Instruktioner för datortentamen 2019-08-20

TDDE24 Funktionell och imperativ programmering del 2

Hjälpmedel

Följande fysiska 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. (Lösa tomma papper tillhandahålls av tentamensvakterna.)

Du har också tillgång till `https://docs.python.org/3/library` via startmenyn.

Icke tillåtna hjälpmedel inkluderar bl.a. alla former av elektronisk utrustning samt böcker och anteckningar utöver det som listats ovan.

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.

Vid denna omtenta kommer poäng *inte* att sättas under tentans gång. Poäng sätts i efterhand.

Du kan även använda tentaklienten för att **ställa frågor**, som går antingen till examinatorn eller till jourhavande (som vid behov kan vidarebefordra). **Ställ alla tentarelaterade frågor via klienten.** 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

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 normalt **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 "lösningar" som uppfyller testfallen utan att vara korrekta. 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 `find_least_close(seq1, seq2)` som tar två sekvenser av heltal, där `seq1` har godtycklig längd medan `seq2` innehåller minst ett tal. Funktionen ska returnera en ny lista som för varje tal `x` i `seq1` innehåller det tal i `seq2` som är längst bort från `x`.

Exempel:

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

Ytterligare villkor:

- Om det finns både ett större och ett mindre tal som är "längst bort" ska du ta det största. Exempel: `find_least_close([10], [5, 8, 12, 15])` returnerar alltid `[15]`, trots att `5` är lika långt bort från `10`.
- Lösningen får vara iterativ eller rekursiv. Det är också tillåtet att använda både list-byggare (list comprehensions) och inbyggda funktioner som arbetar på hela listor.
- Dina funktioner får inte modifiera sina indata.

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

- `assert find_least_close([], [1]) == []`
- `assert find_least_close([10], [5, 8, 12, 15]) == [15]`
- `assert find_least_close([12, 10], [-1000, 5, 8, 12, 15]) == [-1000, -1000]`

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

I Python finns en funktion som heter `zip`, som tar ett godtyckligt antal sekvenser av godtyckliga element, och returnerar en ny lista med tupler, där varje tupel innehåller ett element från varje in-sekvens i tur och ordning[1]. Om sekvenserna är olika långa, ska listan bara innehålla så många tupler som det finns element i den kortaste sekvensen. Exempel:

`zip([1,2,3], [4,5], ['a', 'b', 'c', 'd'])` ger `[(1,4,'a'), (2,5,'b')]`

Att göra: Implementera en egen version av denna funktion, `zip3`, som tar *tre* sekvenser som indata och returnerar en lista av tupler enligt ovan. Funktionen ska finnas i två varianter:

- En som arbetar enligt **rekursiv** lösningsmodell: `zip3_r(s1,s2,s3)`
- En som arbetar enligt **iterativ** lösningsmodell: `zip3_i(s1,s2,s3)`

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

Ytterligare villkor:

- En inputsekvens kan ha godtycklig längd, inklusive 0, och kan innehålla godtyckliga element.
- Du ska inte använda den inbyggda funktionen `zip`
- 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. Däremot är givetvis `len()` tillåten.
- Dina funktioner får inte modifiera sina indata.

Båda funktionerna ska bl.a. klara följande **tester**, där `zip3_x` är antingen `zip3_i` eller `zip3_r`:

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

- `assert zip3_x([], [], [1]) == []`
- `assert zip3_x([1, 2, 3], [4, 5, 6], ['a', 'b']) == [(1, 4, 'a'), (2, 5, 'b')]`

[1] Egentligen tar funktionen ett antal *iterables* och returnerar en *iterator*, men det spelar ingen roll i den här uppgiften.

Uppgift 3

Skriv en funktion `squared_odds(seq)` som tar en godtyckligt nästlad lista `seq` och returnerar en ny lista som har samma nästlade struktur, men där alla udda heltal är "ersatta" med sina kvadrater.

Ytterligare villkor:

- Listor kan nästlas i godtyckligt antal nivåer, och som synes i exemplet ovan kan listan innehålla annat än bara heltal och andra nästlade listor.
- 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 squared_odds([]) == []`
- `assert squared_odds([1, 2, 3]) == [1, 2, 9]`
- `assert squared_odds([-1, [2, 3], ['Hi', 4, [7]]]) == [1, [2, 9], ['Hi', 4, [49]]]`

Uppgift 4

Deluppgift 4a (2p)

Skriv en högre ordningens funktion `each_pair(seq,fn_pair)` som applicerar en funktion `fn_pair(el1,el2)` av två argument på alla efterföljande par av element från en lista `seq`, och samlar ihop alla resultat i en ny lista som alltså blir ett element kortare än `seq`. Exempel:

- `assert each_pair([3, 7, 9, 15, 23, 27, 33], (lambda x, y: y-x)) == [4, 2, 6, 8, 4, 6]`
- `assert each_pair(['Hel', 'lo', 'op'], (lambda x, y: x+y)) == ['Hello', 'loop']`

Exemplet ovan använder alltså en lambdafunktion som räknar ut skillnaderna mellan de på varandra följande talen i listan. Som framgår av exemplet appliceras funktionen alltså på par av element som står omedelbart efter varandra: Först 3 och 7 (skillnad 4), därefter 7 och 9 (skillnad 2), 9 och 15, osv.

Ytterligare villkor:

- Ovan råkar funktionen appliceras på enkla listor av heltal, men den ska inte anta att detta alltid är fallet. Elementen kan ha godtycklig typ.
- Om `seq` innehåller högst 1 element finns ingen möjlighet att anropa `fn_pair`. Då ska `each_pair` istället returnera tomma listan.
- 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 (3p)

Ovanstående funktion "började" med en *tom lista*, och varje gång man skapade ett nytt element med `fn_pair`, *konkatenerades* detta element till listan. Detta kan vi generalisera för att kunna arbeta med ett godtyckligt startvärde `init` och en godtycklig funktion `fn_combine(current,new)` som kombinerar ihop ett tidigare resultat med ett nytt värde som just returnerades av `fn_pair`,

Implementera en ny högre ordningens funktion `combine_pairs(init,seq,fn_pair,fn_combine)` som åstadkommer detta.

Exempel:

- `assert combine_pairs([], [3, 7, 9, 15, 23, 27, 33], (lambda x, y: y-x), (lambda c,n: c + [n])) == [4, 2, 6, 8, 4, 6]`
  - Här börjar man alltså med tomma listan, och när värdet `4` har returnerats av `fn_pair()` är det den andra lambdafunktionen som kombinerar ihop dessa till `[4]`.
- `assert combine_pairs(True, [1, 2, 5, 4], (lambda x,y: x<y), (lambda a,b: a and b)) == False`
  - Här appliceras en jämförelse på varje efterföljande par. Funktionen som kombinerar ihop resultaten är i grund och botten operationen `and`, och som startvärde används `True`. Här testas alltså om listan är sorterad.
- `assert combine_pairs(True, [1, 2, 3, 4], (lambda x,y: x<y), (lambda a,b: a and b)) == True`
- `assert combine_pairs(False, [1, 2, 5, 4], (lambda x,y: x<y), (lambda a,b: a or b)) == True`
  - Här testas istället om det finns *något* par av element som är i korrekt ordning.

Ytterligare villkor:

- Om `seq` innehåller högst 1 element finns ingen möjlighet att anropa `fn_pair`. Då ska `combine_pairs()` istället returnera basvärdet `init`.
- 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.

Uppgift 5

Tänk dig att du ska svara på $n \geq 1$ olika flervalsfrågor, och att fråga nummer k har m_k olika svarsalternativ, där svaret är något av talen $1, 2, \ldots, m_k$. Vilka olika *kombinationer* av svar kan du då ge på hela sekvensen av flervalsfrågor?

Exempel: Du har $n = 3$ frågor. Första frågan har $m_1 = 4$ svarsalternativ (svaret på denna fråga måste måste alltså vara 1, 2 eller 3). Andra frågan har $m_2 = 2$ svarsalternativ (svar 1 eller 2). Tredje frågan har $m_3 = 2$ svarsalternativ (svar 1 eller 2). Då kan du ge följande 16 kombinationer av svar på de 3 frågorna, där vi representerar varje svarskombination som en tupel av svar:

- $(1,1,1)$, $(1,1,2)$, $(1,2,1)$, $(1,2,2)$, $(2,1,1)$, $(2,1,2)$, $(2,2,1)$, $(2,2,2)$, $(3,1,1)$, $(3,1,2)$, $(3,2,1)$, $(3,2,2)$, $(4,1,1)$, $(4,1,2)$, $(4,2,1)$, $(4,2,2)$

Att göra: Nu ska du implementera en funktion `combinations(seq)`. Indata till denna funktion ska vara en sekvens som innehåller antalet svarsalternativ för varje fråga, alltså en sekvens med talen m_1 till m_n. Utdata ska vara en *mängd* som innehåller alla motsvarande svarstupler enligt exemplet ovan.

För att vara mer exakt: Givet en tupel $(m_1, m_2, \ldots, m_n)$ med godtycklig längd $n \geq 1$, där alla $m_k \geq 1$, generera en mängd som innehåller alla tupler $(c_1, \ldots, c_n)$ med n heltal sådana att $\forall k \in [1, n] : 1 \leq c_i \leq m_i$.

Ytterligare villkor och information:

- Du får inte använda funktioner och iteratorer från itertools för att lösa denna uppgift.

Exempel:

- `assert combinations([1]) == {(1,)}`
- `assert combinations([2, 2]) == {(1, 2), (1, 1), (2, 1), (2, 2)}`
- `assert combinations([4, 2, 2]) == {(1, 2, 1), (2, 2, 2), (4, 2, 2), (3, 1, 1), (3, 2, 1), (1, 2, 2), (2, 2, 1), (2, 1, 1), (4, 2, 1), (1, 1, 2), (3, 2, 2), (4, 1, 1), (2, 1, 2), (1, 1, 1), (4, 1, 2), (3, 1, 2)}`

Uppgift 6

En datastruktur för en *kö* brukar ha funktionalitet för att lägga in nya element *sist* i kön och för att hämta ut respektive ta bort ett element som är *först* i kön – ingen får tränga sig!

Dock finns det också en variant av detta som kallas *dubbeländad kö* – engelska *double-ended queue*, förkortning *deque*. Där kan man lägga till och plocka bort element både längst fram och längst bak. Man kan fortfarande inte manipulera *mitten* av kön genom att hämta eller lägga till element där, vilket är en viktig skillnad jämfört med en generell lista av element.

Att göra: Skapa en konkret implementerad datatyp för dubbeländade köer, enligt den *abstrakta datatyp* som vi beskriver i den här uppgiften. Välj en lämplig intern representation för köerna och implementera nedanstående funktioner. Glöm inte att läsa "ytterligare villkor" innan du börjar implementera!

- `make_deque()`: Returnerar en ny tom kö.
- `length(deque)`: Returnerar längden av kön `deque`.
- `front(deque)`: Returnerar första elementet i kön `deque`, utan att ta bort elementet och utan att modifiera kön på andra sätt. Returnerar `None` om kön är tom.
- `back(deque)`: Returnerar sista elementet i kön `deque`, utan att ta bort elementet och utan att modifiera kön på andra sätt. Returnerar `None` om kön är tom.

Ibland vill vi kunna modifiera en existerande kö genom att lägga till eller ta bort element "(d)estruktivt" – alltså genom att modifiera den existerande datastrukturen. Det gör vi med följande 4 funktioner, som inte ska returnera något (vi skiljer på att titta på vilket element som är först/sist, enligt ovanstående funktioner, och att ta bort det första/sista elementet):

- `push_front_d(deque, elt)`: Lägg till `elt` först i kön `deque` destruktivt.
- `pop_front_d(deque)`: Ta bort första elementet i kön `deque` destruktivt.
- `push_back_d(deque, elt)`: Lägg till `elt` sist i kön `deque` destruktivt.
- `pop_back_d(deque)`: Ta bort sista elementet i kön `deque` destruktivt.

Ibland vill vi istället arbeta (f)unktionellt, vilket i det här fallet innebär att vi inte modifierar den existerande kön utan skapar en *ny* kö som innehåller fler element (för `push`) eller färre element (för `pop`). Den gamla kön ska alltså finnas kvar omodifierad efter ett av följande anrop:

- `push_front_f(deque, elt)`: Returnera en kö som motsvarar att `elt` har lagts till först i kön `deque`.
- `pop_front_f(deque)`: Returnera en kö som motsvarar att första elementet i kön `deque` har tagits bort.
- `push_back_f(deque, elt)`: Returnera en kö som motsvarar att `elt` har lagts till sist i kön `deque`
- `pop_back_f(deque)`: Returnera en kö som motsvarar att sista elementet i kön `deque` har tagits bort.

Ytterligare villkor:

- I denna uppgift behöver ni *inte* dokumentera samtliga funktioner med docstrings.
- Funktioner som ska ta bort ett element *ska* hantera en *tom kö* genom att helt enkelt inte ta bort något. Funktionerna får inte krascha.
- Vi har inte definierat en funktion `is_deque(x)`, och övriga funktioner får anta (behöver inte uttryckligen testa) att en `deque`-parameter faktiskt är en `deque`.
- Funktioner som arbetar funktionellt får inte modifiera den gamla existerande kön `deque`, men det är i denna uppgift OK att en funktion modifierar en *ny* kö som den håller på att skapa. Det är först när kön returneras, så att den syns för andra, som den inte längre får modifieras.
- Funktionernas implementation *får* gå direkt på den interna representationen när de manipulerar köer.
- Den första gruppen av funktioner ger 1p. De andra två grupperna av funktioner (destruktivt respektive funktionellt) ger 2p vardera.

Tester för destruktiv version:

```
q = make_deque()
push_front_d(q, 15)
assert length(q) == 1
push_back_d(q, 22)
assert length(q) == 2
assert front(q) == 15
assert back(q) == 22
pop_front_d(q)
assert front(q) == 22
pop_back_d(q)
assert length(q) == 0
```

Tester för funktionell version:

```
q = make_deque()
assert length(q) == 0
q1 = push_front_f(q, 15)
assert length(q) == 0
assert length(q1) == 1
q2 = push_back_f(q1, 22)
assert length(q) == 0
assert length(q1) == 1
assert length(q2) == 2
assert front(q2) == 15
assert back(q2) == 22
# Fortsätt testa pop-funktionerna!
```