TDDE44. DAT1. 2hp.

2024-08-22 kl. 14.00-18.00

Tillåtna hjälpmedel: penna, papper, linjal, suddgummi, godkänd(a) bok/böcker (lärare kontrollerar böcker innan tentan)

Givna filer

Du hittar eventuella filer du kan behöva till uppgifterna i katalogen `~/Desktop/given_files`. Denna katalog är **skrivskyddad**, liksom filerna i den. Behöver du ändra i någon av de givna filerna måste du först kopiera filen till en annan katalog.

Skriva kod

Det rekommenderas att du sparar dina lösningar på skrivbordet. Sökvägen till skrivbordet är `~/Desktop`. Spara din kod för varje uppgift inklusive deluppgifter i *en* fil. Du kan *inte* spara din fil i `~/Desktop/given_files`. Exempel:

- Svaren till uppgift 1.1 (1.1.1, 1.1.2) sparas i filen (`uppg11.py`).
- Svaret till uppgift 1.2 sparas i filen (`uppg12.py`).

Inga doc-strängar behöver skrivas. Inga avdrag kommer ges om koden bryter mot `PEP8` eller `PEP257`.

Begränsningar av hur en uppgift får lösas

I de fall som en uppgift ska lösas på ett speciellt sätt står detta angivet i uppgiften; t.ex. "*Lös uppgiften rekursivt.*" eller "*Lös uppgiften med hjälp av en `for`-loop*".

Om specifika funktioner inte får användas i lösningen står detta också angivet i uppgiften, t.ex. "*Funktionen `max()` får inte användas.*"

Tillåtna moduler att använda på tentan: Följande Python-moduler är *tillåtna* att användas.

- `copy`
- `csv`
- `math`
- `random`
- `sys`

Övriga moduler är otillåtna om inget annat anges.

Notation och termer i uppgifter

När strängar visas i uppgifterna kommer de alltid att vara omgivna av citattecken. Om det står att strängen `"hej"` ska skrivas ut, ska utskriften motsvara `print("hej")`.

- termen *heltal* syftar alltid på datatypen `int` om inget annat anges
- termen *sträng* syftar alltid på datatypen `str` om inget annat anges

Exempel i uppgifter

Tänk på att eventuella exempel till uppgifterna *inte täcker alla möjliga fall*. Ibland kan t.ex. divisioner resultera i svar med precisionsfel. Om du får `1.99999999` som svar vid divisionen `4/2.0` behöver du inte oroa dig.

Kom ihåg att testköra din kod

Inkompletta och delvis felaktiga lösningar *kan* ge delpoäng men kom ihåg att testköra din kod innan du skickar in! **Filer som kraschar vid körning ger normalt sett underkänt/0p på hela uppgiften.** Alltså, om du vill skicka in inkompletta lösningar som kraschar när du testkör så **kommentera ut eventuella anrop till den kraschande koden innan du skickar in**.

Rättning/bedömning av Del 1

Del 1 består av fem uppgifter. För att få godkänt på Del 1 måste **alla uppgifter** (1.1-1.5) vara godkända.

- För att få **godkänt** på en uppgift (t.ex. 1.1) måste samtliga deluppgifter vara godkända.
- Ej godkända uppgifter kan kompletteras i mån av tid.
- *Upprepade inlämningar där du uppenbart gissat dig fram kan komma att underkännas och kan då inte längre kompletteras.* Om du inte förstår feedbacken så be om ett förtydligande istället för att gissa.

Rättning/bedömning av Del 2

Del 2 rättas endast om Del 1 är godkänd. Om du inte fått godkänt på Del 1 men vill få kommentarer på Del 2 kan du kontakta examinator efter att du fått ditt resultat.

Del 2 består av tre frågor som är något mer omfattande och som kräver mer problemlösning än uppgifterna i Del 1.

Du kan få 0-5 poäng per uppgift i Del 2.

Betyg på tentan

- Betyg 3: Godkänd på Del 1
- Betyg 4: Godkänd på Del 1 + 5 poäng på Del 2.
- Betyg 5: Godkänd på Del 1 + 10 poäng på Del 2.

Del 1

För att bli godkänd på Del 1 måste alla uppgifter vara godkända.

Uppgift 1.1

- För att få godkänt på denna uppgift skall **båda** deluppgifterna lösas.
- En av uppgifterna skall lösas med en `while`-loop och en skall lösas med `for`-loop.
- Comprehensions och generatorer får *inte* användas för att lösa uppgifterna.
- Spara lösningarna till båda deluppgifterna i Uppgift 1.1 i en fil med namnet `uppg11.py`.
- Filen skickas in via Studentklienten som '*Assignment #1*'.

Uppgift 1.1.1 - lös med lämplig loop

Multiplikation av heltal kan ses som upprepad addition av ett tal till sig självt, dvs

$$n \times x = \underbrace{x + ... + x}_{n \text{ gånger}}$$

Exempelvis $3 \times 2 = 2 + 2 + 2$. I Python används normalt operatorn `*` för multiplikation.

Skriv funktionen `mult_as_rep_add(factor1, factor2)` som utför multiplikation av *heltal* som upprepad addition. Det är *inte* tillåtet att använda multiplikationsoperatorn `*` annat än för att testa att man får rätt resultat.

Exempel

```
mult_as_rep_add(2, 0)
0
mult_as_rep_add(2, 1)
2
mult_as_rep_add(5, 3)
15
mult_as_rep_add(10, 10)
100
```

Uppgift 1.1.2 - lös med lämplig loop

Skriv funktionen `years_to_cutoff(start_val, rate, cutoff)` vars argument består av tre positiva tal. `start_val` är ett startvärde, `rate` är årlig tillväxt (rate = 0.1 innebär årlig tillväxt på 10%) och `cutoff` definierar tröskelvärdet. Funktionen ska returnera hur många år det tar för värdet att överstiga tröskelvärdet. Uppgiften skall lösas med en lämplig loop.

Exempel

```
>>> years_to_cutoff(100, 0.1, 115)
2
>>> years_to_cutoff(100, 0.1, 200)
8
>>> years_to_cutoff(120, 0.1, 200)
6
>>> years_to_cutoff(100, 0.05, 200)
15
>>> years_to_cutoff(120, 0.05, 200)
11
```

Uppgift 1.2 - lös med rekursion

- Spara lösningen till Uppgift 1.2 i en fil med namnet `uppg12.py`.
- Filen skickas in via Studentklienten som '*Assignment #2*'.

Vi vill beräkna värdet av ett univariat polynom i en viss punkt `x`. Skriv en funktion `polyval` som tar in en lista av tal, `coeffs`, och ett tal, `x`, och returnerar

$$a_0x^0 + \ldots + a_{n-1}x^{n-1} + a_nx^n$$

där $a_0, ..., a_n$ är elementen i koefficientlistan.

Uppgiften skall lösas med rekursion och du får dela upp din lösning i flera funktioner vid behov.

Tips: Vill man bara skriva en funktion kan det vara lämpligt med en tredje parameter till `polyval`, t.ex. kallad `n`, med default-värdet `0`.

Exempel

```
>>> polyval([1,5,7], 3) # 1*3^0 + 5*3^1 + 7*3^2
79
>>> polyval([], 3) # Så här hanteras det tomma polynomet.
0
>>> polyval([7, 9], 92)
835
>>> polyval([5, 8, 1, 6], 12)
10613
```

Uppgift 1.3

- Spara lösningen till Uppgift 1.3 i en fil med namnet `uppg13.py`.
- Filen skickas in via Studentklienten som '*Assignment #3*'.

Vi använder nästlade listor för att representera skyskrapor som syns över horisonten, en "skyline". Nedan är två exempel.

```
sl1 = [
    [0, 0, 0, 0, 0],
    [0, 1, 0, 0, 0],
    [1, 1, 0, 1, 0],
    [1, 1, 1, 1, 1],
    [1, 1, 1, 1, 1]
]

sl2 = [
    [0, 0, 1, 1],
    [1, 1, 1, 1],
    [1, 1, 1, 1]
]
```

Både antalet inre listor och längden på dessa inre listor är godyckligt. Alla inre listor för *en speficik skyline* kommer dock vara lika långa. Siffran `1` används för att representera byggnaderna och `0` representerar himmelen.

Din uppgift är att skriva funktionen `get_height_of_tallest_skyscraper(skyline)` som får in ett argument med en liststruktur enligt ovan. Funktionen ska returnera höjden av den högsta skyskrapan.

Exempel

Variablerna `sl1` och `sl2` är definierade enligt ovan.

```
>>> get_height_of_tallest_skyscraper(sl1)
4
>>> get_height_of_tallest_skyscraper(sl2)
3
```

Uppgift 1.4

- Spara lösningen till Uppgift 1.4 i en fil med namnet `uppg14.py`.
- Filen skickas in via Studentklienten som '*Assignment #4*'.

I ett spelprojekt har du fått i uppgift att implementera klassen `PlayerCharacter` ska användas för att ha koll på spelaren. Instanser av `PlayerCharacter` ska ha instansvariablerna `name` och `health`.

`name` innehåller en sträng med spelarens namn, `health` är ett heltal som representerar hur många hälsopoäng spelaren har. Om `health` $\leq$ `0` så räknas spelaren som död.

Implementera klassen enligt nedan:

- När instans skapas av klassen måste man ange namn på spelaren. Namnet sparas i instansvariabeln `name`. När en instans skapas ska värdet på `health` vara `100`. Exempel på skapande av ny instans: `PlayerCharacter('Player 1')`
- Metoden `take_damage(value)` ska minska instansvariabeln `health` med `value`. `value` antas vara en `int`.
- Metoden `is_dead()` ska returnera `True` om spelaren är död, annars returneras `False`.

Uppgift 1.5

- Spara lösningen till Uppgift 1.5 i en fil med namnet `uppg15.py`.

Ett vanligt verktyg när man analyserar text är ordfrekvens, d.v.s. hur ofta ord förekommer i en text. Din uppgift är att göra upp en ordfrekvenstabell för en text. Texterna som används har genomgått en enkel tokenisering för att de ska vara lättare att arbeta med.

Skriv en function `calculate_word_frequencies(tokenized_text)` som tar in en tokeniserad text som en sträng och returnerar ett dictionary där nyckeln är ett ord och värdet är hur många gånger det ordet förekommer. Skiljetecken behöver inte räknas. Versaliserade varianter av ord (t.ex. ord som inleds med stor bokstav för att ordet förekommer i början av mening) skall inte räknas som ett separat ord (se exemplet med "hej" nedan).

Tokeniseringen har delat upp texten så att varje stycke står på en egen rad, och varje token (ord eller skiljetecken) är omringat av mellanslag. Tokeniseringen gör att vissa grammatiska strukturer står för sig, och det är okej för dessa att tolkas som ord. Till exempel blir strängen `"it's"` efter tokenisering `"it ' s"`, och både `"it"` och `"s"` kan tolkas som ord.

Notera: Din funktion ska ta en sträng och returnera ett dictionary med frekvenser. För att det ska vara enklare att testa ger vi dig en textfil (`moby_dick_tokenized.txt`) som du kan läsa in och analysera. Du bestämmer själv om du använder den eller inte.

Exempel

```
>>> calculate_word_frequencies("Hej hej !\nHej då . ")
{"hej": 3, "då": 1}
```

Några facitfrekvenser ur `moby_dick_tokenized.txt`:

```
"the": 14175
"whale": 1152
"captain": 327
"queequeg": 252
```

Del 2

- För att få betyg 4 krävs 5 poäng från denna del.
- För att få betyg 5 krävs 10 poäng från denna del.

Dela upp dina lösningar i flera funktioner efter behov.

Uppgift 2.1 (5p)

- Spara svaret till Uppgift 2.1 i en fil med namnet `uppg21.py`.
- Filen skickas in via Studentklienten som '*Assignment #6*'.

Vi har en nästlad lista som representerar en bilparkering. `0` står för tom plats och `1` betyder att en bil står där. Alla rader i en parkering har lika många element. Exempel:

```
parkering1 = [[1, 0, 1, 0, 0, 0, 0],
              [0, 1, 0, 0, 0, 0, 0],
              [0, 0, 1, 0, 0, 0, 0],
              [0, 0, 0, 0, 1, 1, 0]]

parkering2 = [[0, 1, 0, 0],
              [0, 0, 0, 1],
              [1, 0, 0, 0],
              [0, 0, 1, 0]]

parkering3 = [[0, 1, 0],
              [1, 0, 1],
              [0, 1, 0],
              [1, 0, 1]]
```

Enligt våra parkeringsregler måste en bil parkera så att den *inte* inte har annan bil bredvid sig i riktningarna norr, öst, söder eller väst. En bil bredvid någon i diagonal riktning går bra.

Skriv funktionen `is_parking_ok(carpark)`, där `carpark` är en parkeringsplats representerad av en nästlad lista enligt ovan. Funktionen ska kontrollera att alla bilar i `carpark` står korrekt. Om alla bilar står korrekt, returneras `True`, annars returneras `False`.

Dela upp din lösning i fler funktioner efter behov.

Exempel

```
>>> is_parking_ok(p_plats1)
False
>>> is_parking_ok(p_plats2)
True
>>> is_parking_ok(p_plats3)
True
```

Uppgift 2.2 (5p)

- Spara svaret till Uppgift 2.2 i en fil med namnet `uppg22.py`.
- Filen skickas in via Studentklienten som '*Assignment #7*'.

Skriv funktionen `subsequences_without_duplicates(values)` vars argument `values` är en lista som innehåller heltal och strängar. Funktionen ska returnera en lista som innehåller alla *unika* delsekvenser från `values` som inte innehåller några dubletter.

En delsekvens från `values` är en sekvens med en längd på minst 1 som återfinns i `values`. Om `values` är `[1, 2, 3]` så har vi följande giltiga delsekvenser:

- `[1]`, `[2]`, `[3]`, `[1, 2]`, `[2, 3]`, `[1, 2, 3]`

Däremot är `[1, 3]` inte en giltig delsekvens.

Ordningen som delsekvenserna kommer i spelar ingen roll. Dvs att om `[[1], [2], [1, 2]]` är det rätta svaret, så är även `[[2], [1, 2], [1]]` rätt svar.

Exempel

```
>>> subsequences_without_duplicates([1, 1])
[[1]]
>>> subsequences_without_duplicates([1, '1'])
[[1], ['1'], [1, '1']]
>>> subsequences_without_duplicates([1, 2, 3, 2])
[[1], [2], [3], [1, 2], [2, 3], [3, 2], [1, 2, 3]]
# Delsekvensen [1, 2, 3, 2] är inte med eftersom värdet `2` finns med två
# gånger. Vidare är delsekvensen `[2]` endast med en gång eftersom alla
# returnerade delsekvenser skall vara unika.
```

Uppgift 2.3 (5p)

- Spara svaret till Uppgift 2.3 i en fil med namnet `uppg23.py`.
- Filen skickas in via Studentklienten som '*Assignment #8*'.

Du strosar runt på en marknad när ett valfritt objekt av intresse fångar din uppmärksamhet. Priset är rimligt, men försäljaren är en robot som inte förstår koncepten "prutning" och "växel", och du vill inte betala mer än vad det kostar.

Skriv en funktion för att ta reda på om du kan betala jämnt för vad du vill köpa, `pay_even(wallet, price)`, där argumenten `wallet` är en lista med heltal (varje heltal är ett mynt) och `price` är ett heltal. Funktionen ska försöka betala summan i `price` jämnt med hjälp av mynten i `wallet`, och ska returnera ett av följande:

- Om priset går att betala jämnt: returnera en lista med mynten som används för att betala, sorterad med de mest värda mynten först.
- Om mynten i plånboken inte räcker för att betala priset: returnera strängen `"kan inte"`
- Om mynten i plånboken överskrider priset, men det inte går att betala jämnt: returnera strängen `"vill inte"`

De myntvalörer som finns i universum för denna uppgift är 100, 20, 10, 5 och 1.

Exempel

```
>>> wallet = [100, 10, 1, 5, 1, 10, 20, 1]

>>> pay_even(wallet, 132)
[100, 20, 10, 1, 1]

>>> pay_even(wallet, 500)
'kan inte'

>>> pay_even(wallet, 4)
'vill inte'
```