Instruktioner - Datortentamen TDDD73 Funktionell och imperativ programmering i Python

Hjälpmedel

Följande hjälpmedel är tillåtna:

- Exakt en valfri bok, t.ex. den rekommenderade kursboken. Boken får ha anteckningar, men inga lösa lappar.
- Exakt ett A4-papper med egna anteckningar om precis vad som helst. Det kan vara hand- eller maskinskrivet, på ena eller båda sidorna.
- Pennor, radergummin och liknande för att kunna skissa lösningar på papper.

Icke tillåtna hjälpmedel inkluderar bl.a. alla former av elektronisk utrustning samt böcker och anteckningar utöver det som listats ovan.

Lösa tomma papper för att göra anteckningar tillhandahålls av tentamensvakterna.

Genomförande

Under datortentan kommer du att arbeta i en begränsad och övervakad miljö. Med hjälp av en särskild tentaklient skickar du in dina svar. Du kan även använda tentaklienten för att ställa frågor till examinator. Besök i skrivsalen kommer endast ske vid allvarligare tekniska problem.

Varje uppgift kan skickas in en enda gång och kommer därefter poängsättas. Det finns alltså inga möjligheter till komplettering.

Betygsättning

Datortentan består av totalt sex uppgifter som vardera kan ge maximalt fem poäng. Uppgifterna är utvalda för att ha olika svårighetsgrad och är ordnade så att den lättaste oftast kommer först. Observera att vad som är lätt eller svårt naturligtvis skiljer sig från person till person, så det är alltid en bra idé att snabbt läsa igenom alla uppgifterna för att bättre kunna prioritera arbetet. Betygssättningen sker enligt följande tabell:

- För betyg 3 krävs minst 12 poäng.
- För betyg 4 krävs minst 19 poäng.
- För betyg 5 krävs minst 25 poäng.

Rättningskriterier

Följande allmänna kriterier används vid poängsättning av uppgifter. Lösningar som bryter mot dessa kommer att få poängavdrag.

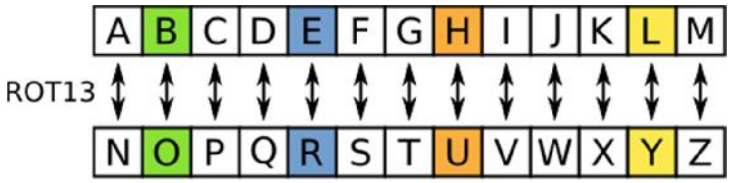
- Funktioner ska ha exakt samma namn som i uppgiften. Detta underlättar rättningen.
- Namn på parametrar, variabler och hjälpfunktioner som inte specificerats explicit i uppgiften ska vara beskrivande och följa namnstandarden.
- Lösningen ska vara välstrukturerad och lämpligt dokumenterad i enlighet med uppgiftsspecifikationen.
- Lösningen ska följa de regler som uppgiften har satt upp. Det kan t.ex. röra sig om att vissa funktioner eller metoder ej får användas, eller att lösningen ska göras enligt en särskild modell.
- Lösningen ska fungera exakt som i körexemplen i uppgiften, om inte texten indikerar något annat.
- Lösningen ska vara generell, d.v.s. den ska inte enbart fungera för de körexempel som finns i uppgiften, utan även för alla andra möjliga indata på samma form. Felkontroller av indata behöver dock normalt inte göras, om inte uppgiften särskilt uttrycker det.

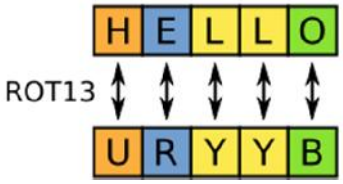
Uppgifter - Datortentamen TDDD73 Funktionell och imperativ programmering i Python

Tisdag 12 januari 2016 kl 14-19

Uppgift 1

När man kommunicerar digitalt är det viktigt med kryptering. I den här uppgiften ska du implementera en mycket enkel krypteringsfunktion i form av ett förskjutningschiffer. Förskjutningschiffer är en enkel typ av chiffer som åstadkoms genom att man placerar två rader med det önskade alfabetet över varandra. Genom att förskjuta den nedre raden med, till exempel, tretton steg ersätts A med N, B med O osv. I detta fall ersätts texten HELLO med URYYB. För att dekryptera texten gör man samma operation i motsatt riktning. Den vanligaste varianten för engelska är ROT-13, där man förskjuter alfabetet 13 steg, se bilden nedan.





Din uppgift är att implementera en funktion `encode(str)`. Du kan utgå från att strängen endast innehåller små bokstäver från det engelska alfabetet. Funktionen returnerar den krypterade versionen av `str`. Om `str` har en udda längd ska du kryptera den med ROT-13, förskjutning 13 steg framåt, annars ska du kryptera den med ROT-N, där man förskjuter alfabetet N steg framåt, där N är längden på strängen delat med 2. Du ska även implementera en funktion `decode(str)` som tar en sträng krypterad enligt ovan och returnerar ursprungssträngen.

```
>>> encode("hello")
"uryyb"
>>> encode("hell")
"jgnn"
>>> decode("uryyb")
"hello"
>>> encode("adam")
"cfco"
```

Observera att du kan skicka in svar på uppgiften endast en gång. De två funktionerna, `encode` och `decode`, ger 3 respektive 2 poäng, och det är möjligt att få poäng på dem oberoende av varandra.

Uppgift 2

En komprimeringsmetod som är vanlig då man har data där samma tecken ofta uppträder flera gånger i följd, är skurlängdskodning (en. *Run Length Encoding*, *RLE*). En sekvens med upprepningar av samma tecken brukar kallas **skurlängd** (en. *run-length*). I skurlängdskodning utförs komprimeringen genom att finna skurlängder av tecken och ersätta dessa med **koder** i kortare längder för att uppnå komprimering. En vanlig kod är tecknet det handlar om och antalet gånger tecknet upprepas.

Anta att vi vill skurlängdskoda följande data: `1 2 2 2 2 5 5 5 5 5 5 1 1 1`
Kodningen ger då utmatning: `1 1 2 4 5 6 1 3`. Detta ska tolkas som en 1:a, fyra 2:or, sex 5:or och tre 1:or.

Skriv två varianter av en funktion som tar en lista och returnerar motsvarande skurlängdskodade lista. Funktionen ska finnas i två varianter: en som arbetar enligt en rekursiv modell (kallad `rle_r`) och en som arbetar enligt en iterativ modell (kallad `rle_i`). Båda funktionerna ska behandla ett element i taget, vilket innebär att du inte får använda inbyggda funktioner eller metoder som behandlar hela listor. Du får inte heller använda listbyggare (eng. *list comprehensions*). Lösningarna ska vara funktionella i bemärkelsen att de inte får modifiera indata.

Funktionerna ska fungera exakt likadant, bortsett från att de ska ha olika lösningsmodeller. Här är några exempel:

```
>>> rle_r([1, 2, 2, 2, 2, 5, 5, 5, 5, 5, 5, 1, 1, 1])
[1, 1, 2, 4, 5, 6, 1, 3]

>>> rle_i(["a", "a", "b", "a", "c", "c"])
["a", 2, "b", 1, "a", 1, "c", 2]
```

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

Uppgift 3

MapReduce är ett ramverk för att processa stora datamängder genom att utnyttja parallellism och massiva datacenter. Ramverket består av två funktioner map och reduce. `map(fnm, seq)` applicerar en funktion `fnm` på varje element i sekvensen `seq` och returnerar resultaten i en ny lista. `reduce(fnr, seq)` reducerar listan `seq` till ett svar genom succesiva anrop till `fnr`. För att räkna ut summan av kvadraterna på alla tal i sekvensen `[1, 2, 3]` kan vi anropa `reduce(lambda x, y: x+y, map(lambda x: x*x, [1, 2, 3]))` vilket ger `14` som svar. En förutsättning är att `map`, `reduce`, `fnm` och `fnr` är strikt funktionella, dvs inte har några sidoeffekter eller modifierar indata.

Deluppgift 3a (2p)

Implementera `map(fnm, seq)` som tar en funktion `fnm` och en lista `[s1, s2, …, sn]` och returnerar en ny lista `[fnm(s1), fnm(s2), …, fnm(sn)]`. Du får inte använda inbyggda funktioner.

```
>>> map(lambda x: x*x, [1, 2, 3])
[1, 4, 9]
```

Deluppgift 3b (2p)

Implementera `reduce(fnr, seq)` som tar en binär funktion `fnr` och en lista `[s1, s2, …, sn]` och returnerar värdet som fås av `fnr(s1, fnr(s2, fnr(s3, ... fnr(sn-1,sn))))`. Du får anta att `seq` innehåller minst 2 element. Du får inte använda inbyggda funktioner.

```
>>> reduce(lambda x, y: x+y, [1, 4, 9])
14
```

Deluppgift 3c (1p)

Implementera en funktion `odd_cubes(n)` som multiplicera kuberna av varje positivt udda tal mindre än eller lika med n genom att använda `map` och `reduce`.

```
>>> odd_cubes(5)
3375
```

Uppgift 4

Levenshteinavståndet är inom datavetenskapen ett mått på hur stor skillnaden är mellan två strängar av symboler. Avståndet beräknas som summan av det antal raderingar, infogningar och substitueringar av tecken som krävs för att transformera den ena strängen till den andra. Det har fått sitt namn efter Vladimir Levensjtejn, som beskrev avståndet 1965. De tre operationerna fungerar så här:

1. Infoga en symbol. Om en sträng s kan delas upp i två strängar u och v, så kan man infoga en symbol x mellan u och v vilket resulterar i uxv. Exempel: Infoga d efter ab i abc ger abdc.
2. Radera en symbol. Om en sträng s kan skrivas som uxv där u och v är strängar och x en symbol, så kan man ta bort symbolen x vilket resulterar i uv. Exempel: Radera c i abc ger ab.
3. Substituera en symbol. Om en sträng s kan skrivas som uxv där u och v är strängar och x en symbol, så kan man byta ut x mot en annan symbol y, vilket resulterar i uyv. Exempel: Substituera det första a:t mot q i aba ger qba.

Din uppgift är att skriva en funktion `edit_distance(s1, s2)` som beräkna det minsta Levenshteinavståndet, även kallat *edit distance*, mellan två strängar `s1` och `s2`. Funktionen behöver inte vara minnes- eller tidseffektiv. Om du önskar får du anta att strängarna aldrig är längre än 100 symboler.

```
>>> edit_distance("abc", "")
3

>>> edit_distance("abc", "ac")
1

>>> edit_distance("lovxxexxx", "xxxxxlove")
7
```

Uppgift 5

Formella språk och logik är två viktiga områden inom datavetenskap. I den här uppgiften ska du implementera en förenklad parser för första ordningens predikatlogik. Grammatiken för språket är:

```
FOPL := "forall" VARIABLE FOPL | "exists" VARIABLE FOPL
     |  PREDICATE (LOGICAL_OP PREDICATE)*
LOGICAL_OP := "and" | "or" | "->"
PREDICATE := PREDICATE_IDENT "(" TERM ("," TERM)* ")" | "not" PREDICATE
TERM := VARIABLE | CONSTANT | FUNCTION_IDENT "(" TERM ("," TERM)* ")"
VARIABLE := "x" | "y" | "z"
CONSTANT := "a" | "b" | "c"
PREDICATE_IDENT := "P" | "Q" | "R"
FUNCTION_IDENT := "f" | "g" | "h"
```

Parsern ska anropas genom funktionen `parse(str)` som tar en sträng och returnerar ett parseträd, uppbyggt av listor i flera nivåer enligt exemplen nedan, där varje instans av reglerna `FOPL`, `PREDICATE` och `TERM` ger en egen nivå. Du får göra antaganden som att det bara finns ett blanksteg mellan två ord. Två exempel:

```
>>> parse("forall x P(x) and Q(x)")
["forall", "x", [["P", "x"], "and", [["Q", "x"]]]]

>>> parse("exists y P(f(x)) -> R(a)")
["exists", "y", [["P", ["f", "x"]], "->", [["R", "a"]]]]
```

Det är ytterst viktigt att din lösning är välstrukturerad och väldokumenterad samt använder lämpliga abstraktioner. Det ska till exempel vara enkelt att ändra vilka identifierare, dvs variabler, konstanter, predikat och funktioner, som finns.

Uppgift 6

Generalisera `map` och `reduce` från uppgift 3 till att hantera godtyckliga listor av listor. Map ska applicera en funktion på varje element i varje sekvens. Reduce ska reducera respektive underlista till ett värde så att listorna i en lista med listor ersätts av de reducerade värdena, sedan kan listan i sin tur reduceras till ett värde. Till exempel, givet listan `[1, [2, 3], 4]` så kommer först underlistan `[2, 3]` reduceras till ett värde *x* och sedan kommer listan `[1, x, 4]` reduceras.

Skriv en funktion `mapreduce(fnm, fnr, seq)` som applicera funktionen `fnm` på varje element i varje sekvens i `seq` och reducerar alla sekvenserna med `fnr` till ett enda värde som returneras. En genomtänkt och bra struktur på lösningen samt lämplig dokumentation krävs. Du får inte använda inbyggda funktioner. Listbyggare är tillåtna.

```
>>> mapreduce(lambda x: x*x, lambda x, y: x+y, [[1], 2, [3, [4]]])
30
```