Instruktioner - Datortentamen TDDD73 Funktionell och imperativ programmering i Python TDDE24 Funktionell och imperativ programmering del 2

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.
- Den implementerade lösningen ska kunna köras inom rimliga tidsramar. Till exempel är en lösning som tar 1 minut för att konkatenera två strängar som är 4 bokstäver långa inte acceptabel.

Uppgifter - Datortentamen TDDD73 Funktionell och imperativ programmering i Python TDDE24 Funktionell och imperativ programmering del 2

Onsdag 4 april 2018 kl 14-19

Uppgift 1

Bowling är en populär sport med anor från medeltiden. Då poängräkningen är lite krånglig är det skönt att ha ett program som kan hålla reda på poängen. Din uppgift är att skriva en funktion `score(pins)` som räknar ut poängen en spelare får om `pins` innehåller en sekvens av hur många käglor spelaren har fått ner i varje kast. Efter två kast eller efter en strike ställs alla 10 käglor upp igen. Får en spelare en strike, dvs får ner alla 10 käglor med ett kast, så får den 10 plus antalet poäng i de två kommande kasten. Får en spelare en spärr, dvs får ner alla 10 käglorna med två kast, så får den 10 poäng plus antalet poäng i nästa kast. Varje serie består av 10 omgångar, om spelaren får en spärr eller strike i sista omgången får den ett respektive två extra kast. Dessa två eller tre kast adderas ihop. Maximal poäng fås om en spelare gör 12 strikes i rad. Om de tre första slagen blir [9, 1, 10, 8, 0], dvs 9/spärr, strike, 8/miss, erhålls alltså följande tre poäng: 20, 38, 46. Om de fyra första slagen blir [10, 10, 4, 6, 8, 1], dvs. strike, strike, 4/spärr, 8/1, så fås följande poäng: 24, 44, 62, 71.

```
>>> score([6, 2, 8, 2, 10, 9, 0, 6, 4, 8, 1, 9, 1, 10, 10, 8, 2, 7])
168
>>> score([10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10])
300
>>> score([1, 3, 3, 6, 2, 5, 9, 0, 0, 5, 0, 0, 4, 5, 5, 3, 1, 8, 7, 2])
69
```

Uppgift 2

Skriv en funktion som tar två listor med tal och returnerar en ny lista som parvis summerar talen, dvs summerar *n*:e talet i första listan med *n*:e talet i andra listan. Funktionen ska hantera att listorna är olika långa genom att ta med talen i den längre listan som de är. Funktionen ska finnas i två varianter: en som arbetar enligt en rekursiv modell (kallad `pairwise_add_r`) och en som arbetar enligt en iterativ modell (kallad `pairwise_add_i`). Du får inte använda inbyggda funktioner eller metoder som behandlar hela listor (`len` är ok). 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:

```
>>> pairwise_add_r([1, 2, 3, 4, 5], [5, 4, 3, 2, 1])
[6, 6, 6, 6, 6]
>>> pairwise_add_i([2, 4, 6], [1, 3])
[3, 7, 6]
```

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

Uppgift 3

Skriv en funktion `find_three_smallest_numbers(seq)` som tar en lista som kan innehålla godtyckligt nästlade listor med godtyckliga element och returnerar en sorterad lista med de tre minsta talen. Om det finns färre än tre tal så returnera alla. Funktionen ska vara icke-destruktiv. Det går bra att lösa uppgiften rekursivt eller iterativt. Tänk som vanligt på att dokumentera funktionen på lämpligt sätt.

```
>>> find_three_smallest_numbers([1, 2, 3])
[1, 2, 3]
>>> find_three_smallest_numbers([[1], [[2], 3]])
[1, 2, 3]
>>> find_three_smallest_numbers([[1, [[1]]], 1, [[1], 2], 3])
[1, 1, 1]
>>> find_three_smallest_numbers([[1], 3])
[1, 3]
>>> find_three_smallest_numbers([[1, 'du', [['e']]], [1, "bra"], [[1], 2]])
[1, 1, 1]
```

Uppgift 4

Deluppgift 4a (3p)

Skriv en högre ordningens funktion `pairwise_apply(f)` som tar en binär funktion `f` som argument och returnera en ny funktion som tar en lista som argument och applicerar funktionen `f` på varje par av element i listan. Om listan innehåller ett udda antal element ska den nya funktionen ignorera det sista elementet.

```
>>> def multiply(x, y):
...     return x*y
>>> f = pairwise_apply(multiply)
>>> f([1, 2, 3, 4, 5, 6])
[2, 12, 30]
```

Deluppgift 4b (2p)

Skriv ett uttryck som använder `pairwise_apply` för att implementera `pairwise_add` som den definieras i uppgift 2 (utan restriktioner på inbyggda funktioner och metoder). Det ska vara exakt ett uttryck `pairwise_add = …` som i sig ska lösa uppgiften. Du kan t.ex. inte definiera extra funktioner.

Uppgift 5

Subset sum är ett klassiskt problem. Givet en multimängd, dvs en mängd som kan innehålla samma tal flera gånger, avgör om någon delmängd av tal summerar till 0. Exempelvis, om vi har talen [-7, -3, -2, 5, 8] så är svaret ja då summan av -3, -2 och 5 är 0. Skriv en funktion `subset_sum(numbers, sum)` som tar en lista med tal `numbers` och en summa `sum` och returnerar sant om och endast om det går att hitta en delmängd av tal som summerar till `sum`. Funktionen behöver inte vara minnes- eller tidseffektiv. Det är till exempel acceptabelt att lösningen provar alla alternativ.

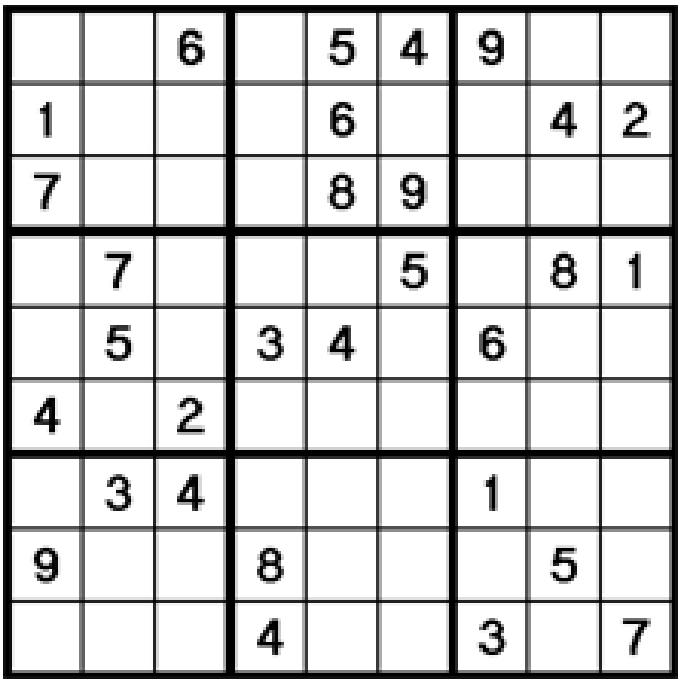
```
>>> subset_sum([-7, -3, -2, 5, 8], 0)
True
>>> subset_sum([-7, -3, -2, 5, 8], 5)
True
>>> subset_sum([-7, -3, -2, 5, 8], 7)
False
>>> subset_sum([3, 34, 4, 12, 5, 2], 9)
True
```

Uppgift 6

Sudoku är ett klurigt spel där du ska fylla i siffror i rutor så att varje rad, varje kolumn och 3x3 ruta innehåller varje siffra 1-9 exakt en gång. Din uppgift är designa och implementera en abstrakt datatyp för Suduko. Datatypen ska kunna användas av en lösare för att hålla reda på vilka siffror som kan finnas i varje ruta och ska alltid vara konsistent. Varje operation ger 0.5 poäng förutom `set_value` som ger 2 poäng. Enkla tester ska bifogas till lösningen.

- `create_board()` – Returnera en ny sudoko utan några siffror ifyllda så att varje ruta kan ha vilket värde som helst.
- `row(b, r)` – Returnera en lista med cellerna i rad `r`, där `r` är ett värde mellan 1 och 9, och där varje cell är en lista med vilka värden som den cellen kan ta.
- `column(b, c)` – Returnera en lista med cellerna i kolumn `c`, där `c` är ett värde mellan 1 och 9, och där varje cell är en lista med vilka värden som den cellen kan ta.
- `square(b, s)` – Returnera en lista med cellerna i ruta `s`, där `s` är ett värde mellan 1 och 9, och där varje cell är en lista med vilka värden som den cellen kan ta.
- `remove_value(b, r, c, v)` – Ta bort värdet `v` från mängden av möjliga värden för cellen (`r`, `c`) på brädet `b`.
- `set_value(b, r, c, v)` – Tilldela cellen (`r`, `c`) på brädet `b` värdet `v`. Alla rader, kolumner och rutor ska uppdateras. Funktionen ska använda `remove_value` för att uppdatera brädet.

- `set_row(b, r, values)` – Tilldela raden `r` på brädet `b` värdena i `values`, om något värde är 0 betyder det att inget värde ska tilldelas den kolumnen. Funktionen ska använda `set_value` för att tilldela värden till celler.



Följande är ett körexempel som fyller i ett Sudoku enligt bilden ovan:

```
>>> b = create_board()
>>> set_row(b, 1, [0, 0, 6, 0, 5, 4, 9, 0, 0])
>>> set_row(b, 2, [1, 0, 0, 0, 6, 0, 0, 4, 2])
>>> set_row(b, 3, [7, 0, 0, 0, 8, 9, 0, 0, 0])
>>> set_row(b, 4, [0, 7, 0, 0, 0, 5, 0, 8, 1])
>>> set_row(b, 5, [0, 5, 0, 3, 4, 0, 6, 0, 0])
>>> set_row(b, 6, [4, 0, 2, 0, 0, 0, 0, 0, 0])
>>> set_row(b, 7, [0, 3, 4, 0, 0, 0, 1, 0, 0])
>>> set_row(b, 8, [9, 0, 0, 8, 0, 0, 0, 5, 0])
>>> set_row(b, 9, [0, 0, 0, 4, 0, 0, 3, 0, 7])
>>> row(b, 2)
[[1], [8, 9], [3, 5, 8, 9], [7], [6], [3, 7], [5, 7, 8], [4], [2]]
>>> set_value(b, 2, 4, 7)
>>> row(b, 2)
[[1], [8, 9], [3, 5, 8, 9], [7], [6], [3], [5, 8], [4], [2]]
>>> column(b, 3)
[[6], [3, 5, 8, 9], [3, 5], [3, 9], [1, 8, 9], [2], [4], [1, 7], [1, 5, 8]]
>>> square(b, 1)
[[2, 3, 8], [2, 8], [6], [1], [8, 9], [3, 5, 8, 9], [7], [2, 4], [3, 5]]
```