Pythonkurs, Övningssession Funktioner

10 mars 2020

1 Upprepning

1.1 Inledning

Funktioner har en väldigt viktig uppgift när det kommer till att undvika upprepning av samma kod. Detta gäller främst när upprepningen sker på olika platser i koden. Är upprepningarna efter varandra är det ofta bättre med en loop. Den här uppgiften syftar till att illustrera fördelarna med ett sådant sätt.

1.2 Uppgifter

1. Skapa ett program utan funktioner som tar ett heltal som input och sedan skriver ut följande:

   - Om talet är större än 5: En sträng bestående av talet som är lika lång som talet. Ex:

   ```
   Ange ett tal: 7
   7777777
   ```

   - Om talet är 5 eller mindre: En rad med varje sådan sträng som ovan upp till och med talet. Till exempel:

   ```
   Ange ett tal: 4
   1
   22
   333
   4444
   ```

2. Skapa en ny version som använder sig av funktioner.

1.3 Utförande

1.3.1 Utan funktioner

1. Skriv ett program som frågar efter ett tal

2. Test om numret är större eller mindre än fem
3. Om det är fem eller mindre, loopa över alla tal upp till talet
4. Konvertera det aktuella talet i loopen till en sträng.
5. Multiplicera strängen med talet
6. Skriv ut strängen
7. Om det är större än fem, hoppa över loopen
8. Konvertera det inmatade talet till en sträng
9. Multiplicera strängen med talet
10. Skriv ut strängen
11. Testkör programmet

1.3.2 Med funktioner

1. Öppna en ny fil
2. Definiera en funktion som tar ett tal som argument och gör följande:
   (a) Konverterar talet till en sträng
   (b) Multiplicerar strängen med talet
   (c) Skriver ut strängen
3. Kopiera förra uppgiften, men ersätt de delar som går med ett anrop till funktionen du just definierade
4. Testkör och se så att resultatet blir samma som i förra uppgiften.

1.4 Kommentarer

I det här exemplet gör villkoret att det inte bara går med en loop, åtminstone inte utan besvär. Det är visserligen bara två platser, så man kan lätt frestas att ta till copy-paste. Men gör man det så behöver man fundera ett varv till, för det orsakar många fel.

2 Underhållbarhet

2.1 Inledning

En ganska vanlig programmeringsuppgift är att utföra ändringar i existerande kod. Om funktioner har använts på ett bra sätt, så behöver man ofta bara göra ändringen på en plats. Använder man inte funktioner så får man istället flera ställen att ändra på, vilket skapar en risk att man missar någon ändring. Programmet riskerar då att få ett inkonsekvent beteende, vilket kan vara mycket besvärligt att upptäcka.

2.2 Uppgifter

1. Ändra båda programmen från uppgift 1 i den här sessionen så att den skriver ut "svar: " framför varje sträng.

2.3 Utförande

Gå igenom koden och ändra i alla print.

2.4 Kommentarer

Här är en fingervisning om varför man inte ska köra copy-paste. Vad hade hänt om du hade missat att ändra på en av platserna? Här är det en rätt kort och överskådlig kod, men när koden växer så blir det ganska snabbt ohanterligt.

3 Flödesscheman

3.1 Inledning

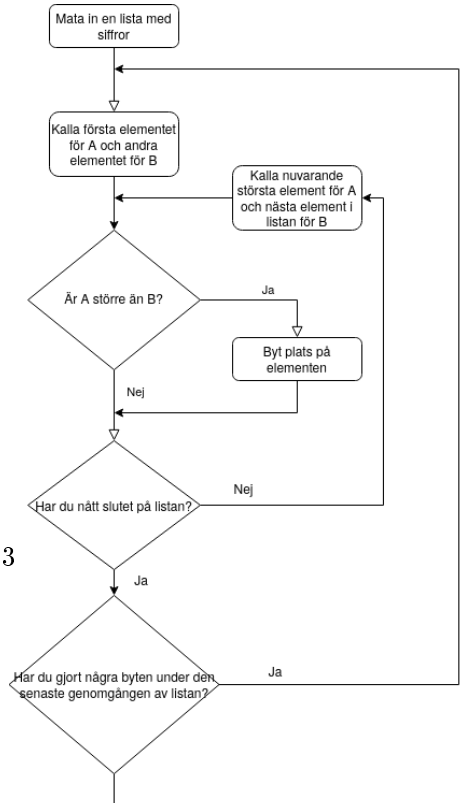
Flödesscheman är ett vanligt sätt att illustrera förlopp. Ibland är det något man som utvecklare själv väljer, ibland är det något man får som en typ ac specifikation. Hur som helst är det användbart att förstå sig på dem. I den här övningen kommer du att implementera en sorteringsalgoritm som kallas bubbelsortering (bubble sort). I de flesta fall finns det bättre sorteringsalgoritmer att använda, men den har ändå sina tillämplningar, och fördelen att den kräver relativt lite och relativt lättbegriplig kod.

3.2 Uppgifter

Skapa ett program som gör en bubbelsortering enligt Figur 1.

3.3 Utförande

1. Skapa en ny fil
2. Definiera funktioner som gör varje del av flödesschemat
3. Skriv grundscriptet som kör funktionerna i den ordning du vill.
4. Testkör programmet med lite olika listor



3.4 Kommentarer

Tidigare programmering följde ofta den här typen av mönster, med en lista som körs uppifrån och ned, förutom där den hoppar i koden. En sådan kod, med hopp till olika platser, kallas för spaghettikod och blir väldigt svårläst väldigt fort. Ett tydligare sätt är då att använda funktioner, men också loopar. Ett alternativt sätt att utföra uppgiften, istället för att definiera funktioner, skulle vara att börja med de yttersta looparna i flödesschemat och därefter arbeta sig inåt.

4 Pseudokod

4.1 Inledning

Ett annat sätt att illustrera en programprocess utan att faktiskt programmera den är att skriva pseudokod. Det innebär kod som inte kan läsas av datorn, men som för en människa förklarar vad programmet gör. I likhet med flödesscheman så kan dessa användas både som verktyg av utvecklare, och som specifikation av produktägare. Det senare är dock tämligen ovanligt, eftersom psuedokod dels kräver mer tekniskt kunnande, dels ofta innebär en detaljstyrning som inte är önskvärd. För att illustrera pseudokod så använder vi oss av en annan typ av sorteringsalgorithm. Denna gång quicksort. Quicksort är en väl använd och väldigt effektiv gång algorithm. Den som kommer att användas är den enklast möjliga variantern och det finns en rad optimeringar som går att göra.

4.2 Uppgifter

Skapa ett program enligt följande pseudokod:

```
Funktionen sort som tar emot en list (unsorted_list):
        p = Ett element i unsorted_list
        for element in unsorted_list (förutom p):
                if element<=p:
                        lägg element i en list (smaller)
                else:
```

```
                    lägg element i en list (larger)

    if smaller innehåller mer än ett tal
            sort(smaller)

    if larger innehåller mer än ett tal
            sort(larger)

    sorted_list = list som innehåller: smaller -> p -> larger
    return sorted_list
```

4.3 Utförande

1. Läs igenom pseudokoden och försök förstå vad den gör
2. Översätt de rader som enkelt låter sig översättas till pythonkod. Tänk på att en del rader som består av ren pythonkod fortfarande kan behöva ändras.
3. Fundera en stund över hur du bäst angriper de rader som inte lätt låt sig översättas. Tänk på att man ibland kan behöva titta på flera rader pseudokod tillsammans. Ibland kan det också vara bra med att göra funktioner. Eller skriva om pseudokoden i ny, mer detaljerad pseudokod.
4. Testkör och se så att funktionen fungerar.

4.4 Kommentarer

Pseudokod kan skrivas på en hel del olika sätt. Ibland är den nästa färdig kod, ibland mer som en berättelse. Det gäller att inte stirra sig alltför blind på hur varje rad ska översättas, utan snarare försöka förstå den övergripande bilden. En intressant sak i den här övningen är att funktionen anropar sig själv. Detta kallas rekursion och är fullt tillåtet i python. Gör man det för många gånger så kostar det mycket datorresurser, och tar lång tid. Python har en inbyggd begränsning. Däremot har inte python, som en del andra språk, en funktion för att optimera bort onödiga rekursioner.

5 Läsbarhet och förberedelser

5.1 Inledning

Man läser sin kod mycket fler gånger än man skriver den. Ganska ofta läser man också andras kod. Det gäller alltså att försöka skriva sin kod så lättläst som möjligt. Funktioner som gör en sak och heter något som talar om för läsaren vad de gör tar ofta bort behovet av att gå in och läsa detaljer. En bra hjälp till att definiera sina funktioner är att göra ett strukturerat förarbete. I den här

uppgiften kommer vi att använda både ett flödesschema och en pseudokod. I verkliga uppgifter kan man använda båda, välja en eller använda någon annan metod, allt efter vad som passar. Att direkt börja knacka in kod brukar dock sällan vara ett effektivt arbetssätt.

5.2 Uppgifter

1. Gör ett flödesschema för ett program som tar en lista och ur den väljer ut alla udda tal över ett av användaren givet värde. Om användaren anger ett icke-nummer så ska hen behöva göra om.
2. Skriv en pseudokod för programmet
3. Skriv programmet utan funktioner
4. Skriv programmer med funktioner

5.3 Utförande

1. Skapa en ny fil
2. Gör en lista med slumpmässiga heltal (Förslagsvis 20 tal mellan 0 och hundra)
3. Låt användaren ange ett heltal
4. Lägg till en if-sats med strängmetoden .isdecimal() för att säkerställa att det är en siffra och inget annat som användaren matat in.
5. Skriv kod, utan funktioner, som skriver ut en lista på alla udda tal under användarens angivna värde
6. Öppna en ny fil
7. Skriv, i den nya filen, en funktion för att generera en slumpmässig list. Funktionen ska ta ett argument för hur många tal du ska ha, och en för hur högt de ska gå.
8. Skriv en funktion som frågar efter input och repeterar tills det är en siffra för att sedan returnera en int.
9. Skriv en till funktion som tar en lista och ett tal som argument, och returnerar en lista på tal över det angivna talet
10. Skriv ytterliggare en funktion som tar en lista med tal som argument och returnerar en lista med uddatal
11. Skriv ett program som använder funktionerna för att göra samma sak som i den förra filen.
12. Läs igenom filerna och fundera över vilken du skulle förstå lättast om du inte visste på förhand vad de gjorde.

5.4 Kommentarer

I det här exemplet används bara funktionerna en enda gång, och en sådan situation är inte ovanlig. Anledningen till att man använder funktioner är att det blir betydligt mer lättläst, åtminstone om man har funktionsnamn som hintar om ungefär vad funktionen gör. En användbar princip är att varje funktion ska göra en sak, och göra den saken bra. Det är i enlighet med den principen som vi har valt att göra en funktion som delar listan vid ett tal, och en annan som plockar ut jämna tal. Vi hade förstått kunnat göra en funktion som gör båda sakerna, men då hade vi dels inte vunnit någon större läslighet, dels hamnat i problem om vi senare vill återanvända kod. Det kan ju hända att man vill dela en lista utan att ta ut de jämna talen, eller vice versa.

I exemplet här skrev vi funktionerna först och satte sedan ihop dem i huvudprogrammet, men det är faktiskt relativt vanligt att göra tvärtom, det vill säga först skriver man in funktioner som inte finns, sedan definierar man dem. Så länge definitionen kommer högre upp än användandet så spelar det ingen roll.

6 Lambda-funktioner

6.1 Inledning

Lambda-funktioner är små korta funktioner som har fördelen att de är lätta att hantera ungefär som variabler. Bland annat kan de returneras av andra funktioner samt läggas i diverse komplexa datatyper

6.2 Uppgifter

1. Skapa ett program som frågar användaren efter två tal och ett av de fyra räknesätten (+, -, *, /) och sedan returnerar resultatet som blir om man tillämpar räknesättet på dessa två.
2. Efter att du testkört programmet, bygg ut det så att det även kan hantera modulus-operatorn (%)

6.3 Utförande

1. Skriv en funktion som frågar efter inputen och sedan returnerar den.
2. Skriv en funktion som tar räknesättet som input och returnerar en lambda-funktion som använder det räknesättet (Tips: lambda-funktioner kan vara värden i en dict)
3. Skriv ett program som kör ovanstående funktioner och sedan skriver ut resultatet
4. Testkör programmet
5. Lägg till modulus i räknesätts-funktionen

6.4 Kommentarer

Här kommer verkligen lambda-funktioner till nytta! Alternativet hade varit en rad if-satser, som tveklöst blivit både mer svårläst och svårare att bygga ut. Observera hur du enbart behöver ändra i räknesättsfunktionen för att lägga till ett räknesätt! Det här är en del av vad som brukar kallas "separation of concerns", det vill säga att varje del av programmet bara ska bry sig om sin egen del. Det här är en stor anledning, att det går att ändra lätt!