Instruktioner för datortentamen 2020-01-17 (08:00)

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 `distribute(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.

Resultatlistan $\texttt{seq}_k$ ska innehålla vart `n`:te element från ursprungslistan, med början i element k (där indexering som vanligt börjar med element 0). Funktionen ska returnera en lista som innehåller alla resultatlistor i tur och ordning.

Exempel:

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

Konsekvenser av definitionen:

- Om den ursprungliga listans längd inte är jämnt delbar med `n` kommer de första resultatlistorna att innehålla fler element.
- 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 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.

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

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

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

Det finns många sätt att *komprimera* information så att den tar upp mindre plats. Många sådana metoder är ganska komplicerade, men för vissa typer av information kan enklare tekniker också ge goda resultat.

Tänk dig till exempel att du har en sekvens där du ofta får samma element många gånger i följd. Det kan till exempel vara en svartvit bild där du ofta har långa sekvenser av helt vita pixlar, följt av några sammanhängande svarta pixlar. Då kan du använda så kallad *run length encoding (RLE)*, på svenska *skurlängdskodning*.

Resultatet av skurlängdskodning är en lista med ett jämnt antal element, där elementen på jämna positioner (0, 2, 4, ...) anger ett element ur den ursprungliga sekvensen och ett element $n > 0$ på nästföljande position anger antalet gånger elementet upprepades.

Exempel:

- `rle("ABBBBCCCCCCCCCCAAA") == ['A', 1, 'B', 4, 'C', 10, 'A', 3]`
  (Den ursprungliga teckensekvensen bestod av 1 'A', 4 'B', 10 'C' och 3 'A'.)

Att göra: Skriv två implementationer av skurlängdskodning. Båda ska ta en godtycklig sekvens av element och returnera motsvarande skurlängdskodade lista enligt ovan.

- `rle_i(seq)` ska arbeta enligt **iterativ** lösningsmodell.
- `rle_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:

- Inputsekvensen `seq` kan ha godtycklig längd, inklusive 0, och kan innehålla godtyckliga element.
- 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 `rle` är antingen `rle_i` eller `rle_r`. Skapa gärna fler tester med inspiration av villkoren ovan!

- `assert rle([1,2,2,2,2,5,5,5,5,5,5,1,1,1]) == [1,1,2,4,5,6,1,3]`
- `assert rle(["a","a","b","a","c","c"]) == ["a",2,"b",1,"a",1,"c",2]`

Uppgift 3

Skriv en funktion `reverse_all` som tar en godtyckligt nästlad lista `seq` med godtyckliga element och returnerar en ny lista som har samma nästlade struktur, men där alla *listor* – på toppnivå och på nästlade nivåer – är reverserade (“baklänges").

Exempel:

- `reverse_all([1, 2, 3, [4, 5, ['x', 7]], 8]) ==`
  `[8, [[7, 'x'], 5, 4], 3, 2, 1]`

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 reverse_all([]) == []`
- `assert reverse_all([1, 2, 3]) == [3, 2, 1]`
- `assert reverse_all([-1, [2, 3], ['Hi', 4, [7]]]) == [[[7], 4, 'Hi'], [3, 2], -1]`

Uppgift 4

Deluppgift 4a (3p)

Om vi ska integrera en funktion $f(x)$ från a till b *numeriskt*, kan vi använda oss av följande approximation:

$$\int_a^b f(x)\,dx \quad \approx \quad (b-a)\cdot f\left(\frac{a+b}{2}\right).$$

Här använder vi oss av en rektangulär approximation, där vi beräknar funktionsvärdet $f((a+b)/2)$ i mitten av intervallet, och multiplicerar detta med bredden av intervallet, alltså $(b-a)$.

Skapa nu en högre ordningens funktion `integrate(f)` som tar en unär funktion `f` och returnerar en ny funktion med två argument, `a` och `b` enligt ovan. Denna nya funktion ska returnera ett approximativt värde på $\int_a^b f(x)\,dx$ enligt ovanstående formel.

Denna funktion testas indirekt av exempel i deluppgift 4b.

Ytterligare villkor:

- a och b kommer att vara tal (heltal eller flyttal).
- 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 `integrate` för att definiera en ny funktion `intsquare(a,b)`, som tar två numeriska parametrar `a` och `b` och returnerar ett approximativt värde på $\int_a^b 3x^2$ enligt ovanstående approximation.

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

Exempel:

- `assert intsquare(2, 4) == 54`
- `assert intsquare(-1, 10) == 668.25`

Extra information utskickad under tentans gång:

- Funktionen integrate() returnerar en ny funktion. Man ska anropa integrate() 1 gång för att skapa den nya funktionen intsquare(). Sedan ska man kunna anropa intsquare() hur många gånger som helst, med olika värden på a och b, utan att integrate() anropas igen. Man anropade ju bara integrate() för att *skapa* intsquare().
- Om du är osäker: Lägg en utskrift i integrate() så du ser exakt när den anropas. Prova sedan att anropa intsquare() många gånger.

Uppgift 5

Deluppgift 5a (4p)

Vi ska nu titta på *delsekvenser*. Sådana kan definieras på flera ekvivalenta sätt.

- Anta att du har en sekvens `seq`. Då är `subseq` en *delsekvens* av `seq` om och endast om samtliga element i `subseq` förekommer i `seq`, i samma ordning.

  Denna definition kan verka något tvetydig om man har flera element med samma värde, som i sekvenserna `(1,1)` och `(1,2,1)`. Se då nedanstående alternativ.

- Anta att du har en sekvens `seq`, och raderar elementen på 0 eller flera positioner ur denna sekvens. Resultatet är då en *delsekvens* till `seq`. (Detta är inte nödvändigtvis det bästa sättet att faktiskt skapa delsekvenser!)

Att göra: Implementera funktionen `subsequences(seq)`, som returnerar *mängden av alla delsekvenser till en godtycklig sekvens*.

Ytterligare villkor och information:

- För att förenkla implementationen behöver funktionen bara fungera när `seq` är en *tupel av heltal*, och även delsekvenserna representeras då som *tupler av heltal* (se exemplen nedan).
- Du får inte använda funktioner eller annan funktionalitet från `itertools` för att lösa denna uppgift.
- Använd onlinedokumentationen om du behöver friska upp minnet om t.ex. mängdoperationer.

Exempel:

- `assert subsequences((1, 1)) == {(1,), (1, 1), ()}`
- `assert subsequences((1, 2)) == {(1,), (1, 2), (2,), ()}`
- `assert subsequences((1, 2, 3)) == {(1, 3), (1, 2), (2,), (1, 2, 3), (2, 3), (1,), (), (3,)}`

Deluppgift 5b (1p)

Implementera funktionen `common_subsequences(seq1,seq2)` som hittar alla *gemensamma delsekvenser* mellan `seq1` och `seq2`. Med andra ord, funktionen ska hitta alla sekvenser som är delsekvenser till både `seq1` och `seq2`.

Exempel:

- `assert common_subsequences((1, 2, 3), (1, 2, 1)) == {(1,), (1, 2), (2,), ()}`
- `assert common_subsequences((1, 2, 1, 2), (1,)) == {(1,), ()}`

Uppgift 6

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

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

Exempel:

- 666 kan primtalsfaktoriseras till $2 \cdot 3 \cdot 3 \cdot 37$ där alla faktorer är primtal (och då det finns mer än en primtalsfaktor är talet alltså sammansatt)
- 666 har siffersumman $6 + 6 + 6 = 18$
- Primtalsfaktorerna $2 \cdot 3 \cdot 3 \cdot 37$ har siffersumman $2 + 3 + 3 + 3 + 7$ vilket också är 18
- Alltså är 666 ett Smith-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 **smith(n)**, som tar ett heltal $n > 0$ och returnerar `True` om n är ett Smith-tal, annars `False`.

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

Exempel:

- `assert smith(4) and smith(22)`
- `assert not smith(984)`
- `assert smith(985)`

Extra information utskickad under tentans gång:

- Det står att talet måste vara sammansatt, "inte ett primtal". Vi vill klargöra att talet 1 inte räknas som sammansatt: Sammansatta tal är *produkten* av två eller flera primtal, t.ex. 4=2*2.
- Mängd heter "set" på engelska. Om du funderar på ordning, testa om {1,2}=={2,1} eller inte.

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