Zeichenketten und Array

1 Taschenrechner für Brüche (A4-Sose2015 Zeichenketten)

Lernziele:

- Benutzereingaben von der Konsole einlesen.
- Zeichenketten analysieren und als Kommandos interpretieren.
- Zeichenketten zerlegen.
- Zeichenketten in Zahlen wandeln.

Schreiben Sie ein Ruby-Programm ***bruch_rechner.rb***, das Kommandos von der Konsole liest und diese in Operationen für Brüche umwandelt. Der Bruchrechner merkt sich zu jedem Zeitpunkt zwei Brüche x und y. Zu Beginn ist x= $\frac{1}{1}$ und y= $\frac{1}{1}$.

Jede Zeile enthält genau einen der nachfolgenden Ausdrücke:

Kommando	Erläuterung
n/d	Eingabe eines Bruches $\frac{n}{d}$. Der Nenner darf nicht null sein. Wenn der Nenner null ist, soll eine Fehlermeldung ausgegeben werden und auf eine neue Eingabe gewartet werden. Dieses Kommando ersetzt y ← x und x ← $\frac{n}{d}$.
n	Gleichbedeutend mit $\frac{n}{1}$.
+	Ersetzt x ← (x+y).
-	Ersetzt x ← (x-y).
*	Ersetzt x ← (x*y).
/	Ersetzt x ← $\frac{y}{x}$.
--	Ersetzt x ← -x.
//	Ersetzt x ← $\frac{1}{x}$.
**	Ersetzt x ← x^2.
x	Beendet das Programm Bruchrechner.

Nach jeder Eingabe sollen x und y ausgegeben werden.

Das Ruby-Projekt ***A4-Sose2015 Zeichenketten*** enthält die Klasse ***MyRational***, die die Grundrechenarten für Brüche implementiert (+, -, *, neg, reziprok).

Testen Sie Ihre Lösung mit der folgenden Eingabe:

```
2
3/5
+
-
*
/
--
//
**
```

Das Ergebnis ist $\frac{9}{25}$.

2 Verschlüsselte Botschaften (A4-Sose2015 Zeichenketten)

Lernziele:

- Den Unterschied zwischen Zeichenketten und Zeichen kennenlernen.
- Die Rechner-interne Darstellung der Zeichen als Zahl kennenlernen.
- Die lesbare Darstellung von Zahlen von der Rechner-internen unterscheiden können.
- Den Modulo Operator in Intervallen mit einem Offset anwenden können.
- Umgang mit Dateien lernen.
- Dateien zeichenweise lesen und schreiben können.

Verschlüsselung von Klartext macht den Inhalt des Textes für den Menschen unlesbar. Wenn der Schlüssel bekannt ist, kann der verschlüsselte Text (Chiffretext) jedoch wieder in Klartext gewandelt werden.

Eine sehr einfache Verschlüsselungsmethode ist der Caesar- Chiffrierung. Jeder Buchstabe im Klartext wird durch einen Buchstaben ersetzt, der in einem bestimmten festen Abstand **d** hinter dem Klartextbuchstaben im Alphabet steht. Nach dem „Z“ beginnt die Zählung wieder mit dem „A“. Der Code für die Chiffrierung ist der Abstand **d**, der nur im offenen Intervall (-26,26) liegen kann. Es sollen nur Großbuchstaben verarbeitet werden. Alle anderen Zeichen sollen unverändert bleiben.

Beispiel: Wenn **d=3,** wird jeder Buchstabe durch den 3’ten nach diesem Buchstaben ersetzt

```
A B C D E F G H I J K L M N O P Q R S T U V W
```

↓ …

```
D E F G H I J K L M N O P Q R S T U V W X Y Z
```

Wenn wir eine aktuelle Sportmeldung vom 15.04.2015

```
KLOPP VERLAESST DORTMUND -- ALS NACHFOLGER WIRD TUCHEL GEHANDELT.
```

mit d=3 verschlüsseln, erhalten wir:

```
NORSS YHUODHVVW GRUWPXQG -- DOV QDFKIROJHU ZLUG WXFKHO JHKDQGHOW.
```

Was ist zu tun?

Schreiben Sie eine Klasse ***Caesar***, die die Verschlüsselung repräsentiert. Die Klasse ist wie folgt spezifiziert:

- Objekte der Klasse ***Caesar*** werden mit dem Code, einem Integer für den Abstand d, erzeugt.
- Die Methode ***encode(c)*** verschlüsselt das Klartextzeichen c (Zeichencode) und liefert das entsprechende Chiffrezeichen (als Zeichencode) zurück.

Schreiben Sie ein Ruby-Script ***caesar_coder.rb***, das einen Klartext **zeichenweise** aus einer Datei liest und einen Caesar-Verschlüssler verwendet um den Klartext zu verschlüsseln. Das Ergebnis der Verschlüsselung soll sowohl auf der Konsole als auch in einer Datei ausgegeben werden.

Hinweis:

Ihr Programm kann auch decodieren / entschlüsseln. Wenn wir einen Codierer mit **d=-3** auf den verschlüsselten Text

```
NORSS YHUODHVVW GRUWPXQG -- DOV QDFKIROJHU ZLUG WXFKHO JHKDQGHOW.
```

ansetzen, dann erhalten wir wieder den ursprünglichen Text.

```
KLOPP VERLAESST DORTMUND -- ALS NACHFOLGER WIRD TUCHEL GEHANDELT.
```

Das ist für Sie in der Entwicklung ein guter Test!

Konventionen:

- Nennen Sie die Datei, die den Klartext enthält ***plain.txt***.
- Nennen Sie die Datei, die den verschlüsselten Text enthält ***encoded.txt***.
- Nennen Sie die Datei, die den entschlüsselten Text enthält ***decoded.txt***.

Hilfestellung:

Da Ruby den Datentyp Character nicht ‚kennt', müssen wir für diese Aufgabe Dateien byte-weise lesen, um den Zeichencode (-> ASCII Tabelle), einen Integer-Wert, zu erhalten. Den Integer-Wert benötigt die Klasse ***Caesar*** für die Codierung. Für die Ausgabe des Ergebnisses der Codierung benötigen wir allerdings lesbare Zeichen. Dafür müssen wir Ruby ‚mitteilen' den Integer als Zeichen zu interpretieren.

In der Datei ***zeichen_lesen_und_schreiben.rb*** finden Sie

- Beispiele für das byte-weise Lesen aus Dateien
- Beispiele, wie aus Integer-Werten bei der Ausgabe auf Konsole oder Datei wieder lesbare Zeichen werden.

Experimentieren Sie mit dem Skript, solange bis Sie den Beispielcode verstehen und anwenden können.

3 Wiederholte Strings (A4-Sose2015 Zeichenketten)

Lernziele:

- Das Zerlegen von Zeichenketten üben.
- Ein Beispiel für nicht triviale Wertgleichheit von Objekten kennenlernen.
- Ein Muster für das Kopieren von Objekten kennenlernen.
- Den Aufruf interner Methoden üben.
- Den Umgang mit ***self*** üben.
- Wiederverwendung von Methoden zur Problemlösung üben.
- Die < Relation auf Objekte eigener Klassen anwenden.
- Die Methode ***to_s()*** implementieren und den Zusammenhang zu ***puts*** begreifen.

Schreiben Sie eine Klasse ***ReplizierterString***, die einen String repräsentiert, der aus einem mehrfach wiederholten Wort besteht. Objekte der Klasse ***ReplizierterString*** enthalten ein Wort ***wort*** und einen nicht negativen Zähler ***zaehler*** für die Wiederholung des Wortes. Die leere Zeichenkette ist für ***wort*** zulässig.

Beispiel: Ein ***ReplizierterString*** mit ***wort***=“jaja“ und ***zaehler***=2 ergibt ausgeschrieben "jajajaja".

Die ausgeschriebene Form ist **nicht eindeutig**: 2x"jaja" entspricht 4x"ja" entspricht 1x"jajajaja"

Die Klasse ***ReplizierterString*** hat die folgenden Methoden:

initialize(wort,zaehler)	Initialisiert einen replizierten String mit Wort und Zähler.
kopiere()	Erzeugt eine Kopie eines replizierten Strings (von ***self***).
Reader für wort und zaehler	Verwenden Sie die in Ruby übliche Kurzform.
to_s()	Liefert die ausgeschriebene Fassung eines replizierten Strings.
==(other_rep_string)	Liefert ***true***, wenn ***self*** und ***other_rep_string*** gleich sind, sonst ***false***.
normalized?()	Liefert ***true***, wenn das Wort des replizierten Strings keine wiederholten Zeichenketten enthält.
normalize()	Maximiert den Zähler des replizierten String. D.h., wenn das Wort aus wiederholten Zeichenketten besteht, dann wird das Wort durch die kürzeste dieser Zeichenketten ersetzt und der Zähler entsprechend angepasst. Die Methode liefert einer Referenz auf ***self*** zurück.
<<(a_string)	Hängt eine Zeichenkette hinten an den replizierten String. Der Zähler bleibt unverändert.
+(wert)	Erhöht den Zähler des replizierten Strings um wert. Das Wort bleibt unverändert.
<(other_rep_string)	Prüft, ob ***self*** ein echter Präfix von ***other_rep_string*** ist.

Alle Methoden, für die der Zähler negativ wird, sollen den Fehler ***ArgumentError*** werfen. Das erreichen Sie, indem Sie den Ausdruck ***raise ArgumentError, "zaehler darf nicht negativ werden"*** verwenden.

Es ist **nur** für die Methode ***to_s*** erlaubt mit dem ausgeschriebenen String zu arbeiten. Verwenden Sie in allen anderen Methoden nur Wort und Zähler. Sie dürfen dazu interne Methoden, außer ***to_s*** verwenden.

Beispielanwendung:

```
rp = ReplizierterString.new("jaajaa",3)
rp2 = ReplizierterString.new("jajaja",2)
rp3 = ReplizierterString.new("kk",6)
rp4 = ReplizierterString.new("jahu",1)

puts rp.normalized?()  # false
puts rp2.normalized?() # false
puts rp3.normalized?() # false
puts rp4.normalized?() # true

p rp.normalize() # #<ReplizierterString:0x00000003179230 @wort="jaa", @zaehler=6>
p rp2.normalize() # #<ReplizierterString:0x000000031791e0 @wort="ja", @zaehler=6>
p rp3.normalize() # #<ReplizierterString:0x00000003179190 @wort="k", @zaehler=12>

puts rp2 # jajajajajaja
puts rp3 # kkkkkkkkkkkk
puts rp2.+(-2) # jajajaja
p rp2.<<("hu") # #<ReplizierterString:0x000000031791e0 @wort="jahu", @zaehler=4>
puts rp2  # jahujahujahujahu

rp4 = ReplizierterString.new("jahujahu",1)
puts rp2<rp4 #  false
puts rp4<rp2 # true
```

Hinweis:

Die Datei ***ReplizierterStringTest*** enthält Unit-Tests für die Methoden des replizierten Strings. Wenn alle Tests durchlaufen (d.h. mit grünem Häkchen versehen sind), dann verhält sich ihr Programm korrekt.

Ob es sich gemäß Entwurfsvorgaben korrekt verhält, prüfen wir in den Praktika.

4 Game Of Life (A4-Sose2015 GameOfLife)

Lernziele:

1. Mehrdimensionalen Arrays erzeugen und initialisieren können.
2. Mehrdimensionale Arrays befüllen können.
3. Über mehrdimensionale Arrays indiziert und nicht indiziert iterieren können.
4. Nachbarschaftsrelationen in Matrizen über Indizes ausdrücken können.
5. Vorgegebenen Sourcecode analysieren und anwenden können.
6. Die Mächtigkeit zelluläre Automaten spielerisch kennenlernen (die Theorie kommt in Automatentheorie oder Intelligenten Systemen).

Das **Spiel des Lebens** (engl. *Conway's Game of Life*) wurde 1970 vom Mathematiker John Horton Conway entworfen und ist eine spielerische Umsetzung von zweidimensionalen zellulären Automaten aus der Automatentheorie nach Stanislaw Marcin Ulam.

Das Spielfeld ist in n Zeilen und m Spalten unterteilt und mit nxm Zellen besetzt. Zellen sind entweder lebendig oder tot. Zu Beginn des Spiels wird das Gitter mit lebenden und toten Zellen nach einem gewissen Muster initialisiert. Dies ergibt die sogenannte Anfangsgeneration. Mit Hilfe von Spielregeln, die den Zustand der Nachbarzellen berücksichtigen, werden aus dieser Anfangsgeneration sukzessive die Nachfolgegenerationen berechnet und dargestellt.

Spielregeln:

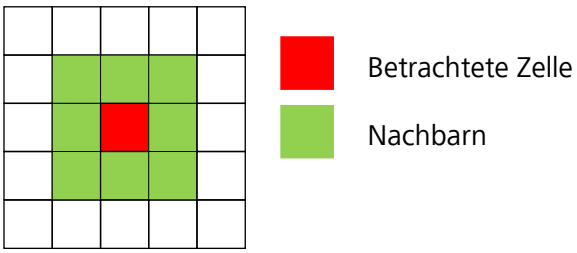
1. Eine tote Zelle mit 3 lebenden Nachbarn, wird neu geboren
2. Eine lebende Zelle mit weniger als zwei lebenden Nachbarn stirbt in der nachfolgenden Generation an Vereinsamung.
3. Eine lebende Zelle mit zwei oder drei lebenden Nachbarn überlebt in der Nachfolgegeneration.
4. Eine lebende Zelle mit mehr als drei lebenden Nachbarn stirbt in der Nachfolgegeneration an Überbevölkerung.

Mit diesen vier Regeln entstehen aus Anfangsmustern während des Spiels komplexe Strukturen. Einige Strukturen sind stabil, einige oszillieren, wachsen oder sterben ganz aus. Manche Strukturen, sogenannte *Gleiter*, bewegen sich auf dem Spielfeld in sich wiederholenden Mustern.

Das Spielfeld ist potentiell unendlich. Da wir mit einem endlichen Spielfeld arbeiten, müssen wir an den Randbereichen des Gitters, den Überlauf beachten und die Nachbarzellen Modulo des Überlaufs berücksichtigen.

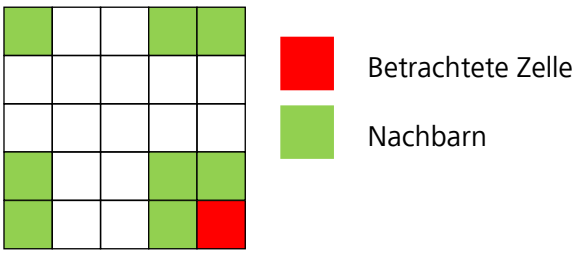
Beispiele für die Bestimmung der Nachbarn:

Im Spielfeld:



Sei i der Zeilen- und j der Spaltenindex. Wenn (i,j) die Position der betrachteten Zelle im Gitter ist, dann sind die Positionen der Nachbarzellen über der Zelle (i-1,j-1), (i-1,j), (i-1,j+1), unter der Zelle (i+1,j-1), (i+1,j), (i+1,j+1), rechts/links der Zelle (i,j+1) / (i,j-1).

Am Rand des Spielfelds:



Sei i der Zeilen- und j der Spaltenindex. Wenn (i,j) die Position der betrachteten Zelle im Gitter ist, dann sind die Positionen der Nachbarzellen über der Zelle (i-1,j-1), (i-1,j), (i-1,(j+1)%m), unter der Zelle((i+1)%n,j-1), ((i+1)%n,j), ((i+1)%n,(j+1)%m), rechts/links der Zelle (i,(j+1)%m) / (i,j-1).

Im den Beispielen ist n=m=5. Da k%5 = k für k < 5, können die Indizes der Nachbarn immer, also auch für Nachbarzellen innerhalb des Spielfelds, %n bzw. %m gerechnet werden.

Was ist zu tun?

Sie sollen das Game of Life in der Toolbox realisieren. Ergänzen Sie dazu die Klasse ***Life*** mit den Methoden.

initialize(n,muster_index)

Objekte der Klasse ***Life*** werden mit einem Parameter für n, die Anzahl der Zeilen und Spalten des Gitters, und einem Index für eine Anfangskonfiguration (Muster) erzeugt.
Im ***initialize*** soll aus dem Muster eine Anfangsgeneration erzeugt und auf der Leinwand dargestellt werden (Aufruf von ***sichtbar_machen***).

Darstellung der Muster:
Die Klasse ***Life*** soll verschiedene Muster in einem 3-dimensionalen Array verwalten. Jeder Eintrag in dem 3-dimensionalen Array entspricht einem Muster. Das Muster, ein zweidimensionales Array, enthält zwei-elementige Arrays, die die Position der lebenden Zellen kodieren. Bsp.: Das Muster [[1,1],[1,2]] beschreibt eine Anfangskonfiguration mit 2 lebenden Zellen auf den Position [1,1] und [1,2]. Das erste Element ist der Zeilen-, das zweite der Spaltenindex.

Erzeugen der Anfangsgeneration:
Die Klasse Life verwaltet die Generationen in einer nxn Matrix. Auf jeder Position der Matrix steht eine Zelle. Zu Beginn wird die Matrix mit Zellen gefüllt. Dann werden die lebendigen Zellen (ergibt sich aus dem Muster) entsprechend markiert.

Das Projekt für diese Aufgabe enthält die bereits implementierte Klasse ***Zelle***. Objekte der Klasse ***Zelle*** repräsentieren die Zellen des Spiels. Zellen kennen ihre Position und ihren Zustand. In Abhängigkeit von ihrem Zustand können Zellen sich korrekt darstellen (Methode ***sichtbar_machen()***). Zustandsänderungen auf den Zellen erreicht man durch Aufruf von ***sterben()*** und ***leben()***. Der Zustand der Zelle lässt sich über das Prädikat ***lebendig?()*** abfragen. Zellen stellen sich relativ zum Gitternetz dar. Dazu muss den Zellen bei der Erzeugung der Offset des Gitternetzes zur Leinwand sowie der Zeilen und Spaltenindex im Gitter mitgegeben werden. Des Weiteren muss für alle lebendigen Zellen das Flag lebendig bei der Erzeugung auf ***true*** gesetzt werden.

sichtbar_machen()

Die Methode macht alle Zellen des Gitters sichtbar. Nutzen Sie dazu einen bekannten Iterator für Arrays und eine geeignete Methode der Klasse ***Zelle***.

unsichtbar_machen()

Die Methode macht alle Zellen des Gitters unsichtbar. Nutzen Sie dazu einen bekannten Iterator für Arrays und eine geeignete Methode der Klasse ***Zelle***. Die Methode ist dann sehr praktisch, wenn mehrere Life Objekte in der Toolbox simuliert werden.

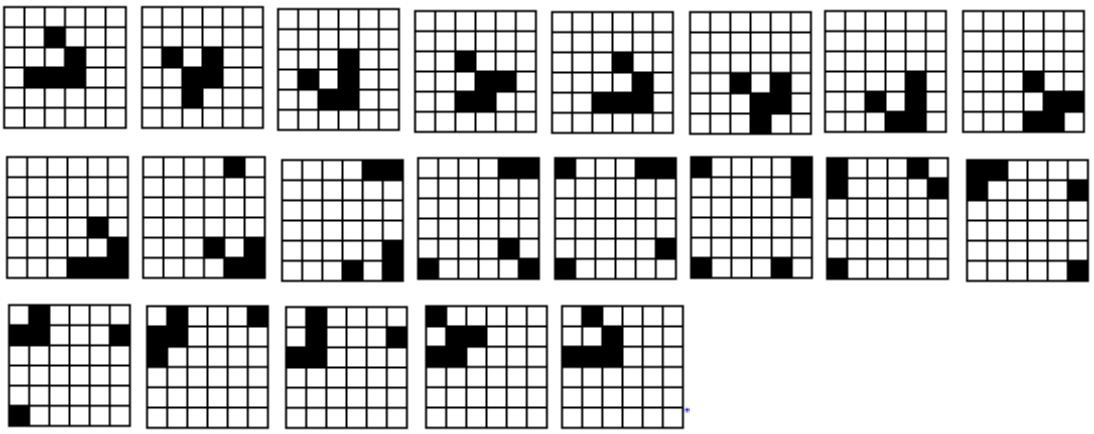
simuliere(schritte)

Die Methode ***simuliere*** hat den Parameter ***schritte***, der die Anzahl der Nachfolgegenerationen bestimmt. Die Methode ist bereits implementiert, ruft aber die interne Methode ***naechste_generation*** auf, die Sie noch implementieren müssen

naechste_generation() Die Methode bestimmt für jede Zelle zunächst die Anzahl der lebenden Nachbarzellen. Schreiben Sie dazu eine interne Methode, die diese Anzahl berechnet. Dann berechnet die Methode auf Basis der 4 Regeln, ob die Zelle in der Nachfolgegeneration neu geboren wird, weiterlebt oder stirbt. Jede Zelle, die in der Nachfolgegeneration eine tote Zelle ist, wird in eine Liste der toten Zellen eingetragen, jede, die in der Nachfolgegeneration lebendig ist, in eine Liste der lebendigen Zellen. Erst nachdem alle Zellen untersucht wurden, dürfen und müssen die Zellen in ihren neuen Zustand wechseln. Abschließend muss die neue Generation sichtbar gemacht werden.

zuruecksetzen Die Methode stellt für ein ***Life*** Objekt den Anfangszustand wieder her. Dazu werden zunächst alle Zellen „getötet“, dann die Zellen der Anfangskonfiguration zum „Leben“ erweckt und abschließend alle Zellen sichtbar gemacht. Diese Methode erleichtert das Testen in der Entwicklungsphase.

Abschließend noch ein Beispiel für einen Gleiter. Die Grafiken wurden durch Aufruf der Methode ***naechste_generation*** (20-mal) für ein 6x6 Gitter erzeugt und dienen Ihnen als Kontrollausgabe.



Suchen Sie im Netz nach mindestens 2 weiteren Beispielen für Anfangskonfigurationen und testen Sie ihr Game of Life mit dieser und den 2 weiteren Beispielen.