Datortentamen 2023-03-15

TDDE24 Funktionell och imperativ programmering del 2

Allmän information

Tentan består av totalt **6 uppgifter** som vardera kan ge maximalt 5 poäng.

Vad som är lätt eller svårt skiljer sig från person till person. Därför bör man inte förutsätta att de första uppgifterna är enklast, utan snabbt läsa genom alla uppgifter för att kunna prioritera arbetet. Vissa uppgifter kan också ge delpoäng för att man löser specifika delar, så om en uppgift verkar svår i sin helhet kanske du ändå vill arbeta med en del av den!

Viktiga ändringar och förtydliganden sedan förra tentan markeras på detta sätt, i blått.

Tillåtna hjälpmedel och verktyg

Följande hjälpmedel och verktyg är tillåtna / tillgängliga:

- **Penna** för att kunna skissa lösningar på papper. (Lösa tomma papper ska finnas tillgängliga från tentavakterna.)
- **Python 3.9**, som vi har använt under senaste kursomgången. Detta finns som kommandot python3.9 på tentadatorerna.

  I terminalfönstret *bör* det också gå att använda Python 3.9 genom att skriva python eller python3, men dessa "alias" fungerar inte nödvändigtvis om du kör program inuti en editor som vscode. Om du får fel version av Python inuti en editor har våra nya svarsmallar en testfunktion som ska tala om det automatiskt vid testkörningen, och det är då upp till dig själv att lösa problemet: Peka ut korrekt version av Python för editorn, eller kör helt enkelt allt med python3.9 från kommandoraden.
- **Ett antal editorer**, inklusive vscode (kommandot `code` på kommandoraden) och gvim. Fler editorer kan finnas tillgängliga från menyer eller från kommandoradsprompten.

  Editorerna har inte alla plugins installerade. Meningen är att man ska få tillgång till de grundläggande menyer och tangentbordsbindningar som man är van vid, inte att man ska ha en fullständig integrerad utvecklingsmiljö. Exempelvis kan man behöva köra program från kommandoraden, och verktyg som debuggers kanske inte fungerar.
- Standardiserade **systemprogram** i kommandoradsprompten eller menyerna.
- **Websidor** under `https://docs.python.org/3.9/` (eventuellt via en "Web Access"-ikon; vi som skapar tentan kan inte själva se tentamiljön). Här finns både biblioteks- och språkreferenser.

Otillåtna hjälpmedel; fusk och vilseledande

Otillåtna hjälpmedel inkluderar alla former av elektronisk utrustning utöver tentadatorerna, samt böcker och anteckningar.

Under datortentan arbetar du i en begränsad och övervakad datormiljö där utvecklingsmiljöer som PyCharm och Thonny inte är tillåtna, och där du har begränsad tillgång till nätet.

Utöver detta gäller följande:

- Man får **inte kopiera text eller lösningar direkt från andra källor.** Man måste skriva koden på egen hand och förstå och beskriva vad den gör.
- Man får **inte kommunicera med andra under tentan**, vare sig för att diskutera uppgifterna eller för andra syften (utom för att ställa frågor till tentavakter, så klart).
- Man får **inte göra tentasvar eller relaterad information tillgängliga för andra** på något sätt under tentans gång. Alla uppgifter ska genomföras helt individuellt.

Vi påminner om att vi är **skyldiga att anmäla** möjligt fusk eller "försök till vilseledande" till disciplinnämnden, utan att själva försöka reda ut om det faktiskt var fusk.

Svarsmallar

För varje uppgift i tentan *skickar vi med* en “svarsmall", med filnamn från ex1.py till ex6.py. **Mallen ska alltid användas som grund till din inlämnade uppgift** och du ska varken döpa om den eller skapa egna filer som lämnas in. **Kopiera** mallfilen från den skrivskyddade katalogen `given_files` till den katalog där du arbetar med uppgifterna, t.ex. `Desktop`. Skriv din kod överst i filen och dina eventuella tester inuti run_tests() – då kan vi enklare testa utan att köra dina egna tester. Provkör med t.ex. “`python3.9 ex1.py`" från kommandoraden.

Mallen hjälper till med detta:

- Testar att man kör rätt version av Python, så att man inte får underliga buggar på grund av fel Pythonversion. Talar annars om vilken version som används.
- Inkluderar “tomma" funktionsdeklarationer för de funktioner du behöver skriva, så du slipper klippa och klistra och inte riskerar att skriva fel. Funktionerna består bara av kommandot `pass`, som du då ska *ersätta* med din egen kod för funktionen. Se upp så du inte får dubbla deklarationer, både vår medskickade och en egen!

  Du får givetvis skapa egna hjälpfunktioner som tillägg.
- Inkluderar eventuella assert-testfall som vi redan har angivit i uppgiften, så du slipper klippa och klistra. (Du behöver ändå tänka på att skapa egna tester, för din egen skull.)
- Ser till att testerna bara körs om man *kör* filen, inte om man *importerar* den (så som vi gör vid rättningen). Det är därför även dina egna tester ska vara inuti run_tests().

Mallen kan se ut *ungefär* så här:

```
# Här i början lägger du din egen kod för uppgift 1.

def the_function_you_should_write(seq: list):
    pass

def check_python_version():
    ...färdig kod som kollar att du kör rätt version av Python...

def run_tests():
    # De här testerna står uttryckligen som assertions på tentan.
    print("Kör uppgiftens tester...")
    assert the_function_you_should_write(10) == 42

    # Här lägger du dina egna tester.  Du kan till exempel skapa egna
    # assertions, eller lägga till andra tester såsom enkla utskrifter
    # av resultatet av en körning.
    print("Kör egna tester...")

    # Här kan du lägga tester där du inte vet korrekta svar men
    # ändå kan skriva ut resultatet.  Kanske det kraschar, kanske
    # det är uppenbart fel...
    print("Kör utskriftstester...")
    print("Resultat 1:", the_function_you_should_write(4711))

    print("Har kört alla tester")

if __name__ == '__main__':
    check_python_version()
    run_tests()
```

Under tentan: Lämna in uppgifter

Med hjälp av **tentaklienten** (som du har fått information om i förväg och som även beskrivs i en fil som ska finnas tillgänglig på tentadatorn) skickar du in dina svar, med en separat inlämning per uppgift. Alla uppgifter måste lämnas in innan tentans slut, då tentasystemet automatiskt stängs!

När du gör en inlämning i tentaklienten anger du också *vilken* uppgift du lämnar in (nummer 1–6). I uppdelade uppgifter lämnas del (a) och del (b) in på samma gång, i samma fil. Var försiktig så att du lämnar in rätt uppgiftsnummer!

Det går bra att skicka in en lösning på samma uppgift flera gånger, och den nya inlämningen ersätter då alltid den tidigare. Utnyttja det: **Testa att lämna in någon lösning tidigt**, så du garanterat vet hur det fungerar, och **riskera inte att missa sluttid / deadline**, utan lämna in din nuvarande minst 5 minuter före sluttiden även om du tror du kan vilja utnyttja resten av tiden för att polera svaret mera.

Det finns en meddelandelogg i klienten, men såvitt vi vet kan du inte själv se exakt vilka filer du har bifogat. Om du är osäker på vad du har skickat in kan du skicka in rätt fil en gång till.

Precis som vid vanliga tentor kommer inga svar att granskas förrän efter tentans slut.

Under tentan: Ställa frågor

Om du har **tekniska problem** (kan inte logga in, har problem med tangentbordet, kan inte skicka in svaret på en fråga ...) kontaktar du tentavakterna som vid behov kan kontakta teknisk jour.

Frågor om tentauppgifterna ställs istället via tentaklienten. Frågorna går då till examinatorn. Tentavakter och teknisk jour ska *inte* svara på tentafrågor.

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 alla eventuella frågor till examinatorn vid ett eller två korta besök i tentasalen. Under denna datortenta kan du visserligen skicka frågor när som helst, men även här kommer frågorna att besvaras “då och då" och **det kan ta någon timme att få svar** – dels går det inte att spendera varje sekund klistrad framför tentaklienten, dels kan det komma många frågor på samma gång.

Ofta kommer många frågor på slutet, så skickar du frågor för sent kan det hända att du inte hinner få svar i tid för att avsluta din lösning innan tentans slut!

Läs alltså genom uppgifterna i början och ställ frågor i god tid!

Under tentan: Informationsutskick

Information som är intressant för flera tentander kan skickas ut via tentaklienten under tentans gång. Detta brukar oftast handla om påminnelser om sådant som redan står i instruktionerna, men som några studenter har missat eller missförstått.

Utskick och svar på frågor syns bara i en meddelandelogg i tentaklienten. Det kommer inga notifikationer eller popup-meddelanden.

Håll alltså koll på tentaklienten och dess meddelandesystem under tentan. Klienten ger INTE automatiska notifikationer om meddelanden kommer!

Viktiga tips om testning – 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 uppgift. De som är skrivna direkt i `assert`-form finns normalt också med i svarsmallen. Det kan också finnas indirekta tips som del av den löpande texten, vilket normalt *inte* läggs med i svarsmallen men kan fungera som inspiration till fler testfall. **Testa allt du kan och skapa egna variationer!** 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. **Skapa egna tester** som täcker fler fall!
- Det går *delvis* att testa genom att **köra implementationen med många olika indata** även om du inte vet vad korrekt svar ska bli (`print` istället för `assert`). Ibland fungerar loopar – testkör t.ex. med värden från 0 till 100! Det är inte ovanligt att koden kraschar direkt, eller skriver ut uppenbart felaktiga svar. Då har du hittat ett fel, utan att veta exakt vad det korrekta svaret skulle vara!
- 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, så testa med olika listor, även med sådana där elementen består av nästlade listor eller tupler eller andra datastrukturer. En *sekvens* behöver inte nödvändigtvis vara en lista, så testa även med tupler och kanske en sträng eller två. Ska det fungera för heltal $n \geq 0$ ska det också fungera för $n = 0$, så testa det. 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.

Att testa är extremt viktigt!

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)?

Men testning fångar inte allt!

Grunden i uppgifterna är alltid att *läsa* och *förstå*. Testerna är inte ett facit.

Egna tester rekommenderas men är ej obligatoriska!

Tester får lämnas in, men det är inte nödvändigt. De påverkar inte poängen.

Rättningskriterier

Brott mot följande allmänna kriterier kan resultera i poängavdrag.

- Lösningen ska givetvis vara **körbar**. Testa alltid **precis innan inlämning** så att din sista finputsning eller dina sista kommentarer inte resulterade i felaktig kod och så att koden inte kraschar när filen importeras av våra granskningsscript! Poängavdrag ges vid icke körbar kod. (Misslyckade tester och assertions är OK *om* de ligger i `run_tests()`, eftersom vi inte kör den funktionen vid vår betygsättning.)
- Lösningen ska följa alla de **specifika regler och villkor** som står i uppgiften.

  Den ska också **fungera exakt som i körexemplen** i uppgiften, om inte texten indikerar något annat – men tänk på att koden kan vara felaktig trots att körexemplen fungerar! Lösningen ska vara **generell** och ska fungera för *alla* indata som följer uppställningen i uppgiften. Att en lösning enbart fungerar för listor med begränsad längd eller för vissa storlekar på indata är exempel på signifikanta fel.
- Funktioner och källkodsfiler ska ha **exakt samma namn** som anges i uppgiften. Vi hjälper numera till med detta genom svarsmallarna.
- Man ska kunna köra funktioner **flera gånger** med olika indata och korrekt resultat, utan att ladda om koden däremellan. Se upp med olika former av globalt tillstånd / globala variabler. Se upp med defaultargument och modifiera dem aldrig. Testa själv att köra många testfall i rad.
- Kod ska vara **lättförståelig**. Det innebär t.ex. att egna namn (på parametrar, variabler med mera) ska vara beskrivande och följa namnstandarden. Det innebär också att lösningen ska vara **välstrukturerad** och tillräckligt **väldokumenterad** för att en granskare **enkelt** ska förstå hur lösningen fungerar och varför den ser ut som den gör.

  Om *docstrings* krävs, beskrivs detta uttryckligen i uppgiftens instruktioner.
- 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 uppgiften.
- Om inte annat sägs ska funktioner **returnera** värden, inte skriva ut dem. Testfall i tentatexten visar normalt returnerade värden, inte värden som har skrivits ut.

Avdrag för felaktig och ofullständig kod ges **även om lösningen är uppenbar** för granskaren, och även om det syns att lösningen "kan ha varit 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 i alltför många fall räknas normalt inte som lösningar och får 0 poäng. Det går oftast 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 anges** i en uppgift, är följande uttryckligen **tillåtet**:

- Att lösa uppgiften **rekursivt eller iterativt, eller med en kombination** av dessa lösningsmodeller (t.ex. hybrider med rekursiva anrop och defaultargument). Med andra ord: Om inget annat sägs kräver vi inte någon specifik form av rekursion, och du kan använda en lösningsform du känner dig bekväm med.
- Att importera och använda **alla vanliga "inbyggda" funktioner** från Pythons standardbibliotek (upp till och med Python 3.9), t.ex. matematiska funktioner från `math`, högre ordningens funktioner som `map()`, och så vidare.
- Att använda **listbyggare** (list comprehensions), generatoruttryck (generator expressions), slicing (delsekvenser) och andra funktionaliteter i språket.
- 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. Se även varningar för defaultargument under rättningskriterier.
- 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). Står det att funktionen ska ta en lista av heltal, får man anta att det är en lista och att den bara innehåller heltal.

  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 **bryta mot "ytliga" kodningsstandarder** i fråga om t.ex. mellanrum, indentering, radbrytningar, radlängd och antal blankrader, så länge som koden fortfarande är *lättläst och lättförståelig*. Samma gäller stil på identifierare (`snake_case`, `camelCase` med mera).

Detta gäller inte om uppgiften gör specifika undantag!

Uppgift 1: Mobilabonnemang (5p)

Använd den givna filen `ex1.py` för hela denna uppgift!

Kopiera den från `given_files` till en skrivbar katalog, t.ex. `Desktop`. Editera den där.

Många mobilabonnemang tillåter en begränsad mängd nedladdning per månad. I denna uppgift vill du gärna titta på din historiska dataanvändning för att se vilket abonnemang som blir bäst för dig i framtiden. Din mobil är snäll nog att lämna ut dessa data som en Pythonlista med följande innehåll:

- Ett strikt positivt tal (som heltalet `3` eller flyttalet `324.25`) innebär att du laddade ner så många MB under en viss dag. Dagar då inget laddades ner *kan* finnas med i listan (som konstanten `0`), men kan också utelämnas helt.
- Konstanten `None` används för att separera månader i listan, och innebär alltså att den nuvarande månaden tar slut och en ny påbörjas.

Som hjälp tittar vi särskilt på dessa randfall:

- Listan kan sakna `None`. Då gäller den alltså en enda månad.

  Om listan dessutom är helt tom gäller den en enda månad då inget laddades ned, vilket innebär att resultatet också behöver innehålla *en* månad utan konsumtion.
- Listan kan innehålla `None` som första element. Då börjar den alltså med en månad då inget laddades ned. Även sådana månader “räknas" som månader med konsumtion 0 och ska finnas med i resultatet. Motsvarande gäller sista elementet i listan.
- Listan kan innehålla `None` flera gånger i rad. Även detta innebär att det finns månader då inget laddades ner, mellan varje par av `None`-element.

Skriv funktionen `datause(seq: list)` som tar indata på detta format och returnerar en lista med:

- Det totala antalet konsumerade MB för varje månad som representeras i indata, i tur och ordning, samt
- Ett sista värde som är antalet konsumerade MB i den *månad* då du använde mobilen mest.

Exempel – skapa gärna egna tester i `run_tests()` med inspiration av villkoren ovan!

- `assert datause([12, None]) == [12, 0, 12]`
- `assert datause([1, 5, 17, 2]) == [25, 25]`
- `assert datause([None, 1, 12, 5, None, 22, 21, None, 2]) == [0, 18, 43, 2, 43]`

⟹⟹⟹⟹⟹⟹⟹⟹ **Fortsätter på nästa sida!** ⟹⟹⟹⟹⟹⟹⟹⟹

Ytterligare villkor:

- Funktionerna som specificeras i uppgiften får **inte** modifiera sina indata.

Allmänna tips och ledtrådar:

- Som vi säger i tentans inledning får man alltid anta att indata är korrekta. I detta fall kommer vi alltså bara att testa med listor `seq` som innehåller tal och konstanten `None`.

Uppgift 2: Run Length Encoding (5p)

Använd den givna filen `ex2.py` för hela denna uppgift!

Kopiera den från `given_files` till en skrivbar katalog, t.ex. `Desktop`. Editera den där.

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 exempelvis att du har en sekvens där du ofta får exakt 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 kan bli en lista av tupler (a, b), där varje sådan tupel anger att det finns $a > 0$ element i rad med värdet b.

Exempel:

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

  Här är alltså indata en sträng, som är en sorts sekvens som innehåller tecken – itererar man över strängen får man ut ett tecken i taget. Indata kan också vara en lista eller tupel.

  En delsekvens som består av identiska element *måste* kodas in "tillsammans" i svaret. Man får alltså inte returnera svaret `[(1,'A'), (2,'B'), (2,'B'), (10,'C'), (3,'A')]`: Även om det egentligen också motsvarar samma teckensekvens (1 'A', 2 'B', ytterligare 2 'B', 10 'C' och 3 'A') måste funktionen detektera att det faktiskt var 4 'B' i rad.
- `rle([1,2,2,2,2,5,5,5,5,5,5,1,1,1]) == [(1,1), (4,2), (6,5), (3,1)]`
- `rle(["a","a","b","a","c","c"]) == [(2,"a"), (1,"b"), (1,"a"), (2,"c")]`
- `rle("") == []` – tomma sekvensen resulterar i tomma listan.

Ytterligare villkor:

- Funktionerna som specificeras i uppgiften får **inte** modifiera sina indata.

Uppgift 2a: RLE med iterativ lösningsmodell (3p)

Implementera funktionen `rle_i(seq)`, som tar en godtycklig sekvens `seq` innehållande godtyckliga element och som returnerar motsvarande skurlängdskodade lista enligt ovan.

Funktionen **måste** vara iterativ och rekursion får inte användas.

Uppgift 2b: RLE med rekursiv lösningsmodell (2p)

Implementera funktionen `rle_r(seq)`, som tar en godtycklig sekvens `seq` innehållande godtyckliga element och som returnerar motsvarande skurlängdskodade lista enligt ovan.

Funktionen måste 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.

I denna uppgift ska rekursionen behandla *ett element i* `seq` *åt gången*. Alltså kan ett anrop till `rle_r(["a","a","a","a","c","c"])` rekursera med anropet `rle_r(["a","a","a","c","c"])`, men får inte på en gång behandla alla `"a"` för att direkt anropa `rle_r(["c","c"])`.

Uppgift 3: Slå upp element i nästlad lista (5p)

Använd den givna filen `ex3.py` för hela denna uppgift!

Kopiera den från `given_files` till en skrivbar katalog, t.ex. `Desktop`. Editera den där.

Skriv funktionen `lookup_nested(seq: list, table: dict)`, som:

- Tar en godtycklig nästlad lista `seq` med godtyckliga element
- Returnerar motsvarande nästlade lista där alla element `elem` som inte är *listor* har ersatts med resultatet av att slå upp `elem` i `table`, en dictionary – eller med `None`, om `elem` inte finns som nyckel i `table`.

  Ersättningen ska göras på alla listnivåer i den nästlade listan (se exemplen längre ner).

Villkor, tips och ledtrådar:

- Funktionerna som specificeras i uppgiften får **inte** modifiera sina indata.
- Funktionen ska hantera godtyckliga nästlade listor. Det inkluderar djupa listor (många nästlingsnivåer), korta listor, tomma listor, och *godtyckliga* elementtyper. Skapa egna testfall för detta, och testa att det i alla fall inte kraschar!
- Det är oftast enklast att klara av godtyckligt antal nivåer med en rekursiv lösning. Det krävs dock inte att man använder en rekursiv *lösningsmodell*, utan lösningen får gärna kombinera iteration och rekursion. Den kan till och med vara helt iterativ, men det kräver tekniker som vi ännu inte har lärt oss.
- En enkel och tydlig implementation i våra lösningsförslag tar 8 rader, inklusive funktionshuvudet (“def"-raden).

  Du ska så klart inte försöka anpassa antalet rader enligt vad vi råkade få i lösningsförslagen! Men om du är på väg mot 25–30–40 rader för att lösa problemet kan det hända att du har trasslat in dig i en krångligare lösning än du egentligen behöver.

Exempel – skapa gärna egna tester i `run_tests()` med inspiration av villkoren ovan!

- ```
  table = {1: 111, 2: 222, 3: '3', -1: 'Negativt?', 'Hi': 'Hello'}
  # Använder samma dictionary i dessa testfall, men självklart är det
  # bra att även testa med andra dictionaries
  ```
- `assert lookup_nested([1, 2, 3], table) == [111, 222, '3']`
- `assert lookup_nested([-1, [2, 3], ['Hi', 4, [1]]], table) == ['Negativt?', [222, '3'], ['Hello', None, [111]]]`
- ```
  assert lookup_nested([-1, [2, 3], ('Hi', 4, 1)], table) ==
  ['Negativt?', [222, '3'], None]
  # Detta illustrerar att en tupel inte är en lista utan ett vanligt element,
  # och eftersom denna tupel inte finns i table ersätts den med None.
  ```

Uppgift 4: Högre ordningens funktioner (5p)

Använd den givna filen `ex4.py` för hela denna uppgift!

Kopiera den från `given_files` till en skrivbar katalog, t.ex. `Desktop`. Editera den där.

Deluppgift 4a (3.5p)

Skapa en högre ordningens funktion `pairwise_apply(f)`. Denna funktion ska göra följande – se även exemplen nedan.

- Ta som argument en funktion `f`.
- Definiera och returnera en ny funktion, som i sin tur:
  - Tar *två* möjligen tomma godtyckliga listor `seq1` och `seq2` som argument.
  - Returnerar en lista som i tur och ordning innehåller elementen `f(seq1[0]), seq2[0])`, `f(seq1[1]), seq2[1])`, och så vidare.

  Om en lista (`seq1` eller `seq2`) innehåller fler element än den andra, ska de överflödiga elementen i denna lista ignoreras. Exempel ges i slutet av deluppgift 4b.

Exempel:

- Anta att vi redan råkar ha definierat en funktion `f` som tar två argument. Till exempel kan `f` vara funktionen `plus(a,b)` som returnerar `a+b`.
- Anta att vi sätter `ourfun = pairwise_apply(plus)`. Anropet till `pairwise_apply(f)` ska då returnera en funktion, och `ourfun` är nu denna returnerade funktion – som tar två listor `seq1` och `seq2` som argument, och som applicerar `f` på par av element enligt ovan.
- Om vi nu anropar `ourfun([1, 2, 3], [7, 9, 11])` ska detta returnera resultatlistan `[f(1,7), f(2,9), f(3,11)]`, vilket är `[plus(1,7), plus(2,9), plus(3,11)]`, vilket är `[8, 11, 14]`.
- Om vi istället anropar `ourfun([1, 2, 3], [7, 9])` ska detta returnera resultatlistan `[f(1,7), f(2,9)]`, vilket är `[plus(1,7), plus(2,9)]`, vilket är `[8, 11]`. Den andra listan hade färre element, så ett element i den första (`3`) användes aldrig.
- Använd gärna detta exempel som inspiration till egna testfall!

Ytterligare villkor:

- Funktionerna som specificeras i uppgiften får **inte** modifiera sina indata.

Villkor, tips och ledtrådar:

- Tänk på att `pairwise_apply` ska kunna applicera *godtyckliga* funktioner `f` (med två parametrar) på element från *godtyckliga* listor. Att det handlade om heltal och addition i exemplet var just bara ett exempel, så detta ska inte på något sätt synas i definitionen av `pairwise_apply`.

Deluppgift 4b (1,5p)

Använd `pairwise_apply` för att definiera `pairwise_multiply(seq1, seq2)`, där varje returnerat element är *produkten* av två element från `seq1` och `seq2`. Detta ska definieras med hjälp av ett enda uttryck:

- `pairwise_multiply = ...`

Du ska göra detta **utan** att definiera egna namngivna hjälpfunktioner utöver `pairwise_apply`. Detta innebär att du behöver använda ett lambdauttryck.

Exempel:

- `assert pairwise_multiply([1, 2, 3], [7, 9, 11]) == [7, 18, 33]`
- `assert pairwise_multiply([3], [7, 9, 11]) == [21]`
- `assert pairwise_multiply(["ab", "cd"], [4, 2]) == ['abababab', 'cdcd']` # Att multiplicera en sträng med ett tal gör att strängen repeteras

Uppgift 5: Alternerande hopp i listor (5p)

Använd den givna filen `ex5.py` för hela denna uppgift!

Kopiera den från `given_files` till en skrivbar katalog, t.ex. `Desktop`. Editera den där.

Översikt: Givet två sekvenser av heltal, använd ett värde från ena sekvensen för att hoppa till ny position i den andra sekvensen, och vice versa. Upprepa till ett slutvillkor är uppnått.

Obs: Denna funktion ser mer komplicerad ut än den egentligen är! En anledning till att beskrivningen är lång är att vi vill illustrera ett exempel.

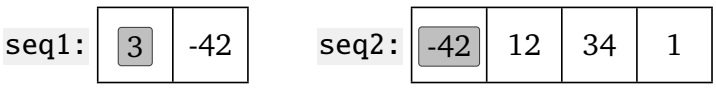
Implementera funktionen `alternating_jumps(seq1, seq2, maxjumps: int)`, som:

(A) Tar två godtyckliga heltalssekvenser `seq1` och `seq2` samt ett heltal `maxjumps` som anger ett maximalt antal "hopp".

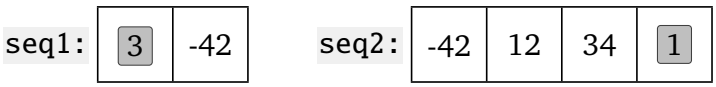
Vi garanterar att `len(seq1)` ≥ 2, att `len(seq2)` ≥ 2, och att `maxjumps` > 0.

Sekvenserna kan innehålla godtyckliga heltal, även talet `0`.

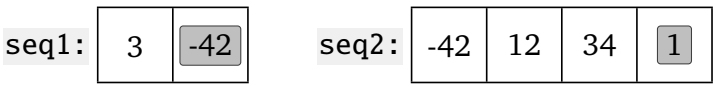
(B) Börjar på första positionen i både `seq1` och `seq2`. Det behövs alltså ett sätt att hålla reda på nuvarande positionen för var och en av dessa sekvenser.

> Om `seq1=[3,-42]` och `seq2=[-42,12,34,1]` kan vi illustrera sekvenserna så här, med en mörkare markering på "nuvarande position":
>
> 
>

(C) Tittar på värdet på nuvarande position i `seq1`, och hoppar så många steg framåt (eller bakåt för negativa tal) i `seq2`.

> I exemplet ovan har vi värdet **3** i `seq1` och går alltså tre steg framåt i `seq2`:
>
> 
>

(D) Tittar på värdet på nuvarande position i `seq2`, och hoppar så många steg framåt (eller bakåt för negativa tal) i `seq1`.

> I exemplet ovan har vi värdet **1** i `seq2` och går alltså `1` steg framåt i `seq1`:
>
> 
>

(E) Ständigt håller reda på de hopp som har gjorts hittills. Vi kan kalla detta för `jumps`.

> I början har vi `jumps=[]`.
> Efter hoppet i punkt (C) hade vi `jumps=[3]`, eftersom vi hoppade 3 steg i `seq2`.
> Efter hoppet i punkt (D) har vi `jumps=[3,1]`, eftersom vi hoppade 1 steg i `seq1`.

(F) Fortsätter på detta sätt (titta på element i `seq1` och byt position i `seq2`, titta på element i `seq2` och byt position i `seq1`, ...) ända till ett av följande tre slutvillkor är uppfyllt. Samtliga slutvillkor exemplifieras också i testfallen. Tänk på att slutvillkoren måste testas vid *varje* hopp, inte bara efter att *två* hopp har gjorts!

- Om vi just har hoppat (steg C respektive D), och resultatet är att vi pekar ut *sista* elementet i *båda* sekvenserna, returneras `("success", jumps)`.

  > I exemplet ovan har vi nått till sista elementet i båda sekvenserna, och därmed returneras `("success", [3,1])`.

- Om vi just har hoppat (steg C respektive D) *utan* nå fram till sista elementet i båda sekvenserna, och vi nu har uppnått det maximala antalet hopp `maxjumps`, får vi inte hoppa fler gånger. Då returneras `("no-more-jumps", jumps)`.
- Om vi är på väg att hoppa (före steg C respektive D), men hoppet i så fall skulle resultera i en ny position som är utanför gränserna för en sekvens (före det första elementet eller efter det sista elementet), kan detta hopp inte genomföras; det är "ogiltigt". Då returneras `("out-of-bounds", jumps)` där `jumps` innehåller de hopp som faktiskt genomfördes.

Ytterligare villkor:

- Funktionerna som specificeras i uppgiften får **inte** modifiera sina indata.

Allmänna tips och ledtrådar:

- **Gör gärna just som vi föreslår ovan!** Förklaringarna är i princip en sorts *pseudokod*, som talar om vad som ska göras och när det ska göras. Vi är exempelvis ofta tydliga med *när* och *hur* villkor bör testas, såsom "om vi *just har hoppat* och resultatet är..." eller "om vi just har hoppat *utan att nå fram till*...". Det kan vara frestande att bryta mot detta, inte minst om man tror att man kan skriva koden effektivare.
  - Kanske du vill testa ett villkor strax *före* ett hopp istället för när vi *just har hoppat*, eller ändra ordning på tester för att det ser bättre ut? Då löper du stor risk att ändra på algoritmen så den inte alls gör vad den ska, och att du inte har tillräckligt många testfall för att upptäcka det.
  - Koden i steg C och steg D gör i princip samma sak men med olika sekvenser, så kanske du vill försöka generalisera och hantera båda steg med samma kod? Då riskerar du också att det trasslar ihop sig, och att du tappar bort dig i vad koden egentligen gör. Här kan det faktiskt vara enklare att låta funktionen innehålla ungefär samma kod två gånger i rad.
- Blanda inte ihop *index (positioner i sekvensen)* (som är heltal) med *element i sekvensen* (som också råkar vara heltal).
- Tänk på att `seq1` och `seq2` kan innehålla samma värde flera gånger.

⟹⟹⟹⟹⟹⟹⟹⟹ **Fortsätter på nästa sida!** ⟹⟹⟹⟹⟹⟹⟹⟹

Exempel (skapa gärna egna testfall):

- `assert alternating_jumps([1, -42], [-42, 1], 10) == ("success", [1, 1])`
- `assert alternating_jumps([3, -42], [-42, 12, 34, 1], 10) == ("success", [3, 1])`
  # Läs siffran 3 i seq1 och hoppa 3 steg till slutet i seq2;
  # läs siffran 1 på slutet av seq2 och hoppa 1 steg till slutet av seq1
- `assert alternating_jumps([1, -1, 14], [-1, 1, 14], 10) == ("no-more-jumps", [1, 1, -1, -1, 1, 1, -1, -1, 1, 1])`
  # Stega fram och tillbaka i seq1 och seq2 till hoppen “tar slut"
- `assert alternating_jumps([2], [1, 1, -10, 5], 10) == ("out-of-bounds", [2])`
  # Hoppar 2 steg framåt i seq2; försöker sedan hoppa -10 steg i seq1; kan ej genomföras

Förslag på egna tester:

- Testa med små och stora värden på `maxjumps`. Även om du inte vet vad svaret borde vara ser du i alla fall om det kraschar eller ger ett resultat!
- Testa med värden på `maxjumps` som är precis rätt (`maxjumps=3` när det behövdes tre hopp), eller 1 för lite (`maxjumps=2` när det behövdes tre hopp), eller 1 för mycket.

Uppgift 6: Dubbeländad kö (5p)

Använd den givna filen `ex6.py` för hela denna uppgift!

Kopiera den från `given_files` till en skrivbar katalog, t.ex. `Desktop`. Editera den där.

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*, förkortat *deque* (låter som “deck"). 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.

I den här uppgiften ska du implementera en datatyp för dubbeländade köer.

Gemensamma villkor för hela uppgiften:

- En dubbeländad kö ska kunna innehålla *godtyckliga* element, även om exemplen råkar använda heltal som element.
- Funktioner som ska ta bort ett element *ska* hantera en *tom kö* genom att helt enkelt inte ta bort något. Funktionerna får inte krascha i detta fall.
- Vi har inte definierat en funktion `is_deque(x)` som testar om ett värde faktiskt är en `deque`, och övriga funktioner får anta (behöver inte uttryckligen testa) att en parameter som borde vara en `deque` faktiskt är en `deque`.

  Detta ger också mer frihet i exakt hur man representerar en `deque`.
- Funktionernas implementation *får* gå direkt på den interna representationen när de manipulerar köer. De behöver inte använda selektorer.

Uppgift 6a: Grundläggande och destruktiva funktioner (3p)

Välj en lämplig intern representation för dubbeländade köer och implementera nedanstående funktioner.

- `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.

Vi behöver också kunna modifiera en existerande kö. I första steget vill vi lägga till eller ta bort element “(d)estruktivt" – vilket helt enkelt innebär att vi *modifierar* den existerande datastrukturen, den kö som skickas in som parameter till en funktion. Det gör vi med följande 4 funktioner, som *inte ska returnera något*.

- `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.

Exempel 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
```

Uppgift 6b: Funktionella funktioner (2p)

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

Ytterligare villkor, utöver de tidigare gemensamma villkoren:

- Dessa funktioner ska arbeta funktionellt och får inte modifiera den gamla existerande kön `deque`. Det är dock OK att funktionen internt 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.

Exempel 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 på egen hand!
```