1,5p)

Om man adderar polynomen 5x+3 och -5x+2 erhålls polynomet 0x+5. Vi får alltså en term vars koefficient är 0, helt i onödan. Modifiera koden så att sådana onödiga termer inte lagras i termlistan.

Deluppgift 6B (3,5p)

Inför en ny funktion `mult_poly` som utför multiplikation av polynom. Om det verkar befogat, inför gärna ytterligare primitiva funktioner.