Instruktioner för datortentamen Torsdag 2019-01-17 (8–13)

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 som inte hör till bokens innehåll.
- 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 datormiljö. Med hjälp av en särskild **tentaklient** skickar du in dina svar. Varje uppgift kan **skickas in en enda gång** och kommer därefter att poängsättas. Man kan alltså inte komplettera.

Poängsättning kan *i viss mån* ske redan under tentans gång, men detta är inte garanterat utan beror bland annat på antalet tillgängliga rättare.

Du kan även använda tentaklienten för att **ställa frågor** till jourhavande, som vid behov kan vidarebefordra till examinator. Besök i skrivsalen kommer endast att ske vid allvarligare tekniska problem.

För att underlätta rättningen:

Namnge de inlämnade filerna ex1.py, ex2.py, ..., ex6.py

Betygsättning

Datortentan består av totalt 6 uppgifter som vardera kan ge maximalt 5 poäng. Alla uppgifter är utvalda för att testa olika aspekter av kursinnehållet. De är också tänkta att ha olika svårighetsgrad och är ordnade så att den svårigheten oftast ökar mot slutet. 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 under kursens laborationer. Detta inkluderar att skriva **docstrings för samtliga funktioner**, även hjälpfunktioner som man själv inför. Specialtaggar som `:param:` krävs ej.
- Lösningen ska följa de **specifika regler och villkor** 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 för alla tänkbara indata som följer uppställningen i frågan.

  Man behöver dock normalt **inte felkontrollera** indata, utom när uppgiften särskilt anger det. Funktioner måste alltså ge korrekta svar för korrekta indata enligt uppgiften, men får normalt krascha eller ge felaktiga svar för felaktiga indata (såsom när en funktion som ska hantera heltal ges en sträng som parameter).
- Den implementerade lösningen ska kunna köras inom **rimliga tidsramar**. Till exempel accepteras inte en lösning som tar 1 minut för att konkatenera två strängar som är 4 bokstäver långa. Eventuella undantag anges uttryckligen i frågan.

Namnge de inlämnade filerna ex1.py, ex2.py, ..., ex6.py

Uppgift 1

Vi är intresserade av att hitta det ***minsta udda talet*** och det ***största jämna talet*** i en rak lista av heltal. Din uppgift blir därför att skriva funktionen `minodd_maxeven(seq)`, som returnerar en tupel innehållande dessa två värden i den angivna ordningen:

- `assert minodd_maxeven([0, -15, 2, 1, -1, -50, -2, 55, 56]) == (-15, 56)`

Kopiera gärna in i dessa tester i din egen kodfil för att förenkla din egen testning!

Om listan saknar udda tal, ska det första värdet i den returnerade tupeln vara `None`.
Om listan saknar jämna tal, ska det andra värdet i den returnerade tupeln vara `None`.
Nedan följer ett exempel där listan varken har udda eller jämna tal.

- `assert minodd_maxeven([]) == (None, None)`

Ytterligare villkor:

- Lösningen får vara iterativ eller rekursiv. Det är också tillåtet att använda både listbyggare (list comprehensions) och inbyggda funktioner som arbetar på hela listor.
- Funktionen ska vara funktionell i bemärkelsen att inskickade parametrar inte får modifieras.

Bra att tänka på

- Alla funktioner ska dokumenteras med docstrings. Detta är sista varningen!
- De testfall vi ger i tentan är bara exempel, och är främst till för att förtydliga vad vi menar med beskrivningstexten. Det finns inga garantier för att en lösning som uppfyller de här testfallen är helt korrekt. Skapa gärna egna tester som kan täcka ytterligare fall.

  Testfall *får* men *behöver inte* lämnas in.
- Testa efter varje ändring, även när du *tror* att ändringen var harmlös. Inlämnad kod måste vara körbar.

Namnge de inlämnade filerna ex1.py, ex2.py, ..., ex6.py

Uppgift 2

Skriv en funktion `skiplist(seq)` som tar en sekvens av icke-negativa heltal. Om denna sekvens är **tom** ska funktionen returnera `[]`. Om den är **icke-tom**:

- Det första elementet i `seq` ska tas med i resultatet. Detta element talar också om hur många element som därefter ska *hoppas över* i listan.

  Funktionen ska klara av att färre element än detta finns kvar i listan, och i så fall helt enkelt hoppa över samtliga kvarvarande element!

- Om `seq` inte tar slut efter att dessa element har hoppats över, ska nästa element finnas med i resultatet, och talar också om hur många element som ska hoppas över därnäst.
- Och så vidare...

Exempel:

- `skiplist([1, 2, 3, 4, 5, 6])` blir `[1, 3]`, eftersom vi ska ta med värdet `1`, hoppa över ett element (elementet `2`), ta med nästa element (`3`), hoppa över tre element (elementen `4`, `5`, `6`), och sedan är listan slut.

Funktionen ska finnas i **två varianter** som ska fungera "exakt likadant", bortsett från att de ska ha olika lösningsmodeller:

- `skiplist_r` ska arbeta enligt en rekursiv modell.
- `skiplist_i` ska arbeta enligt en iterativ modell.

Ytterligare villkor:

- "Överhoppandet" av ett antal element får ske i ett enda steg, oavsett lösningsmodell.
- Du får inte använda inbyggda funktioner som behandlar alla element i hela listor. Däremot är givetvis `len()` tillåten.
- Du får inte använda listbyggare (eng. list comprehensions).
- Lösningarna ska vara funktionella i bemärkelsen att de inte får modifiera listan som skickas in.

Funktionerna ska bland annat klara följande **tester**.

```
assert skiplist_i([]) == []
assert skiplist_i([1]) == [1]
assert skiplist_i([5, 4, 3, 2, 1, 2, 3, 4, 5, 6, 7, 8]) == [5, 3, 7]
assert skiplist_r([]) == []
assert skiplist_r([1]) == [1]
assert skiplist_r([5, 4, 3, 2, 1, 2, 3, 4, 5, 6, 7, 8]) == [5, 3, 7]
```

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

Uppgift 3

Skriv en funktion `length_of_each(lst)`, som:

- Tar en lista som kan innehålla godtyckligt tomma och icke-tomma nästlade listor
- Returnerar en lista med samma nästlade struktur, men där man för varje element `x` som *inte* är en lista istället har med *längden* av det ursprungliga elementet (`len(x)`), där elementet till exempel kan vara en sträng.

Tänk på att eftersom listan är nästlad, kan även elementen vara godtyckliga nästlade listor.

Ytterligare villkor:

- Funktionen ska vara funktionell i bemärkelsen att inskickade parametrar inte får modifieras.
- Det är tillåtet att lösa uppgiften rekursivt eller iterativt, eller med en kombination av dessa metoder.
- Det är tillåtet att använda inbyggda funktioner som arbetar på hela listor. Det är också tillåtet att använda listbyggare (list comprehensions), men dessa arbetar inte själva på nästlade listor.
- Det är tillåtet att anta att alla element som inte är listor stödjer `len()`.

Exempel:

- `assert length_of_each(["hej", "på", "dej"]) == [3,2,3]`
- `assert length_of_each(["hej", "på", ["alla", "som", "skriver", "tenta"], "i", [[["Python"]]]]) == [3, 2, [4, 3, 7, 5], 1, [[[6]]]]`

Uppgift 4

Deluppgift 4a (3p)

Skapa en högre ordningens funktion `pairwise_apply(f)`. Funktionen ska **ta** en binär funktion `f` som argument och **returnera** en ny funktion, som i sin tur:

- Tar *två* möjligen tomma listor `l1` och `l2` som argument
- Applicerar funktionen `f` på de första elementen i `l1` och `l2`, nästa element i `l1` och `l2`, och så vidare (exempel nedan)
- Returnerar en lista som innehåller alla resultaten, i samma ordning som i de ursprungliga listorna.

Exempel:

- Anta att vi sätter `newfun = pairwise_apply(f)`, där `f` är någon godtycklig funktion.
- Då ska `newfun([1, 2, 3], [7, 9, 11])` returnera `[f(1,7), f(2,9), f(3,11)]`.

Ytterligare villkor:

- Om en lista innehåller fler element än den andra, ska de överskjutande elementen ignoreras. Exempel ges i slutet av deluppgift 4b.
- Funktionen ska vara funktionell i bemärkelsen att inskickade parametrar inte får modifieras.
- Det är tillåtet att använda inbyggda funktioner som arbetar på hela listor. Det är också tillåtet att använda listbyggare (list comprehensions).

Deluppgift 4b (2p)

Använd `pairwise_apply` för att definiera `pairwise_multiply(l1, l2)`, där varje returnerat element är *produkten* av två element från `l1` och `l2`. Detta ska definieras med hjälp av ett enda uttryck:

- `pairwise_multiply = ...`

Du ska göra detta utan att definiera egna hjälpfunktioner utöver `pairwise_apply`.

Exempel:

- `assert pairwise_multiply([1, 2, 3], [7, 9, 11]) == [7, 18, 33]`
- `assert pairwise_multiply([3], [7, 9, 11]) == [21]`

Uppgift 5

Anta att du har en lista av ord som du vill skriva ut som följer:

- Orden ska skrivas ut som ett stycke, där varje par av ord separeras av antingen exakt ett mellanslag eller en radbrytning.
- Ingen rad får ha färre än `minlen` tecken, inte ens den sista, och ingen rad får ha fler än `maxlen` tecken. I tecken räknar vi in mellanslag, men inte radbrytningar.

Exempelvis kan man skriva texten i den sista punkten ovan med `minlen=37`, `maxlen=41`:

```
40  Ingen rad får ha färre än minlen tecken,
37  inte ens den sista, och ingen rad får
41  ha fler än maxlen tecken. I tecken räknar
41  vi in mellanslag, men inte radbrytningar.
```

Att göra:

Skriv funktionen `can_break_lines(words, minlen, maxlen)`, med följande parametrar:

- `words`, en icke-tom lista av ord. Ett ord är en icke-tom textsträng som enbart består av skrivbara tecken, utan mellanslag eller radbrytningar med med t.ex. punkter.
- `minlen`, ett heltal strikt större än 0.
- `maxlen`, ett heltal där `minlen <= maxlen`.

Funktionen ska returnera `True` om det går att radbryta texten enligt ovanstående beskrivning så att radlängdsvillkoren uppfylls, och annars `False`.

Ytterligare information och villkor:

- I denna uppgift behöver raderna *inte* vara så långa som möjligt inom gränserna, även om detta kanske skulle vara önskvärt i verkligheten.
- Funktionen behöver *inte* vara minnes- eller tidseffektiv. Det är till exempel acceptabelt att lösningen provar alla alternativ.
- Lösningen får vara iterativ eller rekursiv. Det är också tillåtet att använda både listbyggare (list comprehensions) och inbyggda funktioner som arbetar på hela listor.
- Funktionen ska vara funktionell i bemärkelsen att inskickade parametrar inte får modifieras.

Exempel:

- `w=["Ingen", "rad", "utom", "den", "sista", "får", "ha", "färre", "än", "minlen", "tecken,", "och", "ingen", "rad", "(överhuvudtaget)", "får", "ha", "fler", "än", "maxlen", "tecken.", "I", "tecken", "räknar", "vi", "in", "mellanslag,", "men", "inte", "radbrytningar."]`
- `assert can_break_lines(w, 32, 45)`
- `assert can_break_lines(w, 28, 34)`
- `assert not can_break_lines(w, 28, 32)`

Uppgift 6

Golomb-sekvensen är en intressant sekvens av heltal, namngiven efter Solomon W. Golomb. Den börjar så här:

$$1, 2, 2, 3, 3, 4, 4, 4, 5, 5, 5, 6, 6, 6, 6, 7, 7, 7, \ldots$$

Vi kan också skriva den så här:

$$a_1 = 1, \quad a_2 = 2, \quad a_3 = 2, \quad a_4 = 3, \quad a_5 = 3, \quad a_6 = 4, \quad a_7 = 4, \quad a_8 = 4, \quad a_9 = 5, \ldots$$

Varifrån kommer detta? Jo, sekvensen är en *icke minskande* sekvens (nästa tal är alltid samma eller större), där varje tal a_i talar om *antalet* gånger som just i inträffar i sekvensen. Om vi då börjar med att sätta startvillkoret $a_1 = 1$, ser vi:

- Vi vill ha ett värde på a_2. Det får inte vara 1, för $a_1 = 1$ betyder att vi ska använda "*en* etta" och vi ju har redan använt talet 1 en gång i sekvensen. Då kan man bevisa att nästa tal i sekvensen måste vara nästa heltal, det vill säga 2 (men själva beviset utelämnar vi här).
- Nu ska a_3 få ett värde. Vi har bara använt "en tvåa" och vi skulle ha två tvåor, så vi måste även ha $a_3 = 2$.
- Nu ska a_4 få ett värde. Det får inte vara 2, för vi har redan använt 2 två gånger. Då måste nästa tal bli nästa heltal, $a_4 = 3$.

Att göra

Skriv en funktion `golomb(n)` som tar ett icke-negativt heltal `n` och returnerar en lista som innehåller de första `n` talen i den oändliga Golomb-sekvensen. Exempel:

- `assert golomb(0) == []`
- `assert golomb(4) == [1, 2, 2, 3]`
- `assert golomb(10) == [1, 2, 2, 3, 3, 4, 4, 4, 5, 5]`