Datortentamen 2020-08-18

TDDE24 Funktionell och imperativ programmering del 2

Distanstentamen: Flera skillnader

Detta är vår första distanstentamen i TDDE24. Detta leder till en del skillnader i utförandet. Läs genom instruktionerna noga för att förstå hur tentan ska genomföras.

Distanstentamen: Tillåtna och otillåtna hjälpmedel

- Användning av resurser på Internet, inklusive kursens websidor, är uttryckligen *tillåten*. Som tidigare kan websidor under `https://docs.python.org/3/` vara användbara. Detta inkluderar både biblioteks- och språkreferenser.
- Man får trots detta **inte kopiera text eller lösningar direkt från andra källor.** Man måste skriva koden på egen hand och förstå vad den gör.
- Man får **inte kommunicera med andra under tentan**, vare sig för att diskutera uppgifterna eller för andra syften – precis som under en vanlig tenta. Tentan genomförs alltså på egen hand, isolerat. Det gäller både kurskamrater och andra som kan finnas tillgängliga hemma, på telefon, på nätet, eller liknande.
- Man får **inte göra tentasvar eller relaterad information tillgängliga för andra** på något sätt under tentans gång, fram till **16:00** (för att ge marginaler då vissa har förlängd tid). Alla uppgifter ska genomföras helt individuellt.
- Vi påminner om att vi är skyldiga att **anmäla** möjligt fusk till disciplinnämnden, utan att själva försöka reda ut om det faktiskt var fusk.

Distanstentamen: Grundläggande skillnader i uppgifter

Tentauppgifternas natur och stil måste till viss del ändras för att fungera bättre i en distanstenta med tillgång till Internet. Var därför extra noga med att läsa genom alla delar av varje enskild uppgift, och tänk inte att allt ska fungera som tidigare!

- Då vi inte följer tidigare tentamensstruktur försöker vi inte heller ordna uppgifterna i svårighetsordning.
- Vi kan inte i förväg ange några poänggränser. Det kan man bara göra när man har stor erfarenhet av hur poängen motsvarar de faktiska kunskaperna som vi ska examinera. Examinatorns uppgift är alltid att bedöma uppfyllande av kursmål, och poänggränser är bara en av många olika metoder att göra detta.
- För att vi ska se hur du har testat ska du **lämna in alla testfall som du hittar på!** Detta gäller även eventuella testfall som misslyckas. Testfall ska använda `assert`.

Distanstentamen: Datormiljö

I och med distanstentan behöver du själv ha tillgång till en datormiljö där du kan arbeta med dina uppgifter. **Vi kan inte hjälpa till med att sätta upp den miljön.**

- Du ska använda **Python 3**, inte Python 2! Vi har undervisat i Python 3 i kursen.

  Detta kan finnas som kommandot `python3` på din dator. Det kan också finnas som kommandot `python` – testa då med `python --version` så att det verkligen är en version av Python 3.

  Du är själv ansvarig för att installera Python 3 om det inte redan finns på din dator. För Windows kan du antagligen använda “installer"-filerna på slutet av `https://www.python.org/downloads/release/python-385/`. Troligen vill du ha en x86-64-baserad installerare då du antagligen har en 64-bitars CPU.

  Mer detaljer: Vi kommer att testa koden under **Python 3.8**, så det är OK att använda nya features upp till och med denna version. Det går också bra att använda Python 3.7, som vi har använt under kursens gång, eller version 3.6. Använder du ännu tidigare versioner av Python 3 ska det förhoppningsvis också fungera, men skriv då en kommentar om din exakta version av Python (enligt `--version`) i varje inlämnad fil.

- Du får använda vilka utvecklingsmiljöer eller editorer du vill, men är själv ansvarig för att de fungerar.

- Det är *möjligt* att Thinlinc eller tjänster som `ssh.edu.liu.se` är tillgängliga under tentan. I så fall är det tillåtet att använda dem. Det är också möjligt att dessa tjänster är otillgängliga, eller att de finns tillgängliga till en början men går ner under tentans gång, så att du inte längre kan komma åt ditt arbete. Detta är tyvärr inget som vi i kursledningen kan göra något åt: LiU:s policy är att tentorna *ska* genomföras på distans och att man är ansvarig för sin egen datormiljö.

Är du osäker på vilken version av Python som används “internt" av en editor eller utvecklingsmiljö: `assert 5/2==2.5` fungerar i Python 3, men ger fel i Python 2. Lägger du detta först i varje källkodsfil ser du vid testkörningen om miljön använder rätt version.

Frågor och information under tentans gång

Tentarelaterade frågor skickas via *epost* direkt till `jonas.kvarnstrom@liu.se`.

Spara inte frågorna till slutet. På en vanlig tenta går man genom alla uppgifter så snart man får tillgång till dem, så man kan ställa eventuella frågor till examinatorn när han/hon besöker tentasalen. Under denna distanstenta kan du visserligen skicka frågor när som helst, men även här kommer frågorna bara att besvaras “då och då" – dels går det inte att spendera varje sekund klistrad framför epostklienten, dels kan det komma många frågor på samma gång. Skickar du frågor under sista halvtimmen kan det hända att du inte hinner få svar i tid för att avsluta din lösning innan tentan avslutas!

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.

Information som är intressant för flera tentander kan skickas ut via en epostlista under tentans gång. Håll alltså koll på din egen epost under tentan.

Inlämning av svar

Inlämning sker via Lisam. Vi använder inte “quiz"-systemet med en separat fråga per uppgift i tentan, utan “inlämningssystemet" där man laddar ner tentan i PDF-format och skriver *ett* gemensamt svar. Själva textsvaret kan vara tomt om man vill. Det viktiga är möjligheten att **bifoga filer**, vilket man ska använda för att bifoga *en separat fil per uppgift* (ex1.py, ex2.py, ...).

Enligt manualen för Lisam ska det gå att lämna in ett svar flera gånger, fram till tentatiden är slut. Utnyttja det gärna för att gradvis uppdatera dina svar med nya lösningar, ifall något t.ex. skulle gå fel med din uppkoppling till Internet mot slutet av tentatiden.

För alla som har tenta *den vanliga tiden 08–13* finns en slutlig hård deadline 13:10 då ditt slutliga svar ska vara inlämnat. Detta är enbart till för att du inte ska missa inlämningen på grund av att klockan går fel med 1 minut eller för att uppkopplingen till Internet fick kortvarig hicka. Använd inte denna utökade tid till att arbeta vidare – om du då får problem med inlämningssystemet har du inga marginaler kvar.

De som har *utökad skrivtid* till 15:00 får även lämna in lösningar via epost till adressen `jonas.kvarnstrom@liu.se`. Lösningarna ska skickas innan 15:10. Epost ska skickas från din universitetsadress.

Om du har tekniska problem relaterade till Lisam eller andra system på universitetet, kontakta då helpdesk på `helpdesk@liu.se` eller 013-285898. Det finns inget som vi i kursledningen kan göra i fråga om teknisk support!

Precis som vid vanliga tentor kommer alla svar att granskas *efter* tentans slut.

Namnge filerna ex1.py, ex2.py, ex3.py, ...!
Detta underlättar rättningen.

Viktiga tips – missa inte poäng i onödan

Utöver rättningskriterierna (nästa sida) vill vi starkt uppmana er att tänka på detta:

- Vi ger ofta flera testfall i varje fråga, både i `assert`-form och i den löpande texten. **Testa dem alla!** Ofta hittar man fel som faktiskt är lätta att korrigera.
- De testfall vi ger är bara exempel och täcker definitivt inte allt – man måste också utgå från beskrivningen i texten. Därför ska man **skapa egna tester** som täcker fler fall – se åtminstone efter vad resultatet blir om du testar din implementation med annan input än den givna!
- Tänk på att **testa specialfall** som *tomma listor*, *tomma tupler*, *negativa tal*, med mera. Testa också *långa* listor eller *djupt nästlade* listor.
- Var noga med att **läsa exakt vad som står**! En *godtycklig lista* är precis vilken lista som helst. En *sekvens* behöver inte nödvändigtvis vara en lista (eller en tupel). Ska det fungera för heltal $n \geq 0$ ska det också fungera för $n = 0$. Ska funktionen ta en sekvens och returnera en lista får den inte returnera en tupel, även om inputsekvensen råkade vara en tupel.
- Tänk på att man ofta ska **klara godtycklig input**. Även om de angivna testfallen bara är heltal, kanske det står att man ska klara godtyckliga listor, och alltså vilka element som helst. **Testa då detta!** Fastna inte heller i att ett exempel bara råkar ange *positiva* tal, eller att en lista råkar vara *sorterad*. Återigen, läs vad uppgiften ska klara och skapa egna exempel som testar varierande input.

Testning är extremt viktig!

Vi ser ofta *många* fel som väldigt enkelt kunde ha upptäckts och fixats om man bara *testade* sin lösning lite mer. Vad är bäst, att få 4 poäng på 4 uppgifter (16 totalt) eller att hinna med en uppgift till men få 2 poäng per uppgift på grund av bristande testning (10 totalt)?

Rättningskriterier

Brott mot följande allmänna kriterier kan resultera i 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).
- Man ska kunna köra funktioner **flera gånger** med korrekt resultat, utan att ladda om koden däremellan. Se upp med olika former av globalt tillstånd / globala variabler.
- 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 givetvis följa de **specifika regler och villkor** som uppgiften har satt upp. Lösningen ska också **fungera exakt som i körexemplen** i uppgiften, om inte texten indikerar något annat.
- Lösningen ska vara **generell**: Den ska inte bara fungera för de körexempel som anges, utan för alla indata som följer uppställningen i frågan. Att en lösning enbart fungerar för listor med begränsad längd eller för vissa storlekar på indata är ett signifikant fel.
- 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 i tentatexten visar normalt returnerade värden, inte värden som funktionerna har skrivit ut.
- Man får använda Pythons standardbibliotek, men **inte bibliotek som måste installeras separat**.

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 som granskaren ska arbeta vidare med.

Lösningar som misslyckas alltför ofta räknas normalt inte som lösningar och får 0 poäng. Det går ofta att få delpoäng för lösningar som misslyckas med vissa specifika fall, men inte för lösningar som bara lyckas med vissa specifika fall.

Tillåtet / icke-krav

Vi får ofta frågor om vad som är tillåtet i en lösning. Om inget annat är specifikt angivet i en uppgift, är följande uttryckligen *tillåtet*:

- Att lösa uppgiften rekursivt eller iterativt, eller med en kombination av dessa lösningsmodeller.
- Att använda vanliga “inbyggda" funktioner från Pythons standardbibliotek, t.ex. matematiska funktioner från `math`.
- Att använda “inbyggda" funktioner som själva arbetar på alla element i en lista, sekvens eller iterator, t.ex. `map()`.
- Att använda listbyggare (list comprehensions) och generatoruttryck (generator expressions).
- Att skapa hjälpfunktioner, nästlade eller icke nästlade. Lösningen i sin helhet måste så klart fortfarande följa den tänkta lösningsmodellen, om en sådan är angiven.
- Att addera defaultargument till funktioner, så länge som funktionerna fortfarande går att anropa på sådant sätt som visas i uppgiften. Lösningen i sin helhet måste så klart fortfarande följa den tänkta lösningsmodellen, om en sådan är angiven. Defaultargument kan ibland användas på sätt som bryter mot en rekursiv lösningsmodell.
- Att anta att indata följer specifikationen i uppgiften, utan några egna felkontroller. Står det t.ex. att funktionen ska ta en sekvens, behöver man inte själv kontrollera att man faktiskt får en sekvens som parameter (om inte uppgiften särskilt anger detta).

  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).
- Att använda godtyckligt långa rader. Vi ger inga avdrag för radlängder som bryter mot standarden.

Detta gäller inte om uppgiften säger något annat!

Uppgift 1

Skapa en funktion `cut_sequence_at(seq, pred)` som tar en godtycklig sekvens *seq* och konverterar den till en lista av listor. Den ursprungliga listan ska delas vid varje element som uppfyller predikatsfunktionen `pred`, och detta element ska då tillhöra nästa underlista.

Funktionen ska arbeta iterativt över sekvensen `seq`.

(En predikatsfunktion är en funktion av 1 argument, som returnerar sant eller falskt beroende på om ett villkor, ett predikat, är uppfyllt.)

Skapa även en andra funktion, `cut_sequence_recursively(seq, pred)`, som definieras på samma sätt men använder en 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)`: Här är `fac(n-1)` ett rekursivt anrop som *också* beräknar just fakultetsfunktionen, men för ett mindre värde på `n`. Delproblemet har alltså exakt samma definition som det ursprungliga problemet, men ges andra parametrar.

Exempel:

- `assert cut_sequence_at([], lambda x: x % 2 != 0) == [[]]`
- `assert cut_sequence_at([1,2,3,4], lambda x: x % 2 != 0) == [[1,2],[3,4]]`
- `assert cut_sequence_at([4,2,3,4,8,12,5,2,1], lambda x: x % 2 != 0) == [[4,2], [3,4,8,12], [5,2], [1]]`

Ytterligare villkor:

- Dina funktioner får **inte** modifiera sina indata.
- Du får **inte** använda listbyggare (*list comprehensions*), generatoruttryck (*generator expressions*, som vi inte har diskuterat så mycket i kursen) eller inbyggda funktioner som behandlar alla element i hela sekvenser, utan måste själv iterera eller rekursera över sekvenserna, *ett element i taget*.

  Funktioner som inte behandlar de enskilda elementen, såsom `len()`, är tillåtna.

Mer information på nästa sida!

Sista påminnelsen!

- Dokumentera funktioner med docstrings!
- Skapa många egna tester – prova med många olika indata!
- **Inkludera dina testfall, med användning av 'assert', i din inlämnade fil – även testfall som misslyckas! Detta kommer också att kunna påverka bedömningen av din lösning.**
- Gör aldrig någon ändring utan att testa, även när du *tror* att ändringen var harmlös. Testa omedelbart före inlämningen.

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

Uppgift 2

Skapa en funktion `nested_median(nest)` som räknar ut medianen av de heltal (`int`) som förekommer i en godtycklig nästlad lista `nest`.

Listan `nest` kan alltså innehålla andra listor, som då också ska behandlas. Den kan också innehålla heltal, vars median ska beräknas. Slutligen kan den innehålla element som varken är listor eller heltal. Dessa element ska ignoreras i beräkningen av medianen.

Du får förutsätta att det finns minst 1 heltal någonstans, på någon nivå, i den nästlade listan `nest`.

Vad är medianen?

Wikipedia säger:

> Antag en ordnad följd av n värden. Medianen är det mittersta värdet om n är udda. Om n är jämnt är medianen medelvärdet av de mittersta värdena, det vill säga om a < b < c < d är medianen medelvärdet av b och c.
>
> Om exempelvis n är 7 är medianen värdet med positionen 4 och om n är 8 är medianen medelvärdet av talen med positionerna 4 och 5.

Exempel:

- `nested_median([1,1,1,1,2]) == 1`
- `nested_median([1,2,3,4,5]) == 3`
- `nested_median([3,2,1,5,5]) == 3`
- `nested_median([3,2,[1,5],5]) == 3`
- `nested_median([[3,2,[1,5],5]]) == 3`
- `nested_median([3,2,"Hello",[1,5],("Hello", "world"),5,9.3]) == 3`

Ytterligare villkor:

- Dina funktioner får **inte** modifiera sina indata.
- Då en huvudpoäng med denna uppgift är att hantera nästlade listor, ges inga poäng för lösningar som bara fungerar med icke-nästlade listor.

Uppgift 3

Om uppgiften verkar svår: Man kan få delpoäng för testfunktionen, se "Testning" (nästa sida)!

Vi ska nu behandla en lista `seq` av heltal, där vi börjar på index `pos==0` (första elementet) och vill komma till index `pos==len(seq)-1` (sista elementet). Vi ska förflytta oss i 0 eller flera "steg", och i varje sådant steg får vi fritt välja mellan tre nya index att förflytta oss till:

- Vi kan välja att flytta till index `pos + 1`, om detta index existerar i listan
- Vi kan välja att flytta till index `pos + 2`, om detta index existerar i listan
- Vi kan välja att flytta till index `pos + seq[pos]`, om detta index existerar i listan

Om vår lista är `seq=[5,3,1,9,12,1,1,1,1,1,1,1,1,1,1,1,1]` börjar vi alltså på index 0 och kan i nästa steg gå till index `0+1==1`, index `0+2==2`, eller (då vi just nu pekar på ett element med värdet 5) index `0+5==5`.

Din uppgift är att skriva en funktion `min_jumps(seq)` som tar en godtycklig lista `seq` av $n \geq 0$ strikt positiva heltal $k \geq 1$ och beräknar en sekvens av sådana "hopp" som flyttar indexet till ett läge där `pos==len(seq)-1`, så att indexet pekar ut det allra sista elementet. Som synes i exemplen nedan representeras en hoppsekvens som en tupel av hopp, där varje hopp representeras som en sträng:

- `'+1'` representerar att man flyttar till index `pos + 1`
- `'+2'` representerar att man flyttar till index `pos + 2`
- `'+k'` representerar att man flyttar till index `pos + seq[pos]`

Utöver att hoppsekvensen ska vara korrekt ska den också ha *minimal längd*, alltså innehålla ett minimalt antal hopp. Så länge listan inte är tom kan man ju alltid trivialt nå till sista elementet genom att hoppa 1 eller 2 steg framåt i taget (så en lösning som alltid gör på det sättet ger inga poäng), men ofta kan man få färre hopp genom att använda sig av den tredje möjligheten.

Exempel:

- `min_jumps([]) == None`, för att indikera att ingen lösning finns – det finns inget element, så man kan inte peka på det sista elementet.
- `min_jumps([7]) == ()`, då positionen redan pekar ut det sista elementet.
- `min_jumps([5,3,1,9,12,1,1,1,1,1,1,1,1,1,1,1,1])` ska returnera ett av de två alternativen `('+1', '+k', '+k')` eller `('+2', '+2', '+k')`, eftersom båda alternativen ger 3 hopp vilket är det minsta möjliga antalet. Första alternativet ger 1 steg framåt, 3 steg framåt, 12 steg framåt. Andra alternativet ger 2 steg framåt, 2 steg framåt, 12 steg framåt.
- `min_jumps([5,3,1,9,12,1,1,1,1,1,1,1,1,1,1,1,1])` kan inte längre hoppa 12 steg, då man i så fall skulle gå förbi slutet av sekvensen. Istället finns ett stort antal korrekta lösningar av längd 5, till exempel tupeln `('+1', '+2', '+k', '+1', '+2')`.

Fortsättning på nästa sida!

Tips:

Tänk på att om du i exemplet `seq=[5,3,1,9,12,1,1,1,1,1,1,1,1,1,1,1,1]` börjar med att hoppa framåt 5 steg, missar du möjligheten att komma till elementet med index 4, där du kunde ha hoppat framåt 12 steg. Det är alltså inte uppenbart från värdet på ett visst element om du ska använda det för att hoppa många steg framåt eller om du totalt sett får färre hopp om du just nu bara stegar 1 eller 2 steg framåt. Därför måste alla fallen testas.

Ytterligare villkor:

- Dina funktioner får **inte** modifiera sina indata.

Testning (kan ge delpoäng):

Eftersom en given sekvens ofta har *flera* korrekta lösningar fungerar inte den vanliga testmetoden, att man anger *exakt* vilket värde som måste returneras. Det kan också vara jobbigt att räkna upp precis alla tänkbara returvärden.

I sådana fall kan man ibland testa en lösning genom att *applicera* den och se om den fungerar. Du ska därför skriva en egen testfunktion `test_jumps(seq,solution)` som applicerar en lösningstupel på en heltalssekvens och testar om lösningstupeln faktiskt resulterar i att man hamnar på det sista elementet. Det vill säga, för lösningstupeln `('+1', '+2')` ska man börja på position och därefter hoppa 1 steg framåt (+1) följt av 2 steg framåt (+2) och se om man därefter är på korrekt position.

Du kan då *testa testfunktionen* – kontrollera om `test_jumps` själv fungerar korrekt – på följande sätt. Testerna nedan ska passera, och du kan även hitta på egna enkla tester.

- `assert test_jumps([5,3,1,9,12,1,1,1,1,1,1,1,1,1,1,1,1], ('+1', '+k', '+k'))`
- `assert test_jumps([5,3,1,9,12,1,1,1,1,1,1,1,1,1,1,1,1], ('+2', '+2', '+k'))`
- `assert not test_jumps([5,3,1,9,12,1,1,1,1,1,1,1,1,1,1,1,1], ('+2', ))`

När du är säker på att `test_jumps` fungerar som den ska, kan du använda den för att testa de resultat du får från `min_jumps`, med tester i följande stil:

- `myseq = [5,3,1,9,12,1,1,1,1,1,1,1,1,1,1,1,1]`
- `assert test_jumps(myseq, min_jumps(myseq))`

Detta testar att resultatet från `min_jumps` är en lösning på “hopp-problemet" för den givna sekvensen. Det testar *inte* att det är en *minimal* lösning – kanske det finns en kortare – men det är ändå en viktig del av en fullständig test.

Glöm inte att bifoga dina testfall, både de som testar att `test_jumps` fungerar korrekt och de som använder `test_jumps` för att testa att `min_jumps` fungerar korrekt!

Eftersom denna typ av test inte kontrollerar att lösningarna är minimala, är det bra att OCKSÅ försöka kontrollera minimaliteten på annat sätt, t.ex. genom att själv skapa enkla testfall där storleken på en minimal lösning är uppenbar.

Uppgift 4

I denna uppgift ska du skapa en datatyp som på en abstrakt nivå motsvarar en lista av $n \geq 0$ heltal $k \geq 2$ – men som internt lagrar detta som en lista av dictionaries, där varje dictionary representerar heltalets primtalsfaktorer.

Primtalsfaktorer: Ett heltal kan alltid representeras som en multiplikation av primtal. Exempelvis vet vi att $10 == 2 * 5$, där 2 och 5 är primtal. På samma sätt har vi att $40 == 2 * 2 * 2 * 5$, där alla de fyra faktorerna är primtal.

En sådan primtalsfaktorisering av ett tal ska internt representeras som en dictionary där 40 representeras som `{2: 3, 5: 1}`. Med andra ord, 40 har 3 faktorer med värdet 2, och 1 faktor med värdet 5, alltså $2 * 2 * 2 * 5$.

Funktioner att skapa:

- `make_int_list()` ska returnera en tom heltalslista. Denna ska representeras som en tom lista.
- `add_int(int_list,num)` ska lägga till ett heltal `num` ≥ 2 i slutet av `int_list`. Internt i `int_list` ska detta heltal alltså representeras som en dictionary som representerar primtalsfaktorerna för `num` enligt ovan. Se även exempel under “Tester av den interna strukturen".
- `get_int(int_list,pos)` ska returnera det *heltal* som lagrades på en viss position `pos` i listan `int_list` genom att återskapa heltalet från den interna lagringsformen (som alltså var en dictionary).
- `square_all(int_list)` ska modifiera alla element i `int_list` genom att kvadrera dem, alltså ersätta representationen av heltalet n med en representation av heltalet n^2. Den interna representationen ska fortfarande vara en dictionary motsvarande en primtalsfaktorisering.
- Skapa gärna hjälpfunktioner för primtalsfaktoriseringen.

Som synes ovan kan man inte komma åt primtalsfaktoriseringen av ett visst heltal genom att använda de funktioner vi själva har definierat på datatypen. När man adderar ett heltal till listan lagras det som talets primtalsfaktorer, men när man hämtar ut ett värde på en viss position kommer `get_int()` att återskapa det ursprungliga heltalet utifrån primtalsfaktorerna.

Däremot kan man givetvis titta på listans interna struktur med *andra* funktioner än de vi har definierat ovan, i och med att Python inte tillåter oss att “gömma" den interna strukturen. Till exempel kan vi titta på en dictionary genom att plocka ut `int_list[num]` istället för att anropa `get_int(int_list, num)`. Detta kommer att användas av testerna för att verifiera att implementationen följer den tänkta interna lagringen.

Fortsättning på nästa sida

Exempel:

Tester av det externa beteendet hos datatypen:

- `mylist = make_int_list()`
- `add_int(mylist, 40)`
- `add_int(mylist, 10)`
- `assert get_int(mylist, 0) == 40`
- `assert get_int(mylist, 1) == 10`

Tester av den interna strukturen:

- `assert mylist == [{2: 3, 5: 1}, {2: 1, 5: 1}]`

Fler tester av det externa beteendet:

- `square_all(mylist)`
- `assert get_int(mylist, 0) == 1600`
- `assert get_int(mylist, 1) == 100`

Ytterligare villkor:

- I denna uppgift får givetvis `add_int` modifiera sina indata. Övriga funktioner ska inte modifiera sina indata.
- Det är OK om `get_int` kraschar när en felaktig `pos` anges.