Datortentamen 2023-08-15

TDDE24 Funktionell och imperativ programmering del 2

Allmän information

Tentan består av totalt **5 uppgifter**. De flesta kan ge maximalt 5 poäng, medan den sista är mer omfattande och kan ge maximalt 8 poäng.

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

Tillåtna hjälpmedel och verktyg

Följande hjälpmedel och verktyg är tillåtna / tillgängliga:

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

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

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

Otillåtna hjälpmedel; fusk och vilseledande

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

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

Utöver detta gäller följande:

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

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

Svarsmallar

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

Mallen hjälper till med detta:

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

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

Mallen kan se ut *ungefär* så här:

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

def the_function_you_should_write(seq: list):
    pass

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

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

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

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

    print("Har kört alla tester")

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

Under tentan: Lämna in uppgifter

Med **tentaklienten** (som du har fått information om i förväg och som även beskrivs i en fil som ska finnas tillgänglig på tentadatorn) skickar du in dina svar, med en separat inlämning per uppgift. Svaren 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 anger rätt uppgiftsnummer!

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

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

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

Under tentan: Ställa frågor

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

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

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

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

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

Under tentan: Informationsutskick

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

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

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

Viktiga tips om testning – missa inte poäng i onödan

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

- Vi ger ofta flera testfall i varje uppgift. De som är skrivna direkt i `assert`-form finns normalt också med i svarsmallen. Det kan också finnas indirekta tips som del av den löpande texten, vilket normalt *inte* läggs med i svarsmallen men kan fungera som inspiration till fler testfall. **Testa allt du kan och skapa egna variationer!** Ofta hittar man fel som faktiskt är lätta att korrigera.
- De testfall vi ger är bara exempel och täcker definitivt inte allt – man måste också utgå från beskrivningen i texten. **Skapa egna tester** som täcker fler fall!
- Det går *delvis* att testa genom att **köra implementationen med många olika indata** även om du inte vet vad korrekt svar ska bli (`print` istället för `assert`). Ibland fungerar loopar – testkör t.ex. med värden från 0 till 100! Det är inte ovanligt att koden kraschar direkt, eller skriver ut uppenbart felaktiga svar. Då har du hittat ett fel, utan att veta exakt vad det korrekta svaret skulle vara!
- Tänk på att **testa specialfall** som *tomma listor*, *tomma tupler*, *negativa tal*, med mera. Testa också *långa* listor eller *djupt nästlade* listor.
- Var noga med att **läsa exakt vad som står**! En *godtycklig lista* är precis vilken lista som helst, så testa med olika listor, även med sådana där elementen består av nästlade listor eller tupler eller andra datastrukturer. En *sekvens* behöver inte nödvändigtvis vara en lista, så testa även med tupler och kanske en sträng eller två. Ska det fungera för heltal $n \geq 0$ ska det också fungera för $n = 0$, så testa det. Ska funktionen *ta en sekvens och returnera en lista* får den inte returnera en tupel, även om inputsekvensen råkade vara en tupel.
- Tänk på att man ofta ska **klara godtycklig input**. Även om de angivna testfallen bara är heltal, kanske det står att man ska klara godtyckliga listor, och alltså vilka element som helst. **Testa då detta!** Fastna inte heller i att ett exempel bara råkar ange *positiva* tal, eller att en lista råkar vara *sorterad*. Återigen, läs vad uppgiften ska klara och skapa egna exempel som testar varierande input.

Att testa är extremt viktigt!

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

Men testning fångar inte allt!

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

Egna tester rekommenderas men är ej obligatoriska!

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

Rättningskriterier

Brott mot följande allmänna kriterier kan resultera i poängavdrag.

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

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

  Om *docstrings* krävs, beskrivs detta uttryckligen i uppgiftens instruktioner.
- Den implementerade lösningen ska kunna köras inom **rimliga tidsramar**. Till exempel accepteras inte en lösning som tar 1 minut för att konkatenera två strängar som är 4 bokstäver långa. Eventuella undantag anges uttryckligen i uppgiften.
- Om inte annat sägs ska funktioner **returnera** värden, inte skriva ut dem. Testfall i tentatexten visar normalt returnerade värden, inte värden som har skrivits ut.

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

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

Tillåtet / icke-krav

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

- Att lösa uppgiften **rekursivt eller iterativt, eller med en kombination** av dessa lösningsmodeller (t.ex. hybrider med rekursiva anrop och defaultargument). Med andra ord: Om inget annat sägs kräver vi inte någon specifik form av rekursion, och du kan använda en lösningsform du känner dig bekväm med.
- Att importera och använda **alla vanliga "inbyggda" funktioner** från Pythons standardbibliotek (upp till och med Python 3.9), t.ex. matematiska funktioner från `math`, högre ordningens funktioner som `map()`, och så vidare.
- Att använda **listbyggare** (list comprehensions), generatoruttryck (generator expressions), slicing (delsekvenser) och andra funktionaliteter i språket.
- Att skapa **hjälpfunktioner**, nästlade eller icke nästlade. Lösningen i sin helhet måste så klart fortfarande följa den tänkta lösningsmodellen, om en sådan är angiven.
- Att **addera defaultargument till funktioner**, så länge som funktionerna fortfarande går att anropa på sådant sätt som visas i uppgiften. Lösningen i sin helhet måste så klart fortfarande följa den tänkta lösningsmodellen, om en sådan är angiven. Defaultargument kan ibland användas på sätt som bryter mot en rekursiv lösningsmodell. Se även varningar för defaultargument under rättningskriterier.
- Att **anta att indata följer specifikationen i uppgiften**, utan några egna felkontroller. Står det t.ex. att funktionen ska ta en sekvens, behöver man inte själv kontrollera att man faktiskt får en sekvens som parameter (om inte uppgiften särskilt anger detta). Står det att funktionen ska ta en lista av heltal, får man anta att det är en lista och att den bara innehåller heltal.

  Funktioner måste alltså ge *korrekta svar för korrekta indata* enligt uppgiften, men om inte annat anges får de *krascha eller ge felaktiga svar för felaktiga indata* (såsom när en funktion som bara ska hantera heltal ges en sträng som parameter).
- Att **bryta mot "ytliga" kodningsstandarder** i fråga om t.ex. mellanrum, indentering, radbrytningar, radlängd och antal blankrader, så länge som koden fortfarande är *lättläst och lättförståelig*. Samma gäller stil på identifierare (`snake_case`, `camelCase` med mera).

Detta gäller inte om uppgiften gör specifika undantag!

Uppgift 1: Listhantering (5p)

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

Kopiera den från `given_files` till en skrivbar katalog, t.ex. `Desktop`. Editera den där. Se ytterligare instruktioner tidigare i tentan.

Skriv en funktion `find_least_close(seq1: list[int], seq2: list[int])` som tar två sekvenser av heltal, där `seq1` har godtycklig längd medan `seq2` innehåller minst ett tal. Funktionen ska returnera en ny lista som för varje tal `x` i `seq1` innehåller det tal i `seq2` som är längst bort från `x`. Vi är alltså ute efter *största skillnaden*.

Exempel:

- `find_least_close([11], [5, 8, 12, 15]) == [5]`

Ytterligare villkor:

- Om det finns ett större och ett mindre tal som båda är "längst bort" (lika långt bort) ska du ta det största. Exempel: `find_least_close([10], [5, 8, 12, 15])` returnerar alltid `[15]`, trots att `5` och `15` är lika långt bort från `10`.
- Funktionerna som specificeras i uppgiften får **inte** modifiera sina indata.

Ytterligare exempel – skapa gärna egna tester med inspiration av villkoren ovan!

- `assert find_least_close([], [1]) == []`
- `assert find_least_close([10], [5, 8, 12, 15]) == [15]`
- `assert find_least_close([12, 10], [-1000, 5, 8, 12, 15]) == [-1000, -1000]`

Uppgift 2: Kasta om efterföljande element (5p)

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

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

Vi är nu intresserade av funktionen `reverse_pairs(seq: list)`, som tar en lista `seq` av $n \geq 0$ godtyckliga element och returnerar en *ny* lista där varje element på en jämn position $(0, 2, 4, \ldots)$ har bytt plats med det efterföljande elementet. Om listans längd är udda ska det sista elementet behålla sin plats, eftersom det inte finns något efterföljande element att byta plats med.

Exempel:

- `assert reverse_pairs([]) == []`
- `assert reverse_pairs([1, 2, 'x', 4]) == [2, 1, 4, 'x']`
- `assert reverse_pairs([1, 2, 3, 4, 5]) == [2, 1, 4, 3, 5]`

Ytterligare villkor:

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

Tips och ledtrådar:

- I denna typ av uppgifter händer det ibland att studenter blandar ihop *element* med elementens *index* (position) i en lista. Var försiktig med det. Skapa gärna några egna testfall som inte använder enstaka siffror som element, utan applicerar `reverse_pairs()` på listor av t.ex. strängar och *stora* tal.
- Testa också gärna med listor som har flera element (`[1,2,3,1,2,3]`), listor med 1 element, och andra fall som du själv kommer på.

Uppgift 2a: Iterativ lösningsmodell (3p)

Implementera funktionen `reverse_pairs_i(seq: list)`, som ska arbeta enligt **iterativ** lösningsmodell. Funktionen **måste** vara iterativ och rekursion får inte användas.

Uppgift 2b: Rekursiv lösningsmodell (2p)

Implementera funktionen `reverse_pairs_r(seq: list)`, som 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.

I detta fall ska alltså `reverse_pairs_r(seq)` anropa sig själv med en *kortare lista* (hur mycket kortare?) och få tillbaka ett korrekt resultat för denna kortare lista. Därefter ska den använda detta för att beräkna resultatet för *hela* listan `seq`.

Uppgift 3: Textpekare i nästlad lista

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

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

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 (pekare), 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; denna funktionalitet kan användas för att representera ord som inte redan finns lagrade 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`).

Se **exemplen** på nästa sida för att se hur detta fungerar!

Ytterligare villkor för både 3a och 3b:

- Funktionerna som specificeras i uppgiften får **inte** modifiera sina indata.
- Funktionen ska hantera **godtyckliga** nästlade listor. Det inkluderar djupa listor (många nästlingsnivåer), korta listor, tomma listor, och *godtyckliga* elementtyper. Skapa gärna egna testfall för detta, och testa att det i alla fall inte kraschar!
- Det är oftast enklast att klara av godtyckligt antal nivåer med en **rekursiv** lösning. Det krävs dock inte att man använder en rekursiv *lösningsmodell*, utan lösningen får gärna kombinera iteration och rekursion. Den kan till och med vara helt iterativ, men det kräver tekniker som vi inte har diskuterat i kursen.
- Då huvudpoängen med denna uppgift är att hantera nästlade listor, ges **noll poäng** för lösningar som bara fungerar med icke-nästlade listor.

Uppgift 3a: Direkt ersättning (3p)

I första steget skriver du funktionen `expand(mem: list[str], msg: list)`, 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 – skapa gärna egna tester i `run_tests()`:

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

Uppgift 3b: Ersättning med konkatenering (2p)

Nu tar vi uppgiften ett steg till. Funktionen `expand_concat(mem: list[str], msg: list)` 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:

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

Notera att mellanslagen i vissa strängar, som "lycka är", kommer från att heltalet `0` pekar ut `msg[0]` som är just ett mellanslag. Det finns alltså ingen specialhantering som alltid lägger till mellanslag mellan ord.

Uppgift 4: Högre ordningens funktioner (5p)

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

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

Deluppgift 4a (3.5p)

Skriv en högre ordningens funktion `pred_comp(p, t, f)` enligt följande:

- `pred_comp()` tar tre argument som indata: `p`, `t` och `f`.

  Den som anropar `pred_comp()` förutsätts då skicka in tre funktioner, som var och en tar ett argument. Se under "Exempel" nedan, där alla tre inskickade funktioner är `lambda`-funktioner.

- `pred_comp()` skapar och returnerar i sin tur en ny funktion, som tar ett argument `x`. Denna returnerade funktion ska returnera `t(x)` om `p(x)` är sant, annars `f(x)`. Funktionen kan definieras som en inre funktion inuti `pred_comp()`, eller kan vara en lambdafunktion.

Exempel:

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

  Notera att vi här anropar `pred_comp(...)` för att den ska skapa en ny funktion som den returnerar till oss. Det är detta anrop som är understruket ovan. Vi anropar sedan direkt *denna* nya funktion med `-4` som argument, och får tillbaka värdet `4`.

- ```
  add_world = pred_comp(lambda x: x == "", lambda x: x, lambda x: x + "World")
  assert add_world("Hello") == "HelloWorld"
  assert add_world("") == ""
  ```
  ```

Deluppgift 4b (1.5p)

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 `div`, som är en *tupel* på formen (x, y). Den ska definieras med hjälp av `pred_comp`.

(Anledningen till att `safe_div` använder en tupel som argument är att `pred_comp` använder och returnerar funktioner med *ett* argument, inte två. Vi måste alltså "packa ihop" både täljaren och nämnaren till en enhet, en tupel.)

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:

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

Uppgift 5: Implementera mappningar med listor (8p)

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

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

I denna uppgift ska du definiera en datatyp som fungerar ungefär som `dict`, men som internt lagrar *nycklar* och *värden* som nyckel/värde-par i en lista. Vi kallar denna datatyp för `ListDict`.

Uppgiften innehåller ganska många funktioner, men varje funktion är relativt liten och man behöver inte implementera alla funktioner för att kunna få poäng.

I `ex5.py` finns följande kod som SKA användas som en startpunkt, utan att ändras:

```
from typing import NamedTuple, Any
KeyValue = NamedTuple("KeyValue", [("key", Any), ("value", Any)])
ListDict = NamedTuple("ListDict", [("pairs", list[KeyValue])])
```

Allmän information om `ListDict` och uppgiften:

- Ett `KeyValue`-par kallas ofta bara **par**. Fältet `key` är dess **nyckel** och fältet `value` är dess **värde**. Både nyckel och värde kan vara helt godtyckliga Python-värden av godtyckliga datatyper (`Any`).

  En `ListDict` innehåller som synes fältet `pairs`, som är en lista av `KeyValue`-par.

  Om en vanlig `dict` i Python hade definierats som `{"a": 1, "b": 42}`, ska detta alltså i denna uppgift representeras som en `ListDict` vars `pairs`-lista innehåller de två elementen `KeyValue("a",1)` och `KeyValue("b",42)`. Detta kommer du så småningom att se mer om när vi definierar funktioner som `new_listdict()`, som skapar en tom `ListDict`, och `listdict_put()`, som lägger till eller uppdaterar värden i en `ListDict`.

- Uppgiften är att **implementera ett antal funktioner** som opererar på värden av typ `ListDict` och som fungerar enligt specifikationerna nedan.

- Din implementation behöver *inte* kontrollera typer för indata utan får förutsätta att korrekta värden skickas in.

- Du behöver *inte* heller följa någon specifik modell för hjälpfunktioner till datatypen. Koden ska ändå vara lättläst och strukturerad, men behöver alltså *inte* följa den exakta strukturen som användes i almanackslabben med t.ex. interna selektorfunktioner.

- Den interna lagringen av nycklar och värden *ska* fungera enligt specifikationen, vilket t.ex. betyder att `KeyValue`-par *ska* användas enligt beskrivningen. Även detta kommer att testas, inte bara funktionernas returvärden.

- När man testar om en `ListDict` innehåller ett `KeyValue`-par med en viss nyckel, används (som vanligt) `==` för att jämföra nycklar.

Funktioner att implementera:

Följande funktioner ska implementeras. Specifikationerna i texten förtydligas även av ett körexempel efter funktionslistan.

För att få poäng måste du åtminstone implementera de grundläggande funktionerna 1–3 (new, put, get), så att det är möjligt att testa din implementerade datatyp, och dessa funktioner behöver fungera korrekt för allt utom möjligen några ovanliga undantagsfall.

I övrigt behöver inte alla funktioner implementeras, utan du får poäng beroende på vilka funktioner som är definierade (se “ytterligare villkor") och hur väl de fungerar.

1. `new_listdict()`, som returnerar en ny `ListDict` genom `return ListDict([])`.
2. `listdict_put(ld: ListDict, key, value)`, som uppdaterar `ld` enligt angiven nyckel och värde. Om `ld` redan innehåller ett `KeyValue`-par med `key` som nyckel ska alltså detta par tas bort, så att det (precis som i en `dict`) aldrig finns två `KeyValue`-par med samma nyckel i en `ListDict`.

   Tips: Du kan behöva loopa genom elementen i `ld.pairs`, se om ett existerande `KeyValue`-element redan har nyckeln `key`, och i så fall ta bort detta element ur listan. Testa att detta fungerar genom att anropa `listdict_put()` många gånger med samma `ld` och `key`, och se till att det inte blir kvar flera element med samma nyckel.
3. `listdict_get(ld: ListDict, key, default)`. Om `ld` innehåller ett `KeyValue`-par med denna `key` ska detta pars `value` returneras. Annars ska `default` returneras.

   Tips: Iterera helt enkelt genom `ld.pairs` och se om det finns ett element med den önskade nyckeln.
4. `listdict_delete(ld: ListDict, key)`. Om `ld` innehåller ett `KeyValue`-par med denna `key` ska detta par raderas från `ld`, och dessutom ska `True` returneras för att indikera att något raderades. Annars ska `False` returneras.
5. `listdict_contains(ld: ListDict, key)`. Om `ld` innehåller ett `KeyValue`-par med denna `key` returneras `True`. Annars ska `False` returneras.
6. `listdict_values(ld: ListDict)`, som returnerar en mängd (`set`) som innehåller alla *värden* som anges i paren i `ld`.
7. `listdict_from(map: dict)`, som tar en “vanlig" Python-dictionary `map` och returnerar en ny `ListDict` som innehåller samma mappningar – samma nyckel/värde-kombinationer.

   Exempel: Om `ld=listdict_from(map)` och om `map[key]==value`, så ska `listdict_get(ld, key)` returnera `value`.

8. `listdict_update(ld_to: ListDict, ld_from: ListDict)`, som ska uppdatera `ld_to` genom att "överföra/kopiera" samtliga nyckel-värde-par från `ld_from` som ett tillägg till de nycklar och värden som redan finns i `ld_to`. Detta fungerar alltså i princip som `update()` för en vanlig `dict`.

   Notera att `ld_from` inte får modifieras.

   Tänk på att vissa par i `ld_from` kan ha nycklar som redan används i `ld_to`. Då måste de motsvarande paren i `ld_to` "bytas ut" mot nya par från `ld_from`, motsvarande vad som händer när `listdict_put(ld, key, value)` anropas med en nyckel som redan existerar i `ld`. En `ListDict` får aldrig innehålla två par med samma nyckel.

9. `listdict_add_value(ld: ListDict, key, value)`, som är tänkt att användas när värdena i en `ListDict` i sig är listor (se sista delen av exemplet på nästa sida):

   - Om `key` inte redan finns som nyckel i `ld`, adderas ett nyckel-värde-par med nyckel `key` och värde `[value]` (i en lista!).
   - Om `key` redan finns som nyckel i `ld`, och motsvarande värde är en lista, läggs `value` till i slutet av denna lista.
   - Om `key` redan finns som nyckel i `ld`, och motsvarande värde *inte* är en lista, ska ett undantag av typ `TypeError` signaleras.

10. `listdict_internal_sort(ld: ListDict)`, som är ovanlig på så sätt att den inte ska påverka en `ListDict` på något "synligt" sätt. Istället ska den sortera den interna listrepresentationen, `ld.pairs`, med avseende på *nycklar* (`KeyValue`-parens `key`-värden).

    Inbyggda sorteringsfunktioner får användas. Man får anta att alla nycklar i den `ListDict` som ska sorteras faktiskt kan jämföras med varandra (att de stödjer `'<'`), men man får inte anta att de har någon specifik typ.

    För att testa detta skapar du ett eget testfall. Du behöver inte lämna in testfallet, men får göra det om du vill.

Ytterligare villkor:

- Funktioner som uttryckligen är till för att modifiera innehållet i en specifik `ListDict` får givetvis göra detta. I övrigt ska funktionerna inte modifiera sina indata.
- De tre första funktionerna (new, put, get) ger tillsammans 3p.
- Funktionerna delete, from, update ger 1.0p var.
- Funktionerna contains / values / add_value / internal_sort ger 0.5p var.

Partiella exempel (skapa gärna egna testfall): (Filen `ex5.py` innehåller några fler tester, som också körs då du testar.)

```
ld = new_listdict()
assert isinstance(ld, ListDict)
assert listdict_get(ld, 0, 42) == 42
assert listdict_get(ld, "hello", 42) == 42

listdict_put(ld, 0, 30)
listdict_put(ld, "hello", "x")
assert listdict_get(ld, 0, 42) == 30
assert listdict_get(ld, "hello", 42) == "x"
assert listdict_values(ld) == {30, "x"}

listdict_delete(ld, "hello")
assert listdict_get(ld, 0, 42) == 30
assert listdict_get(ld, "hello", 42) == 42
assert listdict_contains(ld, 0)
assert not listdict_contains(ld, "hello")

ld2 = listdict_from({"a": "b", 10: 20, 30: "z"})
assert listdict_get(ld2, "a", 42) == "b"

listdict_update(ld, ld2)
assert listdict_get(ld, "a", 42) == "b"
assert listdict_get(ld2, "a", 42) == "b"
assert listdict_values(ld) == {20, 'b', 30, 'z'}

listdict_put(ld, "numbers", [])
listdict_add_value(ld, "numbers", 10)
listdict_add_value(ld, "numbers", 20)
assert listdict_get(ld, "numbers", 30) == [10,20]
```

Uppgift 6: Finns inte (0p)

Många av våra tentor har 6 uppgifter, men den här har bara 5, på grund av den lite större omfattningen i uppgift 5. Det är möjligt att “uppgift 6" *ser ut* att finnas i tentaklienten på tentadatorn – ignorera det i så fall.