Datortentamen 2021-01-12

TDDE24 Funktionell och imperativ programmering del 2

Distanstentamen: Flera skillnader

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

Distanstentamen: Tillåtna och otillåtna hjälpmedel

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

Distanstentamen: Grundläggande skillnader i uppgifter

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

- Då vi inte följer tidigare tentamensstruktur försöker vi inte heller ordna uppgifterna i svårighetsordning.
- **Poänggränser förutbestäms inte.** Det kan man bara göra när man har *mycket* erfarenhet av hur poängen motsvarar de faktiska kunskaperna som vi ska examinera. Examinatorns uppgift är alltid att bedöma uppfyllande av kursmål, och förutbestämda poänggränser är bara en av många olika metoder att göra detta.

Distanstentamen: Datormiljö

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

- Du ska använda **Python 3**, som vi har undervisat i.

  Detta kan finnas som kommandot `python3` på din dator. Det kan också finnas som kommandot `python` – testa då med `python --version` så att det verkligen är en version av Python 3 och **inte gamla Python 2**.

  Du är själv ansvarig för att installera Python 3 om det inte redan finns på din dator. Detta är ditt eget ansvar och ska ha förberetts i god tid inför tentan. För Windows kan du *antagligen* använda "installer"-filerna på slutet av `https://www.python.org/downloads/release/python-387/` eller `https://www.python.org/downloads/release/python-391`. Troligen vill du ha en x86-64-baserad installerare då du antagligen har en 64-bitars CPU.

  Mer detaljer: Vi kommer att testa koden under **Python 3.9**, så det är OK att använda nya features upp till och med denna version. Det går också bra att använda Python 3.8, som har funnits tillgänglig som en kursmodul, eller Python 3.7 / 3.6, som har varit "standard" utan kursmodul. Använder du ännu tidigare versioner av Python 3 ska det förhoppningsvis också fungera, men skriv då en kommentar om din exakta version av Python (enligt `--version`) i varje inlämnad fil.

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

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

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

Frågor och information under tentans gång

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

Spara inte frågorna till slutet. På en vanlig tenta går man genom alla uppgifter så snart man får tillgång till dem, så man kan ställa alla eventuella frågor till examinatorn vid ett eller två korta besök i tentasalen. Under denna distanstenta kan du visserligen skicka frågor när som helst, men även här kommer frågorna att besvaras “då och då" och **det kan ta en timme att få svar** – dels går det inte att spendera varje sekund klistrad framför epostklienten, dels kan det komma många frågor på samma gång.

Skickar du frågor för sent kan det hända att du inte hinner få svar i tid för att avsluta din lösning innan tentan avslutas!

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

Information som är intressant för flera tentander kan skickas ut via kursens epostlista under tentans gång. Även de som omtentar ska finnas med på den listan. Håll alltså koll på din egen epost under tentan.

Inlämning av svar

Inlämning sker via Lisam. Vi använder inte “quiz"-systemet med en separat fråga per uppgift i tentan, utan “inlämningssystemet" där man laddar ner tentan i PDF-format och gör **en inlämning för hela tentan**.

I inlämningen kan man skriva ett textsvar, men den texten kan vara tom. Det viktiga är möjligheten att **bifoga filer**, vilket man ska använda för att bifoga **en separat fil per uppgift** (`ex1.py`, `ex2.py`, `...`).

Man kan lämna in svar flera gånger, men måste då lämna HELA sitt svar (alla filer)! Utnyttja det gärna för att gradvis uppdatera dina svar med nya lösningar, ifall något t.ex. skulle gå fel med din uppkoppling till Internet mot slutet av tentatiden.

Varje svar ersätter helt det tidigare svaret, så om du först lämnar in `ex1.py`, ska samma fil `ex1.py` **skickas med även i nästa inlämning**!

Namnge filerna ex1.py, ex2.py, ex3.py, ...!
Detta underlättar rättningen. Uppgifter som har a- och b-delar ska också lämnas in i samma fil (3a och 3b i ex3.py).

Deadlines

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

De som har *utökad skrivtid* till 15:00 har en egen inlämningslänk med en slutlig hård **deadline 15:10**.

Om inlämningssystemet av någon anledning skulle släppa genom en försenad uppdatering av en inlämning, räknas bara eventuella tidigare inlämningar som gjordes innan deadline.

Vid tekniska problem

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

Om du inte skulle få kontakt med Lisam under dina sista 10 minuter går det också att lämna in via epost till adressen `jonas.kvarnstrom@liu.se`. Du har samma deadline som anges ovan, och epost ska skickas från din universitetsadress.

Detta är bara för nödfall. Normalt ska inlämningar lämnas via inlämningssystemet! När du lämnar in via epost krävs en hel del extra arbete, och du missar möjligheten att vara anonym.

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 uppgift, ibland i `assert`-form och ibland 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, så testa med olika listor, även sådana med nästlade listor eller tupler eller andra elementtyper. 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.

Testning är extremt viktig!

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

Men testning fångar inte allt!

Grunden i uppgifterna är ändå 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 läses in, t.ex. i kvarlämnade `assert`-satser! Poängavdrag ges vid icke körbar kod, även för *delvis* körbar kod (som kraschar för vissa fall).
- 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.
- 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. Detta är en ändring från tidigare år.
- 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. Eventuella testfall i tentatexten visar normalt returnerade värden, inte värden som funktionerna har skrivit ut.
- Man får använda Pythons standardbibliotek (upp till och med Python 3.9), men **inte bibliotek som måste installeras separat**. Se upp om du har installerat extra bibliotek på din dator. Du får inte heller skriva av andras kod.

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 med 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).
- Att importera och använda vanliga “inbyggda" funktioner från Pythons standardbibliotek (upp till och med Python 3.9), 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 bryta mot “ytliga" kodningsstandarder i fråga om t.ex. mellanrum, indentering, radlängd och antal blankrader, så länge som koden fortfarande är *lättläst och lättförståelig*.

Detta gäller inte om uppgiften säger något annat!

Uppgift 1: Fyllda ovaler (3p)

Vi vill kunna rita ut *fyllda ovaler* med olika storlekar. Tyvärr har vi bara tillgång till en textterminal, så vi får använda "`X`" för punkter som ligger inuti ovalen och "`.`" för punkter som ligger utanför.

Implementera därför en funktion `oval(w,h)` där parametrarna är två heltal `w`>=0 och `h`>=0 motsvarande bredden och höjden på den önskade ovalen. Funktionen ska **returnera en lista** med h element ("rader") indexerade $0..h-1$, där varje element i sin tur är en lista med w element indexerade $0..w-1$ ("kolumner") som är strängen `'X'` om punkten ligger *inuti ovalen* eller strängen `'.'` om punkten ligger *utanför ovalen*. Exempelvis ska `oval(4,3)` ge:

```
[
    ['.', 'X', 'X', '.'], # rad 0
    ['X', 'X', 'X', 'X'], # rad 1
    ['.', 'X', 'X', '.']  # rad 2
]
```

För att beräkna om en viss punkt på en viss rad `row` och kolumn `col` ligger inuti ovalen ska följande funktion användas. Kopiera/klistra den exakt!

```
def inside_oval(width, height, row, col):
    row = row - width / 2 + 0.5
    col = col - height / 2 + 0.5
    return row * row / (width * width) + col * col / (height * height) <= 0.25
```

Extra information utskickad under tentans gång:

Definitionen ovan hade omkastade värden för rad och kolumn. Korrekt definition skickades ut och inga poäng drogs av om man hade använt den ursprungliga definitionen. Korrekt version är:

```
def facit_inside_oval(width, height, col, row):
    row = row - height / 2 + 0.5
    col = col - width / 2 + 0.5
    return row * row / (height * height) + col * col / (width * width) <= 0.25
```

Exempel:

- `assert oval(0,1) == [[]]`
- `assert oval(1,1) == [['X']]`
- `assert oval(3,3) == [['X', 'X', 'X'], ['X', 'X', 'X'], ['X', 'X', 'X']]`
- `assert oval(4,3) == [['.', 'X', 'X', '.'], ['X', 'X', 'X', 'X'], ['.', 'X', 'X', '.']]`
- `assert oval(5,6) == [['.', 'X', 'X', 'X', '.'], ['X', 'X', 'X', 'X', 'X'], ['X', 'X', 'X', 'X', 'X'], ['X', 'X', 'X', 'X', 'X'], ['X', 'X', 'X', 'X', 'X'], ['.', 'X', 'X', 'X', '.']]`

Mer testning: För denna funktion kan det vara svårare att räkna ut *exakt* vad svaret ska vara för egna testfall. Därför blir det svårare att sätta upp egna assertions. I sådana fall bör man

ändå skapa testfall, men använda något annat sätt att verifiera om de verkar vara korrekta. Använd till exempel följande funktion för att skriva ut resultatet av din `oval()`-funktion och se att det ser rimligt ut för många olika storlekar på ovaler. Se till att utskrifterna är bortkommenterade när du lämnar in.

```
def print_nested(nested):
    for row in nested:
        print("".join(row))
```

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

Uppgift 2: PyAssm

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

I språket ska vi ha tillgång till ett antal *register* som vi kan se som variabler i språket. Dessa register är fördefinierade och har namnen "A" till "Z" (stora bokstäver). När man startar ett PyAssm-program körs ska varje register sättas till värdet `0` (heltal noll).

En *instruktion* i PyAssm är en lista på ett av följande format:

- `["LOG", r]` – skriver ut värdet på register `r` på formatet `"[register=värde]"`, till exempel `[A=12]`. Värdet ska skrivas ut med `print()` på sin egen rad och inget annat än detta ska skrivas ut. Utmatningen kommer att användas som en del av rättningen.
- `["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.

Ett *program* i PyAssm är en Python-lista av instruktioner.

Uppgift 2a: 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. Exempel:

```
>>> eval_pyassm([["LOG", "A"]])
[A=0]

>>> 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:

- Dina funktioner får **inte** modifiera sina indata.
- Programmet får inte ha debugutskrifter eller liknande: Bara utskrifterna från `LOG`-instruktioner får förekomma i utmatningen.

Glöm inte att skriva egna testfall. Skriv upp vilka värden du förväntar dig ska skrivas ut.

Uppgift 2b: 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:

```
>>> eval_pyassm([["ADD", "A", 1],
                 ["MUL", "A", 10],
                 ["ADD", "B", 1],       # 3 steps before the last JEQ
                 ["JEQ", "B", "A", 100],
                 ["LOG", "B"],
                 ["JEQ", "B", "B", -3]  # Always equal, so jump!
                ])
[B=1]
[B=2]
[B=3]
[B=4]
[B=5]
[B=6]
[B=7]
[B=8]
[B=9]
```

Uppgift 2c: 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:

```
>>> eval_pyassm([["ADD", "A", 1],
                 ["JSR", 5],       # Hoppa från 1 till 5
                 ["NOP"],
                 ["NOP"],
                 ["JEQ", "B", "B", 10000],
                 ["LOG", "A"],
                 ["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:

```
>>> eval_pyassm([["ADD", "A", 1],  # Index 0
                 ["JSR", 5],
                 ["LOG", "D"],
                 ["NOP"],
                 ["JEQ", "B", "B", 10000],
                 ["LOG", "A"],  # Index 5
                 ["JSR", 10],
                 ["LOG", "C"],
                 ["RET"],
                 ["NOP"],
                 ["LOG", "B"],  # Index 10
                 ["RET"],
                 ])
[A=1]
[B=0]
[C=0]
[D=0]
```

Uppgift 3

I denna uppgift ska vi beräkna en funktion av ett *träd*, som representeras/lagras så här:

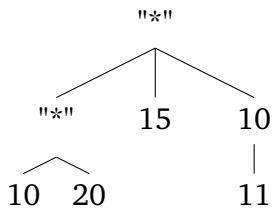
- En **nod** representeras som en lista med minst 1 element.
- Första elementet i en nod är dess **nodvärde**, som kan vara ett godtyckligt Python-värde av godtycklig typ.
- Kvarvarande element i noden (listan) är nodens **barn**, som också är noder.

Som vanligt representeras ett **träd** av dess **rotnod**, och noder utan barn kallas för **löv**. Några exempel på träd är:

- `[15] # En rotnod med nodvärde 15, utan barn`
- `[15, [16], [17]] # En rotnod med nodvärde 15, med två barn`
- `["*", ["*", [10], [20]], [15], [10, [11]]]`

Det senare trädet kan vi också *skriva ut* eller *visualisera* så här, så syns strukturen bättre:

```
["*",
    ["*",
        [10],
        [20]],
    [15],
    [10,
        [11]]]
```



Vi vet alltså redan vad ett *nodvärde* är. Nu vill vi kunna använda detta för att beräkna ett värde för ett helt träd, vilket vi kallar **trädvärde**:

- Trädvärdet för ett *löv* är alltid lika med lövets nodvärde.
- Trädvärdet för en *inre nod*, som alltså har minst 1 barn, beror på nodens nodvärde:
  - Om nodvärdet är strängen "*", är nodens trädvärde lika med *produkten* av *trädvärdena* för alla nodens barn. (Om det bara finns ett barn, är resultatet alltså lika med barnets trädvärde: Produkten av ett enda tal är lika med det talet.)
  - Om nodvärdet är något annat, är nodens trädvärde lika med *summan* av både (1) *nodvärdet* för denna nod och (2) *trädvärdena* för alla nodens barn.

Se även exemplen nedan, som visar konkreta resultat för olika träd.

Uppgift 3a: Trädvärden (3p)

Din uppgift är att skriva funktionen `treeval(tree)`, som beräknar och returnerar trädvärdet för ett träd (där parametern `tree` alltså är trädets rotnod).

Exempel:

- `assert treeval([10]) == 10`
- `assert treeval(["*", [10]]) == 10`
- `assert treeval(["*", ["*", [10], [20]], [15], [10, [11]]]) == 63000`
- `assert treeval([555, ["*", [10], [20]], [15], [10, [11]]]) == 791`

Ytterligare villkor:

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

Uppgift 3b: Trädvärden med *** (3p)

Nu ska vi utöka definitionen av trädvärde:

- Om nodvärdet för en nod *n* är strängen `"***"`, ska nodens trädvärde beräknas via multiplikation, precis som för `"*"`.

  Men i detta fall ska multiplikation användas *även i alla avkomlingar* till *n*, alltså även i sådana noder där trädvärdet annars hade beräknats med addition. Med avkomlingar (descendants) menas alla barn, barnbarn och så vidare.

Detta ska implementeras i en ny funktion, `treeval2(tree)`. Modifiera inte `treeval(tree)`.

Exempel:

- `assert treeval2(["*", [3, [5, [7]]], [11]]) == 165 # (3+5+7)*11`
- `assert treeval2(["***", [3, [5, [7]]], [11]]) == 1155 # 3*5*7*11`
- `assert treeval2([10, ["*", [5, [6]], [7, [8]]], [15, [10]]]) == 200`
- `assert treeval2([10, ["***", [5, [6]], [7, [8]]], [15, [10]]]) == 1715`

Ytterligare villkor:

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

Uppgift 4

Vi ska nu arbeta med *tvådimensionella rutnät* med godtycklig bredd $w > 0$ och godtycklig höjd $h > 0$. Sådana rutnät kan enkelt representeras som nästlade listor i Python:

A	B	C	D
E	F	G	H
I	J	K	L

```
grid1 = [[A,B,C,D],
         [E,F,G,H],
         [I,J,K,L]]
```

Dessa nästlade listor behöver inte vara *regelbundna* – varje rad kan innehålla ett godtyckligt antal element:

```
grid2 = [[A,B,C,D,E,F],
         [G,H],
         [I,J,K,L]]
```

Uppgift 4a: Traversering i alternerande ordning (2p)

Rutnät kan traverseras i olika ordning. Vi vill nu traversera dem i en *alternerande ordning* där första raden i rutnätet traverseras vänster till höger (`A,B,C,D,E,F` i `grid2`), nästa rad från höger till vänster (`H,G`), nästa från vänster till höger igen (`I,J,K,L`), och så vidare.

Skriv funktionen `traverse2(grid)` som tar ett tvådimensionellt rutnät representerat som en nästlad Python-lista i 2 nivåer enligt ovan, där elementen på den lägsta nivån kan vara av vilken typ som helst *utom listor*.

Funktionen ska traverserar rutnätet i *alternerande ordning* enligt ovan, och ska returnera en icke nästlad lista som innehåller alla element som påträffades – i den ordning som de påträffades under traverseringen – *utom* de element som är värdet `None`.

Exempel:

- `traverse2([[1, 2, 3], [4, 5, 6], [7, 8]]) == [1, 2, 3, 6, 5, 4, 7, 8]`
- `traverse2([[1,2,None],[3,None,"a"],[12,34]]) == [1,2,"a",3,12,34]`

Ytterligare villkor:

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

Uppgift 4b: Högre ordningen (1p)

Utöka funktionen från uppgift 4a och skapa `traverse2p(grid,pred)`, som tar ett tvådimensionellt rutnät och en *predikatfunktion* `pred` med 1 parameter (ett element). Funktionen `traverse2p` ska returnera en icke nästlad lista på alla element i `grid` – i den givna ordningen – som *uppfyller predikatfunktionen* `pred`.

Exempel:

- `traverse2p([[1,2,None],[3,None,"a"],[12,34]],`
  `lambda x: isinstance(x,int) and x % 2 == 0) == [2, 12, 34]`

Uppgift 4c: Traversering i godtyckliga dimensioner (2p)

Nu har vi hanterat rutnät i två dimensioner, men det går givetvis bra att utöka dem till flera dimensioner och flera nästlingsnivåer.

Vi definierar då ett **rutnät** som:

- En *lista* där *alla* element är rutnät (och därmed listor), eller
- En *lista* där *inget* element är en lista.

Exempel:

- `[1,2,3]` är ett rutnät, eftersom det är en lista där inget element är en lista.
- `[[1,2,3],[[3,4],[5,6],[7,8]]]` är ett (“oregelbundet") rutnät.
- `[[1,2,3],[3,4,[5,6]]]` är inte ett rutnät, eftersom det finns en nivå där bara vissa element är listor.

Du ska nu utöka funktionen från uppgift 4b och och skapa `traverseNp(grid,pred)`, som klarar godtyckliga *rutnät* enligt denna definition.

Traverseringsriktningen ska nu bestämmas av *nivån*. På toppnivån ska rutnätets element traverseras *från index 0* till slutet. På nästa nästlade nivå ska elementen traverseras *från slutet* till index 0. Nästa nivå traverseras återigen från index 0, och så vidare.

Exempel:

- `traverseNp([1, 2, 3], lambda x: True) == [1, 2, 3]`
- `traverseNp([[1, 2, 3]], lambda x: True) == [3, 2, 1]`
- `traverseNp([[[1, 2, 3]]], lambda x: True) == [1, 2, 3]`
- `traverseNp([[1, 2, 3], [[3, 4], [5, 6], [7, 8]]], lambda x: True) == [3, 2, 1, 7, 8, 5, 6, 3, 4]`
- `traverseNp([[1, 2, 3], [4, 5, 6], [[7, 8], [9, 10], [11, 12]]], lambda x: True) == [3, 2, 1, 6, 5, 4, 11, 12, 9, 10, 7, 8]`

I sista exemplet har vi tre element på högsta nivån; **första** elementet `[1,2,3]` hanteras först och `[[7,8],[9,10],[11,12]]` hanteras sist.

När `[[7,8],[9,10],[11,12]]` hanteras, ska **sista** elementet `[11,12]` hanteras först.

När `[11,12]` hanteras, ska **första** elementet 11 hanteras först.

Ytterligare villkor:

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

Uppgift 5: Delsekvenser med maxavstånd

Vi kan lätt ta reda på om alla element i en sekvens `subseq` *också finns* i samma ordning i en längre sekvens `seq`, där ytterligare element kan vara infogade mellan dem.

Till exempel finns elementen i `[1,2,3]` också i `[1,5,1,2,7,8,9,3]`, i samma ordning, och i `[1,1,1,2,1,3,1,2]`. De finns *inte* i rätt ordning i `[1,5,3,2,1,2]`, men de finns i `[1,2,1,5,2,3]`. Vi hittar också sekvensen `[1,2,1]` inuti `[1,5,2,7,1]`. Sekvensen `[1,1]` finns *inte* i `[1]`; varje element i `subseq` delsekvensen måste ha en egen position i `seq`.

5a: Delsekvenser utan maxavstånd (1p)

Din första uppgift är att implementera funktionen `find_subseq(subseq, seq)`, som ska returnera en lista av *index* där elementen från den godtyckliga sekvensen `subseq` kan hittas i den godtyckliga sekvensen `seq`. Om detta inte går, ska `None` returneras.

Exempel:

- `find_subseq(["a","b"], ["a",5,"b"]) == [0,2]`

5b: Delsekvenser med maxavstånd (5p)

Nu vill vi avgöra om det går att hitta elementen i `subseq`, i rätt ordning enligt ovan, på index i `seq` som är *högst `maxdist` steg från varandra*.

- Elementen i sekvensen `[1,7,3]` finns med i sekvensen `[1,5,1,2,7,8,9,3,3]` på ett avstånd av maximalt 3 steg från varandra: `[1,5,1,2,7,8,9,3,3]`.
- Därmed finns de också på index som är t.ex. maximalt 10 steg från varandra, eftersom $3 < 10$. Vi är alltså inte intresserade av att det längsta avståndet ska vara *exakt* `maxdist`, utan att man hittar index så att varje avstånd är *högst* `maxdist`.
- Men de finns *inte* på index som är maximalt 2 steg från varandra.

Din uppgift är att implementera funktionen `find_with_max_distance(subseq, seq, maxdist)`, som ska returnera en lista av *index* där elementen från den godtyckliga sekvensen `subseq` kan hittas i den godtyckliga sekvensen `seq` så att alla avstånd är högst `maxdist`. Om detta inte går, ska `None` returneras.

Exempel:

- `find_with_max_distance([], [1,5,1], 3) == []`
  (Ett index för varje element i tomma listan...)
- `find_with_max_distance(["a"], ["a",5,"a"], 3) in ([0],[2])`
  (Notera "`in`": Det finns två möjliga lösningar, index 0 eller index 2, och man behöver hitta någon av dem.)
- `find_with_max_distance([1,2], [2,1], 3) is None`
  (Ingen lösning i rätt ordning)
- `find_with_max_distance([1,2,3], [1,9,9,9,1,2,2,9,9,3], 3) == [4,6,9]`

- `find_with_max_distance([1,7,3], [1,5,1,2,7,8,9,3,3], 3) == [2,4,7]` (Med avstånd 4 eller 5 kan flera lösningar finnas)
- `find_with_max_distance([1,7,3], [9,9,9,9,9,9,9,1,5,1,2,7,8,9,3,3], 3) == [9,11,14]` (inte `[2,4,7]`; se nedan)
- `find_with_max_distance([1,7,3], [1,5,1,2,7,8,9,3,7,3], 2) is None`

Extra information utskickad under tentans gång:

En av testerna för 5b adderades till tentan i ett sent stadium för att ge er ett tydligt exempel där första elementet i subseq skulle matchas mot ett *senare* element i seq. Facit blev då fel: "`find_with_max_distance([1,7,3], [9,9,9,9,9,9,9,1,5,1,2,7,8,9,3,3], 3) == [2,4,7]`" skulle vara "`find_with_max_distance([1,7,3], [9,9,9,9,9,9,9,1,5,1,2,7,8,9,3,3], 3) == [9,11,14]`" istället. Detta fel var förhoppningsvis uppenbart (värdet 1 finns ju inte på position 2 i den långa sekvensen), och en korrigering skickades ut klockan 09:02.

Tips:

- Denna uppgift är enklast att lösa med hjälp av rekursion. Vi kräver dock *inte* en rekursiv lösningsmodell då detta inte är poängen med uppgiften.
- Man kan behöva ha en hjälpfunktion eller defaultargument.
- Det kan vara bra att hålla reda på hur många steg det är kvar till man måste hitta nästa element.

Ytterligare villkor:

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