Datortentamen 2020-03-17

TDDE24 Funktionell och imperativ programmering del 2

Uppgifter, poäng och betyg

Denna tenta består av totalt **6 uppgifter** som vardera kan ge maximalt 5 poäng. Alla uppgifter är utvalda för att testa olika aspekter av kursinnehållet. De är också tänkta att ha olika svårighetsgrad och är ordnade så att svårigheten oftast ökar mot slutet. Vad som är lätt eller svårt skiljer sig naturligtvis från person till person, så det är alltid en bra idé att snabbt läsa igenom alla uppgifterna för att bättre kunna prioritera arbetet.

För betyg 3, 4 och 5 krävs minst 12, 19 respektive 25 poäng.

Använd Python 3, inte Python 2!

Vi har undervisat i Python 3 i kursen. Detta ska finnas tillgängligt via kommandot `python3`. Använd detta kommando istället för bara `python`.

Är du osäker på vilken version som används av en editor/miljö: `assert 5/2==2.5` fungerar i Python 3, men ger fel i Python 2. Lägger du detta först i källkoden ser du om miljön använder rätt version. Kör annars testerna med `python3` på kommandoraden.

Datormiljö, tentaklient och inlämning

Under datortentan arbetar du i en begränsad och övervakad datormiljö. Med hjälp av en särskild **tentaklient** skickar du in dina svar, med en separat inlämning per uppgift. Varje uppgift får **skickas in en enda gång**. Man kan alltså inte uppdatera sina svar. Precis som vid vanliga tentor kommer alla svar att granskas efter tentans slut.

Du använder tentaklienten för att **ställa tentarelaterade frågor**. Dessa går antingen till examinatorn eller till jourhavande, som vid behov kan vidarebefordra dem. På detta sätt får man oftast betydligt snabbare svar än på en “vanlig" tenta, där man normalt sparar sina frågor till examinatorn besöker salen och alltså kan få vänta ett par timmar. Det finns dock ingen garanti att man får ett *omedelbart* svar, speciellt om många ställer frågor på samma gång. **Läs alltså genom uppgifterna i förväg och ställ frågor i god tid, precis som på en vanlig tenta!** Har du inte fått svar inom **30 minuter**, skicka en gång till.

Besök i skrivsalen kommer endast att ske vid allvarligare tekniska problem.

Namnge de inlämnade filerna ex1.py, ex2.py, . . . , ex6.py!

Detta underlättar rättningen. Ange inte (a) eller (b) i filnamnet.

Gör en separat inlämning per uppgift (ex1, . . . , ex6)!

När du gör en inlämning i tentaklienten anger du också *vilken* uppgift du lämnar in. I uppdelade uppgifter lämnas dock del (a) och del (b) in på samma gång.

Hjälpmedel och verktyg

Följande fysiska hjälpmedel är tillåtna:

- Pennor, radergummin och liknande för att kunna skissa lösningar på papper. (Lösa tomma papper tillhandahålls av tentamensvakterna.)

Följande elektroniska hjälpmedel är tillgängliga:

- Systemprogram i kommandoradsprompten eller menyerna. Som diskuterats i förväg inkluderar detta flera olika editorer, men de har inte nödvändigtvis 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.
- Websidor under `https://docs.python.org/3/` (via startmenyn). Detta inkluderar både biblioteks- och språkreferenser, med vissa undantag för sidor med kodexempel.

Icke tillåtna hjälpmedel inkluderar bl.a. alla former av elektronisk utrustning utöver tentadatorerna, samt böcker och anteckningar.

Konfigurationsfiler: Vi erbjöd en möjlighet att få med "de nödvändigaste delarna av enklare konfigurationsfiler" till tentan. Enbart en sådan konfiguration har kommit in:

```
setxkbmap -option caps:swapescape
```

Viktiga tips – missa inte poäng i onödan

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

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

Rättningskriterier

Brott mot följande allmänna kriterier kan resultera i poängavdrag.

- Programkod ska vara **körbar**. Testa alltid **precis innan inlämning** så att din sista finputsning eller dina sista kommentarer inte gav felaktig kod! Poängavdrag ges vid icke körbar kod, även för *delvis* körbar kod (som kraschar för vissa fall).
- Man ska kunna köra funktioner **flera gånger** med korrekt resultat, utan att ladda om koden däremellan. Se upp med olika former av globalt tillstånd / globala variabler.
- Funktioner ska ha **exakt samma namn** som i uppgiften. Detta underlättar rättningen.
- **Namn** på parametrar, variabler och hjälpfunktioner som inte specificerats explicit i uppgiften ska vara **beskrivande** och följa **namnstandarden**.
- Lösningen ska vara **välstrukturerad och väldokumenterad** på samma sätt som under kursens laborationer. Detta inkluderar att skriva svenska eller engelska **docstrings** för **samtliga funktioner som definieras med `def` på "toppnivån**", även hjälpfunktioner som man själv inför. Specialtaggar som `:param:` krävs *inte*.

  Det är *inte* ett absolut krav att skriva docstrings för lambda-uttryck eller för nästlade funktioner som definieras inuti andra funktioner. Tänk dock på att koden ändå behöver vara tillräckligt väldokumenterad för att vara lätt att förstå.
- Lösningen ska givetvis följa de **specifika regler och villkor** som uppgiften har satt upp. Lösningen ska också **fungera exakt som i körexemplen** i uppgiften, om inte texten indikerar något annat.
- Lösningen ska vara **generell**: Den ska inte bara fungera för de körexempel som anges, utan för alla indata som följer uppställningen i frågan. Att en lösning enbart fungerar för listor med begränsad längd eller för vissa storlekar på indata är ett signifikant fel.
- Den implementerade lösningen ska kunna köras inom **rimliga tidsramar**. Till exempel accepteras inte en lösning som tar 1 minut för att konkatenera två strängar som är 4 bokstäver långa. Eventuella undantag anges uttryckligen i frågan.
- Om inte annat sägs ska funktioner **returnera** värden, inte skriva ut dem. Eventuella testfall visar normalt returnerade värden, inte värden som funktionerna har skrivit ut.

Avdrag för felaktig och ofullständig kod ges **även om lösningen är uppenbar** för granskaren, och även om det syns att lösningen "var på rätt väg". Det ingår i uppgiften att se till att koden uppfyller specifikationen, och att *själv* upptäcka och åtgärda problem under tentans gång. Den inlämnade koden är ditt slutgiltiga svar, inte ett mellansteg som granskaren ska arbeta vidare med.

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

Tillåtet / icke-krav

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

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

  Funktioner måste alltså ge *korrekta svar för korrekta indata* enligt uppgiften, men om inte annat anges får de *krascha eller ge felaktiga svar för felaktiga indata* (såsom när en funktion som bara ska hantera heltal ges en sträng som parameter).
- Att använda godtyckligt långa rader. Vi ger inga avdrag för radlängder som bryter mot standarden.
- Att lämna in, eller *inte* lämna in, sina testfall (tillåtet men krävs inte).

Detta gäller inte om uppgiften säger något annat!

Att avsluta tentan

Repeterad information från instruktionerna du läste innan du kom till tentan:

- Save all your work on your desktop. Be aware most other folders are read-only. You may create folders for your own convenience but any problem you encounter as a result is your responsibility.
- When you have submitted/handed in all solutions you intend and got the confirmation message “request received" for each of them you should log out. The safest procedure is to save all files and close every open application window one by one until all are closed. Then log out from the main menu and wait until you see the normal login screen. Now you can leave.

Uppgift 1

Begreppet *round robin* kommer från början från *round ribbon*, där man på 1700-talet skrev under petitioner på ett cirkulärt band så att ingen av signaturerna skulle kunna identifieras som den första, “ledaren". Detta begrepp används också i överförd mening inom flera datavetenskapliga områden.

Skriv en funktion `roundrobin(seq)`, som tar en lista `seq=[`$\texttt{seq}_1$`,...,`$\texttt{seq}_n$`]` av $n \geq 1$ godtyckliga sekvenser. Funktionen ska returnera en ny sekvens som innehåller det första elementet från $\texttt{seq}_1$, det första elementet från $\texttt{seq}_2$, ..., det första elementet från $\texttt{seq}_n$, det andra elementet från $\texttt{seq}_1$, det andra elementet från $\texttt{seq}_2$, och så vidare. Varje sekvens får alltså chansen att bidra med ett element i tur och ordning.

När en sekvens $\texttt{seq}_i$ inte har ett element med ett visst index (“elementen tar slut i sekvensen"), hoppas den sekvensen över och man går vidare till nästa (se andra exemplet nedan). När elementen “tar slut" i samtliga sekvenser returneras resultatet.

Exempel:

- `roundrobin([[1,2,3], [4,5,6], [9,8,7]]) == [1,4,9,2,5,8,3,6,7]`
- `roundrobin([[1,2,3], [0], ['abc',6]]) == [1,0,'abc',2,6,3]`
- `roundrobin([[]]) == []`

Ytterligare villkor:

- Lösningen får **inte** använda funktionalitet från modulen `itertools`.
- Dina funktioner får **inte** modifiera sina indata.

Sista påminnelsen

- Dokumentera funktioner med docstrings!
- Skapa många egna tester – prova med många olika indata!
- Gör aldrig någon ändring utan att testa, även när du *tror* att ändringen var harmlös. Testa omedelbart före inlämningen.

Namnge de inlämnade filerna ex1.py, ex2.py, ..., ex6.py!

Uppgift 2

Det finns många applikationer där man vill samla in och analysera data från olika sensorer.

Anta att varje sådan sensor skickar information som tupler (n, v) där första elementet n i tupeln är det unika namnet på sensorn (en sträng) och det andra elementet v är det aktuella mätvärdet (ett tal). Anta vidare att mottagaren tar emot sådana tupler från ett antal sensorer och att dessa hamnar i en lista i tidsordning, inte separerat per sensor:

```
[('a', 2), ('b', 0), ('a', 6), ('c', 0), ('b', 1)]
```

Din uppgift är att skriva två olika implementationer av funktionen `by_sensor(seq)`, som tar en lista med $m \geq 0$ sådana mätvärdestupler och returnerar en dictionary, där varje sensornamn n som uppträder i `seq` associeras med en lista på de mätningar som har kommit in för denna sensor. Mätningarna för en given sensor ska komma i samma ordning som de gjorde i `seq`.

Exempel:

- `by_sensor([('a', -1), ('a', 1)]) == {'a': [-1, 1]}`
- `by_sensor([('a', 2), ('b', 0), ('a', 6), ('c', 0), ('b', 1)]) == {'a': [2, 6], 'b': [0, 1], 'c': [0]}`

De två implementationerna av `by_sensor` ska fungera enligt beskrivningen ovan men använda olika lösningsmodeller:

- `by_sensor_i(seq)` ska arbeta enligt **iterativ** lösningsmodell.
- `by_sensor_r(seq)` ska 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. Ett exempel är fakultetsfunktionen där `fac(n) = n * fac(n-1)`: Här är `fac(n-1)` ett rekursivt anrop som *också* beräknar just fakultetsfunktionen, men för ett mindre värde på `n`. Delproblemet har alltså exakt samma definition som det ursprungliga problemet, men ges andra parametrar.

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

Ytterligare villkor:

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

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

- Dina funktioner får **inte** modifiera sina indata. Dock får man så klart modifiera utdata, alltså resultatet man får från en funktion.

Uppgift 3

Pekare är vanliga inom programmering för att hänvisa till data som finns på en annan plats. Pekare kan användas till mycket, t.ex. för att komprimera en text. Istället för att lagra varje tecken kan texten innehålla pekare till textstycken som då kan återanvändas flera gånger.

Din uppgift är att skriva två funktioner som expanderar en text som innehåller pekare. Pekarna är helt enkelt index i en lista, som då fungerar som ett “minne".

Funktionerna ska ta en lista `mem` med strängar man kan peka på, och en godtycklig nästlad lista `msg` där elementen är strängar, heltal, eller nästlade listor (där elementen i sin tur är strängar, heltal, eller nästlade listor, osv). Ett heltal i `msg` pekar ut alltså ut en sträng genom att ange dess position i `mem`, medan en sträng används som den är, i de fall ett ord inte redan finns lagrat i minnet.

Alla listor kan ha godtycklig längd. Du kan anta att alla pekare är giltiga (att det faktiskt finns ett element med det angivna indexet i `mem`).

I exemplen nedan antar vi:

```
mem = [' ','att','lycka','tenta','till','på','är','kanske','tentan']
```

Ytterligare villkor för både 3a och 3b:

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

Uppgift 3a: Direkt ersättning (3p)

I första steget skriver du funktionen `expand(mem, msg)`, som i princip returnerar resultatet av att byta ut heltalen i `msg` mot de strängar som heltalen pekar ut i `mem`. Denna funktion ska alltså bevara nästlingsstrukturen hos `msg`.

Exempel utan och med nästling:

- `expand(mem, [2, 6, 1, 3]) == ['lycka', 'är', 'att', 'tenta']`
- `expand(mem, [2, 4, 'med', 8]) == ['lycka', 'till', 'med', 'tentan']`
- `expand(mem, [2, 6, [7, 'att', []], 3]) ==`
  `['lycka', 'är', ['kanske', 'att', []], 'tenta']`

Fortsätt med uppgift 3b på nästa sida

Uppgift 3b: Ersättning med konkatenering (2p)

Nu tar vi uppgiften ett steg till. Funktionen `expand_concat(mem, msg)` ska fungera som `expand()`, men varje *sammanhängande (kontinuerlig) delsekvens* av minst 1 heltal och/eller sträng (på samma nivå i listan) ska ska resultera i *en* ny *sammansatt* (konkatenerad) sträng. Sådana delsekvenser kan alltså “brytas" av en nästlad lista.

Som förut ska funktionen bevara nästlingsstrukturen hos `msg`.

Exempel utan och med nästling:

- `expand_concat(mem, [2, 6, 1, 3]) == ['lyckaäratttenta']`
- `expand_concat(mem, [2, 0, 6, 0, 1, 0, 3]) == ['lycka är att tenta']`
- `expand_concat(mem, [2, 0, 6, [7, 0, 'att', []], 3, 0]) == ['lycka är', ['kanske att', []], 'tenta ']`
- `expand_concat(mem, [[[3, 3, [], [], 3]]]) == [[['tentatenta', [], [], 'tenta']]]`

Varje nästlad lista hanteras separat och nästlingsnivåerna bevaras i resultatet. Tomma listan resulterar enbart i tomma listan, inte en tom sträng. Om inga strängar eller heltal finns mellan två nästlade listor, ska ingen sträng heller adderas (som i sista exemplet ovan).

Uppgift 4

Deluppgift 4a (3p)

Skriv en högre ordningens funktion `pred_comp(p, t, f)` som tar tre funktioner `p`, `t` och `f` som indata och returnerar en ny funktion. Funktionerna `p`, `t` och `f`, som kommer att anges av anroparen, kommer var och en att ta *ett* argument. Den returnerade funktionen ska också ta ett argument `x` och den ska returnera `t(x)` om `p(x)` är sant, annars `f(x)`.

Ytterligare villkor:

- Inga specifika villkor för denna uppgift.

Exempel:

- `pred_comp(lambda x: x > 0, lambda x: x, lambda x: -x)(-4) == 4`
- `pred_comp(lambda x: x < 0, lambda x: x, lambda x: -x)(-4) == -4`

Deluppgift 4b (2p)

Skapa en funktion för att utföra säker division av `x/y`, dvs. en funktion som kan hantera fallet när `y==0`. Vid försök till division med 0 ska funktionen returnera 0.

Funktionen ska heta `safe_div(div)` och ska ta *ett* argument, som är *tupeln* `div` $= (x, y)$. Den ska definieras med hjälp av `pred_comp`.

Funktionen ska utföra flyttalsdivision. Var extra noga med att du använder `python3` när du testar uppgiften, eftersom divisionsoperatorn skiljer sig åt mellan Python 2 och 3.

Ytterligare villkor:

- Funktionen `pred_comp` ska bara anropas en gång: För att *skapa* den nya funktionen `safe_div`. Sedan ska man kunna *anropa* `safe_div` hur många gånger som helst, med olika parametrar, utan att `pred_comp` anropas igen.

  Om du är osäker på vad som händer, lägg in en tillfällig utskrift i `pred_comp` för att se varje anrop, och anropa sedan `safe_div` flera gånger.

Exempel:

- `safe_div((10, 5)) == 2`
- `safe_div((10, 4)) == 2.5`
- `safe_div((10, 0)) == 0`

Uppgift 5

Nu ska vi titta på ett problem som kan vara mindre vanligt nuförtiden, men ändå är algoritmiskt intressant: Att ta reda på hur man ger *önskad växel* med *minimalt antal mynt*.

Anta att det existerar mynt i $n \geq 1$ heltalsvalörer `[v1,v2,...,vn]` enheter, där vi vet att `v1` $>$ `v2` $> \ldots >$ `vn` $>$ `0`. Som ett praktiskt exempel kan vi ta de svenska mynten, som har de 4 valörerna `[10,5,2,1]` kronor. För att ge 53 kronor i växel med minimalt antal mynt ska vi ge 5 st 10-kronor, 1 st 2-krona och 1 st 1-krona. Detta kan vi representera som svarslistan `[5,0,1,1]`, där antalen har samma index (står på samma position) som i listan över valörer. Svaret `[5,0,0,3]` vore felaktigt trots att det också resulterar i 53 kronor, eftersom det använder onödigt många mynt.

Om det istället finns mynt i valörerna `[4,3,1]` kronor, och vi vill ge `6` kronor i växel, får vi minimalt antal mynt genom att använda 2 st 3-kronor. Svaret blir `[0,2,0]`.

Om det finns mynt i valörerna `[10,5]` kronor, och vi vill ge tillbaka `3` kronor i växel, är detta omöjligt. Detta kan representeras som en svarslista som någonstans innehåller positiva oändligheten, representerad som `float('inf')`. Detta diskuteras mer senare.

Uppgift:

Skriv funktionen `min_change(denominations,change)`, som tar en sorterad lista med unika positiva heltalsvalörer `denominations=[v1,v2,...,vn]` och ett heltalsbelopp `change`$>$`0` enligt ovan, och returnerar en lista av längd `n` på antalet mynt per valör – eller en lista som någonstans innehåller positiva oändligheten för att indikera att växlingen är omöjlig, enligt ovan. Svaret ska minimera *antalet* mynt i växel.

Exempel:

- `min_change([10, 5, 2, 1], 53) == [5, 0, 1, 1]`
- `min_change([4, 3, 1], 6) == [0, 2, 0]`
- `float('inf') in min_change([10, 5], 3)` # Svarslistan innehåller oändligheten

Ytterligare villkor:

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

Ytterligare information och tips:

- En rekursiv implementation, där varje rekursivt anrop löser ett eller flera äkta delproblem, är troligen enklast. Lösningen måste inte vara särskilt effektiv, utan kan testa olika alternativ (“brute force").
- Tänk noga på vad man kan använda som delproblem. Tips: Det första myntet man väljer är antingen av den största typen eller av någon annan typ. Hur fortsätter man lösningsgången i dessa fall?
- Vi har alltså valt att använda *en lista med tal som innehåller float('inf')* för att representera ett svar på ett omöjligt problem (eller delproblem). Till exempel kan denna

lista bara vara `[float('inf')]`, eller kanske `[float('inf'), 4, 0, 2]`. I detta fall behöver antalet element inte vara samma som antalet valörer i `denominations`.

Det finns en god anledning till denna representation: Summan av myntantalen i en sådan lista blir alltid oändlig, och alltså större än för alla *fungerande* lösningar, som ju har ändligt antal mynt. Detta kan underlätta jämförelsen mellan olika dellösningar.

Man behöver använda `float('inf')` i koden, men detta *skrivs ut* som `inf` när du tittar på resultatet.

- I vissa fall finns det flera möjliga svar som minimerar antalet mynt. Då spelar det ingen roll vilket av dessa korrekta svar som returneras.
- Notera att man antar att inga mynt “tar slut", utan att det alltid finns tillräckligt många mynt i “kassan".

Uppgift 6

Matriser spelar en väldigt stor och viktig roll i både matematiken och datavetenskapen. En matris består av ett antal rader där varje rad har lika många element (kolumner). Följande matris, som kan skrivas med runda parenteser eller hakparenteser, är en 4×3-matris.

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

Indexering börjar på rad/kolumn 1. Elementet $A[2, 3]$, som ofta benämns $a_{2,3}$, är 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:

- Glöm inte dokumentationen för varje funktion.
- List comprehensions är uttryckligen tillåtna för denna uppgift.

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 `fun` har applicerats på varje element i `matrix`. Se körexemplet i slutet av uppgiften.

- `plus(matrix1, matrix2)` – Returnera en ny matris som är summan av `matrix1` och `matrix2`.

  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(matrix1, matrix2)` – returnera en ny matris av rätt dimension, som är produkten av `matrix1` och `matrix2`.

  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 `matrix1` och `matrix2` har "rätt" dimensioner enligt ovan, så att produkten är definierad!

  Om $C = AB$, så gäller:

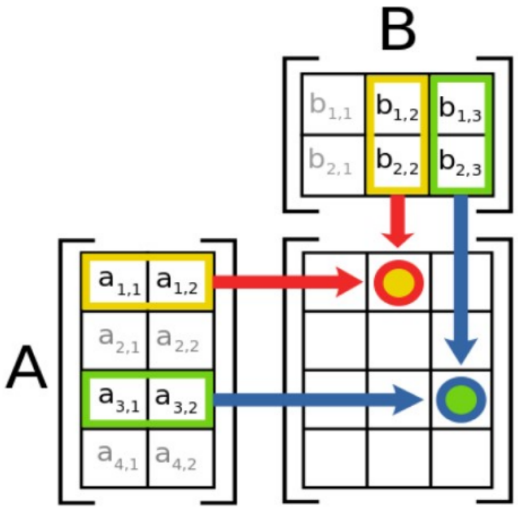
$$c_{i,j} = a_{i,1}b_{1,j} + a_{i,2}b_{2,j} + \ldots + a_{i,n}b_{n,j} = \sum_{r=1}^{n} a_{i,r}b_{r,j}$$

  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]]
>>> rows(m1)
2
>>> columns(m1)
3

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

>>> m3 = [[3, 1], [2, 1], [1, 0]]
>>> plus(m2, m3)
[[4, 0], [2, 4], [3, 1]]

>>> times(m1, m3)
[[5, 1], [4, 2]]

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