Uppgift 1 – Styrstrukturer och kumulativa beräkningar

Många som är intresserade av brädspel klär alla spelkort i spelen med plastfickor. Plastfickor skyddar korten och ger de en närmast oändlig hållbarhet, kort som inte lever i plastfickor slits snabbt och gör att spelet blir ospelbart efter ett antal genomspelningar. Plastfickor är tyvärr inte gratis så det är teoretiskt sett inte alltid värt att ”sleeva” ett spel[1]. Ditt jobb är att skapa ett program som med styrstrukturer och olika indata räknar ut om det är värt att sleeva ett spel. Beräkningen ska ske med upprepningssatser (while- och/eller for-loopar).

Ett spel utan plastfickor börjar på sitt ursprungsvärde. Första gången spelet spelas så minskar värdet med 2 kr. Men andra gången minskar värdet med 2.02 kr, tredje gången med 2.0402 kr. Värdeminskningen ökar alltså med 1% varje gång spelet spelas.

Ett spel med plastfickor minskar aldrig i värde, men plastfickorna behöver bytas ut efter ett visst antal genomspelningar och kostar pengar. Ett spel med plastfickor anses därmed ha ett justerat värde som är differensen mellan spelets ursprungsvärde och hur mycket man spenderat på plastfickor (spelets ursprungsvärde – pengar spendarade på plastfickor). I första körexemplet innebär detta att det justerade värdet för spelet är 600kr under de första 100 genomspelningarna (1000 för spelets värde minus 400 för plastfickorna). Efter 100 genomspelningar är värdet 200kr och så vidare.

När spelets justerade värde med sleeves är högre än värdet för ett spel utan sleeves avslutar programmet och skriver ut hur många gånger man behöver spela spelet för att det ska vara värt att sleeva det(se körexemplet). Om det aldrig blir värt att sleeva spelet ska programmet avslutat och resultatet ska skrivas ut (se andra körexemplet).

Körexempel:

```
python3 uppgift1.py
Mata in spelets ursprungsvärde: 1000
Mata in antal kort: 400
Mata in kostnaden för en plastficka: 1
Mata in hur länge plastfickor håller: 100
Det är värt att sleeva spelet efter du spelat det 162 gånger.
```

```
python3 uppgift1.py
Mata in spelets ursprungsvärde: 1000
Mata in antal kort: 500
Mata in kostnaden för en plastficka: 1
Mata in hur länge plastfickor håller: 100
Det blir aldrig värt att sleeva spelet.
```

KRAV: Din lösning ska innefatta minst en lämplig funktion som du skapat.

TIPS: Beroende på hur man gör beräkningen kan avrundningsfel finnas, det är ok.

1 Detta är endast en tentauppgift, Pontus anser att det i princip alltid är värt att sleeva spel. Det är trevligare att blanda korten, skyddar dem från folk som äter snacks och sen har spel ofta begränsade print runs...

Uppgift 2 – Styrstrukturer och formatterad utskrift

Brädspelet Root har ett väldigt enkelt system för att lösa ut strider. Spelaren som attackerar slår två tärningar och tar bort så många av motståndarens pjäser som den högsta tärningen visar. Försvarande spelare tar bort så många av motståndarens pjäser som den lägsta tärningen visar. I Root kan en tärning slå 0-3 och det är lika stor sannolikhet för alla talen. Spelet löser detta med en 12-sidig tärning där varje siffra förekommer på 3 sidor av tärningen, men vi kan se på det som en 4-sidig tärning i denna uppgift.

Pontus vill veta hur sannolika olika resultat i striderna är. Christoffer envisas med att ”det här är lätt att bara räkna ut”, men Pontus litar inte riktigt på matematik. Därför vill han att du skriver en simulering av problemet i Python.

Ditt program ska rulla 100.000 par av tärningar och spara dessa resultat i en lämplig databehållare. Kom ihåg att den som attackerar tar det högsta resultatet, det är därför 1-0 finns i körexemplet men inte 0-1. Skriv ut alla resultaten i en tabell som matchar formatet i körexemplet, eftersom vi slumpar data i denna uppgift kan du få små variationer i utdatan för ditt program. Även om resultatet varierar lite ska ordningen av resultaten vara som i körexemplet: 0-0 kommer först, sen 1-0 och så vidare.

Tabellen har följande kolumner:

- Out: Hur många av motståndarens pjäser den attackerande spelare får plocka bort
- In: Hur många av motståndarens pjäser den försvarande spelare får plocka bort
- Occurences: Här visas hur vanligt förekommande denna tärningskombination var i procent, avrundat till 1 decimal. Antalet valutatecken (¤) som skrivs ut är lika med procenten avrundat till ett heltal.

Körexempel:

```
 Out:  In: | Occurences
---------------------------------
    0    0 | ¤¤¤¤¤¤ (6.2%)
    1    0 | ¤¤¤¤¤¤¤¤¤¤¤¤¤ (12.6%)
    1    1 | ¤¤¤¤¤¤ (6.2%)
    2    0 | ¤¤¤¤¤¤¤¤¤¤¤¤¤ (12.5%)
    2    1 | ¤¤¤¤¤¤¤¤¤¤¤¤¤ (12.7%)
    2    2 | ¤¤¤¤¤¤ (6.4%)
    3    0 | ¤¤¤¤¤¤¤¤¤¤¤¤ (12.3%)
    3    1 | ¤¤¤¤¤¤¤¤¤¤¤¤ (12.4%)
    3    2 | ¤¤¤¤¤¤¤¤¤¤¤¤¤ (12.5%)
    3    3 | ¤¤¤¤¤¤ (6.2%)
```

Krav: Din lösning skall innefatta minst en lämplig funktion som du skapat.

TIPS: Funktionen **randint(a, b)** i biblioteket **random** ger dig ett slumpat tal i intervallet **a** till och med **b**

TIPS: Funktionen **round(n, m)** kan användas för att avrunda ett flyttal (**n)** till ett tal med **m** decimaler

Uppgift 3 – Kommandoraden och behållare

Pontus har märkt att han äger för många brädspel. Sättet detta manifesterar sig är att det blivit svårt att välja vilket spel man vill spela, det finns så många och de passar bra vid olika tillfällen. Han skulle kunna sälja spel för att råda bot på problemet, vilket är helt orimligt. Istället får du i uppdrag att skapa ett Pythonprogram för att hjälpa till vid val av spel.

Pontus har lagrat sin spelsamling i en datorbehållare i Python, se den givna filen **uppgift3.py**. Du ska skapa ett program som med kommandoradsargument filtrerar denna datastruktur. Programmet ska kunna hantera följande argument, observera att användaren kan ange dem i valfri ordning och kan välja att ange valfritt antal argument:

- **--age=AGE** detta argument används för att ange åldern på den yngsta deltagaren, endast spel som den yngsta deltagaren kan vara med och spela ska finnas med
- **--players=PLAYERS** detta argument används för att ange antalet spelare som ska spela spelet, endast spel där det aktuella antalet spelare fungerar ska finnas med
- **--description=DESCRIPTION** detta argument anger en sträng som måste finnas med i beskrivningen av spelet.

KRAV: Pontus vill i framtiden kunna bygga ut detta system för att hantera många fler argument än dessa två. Din hantering av argumenten behöver således vara generell och fungera för godtyckligt många argument som heter godtyckliga saker och läses in i en lämplig databehållare. På minst ett ställe kommer du behöva använda en styrstruktur som räknar upp alla argumenten som programmet känner till, men detta vill du göra när filtrerar listan av spel, inte när du läser in argumenten. Du får alltså endast skriva exempelvis ”--age” på ett ställe i programmet.

Körexempel (användarinmatning i fetstil):

```
$ python3 uppgift1.py  --age=15
Root: Root is a highly thematic asym...
Twilight Imperium: Twilight Imperium is an epic s...
Eclipse: Eclipse is a 4X space-themed b...
All Bridges Burning: All Bridges Burning is a histo…

$ python3 uppgift1.py  --age=15 --description='asymmetric'
Root: Root is a highly thematic asym...
All Bridges Burning: All Bridges Burning is a histo...

$ python3 uppgift1.py  --players=4 --age=15 --description='asymmetric'
Root: Root is a highly thematic asym...
```

Uppgift 4 – Lite smartare sökning i strängar

Senast Pontus spelade Root förlorade han[2] mot Filip. Pontus spelade ”Vagabond” i spelet och Filip spelade ”Eyrie Dynasties”. Detta är inget som kan få fortsätta hända, därför har Pontus hittat en podcast som pratar om just Root. Det finns tyvärr många avsnitt och du måste nu hjälpa Pontus skriva ett program som söker efter avsnitt som matchar det han är intresserad av. Pontus hämtade ut hela rss-flödet för podcasten ”Woodland Warmachine” och har gjort dig tjänsten att läsa ut relevant information om alla avsnitt och sparat dessa i en bekant form, nämligen json. När du öppnat en fil i python kan du läsa ut datan med biblioteket **json**. Funktionen **json.load(FILE)** tolkar sedan innehållet i filen och returnerar en datastruktur som du kan arbeta med i Python.

Din uppgift är att skriva ett program som låter användaren mata in ord som denne vill söka efter och sedan ett filnamn (se körexempel). Programmet ska sedan söka efter avsnitt med sökorden och skriva ut de 5 avsnitt som bäst matchar sökorden på formatet *poäng: namn-på-avsnittet*.

Hur många poäng ett avsnitt får representerar hur bra avsnittet matchar sökorden. Poäng tilldelas ett avsnitt enligt följande regler:

- För varje ord i namnet tilldelas 2 poäng för varje sökord som förekommer i ordet
- För varje ord i beskrivningen tilldelas 1 poäng för varje sökord som förekommer i ordet

Om namnet på avsnittet är ”hejsanhej hopsan” och sökorden är ”Hej san” tilldelas 6 poäng. Ordet **Hej** förekommer i ett av orden i namnet och ordet **san** förekommer i båda orden i namnet. Observera att matchningen ignorerar gemener och versaler. Observera också att **hej** endast räknas som en förekomst, efter ordet **hejsanhej** är ett ord.

Tips: För att få ut en lista över alla ord i strängen **text** kan du använda kommandot **text.split(" ")**

Tips: För att få ut en sträng där alla versaler omvandlats till gemener kan du använda **text.lower()**

Tips: Det finns 2 json-filer du kan arbeta med, test.json och podcast.json, test.json innehåller bara två ”avsnitt” så du för hand kan kontrollräkna om ditt program fungerar som förväntat.

Körexempel:

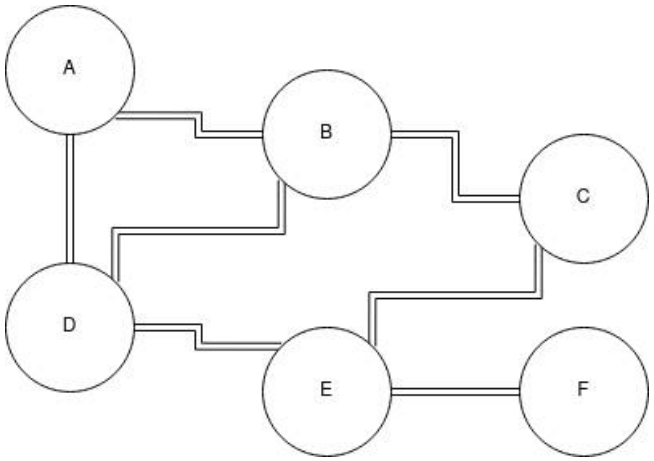
```
python3 uppgift4.py
Mata in din söksträng: root
Välj fil: test.json
7: Rootroot aRoot arootA
6: rootbeerisntroottea butroottea is tea in root

python3 uppgift4.py
Mata in din söksträng: vagabond table talk
Välj fil: podcast.json
9: Episode #07 – Look Who's Table Talking
8: Episode #28 – Look Who‘s Table Talkin‘ Too: Talk Harder
8: Episode #11 – It's Vagabond Week! (featuring Matt & Hunter from SCPT)
6: Episode #47 – Look Who’s Table Talking Now – Tilters in Time
6: Episode #10 – Rover. Wanderer. Nomad. Vagabond.
```

2 Hur svårt det än är att tro på så händer det att Pontus inte vinner när han spelar brädspel. Men det beror för det mesta på att alla runt bordet är orimligt elaka mot honom.

Uppgift 5 – ADT och hantering av referenser

I brädspelet Root finns en karta. Kartan består av gläntor (noder) i en skog som sitter ihop med stigar (bågar). I bilden till höger ser du en förenklad representation av en delmängd av kartan. Det finns 6 st noder (namngivna A-F) som sitter ihop med 7 st bågar.



Din uppgift är att representera denna karta som en abstrakt datatyp. Varje nod representeras av en **Dictionary**. För varje nod ska följande information sparas i rimligt namngivna nyckel-värde-par:

- En **List** med spelpjäser som står i noden. Spelpjäserna är bara strängar.
- En **Dictionary** som representerar alla bågar som hör till noden. Nycklarna är namnet på noden som bågen leder till och värdena referenser till andra noder. Noden **C** ska allså ha 2 nyckel-värde-par där nyckeln är ”B” respektive ”E” och värdet är en referens till nod **B** respektive **E.**

Följande adt-funktioner behöver finnas och ska fungera med det givna huvudprogrammet i **uppgift3.py**:

- create_node: Skapar en nod enligt beskrivningen ovan och returnerar denna.
- add_arc: tar en nod, ett namn på en nod, och en referens till en nod. Funktionen lägger till en båge i noden med namnet och referensen som skickades med som parametrar.
- add_piece: tar en nod och en sträng. Lägger till strängen i listan av spelpjäser.
- remove_piece: tar en nod och en sträng. Tar bort första förekomsten av strängen i listan av spelpjäser.
- get_arc: tar en nod och en sträng. Strängen är namnet på noden som skall returneras. Använd denna för att slå upp rätt båge och returnera en referens till noden som bågen leder till.
- display: tar en nod. Skriver ut noden så att formatet stämmer överens med körexemplet.

Körexempel (användarinmatning i fetstil):

```
Clearing:
paths: ['B', 'D']
pieces: []

Clearing:
paths: ['B', 'D']
pieces: ['Cat']

Clearing:
paths: ['B', 'D']
pieces: ['Cat', 'Bird']
```