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 19 april 2017 kl 14-19

Uppgift 1

Internet of Things (IoT) spås en lysande framtid. Många IoT-applikationer handlar om att samla in och analysera data från olika platser genom utplacerade sensorer. Anta att varje sensor skickar information som par där första värdet i paret är det unika namnet på sensorn och det andra värdet är det aktuella mätvärdet. Skriv ett funktion `analyze_data(seq)` som tar en lista med mätvärden och returnerar en dictionary med en lista för varje sensor som innehåller antal mätningar, min, och max. Nedan finns några exempel.

```
>>> analyze_data([('a', -1), ('a', 1)])
{ 'a': [2, -1, 1] }
>>> analyze_data([('a', 2), ('b', 0), ('a', 6), ('c', 0), ('b', 1)])
{ 'a': [2, 2, 6], 'b': [2, 0, 1], 'c': [1, 0, 0] }
```

Uppgift 2

Skriv en funktion om tar två sorterade listor och skapar en ny sorterad lista genom att slå samman listorna. Alla element från båda listorna ska vara med. Funktionen ska finnas i två varianter: en som arbetar enligt en rekursiv modell (kallad `merge_r`) och en som arbetar enligt en iterativ modell (kallad `merge_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:

```
>>> merge_r([1, 3, 5], [2, 4, 6])
[1, 2, 3, 4, 5, 6]
>>> merge_i([2, 4, 6], [1, 3, 5])
[1, 2, 3, 4, 5, 6]
>>> merge_i([1, 2], [1, 2])
[1, 1, 2, 2]
```

De båda funktionerna `merge_r` och `merge_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 `pred_comp(p, t, f)` som tar tre funktioner *p*, *t* och *f* som indata och returnerar en ny funktion. Funktionerna *p*, *t* och *f* tar var och en ett argument. Den returnerade funktionen ska också ta ett argument *x* och den ska returnera *t(x)* om *p(x)* är sant annars *f(x)*.

```
>>> pred_comp(lambda x: x > 0, lambda x: x, lambda x: -x)(-4)
4
>>> pred_comp(lambda x: x < 0, lambda x: x, lambda x: -x)(-4)
-4
```

Deluppgift 3b (2p)

Skriv en funktion `safe_div(x, y)` som använder `pred_comp` för att utföra säker division av *x*/*y*, dvs. som kan hantera att *y* kan vara 0. Vid division med 0 ska funktionen returnera 0.

```
>>> safe_div(10, 5)
2
>>> safe_div(10, 0)
0
```

Uppgift 4

Skriv en funktion `remove_duplicates` som tar en lista som kan innehålla godtyckligt nästlade listor och returnerar en ny lista där alla listelement som är exakt lika som föregående har tagits bort. 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.

```
>>> remove_duplicates([1, 1, 1])
[1]
>>> remove_duplicates([[1], [1], [1]])
[[1]]
>>> remove_duplicates([[2, 1, 2], [2], [1, 1, 1, 1, 2]])
[[2, 1, 2], [2], [1, 2]]
>>> remove_duplicates([2, [2], 2, [2]])
[2, [2], 2, [2]]
>>> remove_duplicates([[1, 1, [2, 2, 2, 2]], [1, 1, [2]]])
[[1, [2]]]
```

Uppgift 5

Vi vill ha en abstrakt datatyp för att representera prioritetsköer. En prioritetskö består av ett godtyckligt antal *element* där det är möjligt att plocka ut det störst/minsta elementet samt att lägga till nya element till kön. Välj en lämplig representation och definiera nedanstående primitiva funktioner. Alla funktioner ska vara väl dokumenterade.

`make_priority_queue()` : Returnerar en ny tom prioritetskö.

`length(pq)` : Returnerar längden på kön `pq`.

`front(pq)` : Returnerar största elementet i prioritetskön `pq`.

`back(pq)` : Returnerar minsta elementet i prioritetskön `pq`.

`push_d(pq, elt)` : Lägg till `elt` till prioritetskön `pq` destruktivt och returnera kön.

`pop_d(pq)` : Ta bort största elementet i priotetskön `pq` destruktivt och returnera kön.

`push_f(pq, elt)` : Lägg till `elt` till prioritetskön `pq` funktionellt och returnera den nya kön.

`pop_f(pq)` : Ta bort största elementet i priotetskön `pq` funktionellt och returnera den nya kön.

De fyra första funktionerna ger tillsammans 1p. De destruktiva respektive funktionella funktionerna ger 1p vardera.

Här följer några exempel på hur primitiverna ska fungera. Tänk på att utdata kan skilja sig något beroende på ditt val av representation och implementation.

```
>>> pq = make_priority_queue()
>>> pq1 = push_d(pq, 1)
>>> length(pq)
1
>>> length(pq1)
1
>>> push_d(pq, 2)
[2, 1]
>>> front(pq)
2
>>> back(pq)
1
>>> pop_d(pq)
[1]
>>> front(pq)
1
>>> pop_d(pq)
[]
```

```
>>> pq = make_priority_queue()
>>> pq1 = push_f(pq, 1)
>>> length(pq)
0
>>> length(pq1)
1
>>> pq2 = push_f(pq1, 2)
>>> front(pq2)
2
>>> back(pq2)
1
>>> pq3 = pop_f(pq2)
>>> front(pq3)
1
>>> pop_d(pq3)
[]
```

Uppgift 6

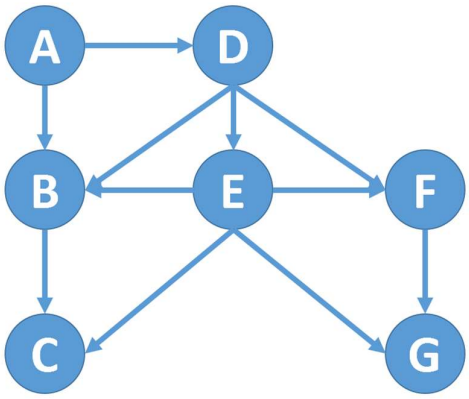
Topologisk sortering handlar om att ta en riktad acyklisk graf och hitta en linjär ordning av noderna så att en nod läggs till först när alla föräldrar till noden redan har lagts till. Det är vanligt att grafen representerar beroenden till exempel för ett schemaläggningsproblem så att uppgifter representeras som noder och det finns en båge från en nod till en annan om den första uppgiften måste utföras innan den andra uppgiften kan utföras. En enkel algoritm för att hitta en topologisk ordning är Kahns algoritm. Algoritmen ser ut så här:

Kahns algoritm för topologisk sortering:

1. L ← tom lista som kommer innehålla de sorterade noderna
2. S ← mängd av noder utan inkommande bågar
3. så länge S inte är tom gör
4. ta bort den fösta noden n från S
5. lägg till n sist i L
6. för varje nod m med en båge från n till m gör
7. ta bort bågen från grafen
8. om m inte har några fler inkommande bågar så
9. lägg till m till S
10. om grafen fortfarande har bågar kvar
11. returnera fel grafen har minst en cykel
12. annars
13. returnera L

Din uppgift är att implementera en funktion `topsort(graph)` som använder Kahns algoritm för att returnera noderna i topologisk ordning. Om det finns flera möjliga noder att välja på så ta dem i lexikografisk ordning. För ett exempel se nedan. Till din hjälp har du färdig kod för att skapa och manipulera grafer (`graph.py`). Funktionen ska inte vara destruktiv och alla funktioner ska vara väl dokumenterade. Om man kör algoritmen på grafen nedan kommer följande hända:

1. Endast A saknar ingående bågar.
2. Lägg till A till L. Ta bort A från grafen och bågarna till B och D. D har nu inga ingående bågar.
3. Lägg till D till L. Ta bort D från grafen och bågarna till B, E och F. E har nu inga ingående bågar.
4. Lägg till E till L. Ta bort E och bågarna till B, C, F och G. B och F har nu inga ingående bågar.
5. Lägg till B till L. Ta bort B och bågarna till C. C och F har nu inga ingående bågar.
6. Lägg till C till L. Ta bort C från grafen. F har nu inga ingående bågar.
7. Lägg till F till L. Ta bort F från grafen. G har nu inga ingående bågar.
8. Lägg till G till L.




```
>>> g = make_directed_graph(['A', 'B', 'C', 'D', 'E', 'F', 'G'], [('A',
'B'), ('A', 'D'), ('B', 'C'), ('D', 'B'), ('E', 'B'), ('E', 'C'), ('D',
'E'), ('D', 'F'), ('E', 'F'), ('E', 'G'), ('F', 'G')])
```

```
>>> topsort(g)
['A', 'D', 'E', 'B', 'C', 'F', 'G']
```