Datortentamen 2023-01-10

TDDE24 Funktionell och imperativ programmering del 2

Allmän information

Tentan består av totalt **6 uppgifter** som vardera kan ge maximalt 5 poäng, utom sista som kan ge 8 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!

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 menyer och tangentbordsbindningar som man är van vid, inte att man ska ha en fullständig integrerad utvecklingsmiljö som hjälper till med allt man ska göra.
- 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 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. Deklarationerna är bortkommenterade, så du får själv ta bort kommentarstecknen “#". Du får givetvis skapa egna hjälpfunktioner som tillägg!
- Inkluderar eventuella assert-tester som vi redan har angivit i uppgiften, så du slipper klippa och klistra. (Du behöver ändå tänka på att skapa egna tester.)

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.

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: Lista till dictionary (5p)

Använd den givna filen `ex1.py` för hela denna uppgift!
Kopiera den från `given_files` till desktop (eller annan skrivbar mapp) och editera den där. Filen innehåller bland annat testfall och den **ska** användas för inlämningen.

I första uppgiften vill vi göra det lättare att hitta alla strängar (från en lista) som börjar på ett visst tecken. Du ska därför skriva funktionen `split_by_first(seq: list[str])`, som antagligen är lättast att förstå från några exempel:

- `assert split_by_first(['apa', 'bepa', 'arg']) == {'a': ['apa', 'arg'], 'b': ['bepa']}`
- `assert split_by_first(['01', '13', '02', '14', '01']) == {'0': ['01', '02', '01'], '1': ['13', '14']}`
- `assert split_by_first(['Bakom', 'brödbutiken', 'bodde', 'Baskerbosses', 'båda', 'bröder', 'bröderna', 'Basker']) == {'B': ['Bakom', 'Baskerbosses', 'Basker'], 'b': ['brödbutiken', 'bodde', 'båda', 'bröder', 'bröderna']}`

Detta är några av de saker vi kan se från exemplen:

- Resultatet av funktionen är en *dictionary*, vars nycklar är exakt de tecken som förekommer *först* i någon sträng i parametern `seq`.
- Med *tecken* menar vi inte bara bokstäver: Första tecknet kan t.ex. vara '0' eller '1'.
- Varje nyckeltecken mappas till en *lista*, som innehåller alla de ord från `seq` som börjar på detta nyckeltecken: `"a"` mappas till `['apa','arg']` i första exemplet, och `'0'` mappas till `['01', '02', '01']` i andra exemplet.
- Vi ser i andra exemplet att samma ord kan finnas med flera gånger: `['01', '02', '01']`.
- I första exemplet måste `'a'` mappas till värdet `['apa', 'arg']` eftersom orden förekom i den ordningen i den ursprungliga listan. Man får inte mappa till värdet `['arg', 'apa']` istället.

Din uppgift är nu att skriva funktionen `split_by_first(seq: list[str])`, som fungerar enligt beskrivningen och exemplen ovan.

Ytterligare villkor:

- Funktionerna som specificeras i uppgiften får **inte** modifiera sina indata.
- Parametern `seq` är en godtyckligt lång eller kort lista med godtyckliga icke-tomma strängar. Eftersom strängarna inte är tomma har varje sträng alltså ett första tecken. (Om `seq` är tomma listan returneras alltså en tom dictionary.)

Fortsätter på nästa sida.

Allmänna tips och ledtrådar:

- Man kan testa om en nyckel `key` redan finns i en dictionary `d` med hjälp av `key in d`.
- Använder du utskrifter för att testa om du får rätt resultat? Tänk då på att en dictionary inte har någon (nyckel)ordning:
  `{'a': ['apa', 'arg'], 'b': ['bepa']}` är samma sak som
  `{'b': ['bepa'], 'a': ['apa', 'arg']}`.
- En enkel och tydlig implementation i våra lösningsförslag tar 9 rader, inklusive funktionshuvudet ("def"-raden). En något optimerad variant tar 5 rader, och en svårläst och mindre effektiv med bland annat nästlade list och dict comprehensions tar 2 rader.

  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: Se även de inledande exemplen.

- `assert split_by_first(['abc']) == {'a': ['abc']}`

Uppgift 2: Dela upp i dellistor (5p)

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

Uppgift 2a: Dela upp med godtycklig lösningsmodell (3p)

Skriv funktionen `split_lists(seq, sizes)`, där:

- Parametern `seq` är en sekvens (lista eller tupel) av $n \geq 1$ godtyckliga element.
- Parametern `sizes` är en *sträng* bestående av siffror (“0" till “9").

Funktionen ska i normalfallet returnera en *lista av* `len(sizes)` *listor*, där varje dellista innehåller så många element som angavs av motsvarande siffra i `sizes`, i tur och ordning:

- `assert split_lists([1, 2, 0, 4, 7, 6], "132") == [[1], [2, 0, 4], [7, 6]]`
  # "132" ==> ska finnas 1+3+2==6 element i listan, vilket det gör
  # "132" ==> plocka i tur och ordning 1, 3 och 2 element ur listan

Om summan av siffrorna i `sizes` skiljer sig från antalet element i `seq`, är problemet ovan inte lösbart. Då ska funktionen istället signalera detta genom att returnera det ovanliga värdet `(None,None)`:

- `assert split_lists([1, 2, 3, "x"], "12") == (None, None)`
- `assert split_lists([1, 2], "12") == (None, None)`

Parametern `sizes` kan innehålla godtyckliga siffror, även 0, på godtycklig position:

- `assert split_lists([1, 2, 3], "102") == [[1], [], [2, 3]]`

Ytterligare villkor:

- Funktionerna som specificeras i uppgiften får **inte** modifiera sina indata.
- Glöm inte att skapa egna testfall. Testa t.ex. med 0 i början och i slutet, och se till att du alltid får `len(sizes)` listor i svaret.

Uppgift 2b: Dela upp rekursivt (2p)

För ytterligare 2 poäng, skriv funktionen `split_lists_rec(seq, sizes)` som tar samma parametrar och returnerar samma resultat, men är implementerad med en rekursiv lösningsmodell. Det innebär alltså att ett rekursivt anrop ska beräkna svaret på ett äkta delproblem av exakt samma typ (dela upp en kortare `seq` enligt en kortare `sizes`), precis som vi har diskuterat under kursen.

Ytterligare villkor:

- Funktionerna som specificeras i uppgiften får **inte** modifiera sina indata.
- Det går utmärkt att lämna in samma rekursiva lösning även för deluppgift 2a, men du måste då ändå definiera (`def`) båda funktionerna (funktionsnamnen). Finns bara en av funktionerna `split_lists(seq, sizes)` eller `split_lists_rec(seq, sizes)` definierad, markeras det automatiskt att bara en av dem är inlämnad!

Exempel:

- Använd samma testfall som du skapade för 2a, men med `split_lists_rec` istället.

Allmänna tips och ledtrådar:

- Tänk på att du alltid ska lämna tillbaka `len(sizes)` listor. Du måste välja rätt basfall för att detta alltid ska ske.

Uppgift 3: Dubblera udda tal i nästlad lista (5p)

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

Av någon anledning tycker vi inte om udda tal. Funktionen `doubled_odds(seq: list)` ska därför ta en godtycklig nästlad lista `seq` och returnera en ny lista som har samma nästlade struktur, men där alla udda heltal n (av typ `int`) är "ersatta" med dubbla värdet $2 * n$.

Ytterligare villkor:

- Funktionerna som specificeras i uppgiften får **inte** modifiera sina indata.
- Listor kan nästlas i godtyckligt antal nivåer. En lösning behöver klara av detta, vilket oftast är enklast om man gör den rekursiv. 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.
- Som synes i exemplen nedan kan listan innehålla annat än bara heltal och andra nästlade listor. Det är bara nästlade *listor* som man ska rekursera ner i, och det är bara udda *heltal* (int) som ska fördubblas.

Allmänna tips och ledtrådar:

- Funktionen ska hantera godtyckliga nästlade listor. Det inkluderar djupa listor (många nästlingsnivåer), korta listor, tomma listor, och *godtyckliga* elementtyper. Skapa egna testfall för detta, åtminstone genom att skriva ut `doubled_odds(...)` och se att det inte kraschar!
- Överkurs: `True` och `False` är av typen `bool`, *inte* av typen `int` (även om det råkar finnas en speciell jämförelsefunktion så att `True==1` är sant).
- En enkel och tydlig implementation i våra lösningsförslag tar 10 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 med inspiration av villkoren ovan!

- `assert doubled_odds([1, 2, 3]) == [2, 2, 6]`
- `assert doubled_odds([-1, [2, 3], ['Hi', 4, [7]]]) == [-2, [2, 6], ['Hi', 4, [14]]]`
- `assert doubled_odds([-1, [2, 3], ('Hi', 4, [7])]) == [-2, [2, 6], ('Hi', 4, [7])]`

Uppgift 4: Högre ordningens funktioner (5p)

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

Deluppgift 4a (3p)

Skapa en högre ordningens funktion `sum_satisfying(fun, pred)`, som:

- Tar två unära funktioner, `fun` och `pred`, som argument. (Unära funktioner är funktioner som tar exakt 1 argument.)
- Definierar en ny unär funktion (med godtyckligt namn), som tar en godtycklig sekvens `seq` och returnerar *summan* av `fun(x)` för alla element `x` ∈ `seq` som uppfyller `pred(x)`.
- Returnerar denna funktion.

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

Här är `str.isdigit` en unär standardfunktion som tar in en sträng och testar om den sträng bara innehåller siffror, vilket alltså är sant för elementen '10' och '4711'. Resultatet av `len()` för dessa element är 2 respektive 4, och summan är 6.

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

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

Ytterligare villkor:

- Funktionerna som specificeras i uppgiften får **inte** modifiera sina indata.
- Om den returnerade funktionen anropas med en sekvens där inget element uppfyller `pred(x)`, ska 0 returneras (summan av inga värden är 0).

Deluppgift 4b (2p)

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

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

Exempel:

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

Allmänna tips och ledtrådar:

- Kom ihåg att `sum_satisfying()` *returnerar* en funktion. Du kan alltså använda tilldelningen `sum_square_negative_odd = sum_satisfying(...)` för att definiera funktionen `sum_square_negative_odd()`.
- Du kan skicka in två lambdafunktioner som parametrar till `sum_satisfying()`. Det är också tillåtet att först skapa “vanliga" namngivna funktioner som skickas in som parametrar.

Uppgift 5: Matrisdatatyp (5p)

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

Matriser spelar en stor och viktig roll i både matematiken och datavetenskapen. En matris består av ett antal rader där alla rader har lika många element (kolumner). Radnumret anges före kolumnnumret, så följande matris är en 4×3-matris:

$$\begin{pmatrix} 1 & 2 & 3 \\ -1 & 0 & 7 \\ 5 & -9 & 2 \\ 6 & 1 & 5 \end{pmatrix}$$

Till skillnad från Pythons listor börjar vi indexeringen på rad/kolumn 1. Elementet $A[2, 3]$, som ofta benämns $a_{2,3}$, är alltså 7.

Din uppgift är att implementera några vanliga matrisoperationer enligt kommande beskrivningar. I uppgiften har vi inte ett fullständigt fokus på dataabstraktion, då detta skulle göra den totala arbetsmängden för stor för denna tenta. Därför skapas till exempel inte en separat funktion för att konstruera nya matriser. Istället ligger fokuset på en korrekt implementation av själva operationerna.

En matris ska representeras som en lista med listor, där varje nästlad lista är en rad i matrisen. Operationerna som anges nedan ska implementeras för denna representation. De två första funktionerna ger 0.5 poäng, medan övriga ger 1 poäng vardera.

Ytterligare villkor:

- Funktionerna som specificeras i uppgiften får **inte** modifiera sina indata.
- Det är uttryckligen tillåtet att använda list comprehensions i denna uppgift. Andra metoder är också tillåtna.
- Man får att anta att indata är korrekta (“plus" får t.ex. krascha om den ges två matriser av olika storlek).
- Man får anta att varje matris har minst 1 rad och minst 1 kolumn.
- Elementen kommer att vara `tal`, t.ex. heltal eller flyttal.

Allmänna tips och ledtrådar:

- För att inte trassla in sig i index i onödan kan man tänka enligt denna mall:

```
result = []
for i in range(hur många rader vi vill ha i svaret):
    row = []
    for j in range(hur många kolumner vi vill ha):
        element = uträkning av värdet $c_{i,j}$
        row.append(element)
    result.append(row)
```

Operationer att implementera:

- `rows(matrix)` – Returnera antalet rader i `matrix`. Ska användas överallt där man behöver ta reda på antalet rader, även i din implementation av andra operationer.
- `columns(matrix)` – Returnera antalet kolumner i `matrix`. Ska användas överallt där man behöver ta reda på antalet kolumner.
- `transpose(matrix)` – Returnera matrisens transponat.

  Transponatet av en $m \times n$-matris är en $n \times m$-matris där rader och kolumner så att säga har bytt plats. Transponatet till matrisen A betecknas A^T, och vi har att $A^T[i, j] = A[j, i]$. Exempel:

  $$\begin{pmatrix} 1 & 0 & 2 \\ -1 & 3 & 1 \end{pmatrix}^T = \begin{pmatrix} 1 & -1 \\ 0 & 3 \\ 2 & 1 \end{pmatrix}$$

- `map(matrix, fun)` – Returnera en ny matris där varje element $c_{i,j}$ har "ersatts" med fun($c_{i,j}$). Exempel: `map([[1, 0, 2], [-1, 3, 1]], lambda x: -x) == [[-1, 0, -2], [1, -3, -1]]`.
- `plus(m1, m2)` – Returnera en ny matris som är summan av `m1` och `m2`.

  Addition av två matriser förutsätter att matriserna har exakt samma dimensioner. Om A och B är två $m \times n$-matriser, definieras $C = A + B$ genom $c_{i,j} = a_{i,j} + b_{i,j}$. Exempel:

  $$\begin{pmatrix} 1 & 3 & 2 \\ 1 & 0 & 0 \\ 1 & 2 & 2 \end{pmatrix} + \begin{pmatrix} 0 & 0 & 5 \\ 7 & 5 & 0 \\ -2 & 1 & 1 \end{pmatrix} = \begin{pmatrix} 1+0 & 3+0 & 2+5 \\ 1+7 & 0+5 & 0+0 \\ 1+(-2) & 2+1 & 2+1 \end{pmatrix} = \begin{pmatrix} 1 & 3 & 7 \\ 8 & 5 & 0 \\ -1 & 3 & 3 \end{pmatrix}$$

  (Fortsättning på nästa sida)

- `times(m1, m2)` – Returnera en ny matris som är produkten av `m1` och `m2`.

  Produkten AB av två matriser A och B är bara definierad om antalet kolumner i A är lika med antalet rader i B. Anta till exempel att A är en $m \times n$-matris (m rader, n kolumner) och att B är en $p \times q$-matris. Då är AB bara definierad om $n = p$, och resultatet blir då en $m \times q$-matris. (Det är tillåtet att anta att `m1` och `m2` har “rätt" dimensioner, och att krascha annars!)

  Om $C = AB$, så gäller:

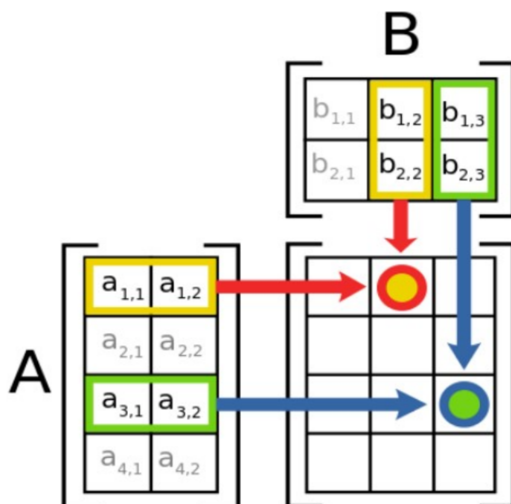
$$c_{i,j} = a_{i,1}b_{1,j} + a_{i,2}b_{2,j} + \ldots + a_{i,n}b_{n,j} = \sum_{r=1}^{n} a_{i,r}b_{r,j}$$

  Ett konkret exempel:

$$\begin{pmatrix} 1 & 0 & 2 \\ -1 & 3 & 1 \end{pmatrix} \times \begin{pmatrix} 3 & 1 \\ 2 & 1 \\ 1 & 0 \end{pmatrix} = \begin{pmatrix} (1\cdot 3 + 0\cdot 2 + 2\cdot 1) & (1\cdot 1 + 0\cdot 1 + 2\cdot 0) \\ (-1\cdot 3 + 3\cdot 2 + 1\cdot 1) & (-1\cdot 1 + 3\cdot 1 + 1\cdot 0) \end{pmatrix} = \begin{pmatrix} 5 & 1 \\ 4 & 2 \end{pmatrix}$$

  Följande bild kan hjälpa till att visualisera operationen (men är egentligen inte nödvändig om du bara vill implementera summan ovan). Raderna i A bestämmer antalet rader i resultatet, och kolumnerna i B bestämmer antalet kolumner i resultatet.

  För att räkna ut det mittre elementet i översta raden, $c_{1,2}$, ska vi titta på motsvarande “gula" rad 1 i A och motsvarande “gula" kolumn 2 i B. Där matchas elementen mot varandra, så att vi multiplicerar $a_{1,1}$ med $b_{1,2}$ och $a_{1,2}$ med $b_{2,2}$; det är därför som A måste ha lika många kolumner som B har rader.



Exempel:

```
m1 = [[1, 0, 2], [-1, 3, 1]]
assert rows(m1) == 2
assert columns(m1) == 3

m2 = transpose(m1)
assert m2 == [[1, -1], [0, 3], [2, 1]]

m3 = [[3, 1], [2, 1], [1, 0]]
assert plus(m2, m3) == [[4, 0], [2, 4], [3, 1]]
assert times(m1, m3) == [[5, 1], [4, 2]]

assert map(m1, lambda x: -x) == [[-1, 0, -2], [1, -3, -1]]
```

Uppgift 6: PyAssm (8p)

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

Nu ska du skriva en emulator för ett enkelt assemblerspråk, som vi kallar PyAssm.

I språket ska vi ha tillgång till 26 *register* som vi kan se som variabler i språket. Dessa register är fördefinierade och har namnen "A" till "Z" (stora bokstäver).

Det första som sker när man startar ett PyAssm-program är att varje register ska sättas till värdet `0` (heltal noll). Därefter utförs alla instruktioner i programmet. Programmet ska hålla reda på en *logg*, en lista med *registervärden* som loggas av `"LOG"` enligt nedan. Denna logg ska sedan returneras.

Ett *program* i PyAssm är en vanlig Python-lista av instruktioner. En *instruktion* i PyAssm är en lista på ett av följande format:

- `["SET", r, n]` – sätter värdet på register `r` till värdet `n`, som är ett tal (heltal, flyttal) i Python.
- `["CPY", r, s]` – sätter värdet på register `r` till värdet som just nu finns i register `s`.
- `["ADD", r, n]` – modifierar värdet på register `r` genom att addera värdet `n`, som är ett tal (heltal, flyttal) i Python.
- `["MUL", r, n]` – modifierar värdet på register `r` genom att multiplicera det med värdet `n`, som är ett tal (heltal, flyttal) i Python.
- `["LOG", r]` – loggar värdet på register `r` genom att addera tupeln `(r, v)` i logglistan, där `v` är det nuvarande värdet i register `r`. Se exemplen på nästa sida: Alla anrop till `eval_pyassm()` returnerar en logglista.

Uppgiften fortsätter på nästa sida.

Uppgift 6a: Första versionen av PyAssm (3p)

Definiera funktionen `eval_pyassm(prog)`, som tar ett giltigt PyAssm-program `prog` med $n \geq 0$ instruktioner och exekverar/emulerar det. Funktionen ska alltså utföra instruktionerna i programlistan från början till slut enligt definitionen av instruktionerna ovan.

Allmänna tips och ledtrådar:

- Alfabetet från A till Z innehåller tecknen ABCDEFGHIJKLMNOPQRSTUVWXYZ.
- Rekursion passar *inte* bra för denna uppgift, eftersom det kan ge svårigheter att implementera hopp i uppgift (b).
- En enkel och tydlig implementation i våra lösningsförslag tar 21 rader, inklusive funktionshuvudet ("def"-raden).

  Du ska så klart inte försöka anpassa antalet rader enligt vad vi råkade få i lösningsförslagen! Detta är bara en vink om ungefär hur mycket kod som kan behövas.

Exempel:

- `assert eval_pyassm([['LOG', 'A']]) == [('A', 0)]`
- `assert eval_pyassm([['SET', 'A', 10], ['MUL', 'A', 5], ['ADD', 'A', 5.25], ['LOG', 'A'], ['CPY', 'B', 'A'], ['LOG', 'A'], ['LOG', 'B']]) == [('A', 55.25), ('A', 55.25), ('B', 55.25)]`

Ytterligare villkor för 2a:

- Funktionerna som specificeras i uppgiften får **inte** modifiera sina indata.
- Det måste gå att köra flera olika program i rad genom att anropa `eval_pyassm` flera gånger. Programmet får alltså inte t.ex. lagra data i globala variabler på ett sätt som gör att olika körningar stör varandra.

Glöm inte att skriva egna testfall.

Uppgift 6b: Hoppinstruktioner (2p)

Med de instruktioner som implementerades i förra deluppgiften kan vi bara skriva program som exekveras linjärt – efter att instruktionen på position (listindex) *pos* har exekverats, går vi alltid vidare till instruktionen på position *pos+1*, ända till programmet är slut. Nu ska vi ändra på det.

Spara gärna undan din fil så att du har den kvar om du skulle råka göra några misstag i denna version (men lämna bara in *en* version!). Lägg sedan till följande två instruktioner:

- `["JEQ", r, s, n]` – *Jump if EQual*: Om denna instruktion exekveras på position *pos* och värdet av register `r` är samma som värdet av register `s` (enligt operatorn `==` i Python), ska man därefter hoppa till (och utföra) instruktionen på position *pos+n*, där `n` är ett godtyckligt heltal som även kan vara negativt. Sedan fortsätter exekveringen som vanligt därifrån.

  Annars går man som vanligt till nästa instruktion på position *pos+1*.

- `["JNE", r, s, n]` – *Jump if Not Equal*: Hoppar till instruktion *pos+n* om värdet av register `r` *inte* är samma som värdet av register `s`.

Se också till att exekveringen *avslutas* om man hoppar till en position som är efter slutet av programmet, som i exemplet nedan.

Ytterligare villkor för 2b: Se uppgift 2a.

Exempel:

- ```
  assert eval_pyassm([
      ['ADD', 'A', 1],            # Position 0
      ['MUL', 'A', 10],
      ['ADD', 'B', 1],            # Jump destination
      ['JEQ', 'B', 'A', 100],
      ['LOG', 'B'],
      ['JEQ', 'B', 'B', -3]]) ==  # Always equal: jump 3 steps back
      [('B', 1), ('B', 2), ('B', 3), ('B', 4), ('B', 5), ('B', 6),
       ('B', 7), ('B', 8), ('B', 9)]
  ```

Allmänna tips och ledtrådar:

- En lösning kan vara att använda en *instruktionspekare*, en variabel vars värde är index för nästa instruktion som ska utföras.
- Tänk på att man ska kunna hoppa framåt *och tillbaka* i programmet.
- Exemplet testar inte allt. Till exempel testas inte `JNE`.
  ```

Uppgift 6c: Subrutiner (3p)

Nu är det dags att implementera *subrutiner*. Spara gärna undan din gamla version av pyassm så att du har den kvar om du skulle råka göra några misstag i denna version (men lämna bara in *en* version!). Utöka sedan emulatorn i uppgift 2b med stöd för tre nya instruktioner:

- `["NOP"]`, som inte gör något men kan "fylla ut" ett program.
- `["JSR", n]` – om denna instruktion exekveras på position *pos*, ska positionen *pos+1* sparas undan på något sätt, så att man senare kan *hoppa tillbaka* till den med en `RET`-instruktion. När positionen är sparad ska programmet hoppa till instruktionen på position `n` i programmet och exekveringen fortsätter som vanligt därifrån.
- `["RET"]` – hoppar tillbaka till den senast undansparade positionen.

Ytterligare villkor för 2c: Se uppgift 2a.

Exempel:

- ```
  assert eval_pyassm([
        ['ADD', 'A', 1],        # Position 0
        ['JSR', 5],             # Hoppa till position 5
        ['NOP'],
        ['NOP'],
        ['JEQ', 'B', 'B', 10000],
        ['LOG', 'A'],      # Position 5
        ['RET']                 # Återvänd till 2
      ]) ==
    [('A', 1)]
  ```

Det måste gå att spara ett godtyckligt antal positioner, så att man kan använda `JSR` för att hoppa till en subrutin, hoppa vidare därifrån till en annan subrutin, hoppa vidare ännu fler gånger, och sedan använda `RET` flera gånger för att återvända "ett steg i taget". I detta exempel gör vi hopp i 2 nivåer:

- ```
  assert eval_pyassm([
        ['ADD', 'A', 1],        # Position 0
        ['JSR', 5],
        ['LOG', 'D'],
        ['NOP'],
        ['JEQ', 'B', 'B', 10000],
        ['LOG', 'A'],           # Position 5
        ['JSR', 10],
        ['LOG', 'C'],
        ['RET'],
        ['NOP'],
        ['LOG', 'B'],           # Position 10
        ['RET']
      ]) == [('A', 1), ('B', 0), ('C', 0), ('D', 0)]
  ```