Grunduppgifter (Uppgift 1-3)

Du får inte ändra given kod om inte annat anges.

Uppgift 1 – Använd behållare

I den givna filen **uppgift1.py** finns två listor, en lista av kort som finns i spelet Root och en lista av spelare. Skriv ett program som skapar en dictionary där nycklarna är spelarna och värdet är en lista av ett slumpmässigt antal kort i intervallet 1 till 3. Korten i listan ska väljas slumpvis. Kod som visar hur man väljer ett slumptal i intervallet samt ett slumpmässigt element i en lista finns i den givna filen. Skriv sedan ut behållaren formatterat som i exemplet nedan:

Funktionella krav:

```
Player: Pontus     has cards: ['Root Tea', 'Cobbler', 'Scouting Party']
Player: Copernicus has cards: ['Better Burrow Bank', 'Anvil']
Player: Alex       has cards: ['Armorers']
Player: Aron       has cards: ['Cobbler', 'Tax Collector', 'Anvil']
```

Uppgift 2 – Abstraktion

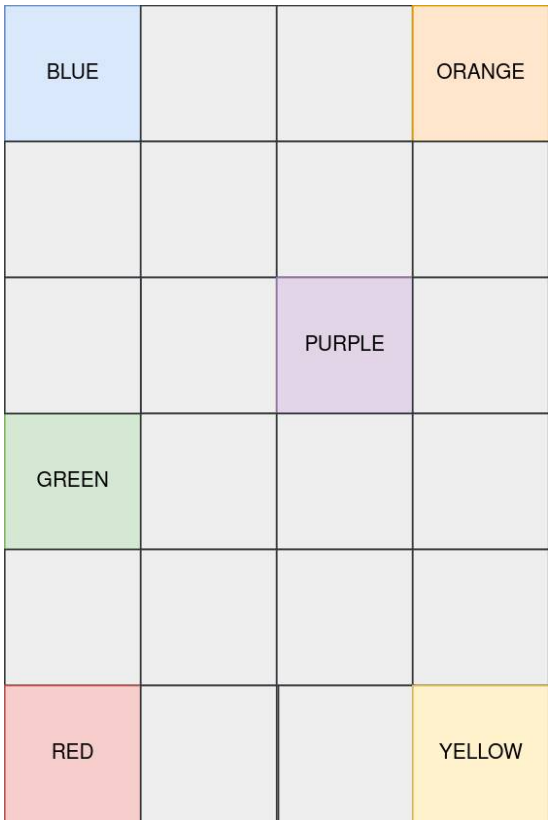
I brädspelet Root finns det en karta som består av ett antal gläntor i skogen, vi ska skapa en adt för en karta som innehåller gläntor. Den underliggande datatypen för kartan ska vara av typen **list** och den underliggande datatypen för en glänta ska vara av typen **dict**. Utgå från den givna filen **uppgift2.py** och implementera de adt-funktioner som anropas i huvudprogrammet. En glänta sparar 3 data: **id** (heltal), **suit** (sträng), **connections** (lista av **id**). Där **id** är ett unikt tal som identifierar gläntan, **suit** är vilken typ av glänta det är och **connections** är en lista av vilka andra gläntor (**id**) som denna glänta är kopplad med.

Funktionella krav (användarinmatning i fetstil):

```
clearing 1: Fox connects to: [2, 3]
clearing 2: Mouse connects to: [1, 4]
clearing 3: Rabbit connects to: [1, 4]
clearing 4: Fox connects to: [2, 3]

Clearings that are connected to Clearing 4:
============================
clearing 2: Mouse connects to: [1, 4]
clearing 3: Rabbit connects to: [1, 4]
```

Uppgift 3 -- Skapa algoritm



I spelet Ahoy spelar en spelare som smugglaren. Smugglarens huvudsyssla är att springa mellan olika platser på spelplanen och hämta last från en plats och lämna den på en annan. I Ahoy så rör man sig alltid ortagonalt (alltså inte diagonalt). Att röra sig från grön till lila på spelplanen kräver alltså att man tar två steg till höger och ett steg uppåt (eller visa versa).

I den givna filen **deliveries.txt** finns ordningen som vi ska besöka de olika platserna. I figuren till höger ser du hur just denna spelplan ska se ut, representera denna med en lämplig datastruktur i ditt program. Spelaren startar alltid uppe i vänstra hörnet. Skriv ett program som besöker platserna i tur och ordning och skriver ut hur långt det var till den nya platsen och sist det totala avståndet man färdats. Ditt program måste självklart fungera om en annan karta skrivs in och för andra filer med platser.

Krav: Minst två funktioner med tydligt avgränsade syften som följer principerna för procedurell abstraktion skall användas för att lösa problemet

Avgränsning: Kartan kommer alltid vara en fyrkant[1] och det kommer aldrig finnas några hinder på kartan.
Avståndet från en koordinat till en annan är alltså alltid summan av differensen i x- och y-led.

Funktionella krav:

```
From: start, to: green, distance: 3
From: green, to: purple, distance: 3
From: purple, to: orange, distance: 3
From: orange, to: red, distance: 8
From: red, to: yellow, distance: 3
From: yellow, to: blue, distance: 8
From: blue, to: purple, distance: 4
From: purple, to: yellow, distance: 4
Total distance: 36 spaces
```

1 Specifikt en fyrsidig polygon där alla hörn är 90 grader. Alltså en rektangel eller en kvadrat.

Avancerade uppgifter (uppgift 4 och 5)

Uppgift 4 – Inkapsling av Data och Gränssnitt

I spelet Ahoy finns olika spelpjäser. I denna uppgift ska du designa och implementera två moduler som kommer utgöra ADTer. En som representerar en spelpjäs (**piece**) och en som representerar ett spelbräde (**board**). De abstrakta datatyperna känner inte till varandras interna implementationer och får endast interageras med genom deras gränssnitt.

Piece representerar en spelpjäs. En spelpjäs har en lämplig underliggande behållare och lagrar två data: namn och en ikon. Namnet är en sträng och en ikon är en sträng som är ett tecken lång. Kom fram till vilka funktioner som behöver finnas i gränssnittet för att kunna uppfylla kravet på huvudprogrammet nedan.

Board representerar ett spelbräde. Boards interna representation och gränssnitt behöver du komma fram till utifrån vad huvudprogrammet nedan ska göra.

Huvudprogrammet som demonstreras i de funktionella kraven (och som du ska implementera) gör följande:

- Skapa ett spelbräde som är 5 rutor brett och 4 rutor högt.
- Skapa och lägg till en spelpjäs på andra platsen från vänster och tredje platsen uppifrån. Denna spelpjäs heter **Patrol** och har ikonen **P**.
- Skapa och lägg till en spelpjäs på fjärde platsen från vänster och fjärde platsen uppifrån. Denna spelpjäs heter **Ship** och har ikonen **S**.
- Tillsvidare (för alltid):
  - Rita ut spelbrädet
  - Ta inmatning från användaren: w, a, s, eller d (ingen felkontroll krävs). Flytta pjäsen som heter Patrol ett steg upp, vänster, ner, respektive höger baserat på inmatningen.

Funktionella krav (användarinmatning i fetstil):

```
python3 uppgift4.py
-------
|     |
|     |
| P   |
|   S |
-------
move (wasd): d
-------
|     |
|     |
|  P  |
|   S |
-------
move (wasd): s
-------
|     |
|     |
|     |
|  PS |
-------
```

```
move (wasd): d
-------
|     |
|     |
|     |
|   P |
-------
move (wasd): w
-------
|     |
|     |
|   P |
|     |
-------
move (wasd):
```

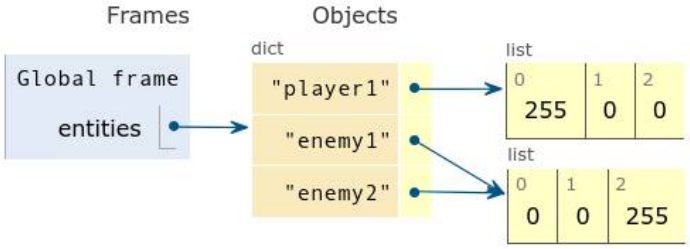
Se nästa sida för icke-funktionella krav! ---->

Icke-funktionella krav:

- Tänk på att huvudprogrammet som implementeras är ett exempel på användning så att ni inte löser problemet med fullständig uppräkning. Det är alltså rimligt att anta att användaren av ditt ADT vill kunna skapa spelplaner med andra dimensioner, andra pjäser kan flyttas, osv.
- Bound-checking (om man springer av spelplanen) kontrolleras inte i denna uppgift.
- Om en pjäs flyttas dit en annan pjäs står så försvinner pjäsen som stod på platsen från början.
- Programmet har inget naturligt sätt att avslutas just nu, det är okej att man behöver döda programmet med **Ctrl + c**

Uppgift 5 – Semantik för Referenser och Omgivningar

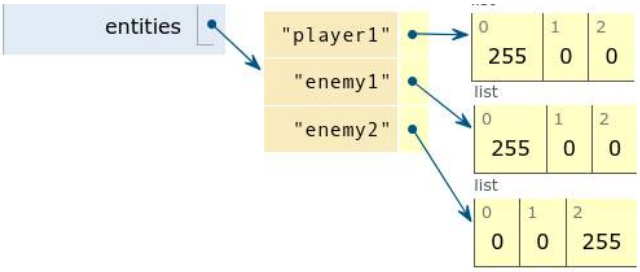
I detta program ska vi jobba med referenser och omgivningar för att hantera Sprites och Textures på ett bra sätt[2]. Enkelt kan man säga att man kan klistra på en textur på en entitet[3] för att färgsätta/rita på denna. Tillsammans bildar dessa en sprite. I denna uppgift kommer våra texturer vara listor som motsvarar färger (tre tal som är hur röd/grön/blå färgen är). Skapa ett program som om det visualiseras i Pythontutor skulle se ut på följande sätt:



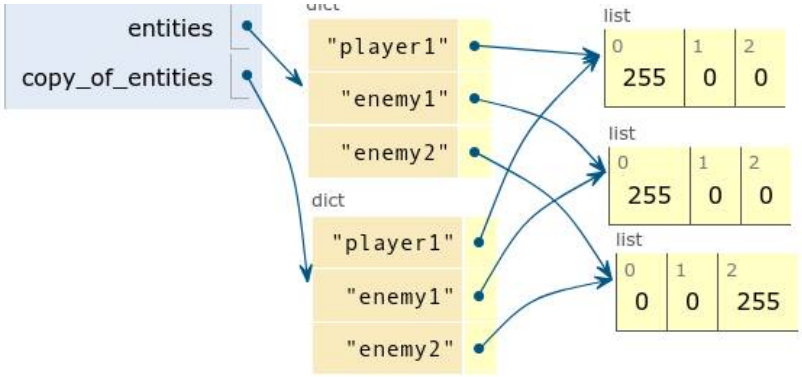
Skapa sedan följande funktioner (Det är upp till dig att komma fram till vilken in- och ut-data funktionerna behöver hantera):

1. **change_texture:** En funktion som ändrar vilken textur en entitet hänvisar till, så att den istället hänvisar till samma textur som en annan entitet.
2. **copy_texture:** En funktion som skapar en kopia av en textur som en entitet hänvisar till och hänvisar om den entiteten till den kopierade texturen.
3. **copy_entities:** En funktion som kopierar alla entiteter (men behåller hänvisningarna till samma texturer).

Nedan följer en visualisering av hur programmet ser ut efter change_texture anropats så att enemy1 istället hänvisar till samma textur som player1. Funktionen copy texture har sedan anropats för player1 för att ge denna sin egen textur.



Nedan följer en visualisering av hur programmet ser ut efter copy_entities har anropats:



2 Detta är förenklat i sammanhanget för att ge dig en chans att visa din förståelse för referenser och omgivningar i python. I verkligheten hanteras detta lite annorlunda vilket du kommer se i TDP005.

3 Lite godtyckligt vad en entitet specifikt är, vanligtvis något som existerar i en spelvärld. I det här fallet representeras en entitet av en sträng.