Datortentamen 2021-03-16

TDDE24 Funktionell och imperativ programmering del 2

Distanstentamen

Detta är en *distanstentamen* i TDDE24. 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 och beskriva 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 **22:00** (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-388/` eller på `https://www.python.org/downloads/release/python-392`. 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. Skulle du använda 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 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 någon 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!

För de som har förlängd skrivtid kommer jag att vara tillgänglig för frågor fram till klockan 21, men även under tiden 19–21 kan svaren dröja och även i detta fall behöver ni tänka på att ställa frågorna i tid.

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

Informationsutskick under tentans gång

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. Detta kan gälla:

- Korrigeringar till tentan.
- Påminnelser om sådant som står i instruktionerna men som flera studenter har missat.

Håll alltså koll på din egen epost under tentan.

Inlämning av svar

Inlämning sker via Lisam. Vi använder “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 14–19* finns en slutlig hård **deadline 19:10** 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.

Har du *utökad skrivtid* till 21:00 använder du en separat inlämning med en slutlig hård **deadline 21:10**.

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. **Ange då ditt anonyma nummer, på formen A-99999!**

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 om testning – missa inte poäng i onödan

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

- Vi ger ofta flera testfall i varje uppgift, ibland i `assert`-form och ibland i den löpande texten. **Testa dem alla och skapa egna variationer!** Ofta hittar man fel som faktiskt är lätta att korrigera.
- De testfall vi ger är bara exempel och täcker definitivt inte allt – man måste också utgå från beskrivningen i texten. **Skapa egna tester** som täcker fler fall!
- Det går *delvis* att testa genom att **köra implementationen med många olika indata** även om du inte vet vad korrekt svar ska bli (`print` istället för `assert`). Kraschar koden? Skriver den ut uppenbart felaktiga svar? Då har du hittat ett fel.
- Tänk på att **testa specialfall** som *tomma listor*, *tomma tupler*, *negativa tal*, med mera. Testa också *långa* listor eller *djupt nästlade* listor.
- Var noga med att **läsa exakt vad som står**! En *godtycklig lista* är precis vilken lista som helst, så testa med olika listor, även 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.
- För att misslyckade testfall inte ska få koden att krascha när den *importeras* av våra testscript är det bra att se till att testfallen inte alls körs vid importering av filen. Det gör man genom att lägga testfallen inom följande `if`-sats, så slipper man komma ihåg att kommentera bort testfallen på slutet:

```
if __name__ == '__main__':
    print("Running tests!")
    assert f(12) == 34
    assert ...
```

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 alltid att *läsa* och *förstå* vad som egentligen menas.

Rättningskriterier

Brott mot följande allmänna kriterier kan resultera i poängavdrag.

- Lösningen ska givetvis vara **körbar**. Testa alltid **precis innan inlämning** så att din sista finputsning eller dina sista kommentarer inte resulterade i felaktig kod och så att koden inte kraschar när filen importeras av våra granskningsscript, 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. **Notera att NumPy är ett separat bibliotek som alltså inte får användas!** Du får inte heller skriva av andras kod.

Rättningskriterier (fortsättning)

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.

Rättningskriterier som många missar

Det är relativt vanligt att studenter missar eller ignorerar följande kriterier. Vi har varit snälla vid årets första tenta men kommer att vara hårdare denna gång.

- Lösningen **SKA** vara körbar. Vi **SKA INTE** behöva ändra på koden för att den omedelbart kraschar, t.ex. i testfall som misslyckas.
- Filer och funktioner **SKA** ha korrekta namn.
- Funktioner **SKA** ge korrekta resultat även när man kör dem flera gånger.

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).

  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 gör specifika undantag!

Uppgift 1: Orduppdelning (4p)

Översikt: I den första uppgiften ska strängar delas upp i ord, och orden ska delas upp i listor med vissa specifika längder.

Skapa filen `ex1.py` för hela denna uppgift!

Implementera funktionen `split_fib(s: str)`, som:

- Tar som argument en godtyckligt lång textsträng `s` som innehåller *skrivbara tecken* (bokstäver, siffror, andra symboler) och *mellanslag*. Man får anta att strängen:
  - Innehåller minst ett skrivbart tecken,
  - Varken börjar eller slutar med mellanslag,
  - Inte har två eller flera mellanslag i rad,
  - Inte innehåller andra "osynliga" tecken än mellanslag (till exempel radbrytningar).
- Delar upp strängen i *ord*, vilket definieras som *sekvenser av tecken som inte är mellanslag*. Man ska *inte* dela strängen på andra platser än vid mellanslag. Strängen `"abc def"` innehåller alltså exakt två ord, och varje input `s` innehåller minst ett ord.
- Returnerar en *lista av* $n \geq 1$ *listor av ord*, som vi kan kalla `ret`.

  Antalet ord i varje lista (och därmed även *antalet listor*) ska bestämmas av Fibonacciserien (0, 1, 1, 2, 3, 5, 8, 13, ...); se även illustrerande exempel på nästa sida! Mer specifikt ska varje dellista `ret[k]` ska innehålla exakt F_{k+1} ord, med undantag för den sista listan `ret[n-1]`, som ska innehålla minst 1 och högst F_n ord. Den flexibla längden på den sista listan gör att vi inte blir begränsade till vissa antal ord som "passar ihop" med Fibonacciserien.

  Med F_k menar vi alltså ett tal i Fibonacciserien, definierat nedan, och med `ret[k]` menar vi Pythons indexering i listor, med start på index `k=0`.

Som hjälp får du följande något ineffektiva funktion som räknar ut $F_k = fib(k)$:

```
def fib(k):
    if k == 0:
        return 0
    elif k == 1 or k == 2:
        return 1
    else:
        return fib(k-1) + fib(k-2)
```

Ytterligare villkor:

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

Exempel på nästa sida!

Exempel:

- `assert split_fib("abc def")) == [['abc'], ['def']]`
- `assert split_fib("abc def ghi j klm nopq rst uv wxy z 123 456 /+") == [['abc'], ['def'], ['ghi', 'j'], ['klm', 'nopq', 'rst'], ['uv', 'wxy', 'z', '123', '456'], ['/+']]`

Skapa egna testfall av olika längd och olika antal ord. De får lämnas in, men det är inget krav.

Uppgift 2: Stegvis summering (4p)

Översikt: I denna uppgift summerar du tal i sekvenser. För varje sekvens ska ett antal olika summor skapas, med olika urval av element som summeras.

Skapa filen `ex2.py` för hela denna uppgift!

Uppgift 2a: Summera med godtycklig lösningsmetod (3p)

Skriv funktionen `nth_sums(seq)`, som:

- Tar en sekvens (lista eller tupel) `seq` av $n \geq 0$ godtyckliga tal.
- Returnerar en lista med lika många element, där element nummer k i den nya listan är summan av var $(k+1)$:te tal i `seq` med början på position k i `seq`. Numreringen av elementen börjar som vanligt i Python med element nummer $k = 0$, och exempel på vilka tal som väljs ut ges längre ner.

Ytterligare villkor:

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

Exempel (skapa gärna egna testfall):

- `assert nth_sums([1,2,3,4,5]) == [1+2+3+4+5, 2+4, 3, 4, 5]`
- `assert nth_sums([1,2,3,4,5,6,7]) == [1+2+3+4+5+6+7, 2+4+6, 3+6, 4, 5, 6, 7]`
- `assert nth_sums([1,2,3,4,12,6,7]) == [1+2+3+4+12+6+7, 2+4+6, 3+6, 4, 12,6,7]`

Uppgift 2b: Summera med listbyggare (1p)

För ytterligare 1 poäng, skriv funktionen `nth_sums_lc(seq,nth)` som gör samma sak men:

- Består av exakt 2 rader, en rad för `def nth_sums_lc...` och en rad för `return`.
- Använder en *listbyggare* (list comprehension) för att skapa den lista som ska returneras: `return [ ... for ... ]`.
- Inte använder egendefinierade hjälpfunktioner eller liknande, men gärna får använda inbyggd funktionalitet som “slicing" (indexering för att plocka ut dellistor) och inbyggda funktioner som arbetar på hela listor.

Om du **redan har skrivit på detta sätt i uppgift 2a** gör du en kopia av `nth_sums` och kallar den `nth_sums_lc`. Vi tittar inte på funktionen `nth_sums` när vi bedömer uppgift 2b, utan funktionen `nth_sums_lc` måste i så fall finnas och fungera korrekt.

Ytterligare villkor:

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

Exempel (skapa gärna egna testfall):

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

Uppgift 3: Hitta element i nästlade listor (6p)

Översikt: På vilka *positioner* i en *nästlad lista* finns det *intressanta element* som uppfyller ett predikat? Hitta de första `n` av dessa positioner enligt en specifik ordning.

Skapa filen `ex3.py` för hela denna uppgift!

Obs: Denna funktion kan se mer komplicerad ut än den egentligen är, eftersom vi vill vara mycket strikta med definitionerna för att undvika missförstånd.

Implementera funktionen `find_nested(nl, pred, n)`, som:

- Tar tre argument:
  - Den nästlade listan `nl`. Med “nästlad lista" menar vi en godtyckligt lång lista (typ `list`) där varje element är antingen (a) en nästlad lista, eller (b) en icke-lista, här kallat *baselement*. Ett baselement kan alltså vara av vilken typ som helst utom `list`.

    Alltså är t.ex. `[12, ("Hello", "world"), [[15]]]` en nästlad lista som (på olika nivåer) innehåller tre baselement: `12` och `("Hello", "world")` och `15`. På samma sätt är `[[[12,12],12]]` en nästlad lista med de tre baselementen `12`, `12` och `12`, som vart och ett har en distinkt position i listan.
  - Predikatsfunktionen `pred`, som ska kunna appliceras på alla *baselement*. De baselement som uppfyller detta predikat som uppfyller detta predikat är *intressanta*. Predikatsfunktionen ska alltså *inte* appliceras på nästlade listor.
  - Heltalet $\texttt{n} \geq 0$, som anger hur många *intressanta* baselement vi skulle vilja hitta.
- Går genom baselementen i `nl` i *utskriftsordning*, alltså den ordning de ser ut att förekomma om `nl` skrivs ut. Med detta menar vi till exempel att baselementen i den nästlade listan `[[5,1,[12,[4],7],3,[2]]` behandlas i ordningen $5, 1, 12, 4, 7, 3, 2$.

  Detta betyder inte att listan ska skrivas ut. Den behöver traverseras på något annat sätt.
- Returnerar `(None,None)` om det finns strikt färre än `n` intressanta baselement i `nl`.

  Det är viktigt att just denna specifika tupel returneras, för att på ett entydigt sätt indikera att man inte anser att det finns tillräckligt många intressanta baselement.
- Annars, returnerar en lista som i tur och ordning innehåller de `n` första *listpositionerna* där intressanta baselement förekommer.

  Med *listposition* menar vi en tupel av listindex som tillsammans pekar ut en specifik position i en lista. Den nästlade listan `nl=[[12,34],[[56]]]` har de tre listpositionerna `(0,0)` och `(0,1)` och `(1,0,0)`, vilket direkt motsvarar att baselementen kan hittas genom uttrycken `nl[0][0]`, `nl[0][1]` och `nl[1][0][0]`.

  Med “i tur och ordning" och “första" menar vi att listpositionerna, och baselementen

på dessa positioner, ska undersökas och adderas till i resultatlistan enligt den *utskriftsordning* som definierades ovan.

Ytterligare villkor:

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

Allmänna tips och ledtrådar:

- Rekursion kan vara användbar för att gå genom listorna “på djupet". Iteration kan vara användbar i andra delar.
- Det kan vara bra att skapa en separat hjälpfunktion med fler parametrar.
- Man kan hålla reda på antalet element under tiden man letar *eller* ta hand om detta efter att man har hittat alla relevanta positioner.
- Skapa många testfall och testa med flera olika predikat.

Exempel (skapa gärna egna testfall):

- `assert find_nested([[12, 34], [[56]]], lambda x: True, 3) == [(0, 0), (0, 1), (1, 0, 0)]`
- `assert find_nested([[[['a', ['b'], 12], [[33, 'c', ['c']]]]]], lambda x: isinstance(x, str), 3) == [(0, 0, 0, 0), (0, 0, 0, 1, 0), (0, 0, 1, 0, 1)]`
- `assert find_nested([[[['a', ['b'], 12], [[33, 'c', ['c']]]]]], lambda x: isinstance(x, str), 4) == [(0, 0, 0, 0), (0, 0, 0, 1, 0), (0, 0, 1, 0, 1), (0, 0, 1, 0, 2, 0)]`
- `assert find_nested([[[['a', ['b'], 12], [[33, 'c', ['c']]]]]], lambda x: isinstance(x, str), 5) == (None, None)`

Uppgift 4: Implementera mappningar med listor (8p)

Översikt: I denna uppgift implementerar du en datatyp som motsvarar en `dict` i Python. Uppgiften innehåller ganska många funktioner, men varje funktion är relativt liten och man behöver inte implementera alla funktioner för att kunna få poäng.

Skapa filen `ex4.py` för hela denna uppgift!

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`.

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

I övrigt behöver inte alla funktioner implementeras, utan du får poäng beroende på vilka funktioner som är definierade (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`.
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.
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 gör följande:
   - Om `key` inte 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*.

    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.

Exempel (skapa gärna egna testfall):

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

Uppgift 5: Kan ett tal “delas upp" i primtal? (6p)

Översikt: Talet 13172 kan “delas upp" i primtalen 13, 17 och 2. Nu ska du skriva en funktion som testar om/hur ett godtyckligt heltal kan delas upp i primtal på detta sätt.

Skapa filen `ex5.py` för hela denna uppgift!

I uppgifterna 5a och 5b ska du implementera två varianter av en funktion som:

- Tar ett *strikt positivt heltal* $n \geq 1$, som kan vara *godtyckligt stort*.

  För att förenkla uppgiften får man anta att heltalet *inte innehåller nollor*; med andra ord är 105 och 210 inte giltig input. Vid ogiltig input får funktionen göra vad som helst, inklusive krascha.

- Tar reda på om n kan delas upp i en *sekvens av primtal* som tillsammans formar samma *sekvens av siffror* som det ursprungliga talet n.

  Exempelvis kan talet 123456 delas upp i många talsekvenser, såsom `[12,34,56]` och `[1,23,456]`. Ingen av dessa sekvenser är en sekvens av primtal.

  Å andra sidan kan talet 13172 delas upp i sekvenserna `[13,17,2]` och `[131,7,2]`, där samtliga tal är primtal.

  (Detta ska *inte* göras genom att iterera över olika godtyckliga primtal och se om de går att sätta ihop till det ursprungliga talet, då detta skulle bli alltför ineffektivt.)

- Returnerar en *lista* av sådana primtal om det är möjligt, till exempel listan `[13,17,2]` eller `[131,7,2]` för talet `13172`, och annars returnerar det speciella värdet `(None, None)`.

  Det är viktigt att just denna specifika tupel returneras, för att på ett entydigt sätt indikera om man anser att det inte existerar en lösning på problemet.

Allmänna tips och ledtrådar:

- Du behöver implementera en primtalstestare. Det lägsta primtalet är 2, och ett tal $n > 2$ är ett primtal om och endast om det är *jämnt delbart* med något heltal från 2 upp till och med `int(math.sqrt(n))`.
- Glöm inte att testa primtalstestaren...
- Det kan vara användbart att konvertera tal till strängar och tvärtom.

Uppgiften fortsätter på nästa sida.

5a: Dela upp i 1–3 primtal (3p)

I uppgift 5a implementerar du funktionen `split_in_primes_3(n: int)`, som testar om det går att dela upp heltalet *n* i **1, 2 eller 3 primtal** enligt den tidigare specifikationen. Notera att funktionen då *inte* får returnera en uppdelning i 4 eller flera primtal, utan ska returnera `(None, None)` om ingen uppdelning i högst 3 primtal finns!!

Tips och ledtrådar för 5a:

- Se allmänna tips och ledtrådar på förra sidan!
- Anledningen till att uppgift 5a finns är att den går att implementera på ett annat sätt än 5b, utan rekursion.

  Men om du ändå ska implementera uppgift 5b kanske du vill undvika dubbelarbete genom att börja med 5b, och därefter implementera `split_in_primes_3()` enligt specifikationen med hjälp av ett anrop till 5b-funktionen `split_in_primes_n()`.

Exempel (skapa gärna egna testfall):

- `assert split_in_primes_3(12) == (None, None)`
- `assert split_in_primes_3(13) == [13]`
- `assert split_in_primes_3(1317) in [[13, 17], [131, 7]]`
- `assert split_in_primes_3(13172) in [[13, 17, 2], [131, 7, 2]]`
- `assert split_in_primes_3(79429375) == [79, 42937, 5]`
- `assert split_in_primes_3(3333) == (None, None)`

5b: Dela upp i godtyckligt antal primtal (3p)

I uppgift 5b implementerar du funktionen `split_in_primes_n(n: int)`, som testar om det går att dela upp heltalet *n* i något *godtyckligt* antal primtal enligt ovanstående definitioner. Här finns alltså *ingen gräns* på hur många primtal som kan ingå i uppdelningen.

Tips och ledtrådar för 5b:

- Se även tidigare tips och ledtrådar på förra sidan och under 5a.
- Det kan vara lämpligt att använda rekursion för delar av uppgiften.
- *13* är ett primtal, men tvingar man fram en uppdelning så snart man hittar ett primtal kan man missa möjligheten att första primtalet istället är *131*. För att hitta alla lösningar behöver man *alltid* prova *alla* alternativ: Klippa eller inte?
- Testa även med stora tal. Listor på primtal finns på nätet och kan användas för att “bygga" uppdelningsbara tal som används i testfallen.

Exempel (skapa gärna egna testfall):

- De 5 första testfallen från 5a är giltiga även här.
- `assert split_in_primes_n(3333) == [3, 3, 3, 3]`