Prof. Dr. Stefan Göller
Dr. Da-Jung Cho

Einführung in die Informatik

WS 2019/2020

Übungsblatt 12
31.01.2020 - 06.02.2020

Abgabe: Bis zum 06.02.20 18:00 Uhr über moodle. **Beachten Sie bitte folgende Hinweise zur Abgabe.** Verwenden Sie keine globale Variablen, löschen Sie alle Ihre `print`-Anweisungen (außer natürlich diejenigen, die wir selbst in den Templates bereitstellen!) und fügen Sie möglichen Testcode in den Bereich ein, den wir in den Templates dafür vorgeben, nämlich nach `if __name__ == "__main__"`. Wenn diese obigen Vorgaben nicht eingehalten werden, werden Punkte abgezogen.

Aufgabe 1 (Objektorientierte Modellierung) (7*4=28 Punkte):

Wir befinden uns im Jahre 50 vor Christus. Ganz Gallien ist von den Römern besetzt. Das Zusammenleben von Galliern und Römern läuft sehr harmonisch. Beide Völker messen sich regelmäßig im sportlichen Wettkampf.

Weibliche Gallier haben Namen, die auf "ine" enden, männliche Gallier haben Namen, die auf "ix" enden. Gallier leben in Dörfern. Ein Dorf hat zwei herausragende Positionen: Den Druiden und den Barden, die beide männlich oder weiblich sein können. Ämterhäufung ist zulässig.

Weibliche Römer haben Namen, die auf "a" enden, männliche Römer haben Namen, die auf "us" enden.[1] Römer haben *einen* Imperator beliebigen Geschlechts und arbeiten in Legionen, welche aus einer Reihe von Soldaten besteht, die immer männlich sind. Jede Legion wird von einem Zenturio angeführt, der nicht männlich sein muss und selbst nicht Soldat dieser Legion ist. Gelegentlich sendet der Imperator eine Legion gegen ein gallisches Dorf aus. Es kommt dann zum oben erwähnten sportlichen Wettkampf, den die Gallier gewinnen, weil sie einen Zaubertrank haben. Der Druide nimmt aus Altersgründen nicht

[1] Der Name Caesars ist Gaius Julius.

am Wettkampf teil. Anschließend gibt es auf Seiten der Gallier ein Bankett, in dem jeder Bewohner ein Wildschwein isst. Davon ausgenommen ist der Barde, der sicherheitshalber an einen Baum gebunden wird.

Setzen Sie nun die obigen Sachverhalte mittels objektorientierter Programmierung um. In der Rumpfdatei zu dieser Aufgabe finden Sie eine Definition der Klasse `Mensch`, welche die Attribute `name` (eine Zeichenkette) und `weiblich` (ein `bool`) hat. Beachten sie, dass die Klasse keine Setter-Methode für das Attribut `weiblich` hat, weil das inhaltlich nicht sinnvoll ist. Lösen Sie die Aufgabe vollständig innerhalb dieser Rumpfdatei. Ihr Code für Aufgabenteil g) dient als Ergebniskontrolle und soll, genauso wie der vorgegebene Testcode, fehlerfrei (insbesondere ohne Exceptions und Fehlermeldungen) durchlaufen.

a) Erstellen sie zunächst eine Klasse `Gallier`, welche von `Mensch` erben soll. Instanzen der Klasse Gallier haben ein zusätzliches Attribut, welches angibt, wie viele Wildschweine der betreffende Gallier gegessen hat. Die Methode `get_wildschweine(self)` soll den Wert dieses Attributs zurückgeben, mit der Methode `iss_wildschwein(self)` soll der betreffende Gallier ein Wildschwein essen.

b) Erstellen Sie eine Klasse `Roemer`, welche auch von `Mensch` erbt. Instanzen dieser Klasse haben ein zusätzliches Attribut, welches speichert, wie oft der betreffende Römer einen Wettkampf verloren hat. Mit diesem Attribut soll mittels der beiden Methoden `verliere(self)` und `wie_oft_verloren(self)` interagiert werden. Zusätzlich hat die Klasse `Roemer` die Klassenvariable `imperator` vom Typ `Roemer`, welche speichert, wer Imperator ist. Mit der Methode `werde_imperator(self)` soll ein `Roemer` zum Imperator werden können. Stellen Sie sicher, dass es immer einen Imperator gibt, sobald mindestens ein `Roemer` existiert.

c) Setzen Sie für die Klassen `Gallier` und `Roemer` die Namenskonventionen der beiden Völker durch, indem Sie in den Initialisierungsfunktionen und Getter-Methoden, wenn nötig, die passenden Namensendungen an einen gewünschten Namen anhängen. Soll beispielsweise ein männlicher `Gallier` mit dem Namen `"Un"` erstellt werden, verändern Sie den Namen zu `"Unix"`. Soll der Name einer `Roemer`in nach `"Lil"` geändert werden, heisst sie danach `"Lila"`. Passende Namen werden nicht verändert: eine `Gallier`in, der `"Hermine"` heissen soll kann auch so heißen.

d) Erstellen Sie eine Klasse `Dorf`, welche Attribute `bewohner` (eine Menge (`Set`) von `Galliern`) sowie `druide` und `barde` (jeweils `Gallier`) haben, und auch in dieser Reihenfolge an die Initialisierungsfunktion übergeben werden. Erstellen Sie für die Attribute `druide` und `barde` Getter- und Setter-Methoden, und beachten Sie, dass Druide und Barde auch Bewohner sind. Erstellen Sie für das Attribut `bewohner` eine Getter-Methode.

e) Erstellen Sie weiterhin eine Klasse `Legion`, welche Attribute `soldaten` (eine Menge (`Set` von `Roemern`) und `zenturio` (ein `Roemer`) hat, die in dieser Reihenfolge an

die Initialisierungsfunktion übergeben werden. Das Attribut `zenturio` soll normale Getter- und Setter-Methoden bekommen, das Attribut `soldaten` soll mittels der Methoden `rekrutiere(self, ein_roemer)` bzw. `pensioniere(self, ein_roemer)` manipuliert werden. Das zweite Argument ist dabei jeweils vom Typ `Roemer`. Beachten Sie die Regeln dafür, wer Soldat sein kann–wer nicht Soldat sein kann, wird nicht rekrutiert.

f) Schreiben Sie schließlich eine Funktion `wettkampf(ein_dorf, eine_legion)`, welche einen sportlichen Wettkampf zwischen ihren beiden Argumenten simuliert und zeilenweise (In der Form `"Gallier XYine misst sich mit Römer/Zenturio ABCus"`) ausgibt, welcher Gallier sich mit welchen Römern misst. Die genaue Aufteilung der Römer auf die Gallier bleibt Ihnen überlassen, es soll aber, sofern genug Römer da sind, jeder geeignete Gallier mindestens einen Römer abbekommen. Beachten Sie, dass sich jeder Römer pro Wettkampf mit genau einem Gallier misst, dass aber der Zenturio auch teilnimmt. Vergessen Sie anschließend das Wildschweinbankett nicht.

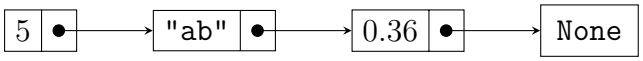
g) Fügen Sie ihren Testcode an der gekennzeichneten Stelle in der Datei ein. Erstellen Sie die geforderten Instanzen in der angegebenen Reihenfolge.

- Erstellen Sie `Gallier`innen `Laureline`, `Canine` und `Apfelsine` in gleichnamigen Variablen, wobei Sie der Initialisierungsfunktion für `Laureline` nur die Zeichenkette `"Laurel"` übergeben.
- Erstellen Sie `Gallier` `Praefix`, `Infix` und `Postfix` in gleichnamigen Variablen, wobei Sie bei letzterem nur die Zeichenkette `"Postf"` übergeben.
- Erstellen Sie `Roemer`innen `Salta`, `Mendoza` und `Ushuaia` in gleichnamigen Variablen, wobei Sie der Initialisierungsfunktion für `Salta` nur die Zeichenkette `"Salt"` übergeben.
- Erstellen Sie `Roemer` `Primus`, `Secundus`, `Tertius`, `Quartus` und `Quintus` in gleichnamigen Variablen, wobei Sie bei letzterem nur die Zeichenkette `"Quint"` übergeben.
- Erstellen Sie ein `Dorf` `Oelixdorf` mit den Bewohnern `Laureline`, `Apfelsine` und `Praefix`. Der Druide ist `Laureline`, der Barde ist `Praefix`. Erstellen Sie ein Dorf `Bekdorf` mit den Bewohnern `Laureline`, `Postfix` und `Infix`. Der Druide ist `Infix`, der Barde ist `Canine`.
- Erstellen Sie Legion `Hispana` mit Zenturio `Salta` und Soldaten `Quintus, Quartus, Tertius, Mendoza`.

Möge Ihnen der Himmel nicht auf den Kopf fallen.

Aufgabe 2 (Verkettete Listen) (8*4=32 Punkte):

In dieser Aufgabe beschäftigen wir uns mit dem Datentyp der verketteten Liste. Anders als bei den aus Python bekannten Listen besteht eine solche Liste aus Paaren, die jeweils einen Eintrag (also z.B. eine Zahl oder Zeichenkette) speichern, sowie das nächste Paar. Anstelle der gesamten Liste muss man sich daher nur das erste Paar merken. Dieses zeigt dann auf das zweite Paar, und so weiter. Man kann also insbesondere nicht auf den beispielsweise vierten Eintrag zugreifen, sondern muss sich die Liste von Paar zu Paar entlanghangeln. Graphisch kann man sich das wie folgt vorstellen:



Formal lässt sich eine verkettete Liste induktiv definieren:

(1) `None` ist eine verkettete Liste.

(2) Wenn L bereits eine verkettete Liste ist und x ein Wert (z.B. eine Zahl oder Zeichenkette), dann ist auch (x, L) eine verkettete Liste.

Also hat die oben dargestellte verkettete Liste folgende formale Entsprechung:

$$(5, (\texttt{"ab"}, (0.36, \texttt{None})))$$

Wir wollen diese Datenstruktur nun in Python modellieren. Dazu schreiben wir eine Klasse `Vliste`, welche eine verkettete Liste modellieren soll.

a) Erstellen Sie eine Klasse `Vliste`, die Attribute `eintrag` und `nachfolger` hat, welche der Initialisierungsfunktion in dieser Reihenfolge übergeben werden. Auf Getter- und Setter-Methoden verzichten wir dieses Mal. Legen sie nun obige Beispielliste als Instanz der Klasse `Vliste` an und greifen Sie auf das dritte Element dieser Liste zu.

b) Implementieren Sie die Instanzmethode `gleich(self, other)` für die Klasse `Vliste` so, dass zwei `Vliste`n als gleich angesehen werden, wenn sie die selben Einträge in der selben Reihenfolge enthalten.[2]

c) Implementieren Sie die Instanzmethode `append`, welche neben `self` noch einen Wert entgegennimmt, und ihn als neuen Eintrag an die entsprechende `Vliste` anhängt.

[2]Normalerweise verwenden wir hier die Funktion `__eq__`. Das kollidiert aber mit der Sternchenaufgabe weiter unten. Die Funktion `gleich` soll sich aber so verhalten, wie es normalerweise die Funktion `__eq__` tun würde.

Beispielsweise soll `Vliste(1, Vliste(2,None)).append(3)` als `gleich` zu der durch `Vliste(1, Vliste(2, Vliste(3, None)))` kodierten verketteten Liste angesehen werden. Ihre Funktion muss nichts zurückgeben.

d) Implementieren Sie die Instanzmethode `extend`, welche neben `self` noch eine weitere `Vliste` entgegennimmt, und diese an die in `self` kodierte `Vliste` anhängt. Beispielsweise soll `Vliste(1, Vliste(2,None)).extend(Vliste(3, Vliste(4, None)))` als `gleich` zur durch `Vliste(1, Vliste(2, Vliste(3, Vliste(4, None))))` kodierten verketteten Liste angesehen werden. Ihre Funktion muss nichts zurückgeben.

e) Schreiben Sie eine Instanzmethode `filter`, welche neben `self` noch eine Funktion entgegennimmt, welche ihrerseits immer `True` oder `False` zurückgibt. Zurückgegeben werden soll dann eine Liste aus *neuen* Instanzen von `Vlisten`, welche nur die Einträge aus der alten Liste enthält, bei denen als Argument die übergebene Funktion `True` zurückgibt. Beispielsweise soll

```
Vliste(1, Vliste(2, Vliste(3,None))).filter(lambda x: x % 2 == 1)
```

eine verketttete Liste zurückgeben, die `gleich` der durch `Vliste(1, Vliste(3,None))` kodierten Liste ist. Verwenden Sie nicht die in Python eingebaute Funktion `filter`.

f) Schreiben Sie eine Instanzmethode `map`, welche neben `self` noch eine Funktion entgegennimmt. Zurückgegeben werden soll dann eine Liste aus *neuen* Instanzen von `Vlisten`, welche jeweils die Ergebnisse der Funktionsanwendungen der übergebenen Funktion auf die Elemente der alten Liste enthält. Beispielsweise soll der Ausdruck `Vliste(1, Vliste(2, Vliste(3,None))).map(lambda x: x + 2)` eine verkettete Liste zurückgeben, die `gleich` zu der durch `Vliste(3, Vliste(4, Vliste(5,None)))` kodierten Liste ist. Verwenden Sie nicht die in Python eingebaute Funktion `map`.

g) Schreiben Sie eine Instanzmethode `nachListe`, welche die in `self` kodierte `Vliste` als reguläre Python-Liste zurückgibt. Beispielsweise soll `Vliste(1,Vliste(2, Vliste(3, None)))` zu `[1,2,3]` auswerten.

h) Schreiben Sie eine Funktion `nachVListe` (außerhalb der Klasse `Vliste`), welche eine reguläre Python-Liste entgegennimmt und eine `Vliste` zurückgibt, welche dieselben Werte kodiert. Beispielsweise soll `nachVliste([1,2,3])` zu `Vliste(1,Vliste(2, Vliste(3, None)))` auswerten.

i) (Sternchenaufgabe) Implementieren Sie eine Instanzmethode `istzirkulaer`, welche prüft, ob die betreffende Instanz einen Zyklus enthält, also nicht irgendwann in `None` endet. In diesem Fall soll `True` zurückgegeben werden, ansonsten `False`. Die folgende verkettete Liste enthält beispielsweise einen Zyklus.