Datortentamen 2022-08-16

TDDE24 Funktionell och imperativ programmering del 2

Allmän information

Tentan består av totalt **6 uppgifter** som vardera kan ge maximalt 5 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), atom, 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 menyer och tangentbordsbindningar som man är van vid, inte att man ska ha en fullständig integrerad utvecklingsmiljö som hjälper till med allt man ska göra.
- Standardiserade **systemprogram** i kommandoradsprompten eller menyerna.
- **Websidor** under `https://docs.python.org/3.9/` (via "Web Access"-ikonen). 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 utvecklingsmiljöer som PyCharm och Thonny 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 tester inuti run_tests(). Testkör 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. Deklarationerna är bortkommenterade, så du får själv ta bort kommentarstecknen “#". 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 behöver ändå tänka på att skapa egna tester.)

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 hjälp av **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. Alla uppgifter 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 lämnar in 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.

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** – dels går det inte att spendera varje sekund klistrad framför tentaklienten, dels kan det 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.

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. Ibland är de skrivna direkt i `assert`-form och ibland beskrivs de indirekt som del av 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`). 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* vad som egentligen menas. Testerna är inte ett facit.

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: 100 dörrar (5p)

Använd den givna filen `ex1.py` för hela denna uppgift!
Kopiera den från `given_files` till desktop (eller annan skrivbar mapp) och editera den där. Filen innehåller bland annat testfall och den **ska** användas för inlämningen.

Tänk dig att det finns 100 dörrar i rad (nummer 1 till nummer 100) i en lång korridor, och att alla dörrar från början är *stängda*. Vad händer då om du gör detta?

- Gå förbi *varje* dörr (nummer 1, 2, 3, ..., 100) och byt läge på den: Öppna den om den var stängd, eller stäng den om den var öppen.
- Gå förbi *varannan* dörr (nummer 2, 4, 6, ..., 100) och byt läge på den.
- Gå förbi *var tredje* dörr (nummer 3, 6, 9, ..., 99) och byt läge på den.
- Gå förbi *var fjärde* dörr (nummer 4, 8, 12, ..., 100) och byt läge på den.
- ...
- Gå till slut förbi *var hundrade dörr* (bara nummer 100!) och byt läge på den.

Vilka dörrar kommer att vara öppna när du har gjort detta?

Skriv funktionen `doors()`, som utför proceduren som har beskrivits ovan och returnerar en lista med index för de dörrar mellan 1 och 100 som är öppna. Listan ska anges i nummerordning.

Om `doors()` returnerade `[1,2,4,8,16]` skulle det alltså tolkas som att dörrarna 1, 2, 4, 8 och 16 är öppna, medan alla andra dörrar är stängda. (Detta är inte det korrekta resultatet, utan bara ett exempel som förklarar hur returvärdet tolkas.)

Allmänna tips och ledtrådar:

- Det slutliga returvärdet ska vara en lista med index (heltal). Det går utmärkt att använda den representationen även internt, när man itererar och uppdaterar vilka dörrar som är öppna. Det går också att använda andra interna datastrukturer om det gör det enklare att komma på en lösning. Då måste funktionen så klart konvertera detta till det förväntade resultatet innan värdet returneras.
- Beroende på vilken datastruktur du använder internt: Tänk på att en listas första index är 0, medan den första dörren har index 1.

Exempel på nästa sida.

Exempel: Här ger vi exempel på några dörrar som ska vara öppna (`in result`) respektive stängda (`not in result`).

- `result = doors()`
- `assert result == sorted(result) # Testa korrekt ordning`
- `assert 1 in result`
- `assert 2 not in result`
- `assert 3 not in result`
- `assert 4 in result`
- `assert 5 not in result`
- `assert 6 not in result`
- `assert 7 not in result`
- `assert 8 not in result`
- `assert 48 not in result`
- `assert 49 in result`
- `assert 50 not in result`

Uppgift 2: Fusc-sekvensen (5p)

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

Nu är det dags att implementera *fusc-sekvensen* (!), som definieras så här för heltal $n \geq 0$:

- $\mathsf{fusc}(0) = 0$
- $\mathsf{fusc}(1) = 1$
- Om $n > 1$ och n är jämnt: $\mathsf{fusc}(n) = \mathsf{fusc}(n/2)$
- Om $n > 1$ och n är udda: $\mathsf{fusc}(n) = \mathsf{fusc}((n-1)/2) + \mathsf{fusc}((n+1)/2)$

Som man kan se ovan arbetar vi hela tiden med heltal: Indata är ett heltal, och det är alltid *jämna* tal som delas med 2.

Din uppgift kommer att vara att implementera två funktioner (en rekursiv och en iterativ) som tar ett heltal $n \geq 0$ och beräknar värdet av $\mathsf{fusc}(n)$.

Deluppgift 2a: Rekursiv implementation (2.5p)

Den rekursiva implementationen, `fusc_r(n: int)`, är antagligen den enklaste i denna uppgift: Den matematiska definitionen är ju redan rekursiv, så du behöver helt enkelt överföra den matematiska notationen till en definition av en Python-funktion.

Testfall finns på nästa sida.

Deluppgift 2b: Iterativ implementation (2.5p)

I den andra deluppgiften skapar du istället en iterativ implementation, `fusc_i(n: int)`. Denna funktion får alltså *inte* anropa sig själv (eller anropa en hjälpfunktion som sedan anropar sig själv), utan ska vara rent iterativ.

Iterationen inuti `fusc_i(n)` kan t.ex. genomföras så här:

- Lägg in värdena av $\mathsf{fusc}(0)$ och $\mathsf{fusc}(1)$ i en lista som används som *uppslagningstabell*.
- Beräkna i tur och ordning värdet på $\mathsf{fusc}(k)$ för varje heltal `k` från `2` till `n`. I beräkningen av $\mathsf{fusc}(k)$ *anropar* du inte funktioner som $\mathsf{fusc}(k/2)$ rekursivt, utan använder den existerande tabellen för att *slå upp* värden som du redan har beräknat. När $\mathsf{fusc}(k)$ är beräknat sparar du undan det i din uppslagningstabell så det kan användas igen vid beräkning av senare värden.

Iterationen bygger alltså steg för steg upp en lista som innehåller $\mathsf{fusc}(0)$, $\mathsf{fusc}(1)$, $\mathsf{fusc}(2)$, $\mathsf{fusc}(3)$, $\mathsf{fusc}(4)$, och så vidare. Denna lista används för att effektivare beräkna det slutliga värdet $\mathsf{fusc}(n)$, som är det enda värde vi egentligen är intresserade av.

Testfall finns på nästa sida.

Exempel och testfall

Båda funktionerna ska bl.a. klara följande **tester**, där `fusc_x` är antingen `fusc_i` eller `fusc_r`.

- `assert fusc_x(2) == 1`
- `assert fusc_x(3) == 2`
- `assert [fusc_x(k) for k in range(10, 20)] == [3, 5, 2, 5, 3, 4, 1, 5, 4, 7]`

Man kan också testa att båda implementationerna ger samma resultat:

- `for k in range(0, 1000): assert fusc_i(k) == fusc_r(k)`

Uppgift 3: Element i nästlade listor (5p)

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

I denna uppgift vill vi hantera *listor med samma nästlade struktur*. Med detta menar vi t.ex.:

- `seq1 = [1,2,3],`
  `seq2 = [9,8,7]`
- `seq1 = [["a"], 6, [2, (3, 5)]]],`
  `seq2 = [["b"], 5, [1, (1, 1, 42)]]]`

Mer formellt har två listor `seq1` och `seq2` "samma nästlade struktur" om och endast om (1) de är lika långa, och (2) ett av följande villkor gäller för elementen `elem1=seq1[pos]` och `elem2=seq2[pos]` på alla positioner `pos` i de två listorna:

- Både `elem1` och `elem2` är listor, exempelvis `[2, (3, 5)]` och `[1, (1, 1, 42)]`, och dessa listor har också *samma nästlade struktur* som varandra, eller
- Varken `elem1` eller `elem2` är listor (exempelvis tuplerna `(3, 5)` och `(1, 1, 42)`). Sådana element, som inte är listor, kallar vi nu för **grundelement**.

Du ska skriva funktionen `greater_nested(seq1: list, seq2: list)`, som tar två godtyckliga (möjligen tomma) nästlade listor `seq1` och `seq2` med *samma nästlade struktur* och som returnerar en *mängd* (`set`) med de grundelement i `seq1` som är *större* än motsvarande grundelement på "samma" position i `seq2`. För att testa "större än" används operatorn `>`. Exempel:

- `assert greater_nested([1, 2], [15, 4.25]) == set()`
- ```
  assert greater_nested(
  [[['a'], 6, [2, (3, 5)]]],
  [[['b'], 5, [1, (1, 1, 42)]]]) == {2, (3, 5), 6}
  ```
- ```
  assert greater_nested(
  [[['b'], [[[3.14]]], [242, (3, 5)]]],
  [[['a'], [[[2]]], [1, (1, 1, 42)]]]) == {242, 3.14, 'b', (3, 5)}
  ```

Ytterligare villkor:

- Funktionerna som specificeras i uppgiften får **inte** modifiera sina indata.
- Du får förutsätta att jämförelseoperatorn '>' alltid fungerar för två element som är på motsvarande positioner i `seq1` och `seq2`. Funktionen kommer alltså inte att testas med anrop som `greater_nested([2, 1], [4, "a"])` där det inte går att jämföra `1` och `"a"`, och koden behöver inte testa om sådana element skickas in utan tillåts att krascha i dessa fall.

Tips på nästa sida.

Allmänna tips och ledtrådar:

- För att lägga till elementen ur `set2` i `set1` används `set1.update(set2)`.
- Listor kan nästlas i godtyckligt antal nivåer. En lösning behöver klara av detta, vilket oftast är enklast om man gör den rekursiv. 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 ännu inte har lärt oss.
- Vi behandlar nästlade *listor*. Tupler är inte listor, och som synes i exemplet ovan ska man *inte* rekursera ner i dem.
- Listorna kan innehålla *godtyckliga* element, så länge som elementen på motsvarande positioner kan adderas med varandra. Försök *inte* att specialbehandla några andra typer än just listor.

Uppgift 4: Högre ordningens funktioner (5p)

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

Deluppgift 4a (4p)

Curryfiering (engelska **currying**) är ibland användbart i t.ex. funktionell programmering. Något förenklat kan man se detta som att man har en funktion som tar flera argument, och att man nu vill “fixera" värdet på det första argumentet så att man slipper ange det värdet vid nästa anrop. Man vill med hjälp av curryfiering “ta bort" det första argumentet så att man bara behöver ange de övriga argumenten.

Detta kan man göra med en högre ordningens funktion som vi kan kalla `curry`.

(Kuriosa: Namnet kommer från Haskell Brooks Curry, en matematiker och logiker som också har fått ge namn åt de tre programmeringsspråken Haskell, Brook och Curry.)

Anta till exempel att du har funktionen `plus(a,b)` som lägger ihop två tal. Denna funktion tar som synes två parametrar, men genom att anropa `curry(plus, 5)` ska man istället få fram en ny funktion som bara tar ett heltal:

- `plusfive = curry(plus, 5)`
- `assert plusfive(1) == 6`
- `assert plusfive(2) == 7`
- `hundreddiv = curry(div, 100)`
- `assert hundreddiv(5) == 20`

Din uppgift är att skriva den högre ordningens funktionen `curry(fn, v1)`, som tar två parametrar: En funktion `fn` med två argument, och ett annat värde `v1` som (senare) ska skickas in som första parameter till `f`. Funktionen ska då returnera en ny funktion med *ett* argument `v2`, som vid anrop returnerar värdet av `fn(v1, v2)`.

Tips:

- Du kan använda lambdauttryck, men det går också att lösa detta utan lambda.
- För att inte förvirra dig själv i onödan: Använd de parameternamn som vi har använt i uppgiften (`fn`, `v1`, `v2`).

Exempel:

- Se ovan. Testfallen finns även i svarsmallen.

Deluppgift 4b (1p)

I Pythons standardbibliotek finns funktionen `math.pow(x,y)` som returnerar x^y där `x` och `y` är flyttal. Importera denna funktion.

Använd sedan `curry` för att definiera funktionen `pow2(z)` som returnerar 2^z. Detta ska definieras med hjälp av ett enda uttryck och utan att använda `def`:

- `pow2 = ...`

Du ska inte definiera egna hjälpfunktioner utöver `curry`.

Exempel:

- `assert pow2(3) == 8`
- `assert pow2(8) == 256`

Uppgift 5: Trie (5p)

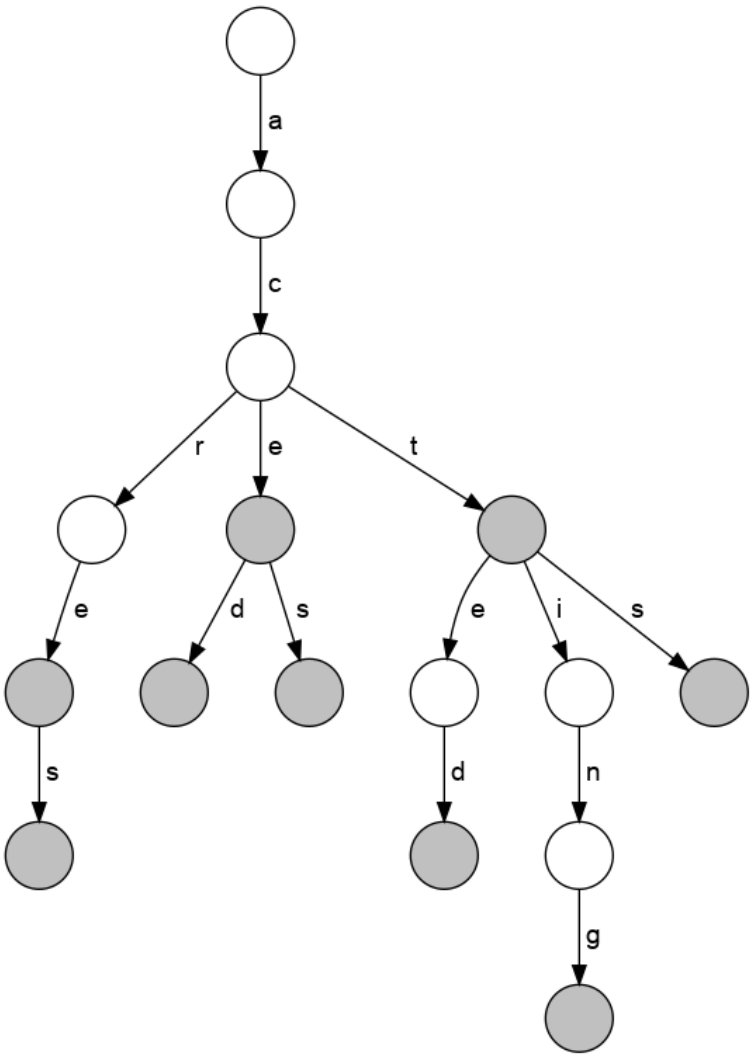
Använd den givna filen `ex5.py` för hela denna uppgift!

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.

I exemplet nedan kan vi följa vägen från *rotnoden* (överst) nedåt till gråmarkerade *slutnoder* för att hitta orden *ace, 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).

Eftersom en nod kan representera ett ord (“ace") trots att den har barn (“aces") behöver noden alltså själv hålla reda på om den är en *slutnod* (grå) eller inte.

Den behöver dessutom hålla reda på sina *barn* (noll eller flera), och vilken *bokstav* som används för att nå varje barn: Efter “act" kan vi nå tre olika barnnoder via bokstäverna “e", “i" respektive “s".



Deluppgift 5a: 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 klara följande anrop:

- `create_trie()` skapar och returnerar en tom Trie.
- `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!
- `word_in_trie(trie, word: str)` avgör om ett ord (en icke-tom sträng) finns med i `trie`. Funktionen ska returnera `True` om ordet word finns i `trie`, och annars `False`. Se de kommande exemplen!

Exempel:

```
trie = create_trie()

for word in ["ace", "aced", "aces", "acre", "acres", "act",
        "acted", "acting", "acts"]:
    add_word(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 5b: Fortsättning på prefix (1p)

I uppgift 5b 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`.

  (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.)

Exempel:

- Givet testen från deluppgift 5a:
  `assert find_all_matches(trie, "ace") == {"ace", "aced", "aces"}`.

Uppgift 6: Run Length Encoding (5p)

Använd den givna filen `ex6.py` för hela denna uppgift!

Det finns många sätt att *komprimera* information så att den tar upp mindre plats. Många sådana metoder är ganska komplicerade, men för vissa typer av information kan enklare tekniker också ge goda resultat.

Tänk dig till exempel att du har en sekvens där du ofta får samma element många gånger i följd. Det kan till exempel vara en svartvit bild där du ofta har långa sekvenser av helt vita pixlar, följt av några sammanhängande svarta pixlar. Då kan du använda så kallad *run length encoding (RLE)*, på svenska *skurlängdskodning*.

Resultatet av skurlängdskodning kan bli en lista av tupler (a, b), där varje sådan tupel anger att det finns $a > 0$ element i rad med värdet b.

Exempel:

- `rle("ABBBBCCCCCCCCCCAAA") == [(1, 'A'), (4, 'B'), (10, 'C'), (3, 'A')]` (Den ursprungliga teckensekvensen bestod av 1 'A', 4 'B', 10 'C' och 3 'A'.)

Att göra: Skriv två implementationer av skurlängdskodning. Båda ska ta en godtycklig sekvens av element och returnera motsvarande skurlängdskodade lista enligt ovan.

- `rle_i(seq)` ska arbeta enligt **iterativ** lösningsmodell.
- `rle_r(seq)` ska arbeta enligt **rekursiv** lösningsmodell. Kom ihåg: Detta innebär inte bara att funktionen anropar sig själv, utan att varje rekursivt anrop ska beräkna och returnera *det korrekta svaret för ett delproblem*. Anroparen använder sedan detta svar för att beräkna sitt resultat för ett större problem. Ett exempel är fakultetsfunktionen där `fac(n) = n * fac(n-1)`, där `fac(n-1)` är ett rekursivt anrop som löser delproblemet för ett mindre värde på `n`.

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

Ytterligare villkor:

- Inputsekvensen `seq` kan ha godtycklig längd, inklusive 0, och kan innehålla godtyckliga element.
- Du får inte använda listbyggare (*list comprehensions*) eller inbyggda funktioner som behandlar alla element i hela listor, utan måste själv iterera eller rekursera över listorna, *ett element i taget*. Funktioner som inte behandlar elementen, såsom `len()`, är tillåtna.
- Funktionerna som specificeras i uppgiften får **inte** modifiera sina indata.

Exempel på nästa sida.

Exempel. Båda funktionerna ska bl.a. klara följande **tester**, där `rle` är antingen `rle_i` eller `rle_r`. Skapa gärna fler tester med inspiration av villkoren ovan!

- `assert rle([]) == []`
- `assert rle([1,2,2,2,2,5,5,5,5,5,5,1,1,1]) == [(1,1), (4,2), (6,5), (3,1)]`
- ```
  assert rle(["a","a","b","a","c","c"]) ==
      [(2, "a"), (1, "b"), (1, "a"), (2, "c")]
  ```
  ```