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 30 mars 2016 kl 14-19

Uppgift 1

En vanlig uppgift är att samla in och bearbeta data. Din första uppgift blir att skriva en funktion `rainfall` som räknar ut hur mycket nederbörd som fallit i genomsnitt. Funktionen ska ta en lista med mätvärden på hur mycket regn som fallit en viss dag som indata, listan innehåller flera sekvenser av mätvärden där en sekvens avslutas med exakt ett negativ tal eller att listan tar slut. Om en sekvens med mättal inte innehåller några observationer ska inget medelvärde ges. Funktionen ska beräkna medelnederbörden för varje sekvens och returnera en lista med dessa medelvärden.

```
>>> rainfall([1, 2, 3])
[2.0]
>>> rainfall([2, 2, 2, -1, 3, 3, 3])
[2.0, 3.0]
>>> rainfall([1.5, 4, 2, -1, 1, -1])
[2.5, 1.0]
>>> rainfall([1.25, 4, 3, -1, 1, -1, 0])
[2.75, 1.0, 0.0]
>>> rainfall([-5])
[]
```

Uppgift 2

I Python finns en funktion som heter `zip` som tar ett godtyckligt antal listor och skapar en lista med tupler där varje tupel innehåller ett element från vardera lista. Om listorna är olika långa så returneras endast så många tupler som det finns element i den kortaste listan. Din uppgift är att göra en egen funktion `zip` som tar exakt två listor som input. Funktionen ska finnas i två varianter: en som arbetar enligt en rekursiv modell (kallad `zip_r`) och en som arbetar enligt en iterativ modell (kallad `zip_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:

```
>>> zip_r([1, 2, 3], ['a', 'b', 'c'])
[(1, 'a'), (2, 'b'), (3, 'c')]
>>> zip_i(['c', 3], [8, 'q', 9, 't'])
[('c', 8), (3, 'q')]
```

De båda funktionerna `zip_r` och `zip_i` ger vardera 2.5 poäng, och det är möjligt att få poäng på dem oberoende av varandra.

Uppgift 3

Skriv en funktion `sum_if(f, p)` som tar två unära funktioner (dvs funktioner som tar ett argument), `f` och `p`, som indata och returnerar en funktion med två heltalsparametrar, `i` och `j`, som summerar `f(x)` för varje värde från `i` till och med `j` där `p(x)` är sann. Funktionen `p(x)` returnerar alltid sant eller falskt för varje värde på `x`.

```
>>> sum_if(lambda x: x*x, lambda x: x%2==1)(0, 5)
35
```

Uppgift 4

Skriv en funktion `unique` som tar en lista som kan innehålla godtyckligt nästlade listor och returnerar en ny lista som endast innehåller unika element. Tänk på att funktionen ska vara icke-destruktiv och att den ska göra varje nivå av listor unik. Det går bra att lösa uppgiften rekursivt eller iterativt. Tänk som vanligt på att dokumentera funktionen på lämpligt sätt. Din funktion behöver inte returnera elementen i exakt samma ordning som i exemplen nedan så länge listorna innehåller rätt element.

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

Uppgift 5

Deluppgift 5a (3p)

Vi vill ha en abstrakt datatyp för att representera grafer. Ett grafobjekt består av ett godtyckligt antal *noder* och ett godtyckligt antal viktade *bågar* som kopplar samman noderna. Bågar kan vara antingen riktade eller oriktade. Välj en lämplig representation och definiera nedanstående primitiva funktioner. Inga primitiver ska vara destruktiva och alla funktioner ska vara väl dokumenterade.

- `make_graph()`: Skapa en ny graf utan noder eller bågar.
- `make_directed_graph(nodes, edges)`: Skapa en ny graf med noder och riktade bågar.
- `make_undirected_graph(nodes, edges)`: Skapa en ny graf med noder och oriktade bågar.
- `add_nodes(graph, nodes)`: Lägg till noder till en graf.
- `add_directed_edges(graph, edges)`: Lägg till riktade bågar till en graf. Skapa nya noder om det behövs så att alla noder som nämns i bågarna finns.
- `add_undirected_edges(graph, edges)`: Lägg till oriktade bågar till en graf. Skapa nya noder om det behövs så att alla noder som nämns i bågarna finns.
- `get_edges(graph)`: Returnera alla bågar i grafen.
- `get_nodes(graph)`: Returnera alla noder i grafen.
- `get_neighbors(graph, node)`: Returnera alla grannar till en nod, dvs alla noder som går att nå direkt med en båge i grafen.

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.

```
>>> g = make_directed_graph(['A', 'B', 'C', 'D', 'E', 'F', 'G'], [('A',
'B', 7), ('A', 'D', 5), ('B', 'C', 8), ('B', 'D', 9), ('B', 'E', 7), ('C',
'E', 5), ('D', 'E', 15), ('D', 'F', 6), ('E', 'F', 8), ('E', 'G', 9), ('F',
'G', 11)])
>>> get_nodes(g)
['A', 'B', 'C', 'D', 'E', 'F', 'G']
>>> get_edges(g)
[('B', 'C', 8), ('B', 'D', 9), ('B', 'E', 7), ('A', 'B', 7), ('A', 'D', 5),
('E', 'F', 8), ('E', 'G', 9), ('C', 'E', 5), ('D', 'E', 15), ('D', 'F', 6),
('F', 'G', 11)]
>>> get_neighbors(g, 'B')
[('C', 8), ('D', 9), ('E', 7)]
>>> h = make_graph()
>>> add_undirected_edges(h, get_edges(g))
>>> get_nodes(h) == get_nodes(g)
True
```

Deluppgift 5b (2p)

Använd den nya datatypen graf för att skriva en funktion `has_cycle(graph, node)` som returnerar sant om och endast om det finns en cykel i grafen `graph` som startar i noden `node`.

```
>>> has_cycle(g, 'A')
False
>>> has_cycle(h, 'A')
True
```

Uppgift 6

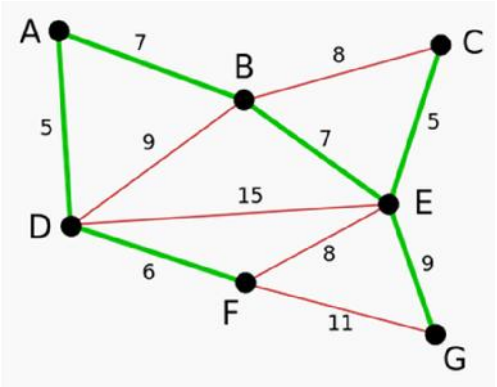
Anta att ett företag har ett antal servrar utplacerade på olika platser som de nu vill koppla samman med dedikerade fiberlänkar. För att koppla samman två servrar krävs det att de antingen har en direkt förbindelse eller har en indirekt förbindelse via någon annan server som den är direktkopplad till. Två servrar kan alltid kopplas samman med en fiber som går fågelvägen mellan dem. Då företaget dessutom vill minimera kostnaden vill de koppla samman alla servrarna så att den totala fiberlängden minimeras. Företaget önskar alltså hitta ett minsta uppspännande träd (*eng. minimum spanning tree*) i grafen av alla möjliga fiberlänkar.

En vanlig algoritm för att hitta ett minsta uppspännande träd är Kruskals algoritm som utgår från en lista med viktade bågar. I det här fallet är bågens vikt avståndet mellan platserna den förbinder.

Kruskals algoritm:
Låt T vara ett tomt träd. Upprepa tills T innehåller alla noder i G

- Låt v vara den kortaste bågen i G som inte märkts som förbrukad
- Märk v som förbrukad
- Om v inte bildar en cykel i T så addera v till T

Din uppgift är att implementera en funktion `mst(edges)` som använder Kruskals algoritm för att hitta ett minsta uppspännande träd givet en lista med bågar, där varje båge är en tupel `(from, to, weight)`där `from` och `to` är noder och `weight` är längden på bågen. Funktionen ska returnera dels totalkostnaden för trädet och dels bågarna som ingår i trädet. För ett exempel se nedan. Funktionen ska inte vara destruktiv och alla funktioner ska vara väl dokumenterade.




```
>>> mst([('A', 'B', 7), ('A', 'D', 5), ('B', 'C', 8), ('B', 'D', 9), ('B',
'E', 7), ('C', 'E', 5), ('D', 'E', 15), ('D', 'F', 6), ('E', 'F', 8), ('E',
'G', 9), ('F', 'G', 11)])
(39, [('A', 'D', 5), ('C', 'E', 5), ('D', 'F', 6), ('A', 'B', 7), ('B',
'E', 7), ('E', 'G', 9)])
```