Datortentamen 2024-01-09

TDDE24 Funktionell och imperativ programmering del 2

Allmän information

Tentan består av totalt **6 uppgifter** som kan ge maximalt 5 poäng.

Poänggränserna sätts inte i förväg för denna tenta, delvis för att tentans sammansättning har ändrats jämfört med tidigare år. Exempelvis har denna tenta inte någon uppgift med krav på rekursiv lösningsmodell, och i flera uppgifter har vi hjälpt till med extra testfall, extra förklaringar och tips, och mer detaljerade genomgångar av tänkta algoritmer. (Det är alltid examinatorns uppgift att avgöra *vilka lösningar som uppnår ett visst betyg*, och poänggränser är bara ett av många möjliga hjälpmedel.)

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 skissa lösningar på papper. (Tomma papper ska finnas tillgängliga.)
- **Python 3.11**, som vi har använt under senaste kursomgången. Detta ska finnas som kommandot python3.11 på tentadatorerna.

  I terminalfönstret *bör* det också gå att använda Python 3.11 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 python3.11 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** med biblioteks- och språkreferenser för Python. Enligt de generella instruktionerna för datortentor ska dessa finnas tillgängliga via en "Web Access"-ikon på skrivbordet (vi som skapar tentan kan inte själva se tentamiljön). Vid eventuella problem, räck upp handen och be tentavakterna tillkalla teknisk jour.

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 den fördefinierade funktionen `run_tests()` – det underlättar vid rättningen. Provkör med t.ex. “`python3.11 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-tester som vi redan har angivit i uppgiften, så du slipper klippa och klistra. (Du kan ändå behöva 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 ett dokument som ska finnas tillgänglig i `given_files`) 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 kan *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.
- **Läs 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 godtyckliga indata**. 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 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 och koden kan vara felaktig även om den klarar alla tester.

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 vid granskning! 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.

  Vi har **inget generellt krav på *docstrings***, men någon form av kommentarer kan som sagt behövas för förståelsen.
- 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.

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 3.11), t.ex. matematiska funktioner från `math`, högre ordningens funktioner som `map()`, och så vidare.
- Att använda **listbyggare** (list comprehensions), generatoruttryck (generator expressions), slicing (delsekvenser såsom `seq[2:5]`) 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 och är alltså problematiska om uppgiften kräver en sådan modell. 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).
- Att **hoppa över "type hinting"** såsom `seq: list[int]` i de funktioner man själv skriver. Vi ger ofta *type hints* i funktionerna som anges i tentan, men det är bara för att hjälpa er se vilka indata som förväntas.

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 `sums(seq: list[int])` som tar en godtyckligt lång lista av godtyckliga heltal som parameter. Funktionen ska returnera en lika lång lista där det *n*:te talet i svaret är summan av de *n* första talen i `seq`.

Exempel:

- `assert sums([1,2,3,4,5]) == [1,3,6,10,15]`.
  (Svaret innehåller tal motsvarande summorna `1`, `1+2`, `1+2+3`, `1+2+3+4` och `1+2+3+4+5`)
- `assert sums([]) == []`
  (Tomma indata ger lika långa utdata)
- Skapa gärna egna tester för olika längder på listor, positiva och negativa tal, och så vidare.

Ytterligare villkor:

- Funktionerna som specificeras i uppgiften får **inte** modifiera sina indata. Med andra ord får funktionen inte addera eller radera tal ur listan `seq`.

Uppgift 2: Merge (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. Se ytterligare instruktioner tidigare i tentan.

Du ska nu skriva funktionen `merge(s1: list, s2: list)` som tar två sorterade listor `s1` och `s2` av godtycklig längd, och som *på ett specifikt sätt* konstruerar en enda sorterad lista som innehåller alla element från dessa. Elementen kan vara av godtycklig typ.

Exempel: Funktionen ska bl.a. klara följande **tester**.

- `assert merge([], [1]) == [1]`
  (Funktionen klarar av tomma listor)
- `assert merge([1, 2, 5, 13], [3, 5, 21]) == [1, 2, 3, 5, 5, 13, 21]`
  (Funktionen kan ta två redan sorterade listor `s1` och `s2` och skapa en ny sorterad lista)
- `assert merge(['a', 'c'], ['b', 'o']) == ['a', 'b', 'c', 'o']`
  (Funktionen är inte begränsad till numeriska element)

Algoritm: Merge-funktionen ska utnyttja att listorna `s1` och `s2` redan är sorterade, genom att i varje steg plocka *ett* element, det första elementet från någon av listorna. I testfallets anrop till `merge([1, 2, 5, 13], [3, 5, 21])` ovan ska exempelvis följande ske:

- `s1==[1,2,5,13]` och `s2==[3,5,21]`. Första elementet i resultatet måste alltid vara första elementet i någon av listorna, alltså 1 eller 3. Det minsta av dessa är 1, så första elementet i resultatet är 1. Det kan nu tas bort från `s1`.
- `s1==[2,5,13]` och `s2==[3,5,21]`. Minst av elementen 2 och 3 är elementet 2, vilket alltså är nästa element i resultatet.
- `s1==[5,13]` och `s2==[3,5,21]`. Nästa element är 3.
- `s1==[5,13]` och `s2==[5,21]`. Nästa element är *någon* av femmorna; det är godtyckligt vilken vi plockar. Låt oss till exempel ta elementet från `s1`.
- `s1==[13]` och `s2==[5,21]`. Nästa element är 5.
- `s1==[13]` och `s2==[21]`. Nästa element är 13.
- `s1==[]` och `s2==[21]`. Nu kan vi inte längre jämföra första elementen, för s1 är tom. Nästa element måste vara 21.
- `s1==[]` och `s2==[]`. Nu har vi plockat alla element från båda listorna, i ordningen `[1,2,3,5,5,13,21]`. Listorna är tomma och resultatet kan returneras.

Tips och villkor på nästa sida.

Tips:

- Tänk på att båda listorna kan vara tomma.
- Kom ihåg att du inte behöver ta hänsyn till att någon kan skicka in felaktiga indata. Du behöver till exempel inte fundera på vad din funktion gör om `s1` eller `s2` är *osorterad*.

Ytterligare villkor:

- Funktionerna som specificeras i uppgiften får **inte** modifiera sina indata. Om du vill *ta bort* element från `s1` eller `s2` behöver du alltså kopiera listorna först. Tänk då på hur man gör en riktig kopia, inte bara en ny pekare till samma lista. (Vi kan inte svara på frågor kring hur man gör detta.)

  Istället för att ta bort element ur listor kan man också välja att hålla reda på ett *index* för varje lista, som talar om hur långt man har kommit.

- Listor kan innehålla godtyckliga element, inklusive *identiska element*.

  Vi garanterar att alla par av element kan jämföras med de vanliga jämförelserna, t.ex. `<` och `<=`. Testa t.ex. med tupler och strängar som element!

- Du får **inte anropa sorteringsalgoritmer** (egna eller inbyggda) för att få elementen i rätt ordning. Du ska istället implementera den *effektivare* metoden som beskrivs i algoritmen ovan, som utnyttjar att listorna `s1` och `s2` redan är sorterade för att slå ihop listorna på ett effektivare sätt än med en fullständig sorteringsalgoritm.

Uppgift 3: Filter 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. Se ytterligare instruktioner tidigare i tentan.

Du ska nu arbeta med **nästlade listor**. Din implementation måste klara av nästling på godtyckligt djup (exempelvis `[[[[[[[[[[10]]]]]]]]]]`), och då är det sannolikt enklast att använda rekursion i implementationen. Vi ställer dock inga specifika krav på *rekursiv lösningsmodell*. Det är fullt tillåtet, och oftast enklare, att använda enkelrekursion istället för dubbelrekursion.

Du ska skriva funktionen `without(nest: list, to_remove: list)`, som tar en nästlad lista `nest` (av godtycklig längd/form, med godtyckliga element) och returnerar motsvarande lista *utan* de element som finns i `to_remove`. Element ska då uteslutas på alla nivåer i den nästlade listan. För enkelhets skull kan `to_remove` inte innehålla element som är listor.

Exempel:

- `assert without([[1], [[2]], [[[3]]], [[[[4]]]]], [1,3]) == [[], [[2]], [[[]]], [[[[4]]]]`
  (Returnerar en nästlad lista på samma form som input, men utan elementen 1 och 3)
- `assert without([[[[[[[[[[10]]]]]]]]]], [10]) == [[[[[[[[[[]]]]]]]]]]`
  (Alla nästlingsnivåer bevaras, även när man tar bort alla element)
- `assert without([[[[[[[[[[10]]]]]]]]]], [5]) == [[[[[[[[[[10]]]]]]]]]]`
  (Ibland tas inga element bort)
- `assert without([[(1,2)], [["b"]], [[[None]]], [[[[42.5]]]]], [1,None]) == [[(1,2)], [["b"]], [[[]]], [[[[42.5]]]]]`
  (Godtyckliga elementtyper. Enbart listor räknas som nästlade – inte t.ex. tupler.)

Tips:

- Tänk på att `nest` kan vara tom. Den kan även *innehålla* tomma listor på godtycklig nivå. Testa gärna!

Ytterligare villkor:

- Funktionerna som specificeras i uppgiften får **inte** modifiera sina indata. Funktionen ska alltså inte ta bort element ur den ursprungliga listan `nest`, utan returnera en *ny* lista motsvarande den ursprungliga minus de oönskade elementen.
- 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!
- Då huvudpoängen med denna uppgift är att hantera nästlade listor, ges **noll poäng** för lösningar som bara fungerar med icke-nästlade listor eller som specialhanterar ett fåtal olika nästlingsdjup.

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. Se ytterligare instruktioner tidigare i tentan.

Deluppgift 4a (2.5p)

Implementera funktionen `split_at(seq, pred)`, som delar upp sekvensen `seq` i dellistor enligt följande:

- `assert split_at([1,2,3,4,2,5], lambda x: x==2) == [[1], [3,4], [5]]`

Det är parametern `pred` som avgör vid vilka element den ursprungliga sekvensen `seq` ska delas. I exemplet ovan definieras `pred` som en lambda-funktion som returnerar sant för element `x` som är lika med `2`, alltså det andra och femte elementet i parametern `[1,2,3,4,2,5]`. Genom att dela sekvensen vid dessa positioner får vi svaret `[[1], [3,4], [5]]`.

Eftersom `split_at()` tar en annan funktion som parameter, är den en *högre ordningens funktion*.

Mer exempel:

- `assert split_at([2,3,4,2,5], lambda x: x==2) == [[], [3,4], [5]]`
  (Om sekvensen delas vid första elementet, blir det en tom lista först i resultatet)
- `assert split_at([1,2,3,4,2], lambda x: x==2) == [[1], [3,4], []]`
  (Om sekvensen delas på slutet, blir det en tom lista sist i resultatet)
- `assert split_at([1,2,2,3,4,2,5], lambda x: x==2) == [[1], [], [3,4], [5]]`
  (Här får vi två delningar efter varandra, och därmed en tom lista mitt i resultatet)
  (Tänk på att det också kan bli flera delningar *först* eller *sist* i `seq`!)
- `assert split_at("abcdeba", lambda x: x=="b") == [["a"], ["c", "d", "e"], ["a"]]`
  (Sekvensen är inte nödvändigtvis en lista, utan även strängar, tupler med mera är sekvenser.)
  (Tips: Alla typer av sekvenser kan hanteras med `for`-loop.)
  (Elementen är av godtycklig typ.)
  (Resultatet blir alltid en lista av listor, även om ursprunget var en annan sekvenstyp.)
- `assert split_at([1,2,3,4,5], lambda x: x % 2 == 0) == [[1], [3], [5]]`
  (Funktionen `pred` kan se ut hur som helst...)

Ytterligare villkor:

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

Deluppgift 4b (2.5p)

Skriv en högre ordningens funktion `add_for_each(seq: list, func)` som tar en rak lista och en funktion som parameter. Funktionen `add_for_each()` ska applicera den givna funktionen `func()` på varje element i listan `seq` och returnera summan av resultaten.

Man får anta att `func(...)` alltid returnerar ett tal.

Exempel:

- `assert add_for_each([1, 2, 3, 4], lambda x: x**2) == 30`
  (Eftersom `1**2 + 2**2 + 3**2 + 4**2 == 1+4+9+16 == 30`)
- `assert add_for_each([], lambda x: x**2) == 0`
  (Listan är tom, och vi returnerar då summan av 0 resultat, vilket är 0)
- `assert add_for_each([[1, 2, 3], [1], [1, 2, 3, 4]], lambda x: len(x)) == 8`
  (Listan `seq` har tre element, som själv råkar vara listor;
  lambdafunktionen applicerar `len` på dessa listor;
  resultat `len([1, 2, 3]) + len([1]) + len([1, 2, 3, 4]) == 3+1+4 == 8`)

Ytterligare villkor:

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

Uppgift 5: Attraktiva tal (5p)

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. Se ytterligare instruktioner tidigare i tentan.

I denna uppgift ska du avgöra om ett *tal* är *attraktivt*. För att göra detta ska du skriva tre funktioner. Funktionerna kan ge delpoäng oberoende av varandra (man kan alltså få poäng för steg 3 även om steg 1 och 2 inte fungerar). Poängsumman bedöms utifrån uppgiften i sin helhet.

Steg 1: Primtal

Om ett tal är attraktivt eller inte beror på vissa egenskaper hos primtal. Vi behöver därför börja med att skriva följande funktion.

- `is_prime(n: int)` ska ta ett heltal `n` ≥ 1 och returnera `True` om heltalet `n` är ett primtal. Annars ska funktionen returnera `False`.

Tips: Det lägsta primtalet är 2. Om ett tal $n > 2$ är *jämnt delbart* med något heltal från och med 2 upp till och med `int(math.sqrt(n))`, är n inte ett primtal (det gick ju att dela upp det) och funktionen ska returnera `False`. Annars *är* det ett primtal och funktionen ska returnera `True`.

Notera alltså att man *inte* behöver försöka dela med alla tal upp till n. **För full poäng ska man bara testa division med tal upp till och med `int(math.sqrt(n))` divisioner**, så att även stora tal kan testas snabbt.

Exempel:

- `assert is_prime(1) == False`
- `assert is_prime(2) == True`
- `assert is_prime(10) == False`
- `assert is_prime(11) == True`
- Alla primtal under 100: 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97
- `assert is_prime(10000000019) == True`
  Detta funktionsanrop returnerar *snabbt* om man bara testar division med tal upp till och med cirka *roten ur* 10000000019 enligt ovan (alltså upp till cirka 100000). Det kan ta *lång tid* om man testar med för många divisioner. (På en testmaskin: 5 millisekunder vid snabb implementation, 5 minuter vid långsam implementation.)

Tips:

- Om du använder `range()`, tänk på att t.ex. `range(10)` räknar upp tal till och med 9, *inte till och med 10*.

Steg 2: Primtalsfaktorisering

Vi behöver också kunna dela upp ett positivt heltal n i sina *primfaktorer*. I detta fall ska vi hitta en *lista* som enbart innehåller *primtal*, så att produkten av alla tal i listan är lika med det ursprungliga talet n. Exempelvis har talet 20 primfaktorerna `[2,2,5]`, eftersom 2 och 5 är primtal och $2 \cdot 2 \cdot 5 = 20$. Implementera därför följande funktion.

- `prime_factors(n: int)` ska ta ett heltal `n` ≥ 2 och returnera en *lista av heltal* som innehåller samtliga primfaktorer i `n`.

  Returvärdet får ange primfaktorerna i godtycklig ordning: `prime_factors(20)` får t.ex. returnera `[2,2,5]` eller `[2,5,2]` eller `[5,2,2]`.

Tips:

- Primtalsfaktoriseringen behöver inte vara effektiv (snabb) för stora tal.

Exempel: Vi sorterar returvärdet innan vi jämför med förväntat resultat, så att ordningen i det faktiska returvärdet inte ska spela någon roll.

- `assert sorted(prime_factors(2)) == [2]`
- `assert sorted(prime_factors(10)) == [2, 5]`
- `assert sorted(prime_factors(20)) == [2, 2, 5]`
- `assert sorted(prime_factors(55)) == [5, 11]`
- Skriv gärna egna tester som åtminstone *skriver ut* resultatet från `prime_factors(n)` för olika `n`, t.ex. i en loop. Se sedan om resultaten ser rimliga ut.

Steg 3: Attraktiva tal

Ett tal är ett **attraktivt tal** (*attractive number*) om antalet primfaktorer i talet är ett primtal.

Som vi har sagt tidigare har talet 20 primfaktorerna `[2,2,5]`. Det finns alltså 3 primfaktorer, och eftersom 3 är ett primtal är 20 ett attraktivt tal. Talet 40 har istället 4 primfaktorer, `[2,2,2,5]`, och eftersom 4 inte är ett primtal är 40 *inte* ett attraktivt tal.

Med hjälp av de tidigare funktionerna kan du nu till slut skriva följande funktion:

- `is_attractive(n: int)` ska ta ett heltal `n` ≥ 2 och returnera `True` om `n` är attraktivt. Annars ska funktionen returnera `False`.

Exempel:

- `assert is_attractive(16) == False`
- `assert is_attractive(20) == True`
- `assert is_attractive(21) == True`
- `assert is_attractive(22) == True`
- `assert is_attractive(23) == False`
- `assert is_attractive(24) == False`
- `assert is_attractive(55) == True`

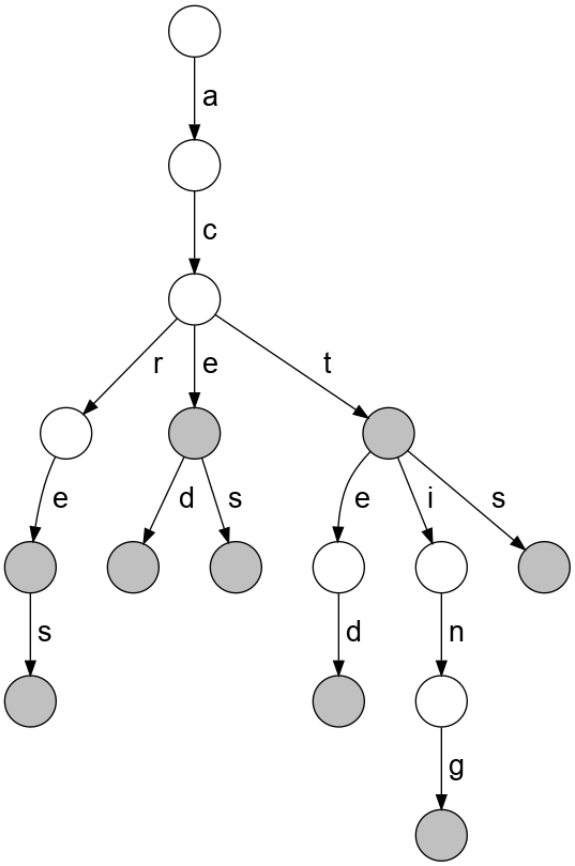
Uppgift 6: Trie (5p)

Använd den givna filen `ex6.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.

En Trie (även kallat “prefixträd") är en trädbaserad datastruktur som kan användas för att **lagra** en mängd strängar, **testa** om en sträng ingår i mängden, och **hitta** alla fortsättningar på ett prefix (förklaras på nästa sida).

Till höger syns ett exempel på en Trie. Där kan vi följa vägen från *rotnoden* (överst) nedåt till gråmarkerade *slutnoder* för att hitta orden *ace* (rakt nedåt!), *aced, aces, acre, acres, act, acted, acting* och *acts*. Effektiviteten uppnås genom att gemensamma prefix bara lagras en gång, även när det finns längre ord som börjar på samma sätt (act/acted). Detta kommer framförallt att göra *testning* snabbare, alltså när man ska avgöra om ett av många liknande ord ingår i en Trie eller inte.



Eftersom en nod kan representera ett ord (“ace") trots att den har barn (“aces") behöver noden själv hålla reda på om den är en *slutnod* (grå) eller inte (vit). Detta lagras lämpligen som ett sant/falskt värde i noden.

Varje nod behöver dessutom hålla reda på sina *barn* (noll eller flera noder), och vilken *bokstav* som används för att nå varje barn: Efter “act" kan vi nå tre olika barnnoder via bokstäverna “e", “i" respektive “s". Barnen kan t.ex. lagras i en `dict` där nyckeln är barnets bokstav och värdet är en nod; när noden först skapas har den inga barn och har då en tom barn-`dict`.

Noden behöver alltså vara *någon* sorts datastruktur som kan hålla reda på (1) om den är en slutnod eller inte, och (2) vilka barn som finns och genom vilken bokstav varje barn nås.

Deluppgift 6a: Grunderna (4p)

Din första uppgift är att själv utveckla och implementera en konkret datastruktur för att representera en Trie, baserat på diskussionen och illustrationen på förra sidan. Du får antingen skapa egna namngivna datatyper eller helt enkelt använda listor, tupler, dictionaries och vad som nu kan tänkas behövas.

Datastrukturen ska manipuleras med följande funktioner, som du ska skriva:

- `create_trie()` skapar och returnerar en tom Trie. Du kan alltså välja själv hur detta ska representeras, så länge som de andra funktionerna du skriver också förstår den representation du har valt.
- `add_word(trie, word: str)` adderar ett nytt ord (en icke-tom sträng) till en Trie.

  Funktionen **ska uppdatera parametern `trie`**, inte returnera en ny Trie.

  Denna funktion ska alltså modifiera sina indata! Till skillnad från de flesta funktioner finns den bara till för att just göra en ändring i en datastruktur.
- `word_in_trie(trie, word: str)` avgör om ett ord (en icke-tom sträng `word`) finns med i `trie` – om den motsvarar ett ord som tidigare hade adderats med `add_word()`.

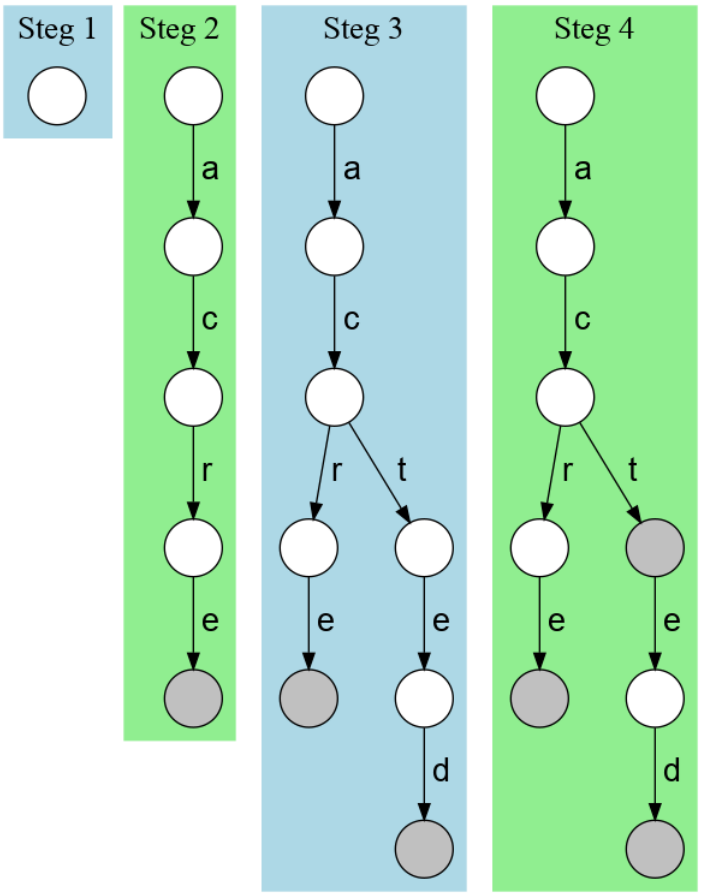
  Funktionen ska returnera `True` om ordet word finns i `trie`, och annars `False`.

Tips:

- Eftersom datastrukturen ska modifieras av `add_word()` kan det vara enklast att undvika datatyper som inte kan modifieras, som tupler och `NamedTuple`.

Se exempel på nästa sida!

Illustrerande exempel: Här syns fyra steg i konstruktionen av en Trie. I denna bild har vi gjort följande:



1) `mytrie = create_trie()`
Resulterar i strukturen i "Steg 1" – en tom Trie. Denna Trie är *inte* en slutnod, eftersom den inte motsvarar ett ord; vi har ju inte lagt till några ord ännu! Rotnoden har noll barn.

Vi *illustrerar* detta med en tom cirkel motsvarande en nod. Hur du *representerar* detta i Python är upp till dig.

2) `add_word(mytrie, "acre")`
Resulterar i strukturen i "Steg 2", där rotnoden har fått ett barn, som i sin tur har barn, och så vidare. I sista noden markeras med grå färg att ordet "acre" är ett ord som faktiskt ingår i vårt Trie, något som alltså också ska framgå genom någon sorts information i noden. De tidigare noderna, som nås via strängarna "a", "ac" och "acr", är vita i illustrationen eftersom dessa strängar inte "ingår" i mängden (vi har inte adderat ordet "ac" till `mytrie`).

3) `add_word(mytrie, "acted")`
Resulterar i strukturen i "Steg 3". Rotnoden hade redan ett barn via "a", som i sin tur redan hade ett barn via "c". Nästa bokstav i "acted" är "t", men vår nuvarande nod (den tredje uppifrån) har bara ett barn för "r". Alltså måste man *lägga till* ett nytt barn för "t". I "Steg 3" har vi då förgrenat strukturen genom att skapa ett nytt barn mitt i trädet.

Den sista nya noden, som nås via hela ordet "acted", är grå.

4) `add_word(mytrie, "act")`
Resulterar i strukturen i "Steg 4". Vi har följt de existerande bågarna för bokstäverna "a", "c" och "t". Alla dessa fanns redan i trädet, så vi behövde inte skapa några nya noder. Däremot måste vi markera det nya ordet – noden som nås via "a", "c" och "t" är nu grå (vi har modifierat värdet som talar om huruvida det är en slutnod eller inte.)

Fortsättning på nästa sida!

Ytterligare exempel och tester:

```
trie = create_trie()

for word in ["ace", "aced", "aces", "acre", "acres", "act",
        "acted", "acting", "acts"]:
    add_word(trie, word)
    assert word_in_trie(trie, word)

for word in ["ace", "aced", "aces", "acre", "acres", "act",
        "acted", "acting", "acts"]:
    assert word_in_trie(trie, word)

for word in "En Trie är en effektiv datastruktur".split(" "):
    assert not word_in_trie(trie, word)
```

Deluppgift 6b: Fortsättning på prefix (1p)

I uppgift 6b ska du implementera följande funktion:

- `find_all_matches(trie, prefix: str)` returnerar en *mängd* (`set`) med alla ord (strängar) in `trie` som börjar på `prefix`. Funktionen får inte göra ändringar i `trie`.

Exempel:

- Givet att man har kört testerna från deluppgift 6a:
  `assert find_all_matches(trie, "ace") == {"ace", "aced", "aces"}`.

  (Notera att ordningen inte spelar roll i en mängd, så om du *skriver ut* resultatet av `find_all_matches()` kan orden komma i en annan ordning än ovan. En *jämförelse* kommer ändå att fungera: `{"ace", "aced", "aces"} == {"aced", "ace", "aces"}`.)