Instruktioner för datortentamen 2020-01-17 (14:00)

TDDE24 Funktionell och imperativ programmering del 2

Hjälpmedel

Följande fysiska hjälpmedel är tillåtna:

- Pennor, radergummin och liknande för att kunna skissa lösningar på papper. (Lösa tomma papper tillhandahålls av tentamensvakterna.)

Du har också tillgång till `https://docs.python.org/3/` via startmenyn. Detta inkluderar både biblioteks- och språkreferenser.

Icke tillåtna hjälpmedel inkluderar bl.a. alla former av elektronisk utrustning samt böcker och anteckningar.

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**. Man kan alltså inte komplettera. Precis som vid vanliga tentor kommer alla svar att granskas efter tentans slut.

Du använder tentaklienten för att **ställa tentarelaterade frågor**. Dessa går antingen till examinatorn eller till jourhavande, som vid behov kan vidarebefordra dem. På detta sätt får man oftast betydligt snabbare svar än på en “vanlig" tenta, där man normalt sparar sina frågor till examinatorn besöker salen och alltså kan få vänta ett par timmar. Det finns dock ingen garanti att man får ett *omedelbart* svar, speciellt om många ställer frågor på samma gång. **Läs alltså genom uppgifterna i förväg och ställ frågor i god tid, precis som på en vanlig tenta!** Har du inte fått svar inom **30 minuter**, skicka en gång till.

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

Tillgängliga verktyg

Som diskuterats i förväg finns flera editorer tillgängliga, men de har inte nödvändigtvis alla plugins installerade. Meningen är att man ska få tillgång till de menyer och tangentbordsbindningar som man är van vid, inte att man ska ha en fullständig utvecklingsmiljö.

Konfiguration

Vi erbjöd en möjlighet att få med “de nödvändigaste delarna av enklare konfigurationsfiler" till tentan. Enbart en sådan konfiguration har kommit in:

```
setxkbmap -option caps:swapescape
```

Uppgifter, poäng och betyg

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 svårigheten oftast ökar mot slutet. Vad som är lätt eller svårt skiljer sig naturligtvis från person till person, så det är alltid en bra idé att snabbt läsa igenom alla uppgifterna för att bättre kunna prioritera arbetet.

För betyg 3, 4 och 5 krävs minst 12, 19 respektive 25 poäng.

Rättningskriterier

Lösningar som bryter mot följande allmänna kriterier kommer att få poängavdrag.

- Programkod ska vara **körbar**. Testa alltid **precis innan inlämning** så att din sista finputsning eller dina sista kommentarer inte gav felaktig kod! Poängavdrag ges vid icke körbar kod, även för *delvis* körbar kod (som kraschar för vissa fall).
- 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 svenska eller engelska **docstrings** för **samtliga funktioner som definieras med `def` på "toppnivån**", även hjälpfunktioner som man själv inför. Specialtaggar som `:param:` krävs *inte*.

  Det är *inte* ett absolut krav att skriva docstrings för lambda-uttryck eller för nästlade funktioner som definieras inuti andra funktioner. Tänk dock på att koden ändå behöver vara tillräckligt väldokumenterad för att vara lätt att förstå.
- 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 också **fungera exakt som i körexemplen** i uppgiften, om inte texten indikerar något annat.
- Lösningen ska vara **generell**: Den ska inte enbart fungera för de körexempel som finns i uppgiften, utan för alla tänkbara indata som följer uppställningen i frågan. Att en lösning t.ex. enbart fungerar för listor med begränsad längd eller för vissa storlekar på indata är ett signifikant fel.

  Man behöver dock **inte felkontrollera** indata, utom när uppgiften särskilt anger det. Funktioner måste alltså ge *korrekta svar för* ***korrekta indata*** enligt uppgiften, men om inte annat anges får de *krascha eller ge felaktiga svar för* ***felaktiga indata*** (såsom när en funktion som bara 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.
- Om inte annat sägs ska funktioner **returnera** värden, inte skriva ut dem. Eventuella testfall visar normalt returnerade värden, inte värden som funktionerna har skrivit ut.

Avdrag för felaktig och ofullständig kod ges **även om lösningen är uppenbar** för granskaren, och även om det syns att lösningen "var på rätt väg". Det ingår i uppgiften att se till att koden uppfyller specifikationen, och att *själv* upptäcka och åtgärda problem under tentans gång. Den inlämnade koden är ditt slutgiltiga svar, inte ett mellansteg att arbeta vidare med.

De **testfall** vi ger i tentan är bara exempel, och är främst till för att förtydliga vad vi menar med beskrivningstexten. Det finns många felaktiga "lösningar" som uppfyller testfallen. **Skapa gärna egna tester** som täcker ytterligare fall. Tänk på specialfall som tomma listor, negativa tal, olika elementtyper, med mera. Testfall *får* men *behöver inte* lämnas in.

Uppgift 1

Skriv en funktion `divide(n, seq)` som delar upp elementen från den godtyckliga sekvensen `seq` i totalt `n` *listor* $\texttt{seq}_0, \ldots, \texttt{seq}_{n-1}$, där $\texttt{n} > 0$ är ett godtyckligt positivt heltal. Funktionen ska sedan returnera en lista av listor, $[\texttt{seq}_0, \ldots, \texttt{seq}_{n-1}]$.

Elementen i listorna $\texttt{seq}_0, \ldots, \texttt{seq}_{n-1}$ ska vara samma som i ursprungslistan `seq`, i samma ordning.

Om antalet element i `seq` är jämnt delbart med `n` ska alla listor $\texttt{seq}_i$ ha lika många element:

- `divide(2, [1, 2, 3, 4, 5, 6]) == [[1,2,3], [4,5,6]]`

Annars ska de överblivna elementen finnas i de första listorna, och den *sista* listan blir kortare:

- `divide(3, (1, 2, 3, 4, 3)) == [[1,2], [3,4], [3]]`

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.
- Lösningen får **inte** använda funktioner eller annan funktionalitet från `itertools`.
- Dina funktioner får inte modifiera sina indata.

Att tänka på:

-
- Eftersom `seq` är en *godtycklig* sekvens kan det hända att den ursprungliga listan innehåller färre element än `n`. Då blir de sista resultatlistorna tomma.

Ytterligare exempel – skapa gärna egna tester med inspiration av villkoren ovan!

- `assert divide(3, [1, 2, 3, 4, 5, 6, 7]) == [[1,2,3], [4,5,6], [7]]`

Extra information utskickad under tentans gång:

I uppgift 1 står:

"Annars ska de överblivna elementen finnas i de första listorna, och den sista listan blir kortare: divide(3, (1, 2, 3, 4, 3)) == [[1,2], [3,4], [3]]"

Det kan också uppstå en situation där man antingen behöver ha flera *kortare* listor eller flera *tomma* listor på slutet. Då detta är den allra första uppgiften accepterar vi båda varianterna.

Först ett exempel där att det är flera tomma listor på slutet:

divide_1(9, [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20]) == [[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12], [13, 14, 15], [16, 17, 18], [19, 20], [], []]

Vi accepterar *också* den mer naturliga lösningen där man har fördelat elementen så jämnt som möjligt, så att man inleder med listor med k element och därefter har listor med k' < k

element:

divide_1(9, [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20]) == [[1, 2, 3], [4, 5, 6], [7, 8], [9, 10], [11, 12], [13, 14], [15, 16], [17, 18], [19, 20]]

Bra att komma ihåg

- Dokumentera funktioner med docstrings! Detta är sista varningen.
- Testa efter varje ändring, även när du *tror* att ändringen var harmlös. Testa omedelbart före inlämningen.

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

Uppgift 2

Vi vill nu kunna ta en sekvens som innehåller $n \geq 0$ godtyckliga heltal och dela upp denna sekvens i sammanhängande delar, listor där talen följer direkt efter varandra. Elementen ska fortfarande komma i samma ordning som i den ursprungliga sekvensen.

Exempel:

- `consecutive_parts((1, 8, 9, 10, 4, 5, 22, 6)) == [[1], [8, 9, 10], [4, 5], [22], [6]]`

Att göra: Skriv två implementationer av `consecutive_parts`. Båda ska ta en godtycklig sekvens av godtyckliga heltal och returnera en gruppering enligt ovan.

- `consecutive_parts_i(seq)` ska arbeta enligt **iterativ** lösningsmodell.
- `consecutive_parts_r(seq)` ska arbeta enligt **rekursiv** lösningsmodell.

  Kom ihåg: En rekursiv *lösningsmodell* innebär inte bara att funktionen anropar sig själv, utan att varje rekursivt anrop ska beräkna och returnera *det korrekta svaret för ett delproblem*. Anroparen använder sedan detta svar för att beräkna sitt resultat för ett större problem. Ett exempel är fakultetsfunktionen där `fac(n) = n * fac(n-1)`, där `fac(n-1)` är ett rekursivt anrop som löser delproblemet för ett mindre värde på `n`.

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

Ytterligare villkor:

- Du får inte använda listbyggare (*list comprehensions*) eller inbyggda funktioner som behandlar alla element i hela listor, utan måste själv iterera eller rekursera över listorna, *ett element i taget*. Funktioner som inte behandlar elementen, såsom `len()`, är tillåtna.
- Dina implementationer får inte modifiera sina indata.

Båda funktionerna ska bl.a. klara följande **tester**, där `consecutive_parts` är antingen `consecutive_parts_i` eller `consecutive_parts_r`. Skapa gärna fler tester med inspiration av villkoren ovan!

- `assert consecutive_parts([]) == []`
- `assert consecutive_parts((-1, 0, 1, 42, 43)) == [[-1, 0, 1], [42, 43]]`

Uppgift 3

Skriv en funktion `find_nearest(nseq1, seq2)` som tar en godtyckligt nästlad lista `nseq1` med $n_1 \geq 0$ heltal, samt en icke-nästlad lista `seq2` med $n_2 \geq 1$ heltal. Inga andra element än heltal (och nästlade listor i `nseq1`) kan förekomma.

Funktionen ska returnera en ny lista som för varje heltal `x` i `nseq1` innehåller det heltal i `seq2` som är närmast `x`. Den nya listan ska i övrigt ha samma nästlingsstruktur som `nseq1`.

Det kan inträffa att det finns två olika tal som båda är närmast: Både 6 och 4 kan vara närmast till 5 (avstånd 1). I detta fall ska du ta det *största* av de två lika nära talen.

Exempel:

- `find_nearest([[11]], [5, 8, 10, 15, 12]) == [[12]]`

Ytterligare villkor:

- Inskickade parametrar får inte modifieras. Det är alltså en helt ny lista som ska returneras.
- 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.

Exempel – skapa gärna egna tester med inspiration av villkoren ovan!

- `assert find_nearest([], [1]) == []`
- `assert find_nearest([10], [5, 8, 10, 15, 12]) == [12]`
- ```
  assert find_nearest([12, [[[10]]]],
                      [3, 9, 7, 5, 11, 5, 13]) ==
                      [13, [[[11]]]]
  ```
  ```

Uppgift 4

Deluppgift 4a (3p)

Skapa en högre ordningens funktion `sum_satisfying(f, p)`, som:

- Tar två unära funktioner, `f` och `p`, som argument.
- Returnerar en ny unär funktion, som tar en godtycklig sekvens `seq` och returnerar *summan* av `f(x)` för alla element `x` ∈ `seq` som uppfyller `p(x)`.

Exempel som summerar längden på "numeriska strängar" – skapa gärna egna tester!

- `assert sum_satisfying(len, str.isdigit)(['10', 'yes', 'no', 'whatever', '4711']) == 6`

Denna funktion testas också indirekt av exempel i deluppgift 4b.

Ytterligare villkor:

- Om den returnerade funktionen anropas med en sekvens där inget element uppfyller `p(x)`, ska 0 returneras (summan av 0 värden är 0).
- Dina funktioner får inte modifiera sina indata.
- Det är tillåtet att använda inbyggda funktioner som arbetar på hela listor. Det är också tillåtet att använda listbyggare (list comprehensions). Både rekursiva och iterativa lösningsmodeller är tillåtna.

Deluppgift 4b (2p)

Använd `sum_satisfying` från uppgift 4a för att definiera en ny funktion `sum_square_negative_odd(seq)`, som tar en godtycklig sekvens av *heltal* och returnerar summan av *kvadraterna* av alla *negativa udda* heltal i denna sekvens.

Funktionen `sum_satisfying` ska bara anropas en gång: När du skapar den nya funktionen `sum_square_negative_odd`. Den ska inte anropas varje gång `sum_square_negative_odd` anropas.

Exempel:

- `assert sum_square_negative_odd([-1, 0, -3, 5, 2, 7]) == 10`

Uppgift 5

I den diskreta matematiken definieras den **kartesiska produkten** av 2 mängder s_1, s_2 som den möjligen tomma mängden $s_1 \times s_2 = \{(e_1, e_2) \mid e_1 \in s_1 \text{ and } e_2 \in s_2\}$. Detta kan lätt generaliseras till kartesiska produkter av $n > 2$ mängder $s_1, s_2, \ldots, s_n$, där elementen är alla n-tupler av formen $(e_1, e_2, \ldots, e_n)$ där $\forall i : e_i \in s_i$.

Exempel:

- $\{1, 2\} \times \{3, 4, 5\} \times \{11\} =$
  $\{(1, 3, 11), (1, 4, 11), (1, 5, 11), (2, 3, 11), (2, 4, 11), (2, 5, 11)\}$

Att göra: Implementera funktionen `cartprod(sets)`, som tar en *lista av $n \geq 2$ godtyckliga mängder* och returnerar *kartesiska produkten* av alla dessa mängder, vilket alltså är en mängd av tupler.

Tips: Man kan skapa en tupel genom att först skapa en lista `seq`, som kan modifieras, och sedan konvertera den till en tupel med hjälp av `tuple(seq)`.

Ytterligare villkor och information:

- Du får inte använda funktioner eller annan funktionalitet från `itertools` för att lösa denna uppgift.

Exempel:

- `assert cartprod([{1,2}, {3,4,5}, {11}]) ==`
  `{(1,3,11),(1,4,11),(1,5,11),(2,3,11),(2,4,11),(2,5,11)}`

Uppgift 6

I matematiken har man identifierat många olika intressanta egenskaper som tal kan uppfylla. Till exempel är ett **Hoax-tal** ett heltal n som uppfyller följande villkor:

- Det är *sammansatt* (inte ett primtal)
- Summan av dess siffror = summan av siffrorna i de *distinkta* talen i dess primtalsfaktorisering[1]

Exempel:

- 136 kan primtalsfaktoriseras till $2 \cdot 2 \cdot 2 \cdot 17$ där alla faktorer är primtal (och då det finns mer än en primtalsfaktor är talet alltså sammansatt)
- 136 har siffersumman $1 + 3 + 6 = 10$
- 136 har de *distinkta* primtalsfaktorerna 2 och 17 (vi hoppar över upprepningarna av 2:an), och dessas siffersumma är $2 + 1 + 7$ vilket också är 10
- Alltså är 136 ett Hoax-tal

Att göra:

Implementera följande funktioner. Det kan också vara bra att skapa andra hjälpfunktioner för att beräkna andra värden som behövs i testet ovan.

- Funktionen **prime(n)**, som tar ett heltal $n > 0$ och returnerar sant om och endast om detta är ett primtal.

  Kom ihåg att det minsta primtalet är 2. För att testa om ett tal `n` är ett primtal kan man t.ex. undersöka resten vid division med samtliga heltal från 2 upp till $\sqrt{n}$. Exempel: `prime(13)==True`.

- Funktionen **hoax(n)**, som tar ett heltal $n > 0$ och returnerar `True` om n är ett Hoax-tal, annars `False`.

Du kan få delpoäng även om enbart en av funktionerna är korrekt.

Exempel:

- `assert hoax(22) and hoax(94)`
- `assert not hoax(859)`
- `assert hoax(860)`

[1] Egentligen måste man också ange en talbas; i denna uppgift använder vi enbart bas 10.