Datortentamen 2022-03-16

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!

Tillåtna hjälpmedel och verktyg

Följande hjälpmedel och verktyg är tillåtna / tillgängliga:

- **Pennor, suddgummin och liknande** för att kunna skissa lösningar på papper. (Lösa tomma papper tillhandahålls av tentamensvakterna.)
- **Python 3.9**, som vi har använt under årets kursomgång. 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, atom, 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/` (via "Web Access"-ikonen). 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, fram till **21:15** (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 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. **Kopiera** mallfilen från `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(). Testkör 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 `tentaklient.pdf`) 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 en 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.

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 flera studenter har missat eller missförstått.

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. Ibland är de skrivna direkt i `assert`-form och ibland beskrivs de indirekt som del av den löpande texten. **Testa dem alla 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 med värden från 0 till 100! Kraschar koden? Skriver den ut uppenbart felaktiga svar? Då har du hittat ett fel.
- 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* vad som egentligen menas.

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 i `run_tests()` är OK, eftersom vi inte kör dessa när vi importerar filerna.)
- 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']`.
- Eftersom listor används spelar också *ordningen* roll: `['13', '14']` är inte samma som `['14', '13']`.

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']}`.

Exempel: Se även de inledande exemplen.

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

Uppgift 2: Merge för 3 listor (5p)

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

MergeSort är en känd sorteringsalgoritm som baseras på principen *söndra och härska*: Man delar upp listor i delar, löser de resulterande delproblemen genom att sortera de mindre listorna var för sig, och slår sedan ihop de resulterande listorna. Det sista steget underlättas då kraftigt av att de listor som ska slås ihop redan är sorterade.

Du ska inte skriva en fullständig sorteringsfunktion. Vi fokuserar istället på en liten del av detta: En funktion som tar två sorterade listor `s1` och `s2` av godtycklig längd, och som på ett specifikt sätt konstruerar en enda sorterad lista som innehåller alla element från dessa.

Funktionen ska finnas i två varianter som ger **vardera 2.5 poäng**, och det är möjligt att få poäng på dem oberoende av varandra.

- En som använder rekursiv lösningsmodell: `merge_r(s1: list, s2: list)`
- En som använder iterativ lösningsmodell: `merge_i(s1: list, s2: list)`

Arbetsgång: Funktionerna ska i varje steg plocka *ett* element från någon av listorna, `s1` eller `s2`. Anropet till `merge_i([1,4,6,7,12], [0,1,10])` ser till exempel att de första elementen i listorna är 1 och 0. Det minsta av dessa är 0, och eftersom listorna är sorterade måste detta också vara det minsta elementet av *alla* element i listorna. Därför ska detta element “plockas" och ska vara första elementet i svaret. Sedan har vi kvar `[1,4,6,7,12]` och `[1,10]`. De första elementen i listorna är då 1 respektive 1. Värdet 1 är då minst, men det spelar inte någon roll vilken lista man plockar det från.

Ytterligare villkor:

- Funktionerna som specificeras i uppgiften får **inte** modifiera sina indata.
- Listor kan innehålla godtyckliga element, inklusive identiska element. Vi garanterar att alla par av element kan jämföras med de vanliga jämförelserna, t.ex. `<` och `<=`. Testa t.ex. med tupler och strängar som element!
- Du ska inte använda existerande sorteringsalgoritmer och inte heller implementera egna sorteringsalgoritmer, utan själv plocka enskilda element i rätt ordning enligt “Arbetsgång" ovan.
- Du får inte använda listbyggare (eng. list comprehensions) eller inbyggda funktioner som behandlar alla element i hela listor. Däremot är givetvis `len()` tillåten.
- 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*. I detta fall ska alltså `merge_r(...)` anropa sig själv med två listor, där en av listorna är kortare än förut. Den ska då få tillbaka ett korrekt *merge*-resultat för dessa två listor.

Fortsätter på nästa sida.

Exempel: Båda funktionerna ska bl.a. klara följande **tester**, där `merge_x` är antingen `merge_i` eller `merge_r`. Skapa gärna egna tester med inspiration av villkoren ovan, och kom då ihåg att parameterlistorna `s1` och `s2` redan ska vara sorterade.

- `assert merge_x([], [1]) == [1]`
- `assert merge_x([1, 2, 8, 13], [3, 5, 21]) == [1, 2, 3, 5, 8, 13, 21]`
- `assert merge_x(['a', 'c'], ['b']) == ['a', 'b', 'c']`

Uppgift 3: Addera element i nästlade listor (5p)

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

I denna uppgift vill vi hantera *listor med samma nästlade struktur*. Med detta menar vi t.ex.:

- `seq1 = [1,2,3],`
  `seq2 = [9,8,7]`
- `seq1 = [[["a"], 6, [2, (3, 5)]]],`
  `seq2 = [[["b"], 5, [1, (1, 1, 42)]]]`

Mer formellt har två listor `seq1` och `seq2` "samma nästlade struktur" om och endast om (1) de är lika långa, och (2) ett av följande villkor gäller för elementen `elem1=seq1[pos]` och `elem2=seq2[pos]` på alla positioner `pos` i de två listorna:

- Elementen är två listor, exempelvis `[2, (3, 5)]` och `[1, (1, 1, 42)]`, som också har *samma nästlade struktur* som varandra, eller
- Inget av elementen är listor (exempelvis `(3, 5)` och `(1, 1, 42)`).

Du ska skriva funktionen `add_nested(seq1: list, seq2: list)`, som tar två godtyckliga (möjligen tomma) nästlade listor `seq1` och `seq2` med samma nästlade struktur och som fungerar enligt följande exempel.

- `assert add_nested([1, 2], [15, 4.25]) == [16, 6.25]`
- `assert add_nested([[['a'], 6, [2, (3, 5)]]], [[['b'], 5, [1, (1, 1, 42)]]]) == [[['ab'], 11, [3, (3, 5, 1, 1, 42)]]]`

Med andra ord ska funktionen traversera båda listorna och *addera* element på motsvarande positioner på alla nivåer: `'a'+'b'` blir `'ab'`. Element som är listor ska givetvis inte adderas, eftersom de är nästlade listor som ska behandlas rekursivt på samma sätt som de ursprungliga listorna. Resultatet ska sedan ha exakt samma nästlade struktur som ursprungslistorna.

Ytterligare villkor:

- Funktionerna som specificeras i uppgiften får **inte** modifiera sina indata.
- Du får förutsätta att addition ('+') alltid fungerar för två element som är på motsvarande positioner i `seq1` och `seq2`. Funktionen kommer alltså inte att testas med anrop som `add_nested([2, 1], [4, "a"])` där det inte går att addera `1` och `"a"`, och koden behöver inte testa om sådana element skickas in utan tillåts att krascha i dessa fall.

Fortsätter på nästa sida.

Allmänna tips och ledtrådar:

- 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.
- Vi behandlar nästlade *listor*. Tupler är inte listor, och som synes i exemplet ovan ska man *inte* rekursera ner i dem.
- Listorna kan innehålla *godtyckliga* element, så länge som elementen på motsvarande positioner kan adderas med varandra. Försök *inte* att specialbehandla några andra typer än just listor.

Uppgift 4: Högre ordningens funktioner (5p)

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

Deluppgift 4a (3p)

Definiera funktionen `make_val_finder(val)`, som tar ett godtyckligt värde `val` och returnerar en ny funktion. Om man anropar den *returnerade* funktionen ska den i sin tur:

- Ta en godtycklig lista `seq` som argument
- Returnera `true` om `val` finns i `seq`, och returnera `false` annars.

Den returnerade funktionen är alltså en *värdeletare* som letar efter ett specifikt värde i en lista, och `make_val_finder` är en funktion som *skapar* en *värdeletare*.

Allmänna tips och ledtrådar:

- Se till att alla funktioner skapas med korrekta parametrar. Som du ser ovan ska både `make_val_finder` och funktionen som `make_val_finder` returnerar ta *en* parameter.
- Detta betyder bland annat att funktionen som returneras inte ska ta `val` som parameter. Det var ju `make_val_finder` själv som tog `val` som parameter.

Exempel:

- `assert not make_val_finder(10)([1,2,3])`
- `assert make_val_finder(10)([1,10,5])`
- `assert make_val_finder("x")([1,10,"x"])`

I första exemplet är `make_val_finder` en funktion, men även `make_val_finder(10)` är en (returnerad) funktion. Den appliceras sedan på `seq=[1,2,3]`.

Ytterligare villkor:

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

Deluppgift 4b (2p)

Använd funktionen `make_val_finder` för att skapa funktionen `contains_14(seq: list)`, som tar en godtycklig lista som parameter och returnerar sant om och endast om parametern `seq` innehåller heltalsvärdet `14`.

För full poäng ska funktionen `make_val_finder` inte anropas inuti `contains_14`. Istället ska `contains_14` *vara* en funktion som skapades och returnerades av `make_val_finder` med en lämplig parameter.

Exempel:

- `assert not contains_14([1,2,3])`
- `assert contains_14([1,14,"x"])`

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, och att anta att indata är korrekta (“plus" får t.ex. krascha om den ges två matriser av olika storlek).

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:

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

- `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:

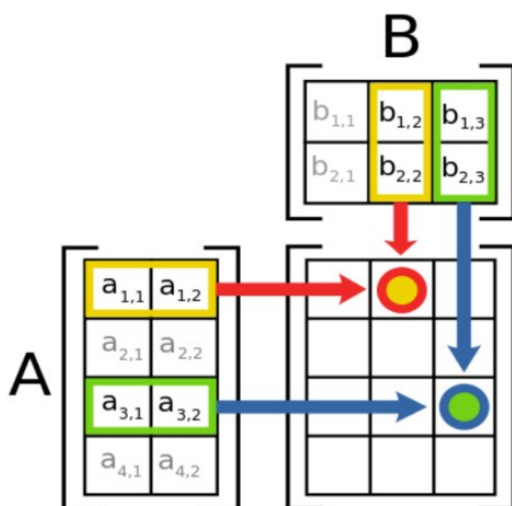
$$c_{i,j} = a_{i,1}b_{1,j} + a_{i,2}b_{2,j} + \ldots + a_{i,n}b_{n,j} = \sum_{r=1}^{n} a_{i,r}b_{r,j}$$

  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. 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: Deranged anagrams (5p)

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

Ett *anagram* av ett ord är ett annat (annorlunda) ord som man har fått fram genom att kasta om bokstäverna i det ursprungliga ordet. Ordet *ettan* är t.ex. ett anagram av *tenta*.

Vad är då ett ord? Normalt handlar det egentligen om ord från ett riktigt språk, till exempel svenska. I den här uppgiften vill vi inte använda någon ordlista för att hålla reda på giltiga ord, så vi definierar ett **ord** som en *sträng* med minst 1 tecken (inte tomma strängen), där alla tecken i strängen är “små" bokstäver från det svenska alfabetet: *a-ö*. Då räknas alltså “orden" *etnat* och *ttnae* som anagram till *ettan*.

Ett anagram är *deranged* om inga bokstäver finns på samma plats som i det ursprungliga ordet. I vårt exempel är *ettan* ett *deranged anagram* av *tenta*. Däremot är *etnat* och *ttnae* inte *deranged anagrams* av *tenta*: Både *etnat* och *tenta* har *n* på tredje positionen, och både *ttnae* och *tenta* har *t* på första positionen.

Du ska nu skriva fyra funktioner som har att göra med *deranged anagrams*. **Du kan få delpoäng även om du bara implementerar vissa av funktionerna**; varje funktion testas alltid tillsammans med facit-implementationen av de övriga funktionerna. Termerna *ord*, *anagram* och *deranged anagram* definieras ovan!

- `is_anagram(word1: str, word2: str)` ska returnera sant om och endast om de två godtyckliga *orden* `word1` och `word2` är *anagram* av varandra.
- `is_deranged_anagram(word1: str, word2: str)` ska returnera sant om och endast om de två godtyckliga *orden* `word1` och `word2` är *deranged anagrams* av varandra.
- `all_anagrams(word: str)` ska returnera en lista som innehåller exakt de ord som är anagram för det godtyckliga ordet `word`, utan dubletter. Tänk på att ett ord inte räknas som anagram till sig själv.

  Listan ska inte innehålla dubletter. Det kan vara enklast att åstadkomma det genom att först skapa en provisorisk lista, som kan innehålla dubletter, och sedan filtrera den.

  Funktionen ska *inte* fungera genom *generate-and-test* – till exempel ska den inte först generera ord som “aaa", “aab", “aac" och sedan testa om dessa är anagram. Detta tar alltför lång tid, speciellt med långa ord. Den ska istället på ett kombinatoriskt sätt generera alla anagram med utgångspunkt i ordet `word`.
- `all_deranged_anagrams(word: str)` ska använda de tidigare funktionerna för att skapa och returnera en lista som innehåller samtliga ord som är *deranged anagrams* för det godtyckliga ordet `word`.

Allmänna tips och ledtrådar:

- Kom ihåg att vi inte kräver att strängarna är verkliga svenska ord för att de ska räknas som anagram.
- Att hitta alla möjliga anagram är en **kombinatorisk** uppgift där man behöver testa

"alla alternativ". Det finns flera sätt att tänka här – man hittar alla sätt att *kasta om* bokstäverna, eller alla sätt att *välja* en bokstav i taget ur det ursprungliga ordet (börjar man med "tenta" kan man i första steget välja "t", "e", "n", "t" eller "a").

- Rekursion är ofta ett lämpligt verktyg i kombinatoriska uppgifter. Man *måste* dock inte ha en *rekursiv lösningsmodell*. Det *går* också att lösa detta iterativt, men då genom tekniker vi inte har lärt ut i kursen.

Exempel: För de funktioner som returnerar listor använder vi `sorted()` nedan. Detta ser till att ordningen i listan inte spelar någon roll.

- `assert not is_anagram("ab", "ac")`
- `assert not is_anagram("ab", "cb")`
- `assert is_anagram("ab", "ab")`
- `assert is_anagram("ab", "ba")`
- `assert is_anagram("abc", "bca")`
- `assert is_anagram("abc", "cba")`
- `assert is_anagram("themorsecode", "herecomedots")`
- `assert not is_deranged_anagram("ab", "ac")`
- `assert not is_deranged_anagram("ab", "cb")`
- `assert not is_deranged_anagram("ab", "ab")`
- `assert is_deranged_anagram("ab", "ba")`
- `assert is_deranged_anagram("abc", "bca")`
- `assert not is_deranged_anagram("abc", "cba")`
- `assert not is_deranged_anagram("themorsecode", "herecomedots")`
- `assert sorted(all_anagrams("a")) == []`
- `assert sorted(all_anagrams("ab")) == ['ba']`
- `assert sorted(all_anagrams("abc")) == ['acb', 'bac', 'bca', 'cab', 'cba']`
- `assert sorted(all_anagrams("abcd")) == ['abdc', 'acbd', 'acdb', 'adbc', 'adcb', 'bacd', 'badc', 'bcad', 'bcda', 'bdac', 'bdca', 'cabd', 'cadb', 'cbad', 'cbda', 'cdab', 'cdba', 'dabc', 'dacb', 'dbac', 'dbca', 'dcab', 'dcba']`
- `assert sorted(all_anagrams("abb")) == ['bab', 'bba']`
- `assert sorted(all_anagrams("abbc")) == ['abcb', 'acbb', 'babc', 'bacb', 'bbac', 'bbca', 'bcab', 'bcba', 'cabb', 'cbab', 'cbba']`
- `assert sorted(all_deranged_anagrams("a")) == []`
- `assert sorted(all_deranged_anagrams("ab")) == ['ba']`
- `assert sorted(all_deranged_anagrams("abc")) == ['bca', 'cab']`
- `assert sorted(all_deranged_anagrams("abcd")) == ['badc', 'bcda', 'bdac', 'cadb', 'cdab', 'cdba', 'dabc', 'dcab', 'dcba']`
- `assert sorted(all_deranged_anagrams("abb")) == []`
- `assert sorted(all_deranged_anagrams("abbc")) == ['bacb', 'bcab']`
- `assert len(all_anagrams("abcdddefgh")) == 604799`
- `assert len(all_deranged_anagrams("abcdddefgh")) == 97650`