Tentamen Programmeringsteknik II 2024-05-29

Skrivtid: 08:00 – 13:00

Nedladdning av skrivningens filer

Du laddar ner filerna för tentan från länken i Inspera som kallas Files.

Kopiera innehållet i filerna, som är i `.txt`-format, in i filer som namnges enligt följande:

`main.txt` –> `main.py`
`m1.txt` –> `m1.py`
`m2.txt` –> `m2.py`
`m2b.txt` –> `m2b.py`
`m2init.txt` –> `m2init.txt` (ändra inte till `.py`)
`m2binit.txt` –> `m2binit.txt` (ändra inte till `.py`)
`m2tokenizer.txt` –> `m2tokenizer.py`
`m3.txt` –> `m3.py`
`m4.txt` –> `m4.py`
`m4_data.txt` –> `m4_data.txt` (ändra inte till `.py`)

Arbete med uppgifterna

- Skrivningen innehåller A- och B-uppgifter. A-uppgifter måste fungera (inlämnade program ska kunna köras och lösa uppgiften) för att godkännas. B-uppgifter kan ge ”poäng” även om de inte löser problemen fullständigt.
- Skriv dina lösningar *på anvisade platser* i filerna `m1.py`, `m3.py` och `m4.py`. I `m2.py` och `m2b.py` måste du delvis själv hitta var ändringarna och tilläggen ska göras.
- I editorerna är det alltid `main.py` som körs. Genom att kommenter/avkommentera rader i `main.py` väljer du vilka uppgifter som ska testköras.
- Det har hänt att enskilda editorer kraschar under tentan. Därför ta, med jämna mellanrum, för säkerhets skull en “backup” av din kod genom att kopiera in din kod i Inspera. T.ex. jobbar du med `m1.py` så kopierar du innehållet från editorn och klistrar in i uppgiftesrutan för `m1.py` i Inspera (ctrl-c för att kopiera, ctrl-v för att klistra in). Om editorn skulle kracha så tappar du inte så mycket tid.

Regler

- Du måste behålla namn på filer, klasser, metoder och funktioner. Funktioner måste gå att anropa exakt på det sätt som står i uppgiften. Om funktioner ska returnera något, så ska det vara enligt beskrivningen i uppgiften.
- Du får inte använda andra paket än de som redan är importerade i filerna såvida inte annat sägs i uppgiften.
- Du får skriva och använda hjälpfunktioner.

Inlämning

För inlämning klistrar du in hela `m1.py`, `m2.py`, `m2b.py`, `m3.py` och `m4.py` i motsvarande ruta i Inspera. En fullständig inlämning består således av alltså sammanlagt 5 filer. Du ska inte ladda upp tokenizern eller någon `init`- eller `data`-fil.

Betygskrav:

3: Minst sex A-uppgifter godkända, varav minst en uppgift godkänd i varje modul.

4: Minst sju A-uppgifter godkända samt antingen två B-uppgifter stort sett korrekta.

5: Minst sju A-uppgifter godkända samt fyra B-uppgifter i stort sett korrekta

En godkänd VA-uppgift räknas som en B-uppgift.

Notera att vi *kan* sänka betygskraven, så det är värt att lämna in tentamen även om man inte strikt uppfyller de angivna kraven ovan.

Uppgifter i anslutning till modul 1

Lösningarna till dessa uppgifter ska skrivas på anvisade platser i filen `m1.py`.

A1: Skriv funktionen `digit_sum(x)` som beräknar och returnerar (den digitala) siffersumman i det icke-negativa heltalet `x`.

Exempel:

```
digit_sum(0)                # should return 0
digit_sum(12)               # should return 3
digit_sum(712)              # should return 10
digit_sum(1021)             # should return 4
```

Uppgiften ska lösas med rekursion och får alltså inte innehålla några iterationer och inte heller använda några av Pythons list- eller sträng-funktioner.

Följande funktion används i uppgift A2 och B1:

```
def foo(n):
    if n<3:
        return 1
    else:
        return n + 2*foo(n-1) - foo(n-2) - foo(n-3)
```

A2: Funktionen är bara användbar för små värden på argumentet `n`. Skriv funktionen `foo_mem` så att den beräknar samma resultat men använder tekniken med memorering (eng ”memoization”) av beräknade resultat för att effektivt kunna beräkna till exempel `foo(100)`. Funktionen ska alltså fortfarande vara rekursiv.

B1: Originalversionen av `foo` tidtogs med nedanstående kod. Till höger visas utskrifterna. Tiderna är i sekunder.

```
for i in (28, 29, 30, 31):
    tstart = time.perf_counter()
    foo(i)
    dt = time.perf_counter()-tstart
    print(f'{i:3d} : {dt:4.2f}')
```

```
 28 : 0.96
 29 : 1.77
 30 : 3.39
 31 : 5.99
```

a) Ange ett Θ-uttryck för tidskomplexiten som mätningarna indikerar!
b) Uppskatta hur lång tid anropet `foo(100)` skulle ta!
c) Ange ett icke-trivialt Ω-uttryck (dvs en *undre* begränsning) för tidskomplexiteten som funktionen *garanterat* har. Med ”icke-trivialt” menas att uttrycket, även om det är en undre begränsning, ska uttrycka det väsentliga beteendet hos tidskomplexiteten!

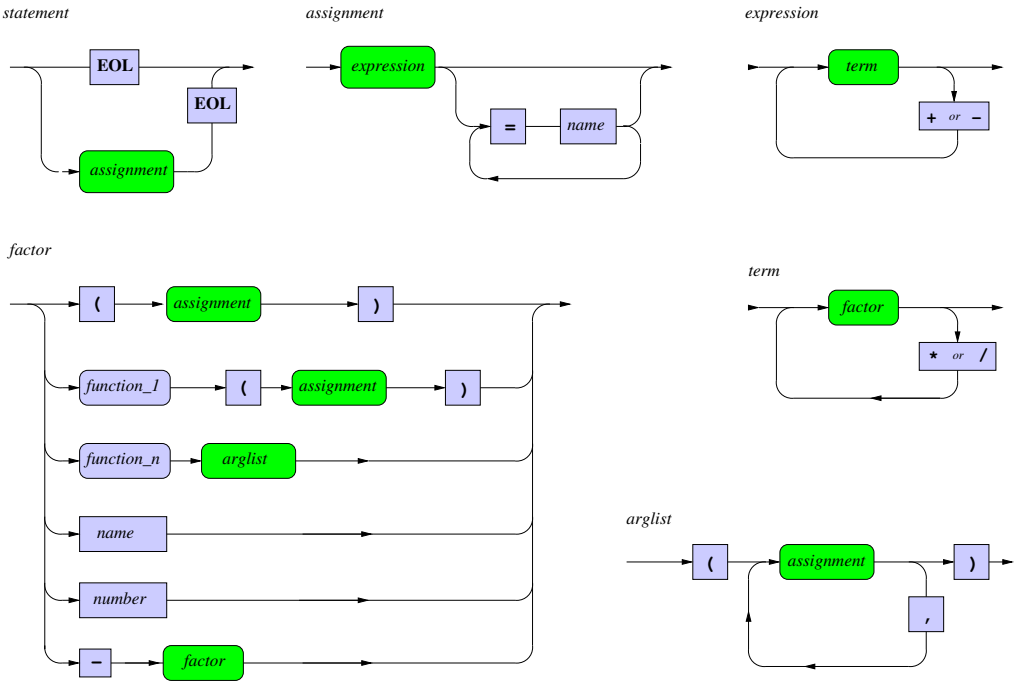
Observera att tidmätningar i examinationssystemen är helt otillförlitliga. Använd de i uppgiften angivna värdena!

Motivera alla svar!

Uppgifter i anslutning till modul 2

Den givna filen `m2.py` implementerar en kalkylator liknande den i den andra obligatoriska uppgiften.

Syntaxen för uttrycken beskrivs av följande diagram:



(Den givna koden hanterar inte funktioner med flera parametrar.) Tokenizern och en initieringsfilen `m2init.txt` finns också med bland de nedladdade filerna.

Programmet börjar med att läsa från filen `m2init.txt` som innehåller testfall för uppgifterna A3 och A4. Om du inte vill att programmet börjar med att läsa testfilen så kan du byta namn på den.

A3: Lägg till || för absolutbelopp. Exempel:

```
1
2  init   : |1-3|                    # Expects 2
3  Result: 2
4
5  init   : ||1-3| - |1-8||           # Expects 5
6  Result: 5
7
8  init   : -|2 + |3-1||              # Expects -4
9  Result: -4
10
11 init   : |1 - 3 +|8|               # Expects SyntaxError
12 *** Syntax error:  Expected '|'
13 Error occurred at '# Expects SyntaxError' just after '|'
14
15 init   : |1 - 2)|                  # Expects SyntaxError
16 *** Syntax error:  Expected '|'
17 Error occurred at ')' just after '2'
```

A4: Modifiera koden så att en sats (”a statement”) kan innehålla flera tilldelningar (assignments) åtskilda av komma tecken.

Exempel:

```
init  : 1, 2=a, 2+2, 1+1+3, 2+1   # Expects 1, 2, 4, 5, 3
1, 2, 4, 5, 3

init  : ans                       # Expects 3
Result: 3

init  : 3=x, 2=y, x*y=z, z*z      # Expects 3, 2, 6, 36
3, 2, 6, 36

init  : 3, , 4,                   # Expects SyntaxError
*** Syntax error:  Expected number, word or '('
Error occurred at ',' just after ','

init  : 9=x, 3, 4/(2*2-4), 8=y    # Expects EvaluationError
*** Evaluation error: Division by zero

init  : ans, x, y                 # Expects 36, 9, 2
36, 9, 2

init  : (1, 2, 3)                 # Expects SyntaxError
*** Syntax error:  Expected ')'
Error occurred at ',' just after '1'
```

Notera att den inbyggda variabeln `ans` sätts till värdet av det sista uttrycket på raden samt att den endast uppdateras om inget syntax- eller evalueringsfel uppstått. Observera också hur resultat skrivs ut!

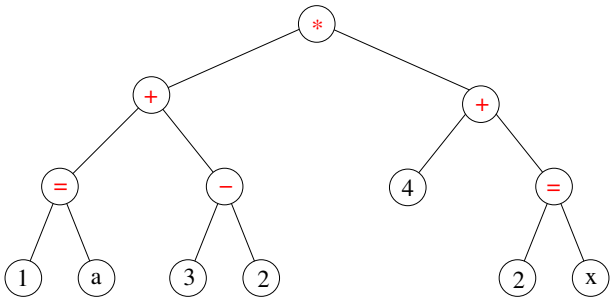
B2: Detta är en ganska omfattande uppgift som innehåller delar från båda den andra och den tredje kursmodulen. Det kan därför vara klokt att spara denna uppgift tills de andra uppgifterna är gjorda.

Uppgiften är strukturerad i deluppgifter varav de två första måste göras i ordning medan de påföljande kan lösas oberoende av varandra.

I den andra kursmodulen skrevs en kalkylator som, med hjälp av en tokenizer och parser, läste och beräknade aritmetiska uttryck. Ett annat sätt är att låta parsern bara konstruera en intern struktur av uttrycket som sedan kan evalueras. Det gör ju att man bland annat kan kontrollera hela uttryckets syntax innan några beräkningar och tilldelningar utförs. Det ger också bättre möjligheter till andra operationer som till exempel derivering och att definiera egna funktioner.

Denna uppgift går ut på att påbörja ett sådant program. Programmet ska hämta indata med tokenizern och bygga upp ett *binärt träd* som representerar uttrycket. De interna noderna står för operationer (`+`, `-`, `*`, `/`, `=`) och löven för konstanter och variabler.

Exempel: Uttrycket `((1=a) + 3 - 2)*(4 + (2=x))` ska representeras av trädet:



När trädet evalueras kommer det att resultera i värdet 12 samt att variabelerna `a` och `x` får värdena 1 respektive 2. Observera att parenteserna *inte* ska finnas med i trädet — det är strukturen som bestämmer evalueringsordning.

Filen `m2b.py` innehåller en begränsad del av den kalkylator som ingick MA2 samt skelett till de klasser som ska utgöra noder i trädet. Uppgiften går ut på att skriva klart dessa klasser samt att modifiera kalkylatorn så att den, i stället för att räkna, bygger upp ett träd som representerar uttrycket. När trädet är klart ska det *evalueras* och värdet skrivas ut. Det ska också gå att skriva ut själva uttrycket som trädet representerar.

De interna noderna ska representeras med hjälp av klassen `Operator` och löven ska representeras av klasserna `Variable` och `Constant`.

Klasserna ska innehålla metoderna `__init__`, `__str__` och `evaluate`.

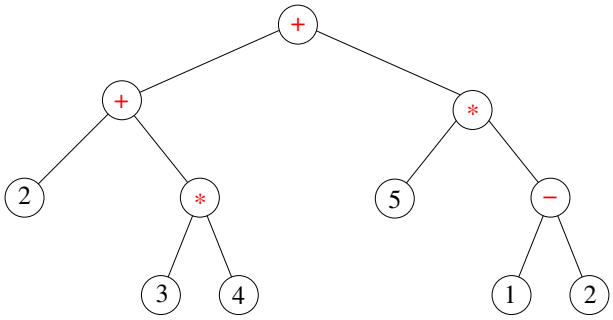
Klassen `Constant` är given och behöver inte ändras. Klasserna `Variable` och `Operator` är delvis givna men kommer att behöva kompletteras och/eller modifieras.

Klassen `Operator` ska representera binära aritmetiska operationer. Dessa objekt utgör de interna noderna i trädet och behöver således instansvariabler för att hålla reda på sin vänster- respektive höger-operand. De ska också innehålla ett *namn* i form av en sträng som identifierar operationen dvs någon av strängarna `'+'`, `'-'`, `'*'` eller `'/'`.

Deluppgift 1: Bygga ett uttryckträd med aritmetiska operatorer

I denna ska uttryck med konstanter och de binära, aritmetiska operatorerena dvs `+`, `-`, `*` och `/` kunna läsas in och lagras i en trädstruktur. Det handlar alltså om att modifiera parserfunktionerna så att de returnerar nod-referenser i stället för numeriska värden. Avvakta med att hantera tilldelningar. Parametern `variables` till parserfunktionerna kan tas bort — den fyller ingen funktion i parsningen.

Exempel: Uttrycket `2+3*4 + 5*(1-2)` ska resultera i trädet



Utskrift:

```
Input : 2 + 3*4 + 5*(1-2)
Parsed: 2+3*4+5*1-2
```

Observera avsaknaden av parenteser i utskriften. Detta åtgärdas i deluppgift 3.

Deluppgift 2: Evaluera uttrycken

För att evaluera (beräkna) trädets numeriska värde ska noderna innehålla metoden `evaluate(variables)`. Denna metod är given för klassen `Constant` men du måste skriva den för klassen `Operator`. Gemensamt för evaluering av de binära operatorerna är att de först ska evaluera sina båda operander och därefter utföra sin egen operation på de två evaluerade operanderna.

Operatorernas funktion ska hämtas ur ett lexikonet `OPERATORS` som till varje möjligt operatornamn lagrar ett lambda-uttryck som definierar operationen (jfr hantering av funktionsobjekt i kalkylatorn i MA2!)

Eftersom man för deluppgift 3, förutom lambda-uttrycket, behöver knyta ytterligare ett värde (en prioritet) till varje operator gör man klokt i att lagra *tupler* som värden i lexikonet med lambda-uttrycket som första element.

Evalueringen igångsätts genom att i `main`-funktionen anropa rotnodens evalueringsfunktion. Se till att `main`-metoden uppdateras med detta! Med samma exempel som ovan blir resultatet så här:

```
Input : 2+3*4+5*(1-2)
Parsed: 2+3*4+5*1-2
Result: 9
```

Deluppgift 3: Förse utskrifterna med parenteser

Modifiera `__str__` i `OPERATOR` så att den sätter ut parenteser vid behov. Behoven styrs av en *prioritet* som hör ihop med varje nod i trädet. I den givna koden är prioriteten för `Constant` och `Variable` satt till 100 vilket innebär att man aldrig behöver sätta ut parenteser kring konstanter och variabler. I lexikonet `OPERATORS` ger man som andra värde (efter lambda-uttrycket) en prioritet. Ge `+` och `-` låg (och samma) prioritet samt `*` och `/` en lite högre (och samma) prioritet. Genom att jämföra sin egen med barnens prioritet kan `__str__`-metoden i `Operator` avgöra om parenteser behöver sättas ut runt respektive barn. Använd prioriteterna i stället för att titta på operationerna — man ska inte behöva ändra på koden om man lägger till nya operatorer!

När det är är gjort ska vårt exempeluttryck ge följande resultat:

```
Input : 2+3*4+5*(1-2)
Parsed: 2+3*4+5*(1-2)
Result: 9
```

Du ska inte sätta ut parenteser om de inte behövs!

Deluppgift 4: Implementera tilldelningar

Tilldelningsoperatorn `=` är visserligen en binär operator men den är lite speciell eftersom den bara ska evaluera den ena operanden. Vi har därför valt att den ska implementeras med en egen klass med namnet `Assignment`.

Skriv den klassen och se till att den kan skapas av parsern. Dess evalueringsfunktion ska, förutom att evaluera den vänstra operanden, också lägga in värdet i lexikonet `variables`.

När det är klart ska det med vårt inledande exempel se ut så här:

```
Input : ((1=a)+(3-2))*(4+(2=x))
Parsed: ((1=a)+(3-2))*(4+(2=x))
Result: 12

Input : vars
   E     : 2.718281828459045
   PI    : 3.141592653589793
   a     : 1
   ans   : 12
   x     : 2
```

Uppgifter i anslutning till modul 3

I detta avsnitt ska du arbeta med filen `m3.py` som innehåller klasserna `LinkedList` och `BST`.

Klassen `LinkedList.py` innehåller kod för att hantera länkade listor av objekt. Listorna är inte sorterade.

Klassen `BST` innehåller kod för vanliga binära sökträd.

A5: Metoden `push(self, x, index=0)` i klassen `LinkedList` ska lägga in en ny nod med värdet `x` på angivet index. Som vanligt betyder index 0 den första platsen, index 1 den andra och så vidare. Den givna metoden är bara implementerad för default-fallet dvs den lägger in först när inget index är givet. Uppgiften går ut på att komplettera metoden så att den fungerar även när ett index ges som parameter. Det ska gå att lägga in ett nytt element efter sista elementet i listan men om ett större index än så anges ska ett `ValueError` skapas.

Exempel: Koden

```
    lst = LinkedList()
    testset = ((4, 0), (1, 0), (2, 1), (3, 2), (5, 4), (9, 0),
               (7, 7), (6, -1))
    for item in testset:
        try:
            lst.push(item[0], item[1])
            print(f'Pushed {item[0]} at index {item[1]}.' +
                  f' Resulting list: {lst}')
        except ValueError as ve:
            print(f'*** {ve}')
```

ska producera följande utskrift

```
Pushed 4 at index 0. Resulting list: (4)
Pushed 1 at index 0. Resulting list: (1, 4)
Pushed 2 at index 1. Resulting list: (1, 2, 4)
Pushed 3 at index 2. Resulting list: (1, 2, 3, 4)
Pushed 5 at index 4. Resulting list: (1, 2, 3, 4, 5)
Pushed 9 at index 0. Resulting list: (9, 1, 2, 3, 4, 5)
*** Index 7 out of range in push
*** Index -1 out of range in push
```

A6: Metoden `remove_largest(self)` i klassen `BST` ska ta bort noden med det största värdet ur trädet och returnera själva värdet.
Skriv klart den *rekursiva* hjälpfunktionen `_remove_largest(node)` så att den fungerar i den givna `remove_largest`.
Tips: Det behövs högst 5 rader kod.

Exempel: Koden

```
    inserts = (5, 1, 8, 2, 12, 3, 4, 6)
    print(f'Inserted values: {inserts}')
    bst = BST(inserts)
    print('Removed values:', end=' ')
    while not bst.is_empty():
        print(bst.remove_largest(), end =' ')
    print(f'\nResulting tree: {bst}')
    try:
        bst.remove_largest()
    except ValueError as ve:
        print(f'*** {ve}')
```

ska producera följande utskrift

```
Inserted values: (5, 1, 8, 2, 12, 3, 4, 6)
Removed values: 12 8 6 5 4 3 2 1
Resulting tree: <>
*** Empty tree in remove_largest
```

B3: Skriv metoden `max_level_sum(self)` i klassen `BST` som returnerar den maximala summan av nycklarna på samma nivå.

Exempel: Koden

```
    tree = BST((4, 1))
    print(f'Expects 4, got {tree.max_level_sum()}')
    tree = BST((4, 7))
    print(f'Expects 7, got {tree.max_level_sum()}')
    tree = BST((4, 7, 9))
    print(f'Expects 9, got {tree.max_level_sum()}')
    tree = BST((4, 7, 9, 3))
    print(f'Expects 10, got {tree.max_level_sum()}')
    inserts = (5, 8, 1, 3, 7, 2, 6, 9)
    print(f'\nInserted keys: {inserts}')
    tree = BST(inserts)
    print('Maximal level sum:', tree.max_level_sum())
    tree.insert(12)
    print('Maximal level sum:', tree.max_level_sum(),
          'after inserting 12')
    try:
        print('Empty tree', BST().max_level_sum())
    except ValueError as ve:
        print(f'*** {ve}')
```

ska producera följande utskrift

```
Expects 4, got 4
Expects 7, got 7
Expects 9, got 9
Expects 10, got 10

Inserted keys: (5, 8, 1, 3, 7, 2, 6, 9)
Maximal level sum: 19
Maximal level sum: 20 after inserting 12
*** Empty tree in max_level_sum
```

Uppgifter i anslutning till modul 4

I detta avsnitt ska du arbeta med filen `m4.py` (nerladdningsnamn `m4.txt` men namnge den i Pythoneditorn till `m4.py`). Du ska även ladda ner en fil kallad `m4_data.txt` och den ska kallas så (inte `m4_data.py`).

Ni får endast importera `functools.reduce` samt moduler/paket för parallellisering i denna sektion för modul 4 (finns även några som redan står i filen `m4.py`). Löser ni uppgifter med andra externa moduler/paket som ni importerar blir det automatiskt underkänt på den uppgiften.

A7: Du har en lista `v` av följande form:

- Listan har jämnt antal element.
- Listan består av x och y-koordinater för olika vektorer, t.ex.

  `v=[1,2.1,4,-2,1,3]`

  representerar tre vektorer

$$(x_1, y_1) = (1, 2.1),$$
$$(x_2, y_2) = (4, -2),$$
$$(x_3, y_3) = (1, 3).$$

Modifiera funktionen `lengths(v)` i `m4.py` så att den uppfyller följande:

- Skapa och returnera en lista där varje element är längden på varje vektor (x_k, y_k) i `v`. För exemplet ovan vill vi returnera

  `[2.3259406699226015, 4.47213595499958, 3.1622776601683795]`

  eftersom

$$\sqrt{x_1^2 + y_1^2} = \sqrt{1^2 + 2.1^2} \approx 2.3259406699226015,$$
$$\sqrt{x_2^2 + y_2^2} = \sqrt{4^2 + (-2)^2} \approx 4.47213595499958,$$
$$\sqrt{x_3^2 + y_3^2} = \sqrt{1^2 + 3^2} \approx 3.1622776601683795.$$

- Listan `v` kan vara godtyckligt lång, men har alltid jämnt antal element (du behöver inte testa detta).
- Använd minst två högre ordningens funktioner (higher order functions) som gåtts igenom i kursen, t.ex. `zip`, `map`, `reduce`, `list comprehension`, `filter`.
- Skriv funktionen på en rad.
- Använd inga andra moduler eller paket än `math` och `functools.reduce` (om du vill använda reduce). Du får inte använda `numpy`.

A8: I `m4.py` finns en funktion definierad, `simulation(n, scaling)`, som **inte** ska modifieras. Funktionen returerar en lista av längd `n`, med slumpvisa flyttal i intervallet $[0, \texttt{scaling}]$.

Modifiera funktionen `run_simulations(n_total, scalings)` i `m4.py` så att den:

- Utför ett antal simuleringar (anrop till `simulation`) parallellt. Importera vilket paket/modul ni vill för detta.
- `scalings` är en lista med olika "skalningar", längden på denna lista avgör hur många processer du ska köra parallellt. Variablen `n_total` är det totala antalet samplingar som ska användas för simuleringarna. Exempelvis, om man anropar

  ```
  >>> run_simulations(1000, [2.5,3,4])
  ```

  så ska vi parallellt göra dessa tre funktionsanrop

  ```
  simulation(334,2.5)
  simulation(333,3)
  simulation(333,4)
  ```

  Obeservera att summan av alla anrops `n` måste bli `n_total` (ovan 334+333+333=1000). Om du i exemplet ovan hade haft `n_total=1001` så ska du göra följande funktionsanrop

  ```
  simulation(334,2.5)
  simulation(334,3)
  simulation(333,4)
  ```

  D.v.s. sprid ut simuleringarna så jämnt som möjligt, och "fyll på" överskjutande värden ur `n_total` i ordning för argumenten i `scalings`.
- Funktionen ska returnera en lista med samma längd som `scalings`, där varje element är **summan** av de returnerade värdena för varje simulering som kördes parallellt.
- Under tentan, testa inte med för många processer (prova med max 4) eller stora `n_total` (det räcker med 100 eller 1000) för att undvika överbelastning.

B4: Antag att du arbetar för en annonsavdelning på ett företag, och du ska få fram information om varor som ni säljer; din chef vill t.ex. veta vilka ålderkategorier det är som handlar de olika varorna för att kunna rikta annonser.

Du får en fil för varje produkt (ni har för en produkt, kallad `m4_data.txt`) där första raden består av modellnamn för varan, standardförtjänst i SEK (det finns ungdoms och pensionsrabatter, se nedan), och antalet köp registrerade i filen. Varje rad i resten av filen innehåller sedan ett tal 10-99 som representerar åldern för en person som köpt varan. Exempelvis ser början på filen `m4_data.txt` ut så här

```
XPZ112 211.90 1000
83
76
73
90
...
```

Modellnamn på varan är `XPZ112`, standardförtjänsten 211.90 SEK och det finns 1000 registrerade köp i filen (ålder 83, 76, 73, 90, ...). Filens totala längd är därför 1001 rader.

Du ska samla ihop datat i ålderskategorier av storlek 10, för alla åldrar (d.v.s. 10-19, 20-29, ..., 90-99); ingen under 10 eller över 99 år har handlat varan.

Du ska modifiera funktionen `get_statistics(filenames,n_processes)` i `m4.py` så att den kan göra följande (läs igenom allt innan du börjar koda).

- Argumentet `filenames` är en lista med filnamn för olika produkter; t.ex.

  ```
  ["m4_data1.txt","m4_data2.txt"]
  ```

- Funktionen ska returnera en lista med följande information för varje vara (varje varuinformation är i sig en lista med sex element)

  1. Modellnamnet på varan.
  2. Standardförtjänst på varan.
  3. En lista med antalet personer i varje ålderskategori (lista med 9 element).
  4. En lista av listor, där varje inre lista hör till en ålderskategori och innehåller alla åldrar för personer i den ålderskategorin; dessa åldrar ska vara sorterade från lägst till högst, och om det finns mer än en person med en viss ålder så ska alla personer vara med i denna lista. (lista med 9 element).
  5. Medelåldern för alla de som köpt varan.
  6. Förtjänsten för varje ålderskategori i SEK (avrunda till närmaste heltal). De som är 20 år och yngre samt 65 och äldre har 10% rabatt.

- Koden ska vara parallelliserad på två nivåer:

  – Varje vara ska hanteras parallellt av vardera `n_processes` processer; d.v.s. har du två varor så ska hela programmet använda `2*n_processes` processer.

  - För varje enskild vara ska du kolla hur stor datafilen är (på rad 1) och sedan distribuera vilka radnummer som ska hanteras i varje process. Du ska alltså inte läsa in all data från filen och distribuera detta, utan läsa filen i varje process och endast hantera rätt information lokalt.

    När varje process är klar måste datat kombineras och statistiken som definierats ovan returneras.

- Skriv även en funktion `print_statistics(filenames, n_processes)` som presenterar informationen från `get_statistics(filenames,n_processes)` på ett överskådligt vis i terminalen (d.v.s printa inte variablen 4 i listan ovan).

  Exempel (ni behöver inte göra exakt så här):

```
>>> print_statistics(["m4_data.txt"],2)
----------
Model: XPZ112
Standard revenue per sale: 211.9
Average age: 60.547
Total revenue: 201368.57
Bins with number of people and revenues:
[10,19]:    26     4958
[20,29]:    56    11739
[30,39]:    80    16952
[40,49]:   120    25428
[50,59]:   169    35811
[60,69]:   165    33247
[70,79]:   161    30704
[80,89]:   132    25174
[90,99]:    91    17355
----------
```

- Använd så mycket högre ordningens funktioner du kan.
- Skriv hur många hjälpfunktioner du vill.
- Importera inga externa moduler eller paket, förutom de som behövs för parallelliseringen (och functools.reduce om du vill använda det).