DOKUMENTATION

Warriors & Gatherers

Vorwort

Diese Dokumentation gehört zum Programm Warriors & Gatherers Version 1.2. Sollten Fehler in der Dokumentation oder im Programm gefunden werden, bittet der Autor um Rückmeldung an maikel@posteo.de.

Changelog

Version 1.1

- Herausforderung Expert hinzugefügt
- Herausforderungen angepasst
- KIs besser randomisiert, sodass sie nicht eine bestimmte Startposition bevor-/ bzw. benachteiligen.
- Dokumentation: Abschnitt über yourAI.py hinzugefügt.

Version 1.2

- Bugfixes

Inhaltsverzeichnis

1 Einleitung . . . 3

2 Das Spielprinzip . . . 3

3 Der Client . . . 3

3.1 Bedienung . . . 3

3.2 Funktionsweise . . . 4

3.3 Fehlermeldungen . . . 4

3.4 yourAI.py . . . 4

3.5 Herausforderungen . . . 5

3.6 Speichern . . . 5

4 Python . . . 5

5 Die Klassen . . . 6

5.1 Direction . . . 6

5.2 UnitType . . . 6

5.3 Unit . . . 7

5.4 RoundState . . . 9

5.5 FieldType . . . 10

5.6 Field . . . 10

5.7 Base . . . 12

5.8 GameConfiguration . . . 13

6 Die Spielmechanik . . . 14

6.1 Resourcenfelder . . . 14

6.2 Resourcen sammeln . . . 15

6.3 Kampf . . . 15

6.4 Bewegen der Einheiten . . . 15

7 Anhang . . . 16

7.1 Einführung in Python . . . 16

1 Einleitung

Warriors & Gatherers ist ein simples, rundenbasiertes, in Python geschriebenes Strategiespiel. Im Gegensatz zu herkömmlichen Spielen dieser Art nimmt der Spieler aber hier nicht direkt Einfluss auf das Spiel. Stattdessen ist es die Aufgabe des Spielers eine KI, eine künstliche Intelligenz, zu programmieren, die für den Spieler die Züge übernimmt.

Diese Dokumentation beschreibt das Spielprinzip und die Spielmechanik, erklärt den Client, gibt eine kurze Einführung in Python und enthält eine ausführliche Beschreibung der vordefinierten Klassen, auf die die KI zugreifen kann.

2 Das Spielprinzip

Das Spielfeld von Warriors & Gatherers ist quadratisch und besteht aus Feldern vom Typ Wasser, Wand, Land oder Basis. Jeder der vier Spieler hat genau eine Basis auf dem Feld. Weiter erscheinen auf dem Spielfeld mehr oder weniger zufällig Resourcen. Das Ziel ist es am Ende des Spiels mehr Resourcen zu haben als die Gegner.
In seiner Basis kann der Spieler zwei Arten von Einheiten erzeugen, Warrior (Krieger) und Gatherer (Sammler). Dabei sind Gatherer günstiger in der Erzeugung und können mehr Resourcen tragen, als Warrior, dafür sind Warrior stärker im Zweikampf. Resourcen werden gesammelt, indem sie von den Einheiten in die Basis getragen werden.
Einheiten können sich nur auf dem Land bewegen und können nicht gegnerische Basen betreten. Dabei kann jede Einheit pro Runde maximal einmal in eine Himmelsrichtung bewegt werden. Betritt eine Einheit ein Feld, auf dem sich schon eine gegnerische Einheit befindet, kommt es zum Kampf, wobei der Angreifer einen Vorteil hat. Eine Einheit kann kein Feld betreten, auf dem sich schon eine Einheit des selben Spielers befindet, es sei denn bei dem Feld handelt es sich um die eigene Basis.

3 Der Client

Im ersten Abschnitt wird kurz beschrieben, wie man den Client bedient. Im zweiten Teil wird erklärt, wie genau der Client ein Spiel erzeugt. Zum Schluss wird auf Fehlermeldungen und Log-Dateien eingegangen.

3.1 Bedienung

Angenommen jeder der vier Spieler hat eine funktionierende KI. Sobald der Client eingestellt wurde, kann das Erzeugen des Spiels durch betätigen des Buttons Run oder mit der Return-Taste gestartet werden. Während das Spiel erzeugt wird, gibt der Ladebalken den Fortschritt der Berechnung an. Nachdem das Spiel erzeugt wurde, kann man es mit den Buttons Play/Pause, « und » bzw. den Tasten Space, ← und → abspielen.
Berechnete Spiele können mit dem Button Save gespeichert, gespeicherte Spiele mit dem Button Load geladen werden.

Warnung. *Wird die Berechnung gestartet, lässt sie sich nicht mehr sauber unterbrechen. Eine Möglichkeit ist einen* `KeyInterruptError` *hervorzurufen, indem man das Kommandozeilenfenster der Anwendung auswählt und* STRG+C *drückt.*
Dies kann unter anderem relevant werden, wenn man in einer KI eine Endlosschleife hat.

3.2 Funktionsweise

Angenommen jeder der vier Spieler hat eine funktionierende KI, das heißt sie erzeugen keine Fehlermeldung bei der Ausführung. Dann werden in jeder Runde nacheinander die KIs der einzelnen Spieler ausgeführt. Dazwischen werden die Einheiten bewegt, Resourcen auf dem Spielfeld erzeugt und die Kämpfe ausgeführt. Eine Runde sieht dementsprechend folgendermaßen aus:

```
KI von Spieler 1 wird ausgeführt
Einheiten werden bewegt, Resourcen werden erzeugt, Kämpfe werden ausgeführt
KI von Spieler 2 wird ausgeführt
Einheiten werden bewegt, Resourcen werden erzeugt, Kämpfe werden ausgeführt
KI von Spieler 3 wird ausgeführt
Einheiten werden bewegt, Resourcen werden erzeugt, Kämpfe werden ausgeführt
KI von Spieler 4 wird ausgeführt
Einheiten werden bewegt, Resourcen werden erzeugt, Kämpfe werden ausgeführt
```

Das heißt die KI gibt das Verhalten des Spielers einer gesamten Runde wider.
Fällt ein Spieler aus, weil er keine Einheiten und keine Resourcen mehr hat, entfällt auch das Erzeugen der Resourcen nach seinem Zug.
Um das Verhalten der KI zu bestimmen, stehen dem Programmierer unter anderem Informationen zur aktuellen Runde, zur eigenen Basis, zu den eigenen Einheiten und zum Spielfeld zur Verfügung. Informationen aus vergangenen Runden können nicht verwendet werden bis, auf eine Ausnahme. Siehe hierzu die Methode `getLastMove()` in 5.3!
Für weitere Informationen siehe 6.

3.3 Fehlermeldungen

Die Log-Dateien tragen denselben Namen wie die KI im laufenden Spiel und haben die Endung *.log. Sie werden in demselben Ordner gespeichert, indem sich das Hauptprogramm befindet. Fehlermeldungen, die bei der Ausführung einer KI entstehen, werden in den entsprechenden Log-Dateien gespeichert.
Auch werden alle anderen Ausgaben, die während der Ausführung einer KI geschehen, in diese Log-Datei gespeichert, insbesondere Ausgaben, die mit dem `print`-Befehl erzeugt werden. Dies erweist sich beim debuggen als sehr nützlich.

3.4 yourAI.py

Dem Spieler stehen zunächst ein Mal zwei Dateien, yourAI.py und yourFunctions.py, zur Verfügung. Es ist vorgesehen in diese Dateien die KI zu programmieren. Immer, wenn der Spieler dran ist, wird yourAI.py einmal ausgeführt.
yourFunctions.py enthält alle Funktionen, die in yourAI.py aufgerufen werden. Hier lassen sich neue Funktionen oder Klassen definieren. Es ist auch möglich weitere Dateien durch den Befehl `import foo` einzubinden.
Prinzipiell kann man natürlich alle Funktionen und Klassen, die man braucht, in yourAI.py definieren, aber der Übersicht hilft es, wenn man diese auslagert. Siehe den Abschnitt über Funktionen in 7.1!
Auch kann man andere PYTHON-Bibliotheken importieren, zum Beispiel durch `import random`.

3.5 Herausforderungen

Es gibt vier Herausforderungen: Noob, Intermediate, Advanced und Expert. Mit jeder bewältigten Herausforderung werden neue Karten und die neue Herausforderung freigeschalten. Der Fortschritt in den Herausforderungen wird automatisch gespeichert.
Es ist eine Dummy-KI eingebaut, die gar nichts tut. Diese lässt sich aufrufen, indem man in ein KI-Feld Dummy reinschreibt.

3.6 Speichern

Spiele können mit dem Button Save gespeichert werden. In der Regel werden die Dateien in den dafür vorgesehenen Ordner /saves/ gespeichert. Beim Speichern werden auch die *.log-Dateien ins selbe Verzeichnis wie das Spiel kopiert.

4 Python

In diesem Abschnitt wird nur sehr kurz auf Python eingegangen. Für weitere Informationen wird auf einschlägige Quellen aus dem Internet verwiesen. Hat der Leser noch keinerlei Programmiererfahrung, wird `http://learnpythonthehardway.org/book` empfohlen. Zwar wird bei learnpythonthehardway.org mit Python 2.x gearbeitet, doch kommen die Unterschiede bei Warriors & Gatherers kaum zu tragen.
Der Client wurde in Python 3.x geschrieben. Der schwerwiegendste Unterschied zu Python 2.x ist wohl

```
Python 2.7.9
>>> print 'hello'
hello

Python 3.4.3
>>> print 'hello'
  File "<stdin>", line 1
    print 'hello'
                ^
SyntaxError: Missing parentheses in call to 'print'
```

Für einzelne Unterschiede sollte man auch hier das Internet befragen.
Das, was Python auf den ersten Blick am stärksten von den meisten anderen Programmiersprachen unterscheidet, zumindest was die Syntax angeht, ist die sogenannte *Einrückungsregel* (englisch: *Indentation rule*). In Python werden Blocks von Schleifen, Funktionen oder If-Abfragen nicht durch Klammern markiert, wie zum Beispiel bei Java oder C/C++/C#, sondern durch dieselbe Einrückung der Zeilen. Eine Stück Code könnte zum Beispiel folgendermaßen aussehen:

```
for k in range(4):
    if k == 1:
        print(k)
    elif k == 3:
        print('zwei')
```

Wichtig ist hierbei Tabulator und Space nicht zu vermischen! In dem Client wurden als Einrückung standardmäßig vier Leerzeichen verwendet. Der Autor empfiehlt diesen Standard beizubehalten.
Python ist eine Skriptsprache, das heißt sie muss in der Regel nicht kompiliert werden, sondern wird von einem Interpreter Zeile für Zeile interpretiert. Dies hat den Vorteil, dass man einzelne Befehlszeilen in den Interpreter eingeben kann und man dadurch direkt eine Ausgabe hat.

Beim definieren von Funktionen muss man darauf achten, dass der Interpreter keine Funktionen ausführen kann, die erst weiter unten im Code definiert werden, da diese noch nicht durch den Interpreter gelaufen und somit von ihm noch nicht erkannt worden sind.
Im Anhang befindet sich noch eine kurze Einleitung in Python , die das wichtigste abdeckt, um den vorliegenden Code zu verstehen. Für weitere Information sei an dieser Stelle ein weiteres Mal auf das Internet verwiesen.

5 Die Klassen

In diesem Abschnitt werden ausführlich alle Klassen, die dem Programmierer zur Verfügung stehen, eingeführt, ihre Methoden erklärt und beschrieben, wie sie in Warriors & Gatherers verwendet werden.
Im Folgenden wird davon ausgegangen, dass die Klassen, wie in yourAI.py, durch

```
from classes import *
```

importiert worden sind.

5.1 Direction

Diese Klasse enthält nur die `Strings` der Himmelsrichtungen, in die man eine Einheit bewegen kann, und hat keine Methoden.

```
>>> Direction.NORTH
'NORTH'
>>> Direction.WEST
'WEST'
>>> Direction.SOUTH
'SOUTH'
>>> Direction.EAST
'EAST'
>>> Direction.STAY
'STAY'
```

5.2 UnitType

Diese Klasse enthält sowohl die Strings zu den beiden Einheitentypen, als auch einige Informationen über die Einheitentypen.

```
>>> FieldType.WARRIOR
'WARRIOR'
>>> FieldType.GATHERER
'GATHERER'
```

Warnung. *Objekte der Klasse* UnitType *sind* ***keine*** *Strings. Möchte man zum Beispiel den Einheitentyp einer Einheit als String, benötigt man eine weitere Methode oder setzt das Objekt der Klasse* UnitType *in die* `str`*-Funktion ein. Siehe dazu das Beispiel zur Mehtode* `getString()`*!*

- `getString()`
  Gibt den Einheitentyp als String zurück.

```
>>> unit
<classes.Unit object at 0x7fa2d48528d0>
>>> unit.getUnitType()
<classes.UnitType object at 0x7f519b87da58>
>>> unit.getUnitType().getString()
'WARRIOR'
>>> str(unit.getUnitType())     # der String eines Objektes der Klasse UnitType
'WARRIOR'
>>> str(unit.getUnitType()) == UnitType.WARRIOR
True
```

- `getMaxResources()`
  Gibt die Anzahl der Resourcen, die der Einheitentyp maximal tragen kann, als `int` zurück.

```
>>> str(unit.getUnitType())
'WARRIOR'
>>> unit.getMaxResources()
30
```

- `getStrength()`
  Gibt die Stärke des Einheitentyps als `float`-Wert zurück. Mit Hilfe dieses Wertes wird der Ausgang eines Kampfes gewürfelt. Siehe dazu 6.3.

```
>>> str(unit.getUnitType())
'WARRIOR'
>>> unit.getMaxResources()
30
```

- `getCost()`
  Gibt die Kosten des Einheitentyps als `int`-Wert zurück.

```
>>> str(unit.getUnitType())
'WARRIOR'
>>> unit.getCost()
60
```

5.3 Unit

Die Klasse UNIT enthält Informationen über eine Einheit wie Einheitentyp, getragene Resourcen und Position. Zusammen mit der Spielerzugehörigkeit können Einheiten eindeutig über eine UnitID identifiziert werden. Weiter lässt sich mit der Methode `moveUnit(direction)` die entsprechende Einheit in eine Himmelsrichtung bewegen.

- `getUnitId()`
  Gibt die UnitID der Einheit als `int`-Wert zurück. Jede Einheit eines Spielers hat eine eindeutige UnitID. Diese wird in der Reihenfolge vergeben, in der die Einheiten erstellt werden.

```
>>> base
<classes.Base object at 0x7fa2d48730f0>
>>> base.getUnits()
[]
>>> base.createUnitWarrior()
>>> base.getUnits()
[<classes.Unit object at 0x7fa2d48528d0>]
>>> base.getUnits()[0].getUnitId()
0
>>> base.createUnitGatherer()
```

```
>>> base.getUnits()
[<classes.Unit object at 0x7fa2d48528d0>, <classes.Unit object at 0
    x7fa2d484bef0>]
>>> base.getUnits()[1].getUnitId()
1
```

- `getUnitType()`
  Gibt den Einheitentyp als UnitType-Objekt zurück.

```
>>> unit
<classes.Unit object at 0x7fa2d48528d0>
>>> unit.getUnitType()
<classes.UnitType object at 0x7f519b87da58>
```

- `getOwner()`
  Gibt den Namen des Spielers, dem die Einheit gehört, als `String` zurück.

```
>>> unit.getOwner()
'yourAI'
```

- `getResources()`
  Gibt die Anzahl der Resourcen, die die Einheit bei sich trägt, als `int` zurück.

```
>>> unit.getResources()
0
```

- `getPosition()`
  Gibt die Position der Einheit auf dem Spielfeld als Liste zweier positiven `int`-Werte, die den Koordinaten entsprechen, zurück. Beachte für das Beispiel die Klasse Field in 5.6.

```
>>> field
<classes.Field object at 0x7f519b87d8d0>
>>> field.getUnitOnField()
<classes.Unit object at 0x7fa2d48528d0>
>>> field.getPosition()
[3, 4]
>>> field.getUnitOnField().getPosition()
[3, 4]
```

- `moveUnit(direction)`
  Das Argument muss vom Typ Direction sein. Siehe dazu 5.1!
  Diese Methode gibt der Einheit den Befehl sich in eine vorgegeben Himmelsrichtung zu bewegen. Ob die Einheit dann tatsächlich bewegt wird hängt von mehreren Faktoren ab. Siehe dazu 6.4!
  Bei mehrmaligem ausführen dieser Methode für ein festes `unit` wird der String `Unit can't move more than once a round!` in die Log-Datei ausgegeben und es wird `False` zurückgegeben.
  Ist das Argument nicht vom Typ Direction wird der String `Wrong direction!` in die Log-Datei ausgegeben und es wird `False` zurückgegeben.

```
>>> unit.moveUnit('beliebiger String')
Wrong Direction!
False
>>> unit.moveUnit(Direction.SOUTH)
>>> unit.moveUnit(Direction.SOUTH)
Unit can't move more than once a round!
False
```

- `getLastMove()`
  Diese Methode gibt die Himmelsrichtung, in die die Einheit zum letzten Mal bewegt wurde, als DIRECTION-Objekt zurück.
  Sollte die Einheit nicht bewegt worden sein, weil sie zum Beispiel sonst eine gegnerische Basis oder ein Feld, auf dem schon eine andere befreundete Einheit ist, betreten würde, gibt diese Methode `Direction.STAY` zurück. Für weitere Informationen siehe 6.4!

```
>>> unit.getLastMove()
'WEST'
>>> unit.moveUnit(Direction.SOUTH)
>>> unit.getLastMove()
'SOUTH'
```

5.4 RoundState

Diese Klasse enthält die Informationen zur aktuellen Runde wie die eigene Basis, die Liste der eigenen Einheiten, ein Abbild des Spielfelds und die aktuelle Rundenzahl.

- `getMap()`
  Gibt ein Abbild des Spielfeldes zurück. Hier gibt es kein Beispiel, da `roundState.getMap()` eine Liste von Listen von Objekten vom Typ FIELD ist und das den Rahmen sprengen würde. Siehe hierzu 5.6!

  Warnung. *`roundState.getMap()` ist nur ein Abbild des Spielfeldes und allem, was sich darauf befindet, inklusive Basen und Einheiten. Möchte man diese manipulieren um etwa Einheiten in der Basis zu bauen oder Einheiten auf dem Spielfeld zu bewegen, muss man die Objekte `roundState.getBase()` und `roundState.getUnits()` verwenden.*

- `getBase()/getOwnBase()`
  Gibt die eigene Basis zurück. Mit Hilfe dieser können unter anderem Einheiten erzeugt werden und enthält die aktuelle Anzahl der Resourcen und eine Liste der eigenen Einheiten.

```
>>> roundState
<classes.RoundState object at 0x7f518b37de67>
>>> roundState.getBase()
<classes.Base object at 0x7fa2d48730f0>
>>> roundState.getOwnBase()
<classes.Base object at 0x7fa2d48730f0>
```

- `getUnit()/getOwnUnit()`
  Gibt eine Liste aller Einheiten des Spielers zurück. Das ist exakt dieselbe Liste, die man durch `roundState.getBase().getUnits()` bzw. `roundState.getBase().getOwnUnits()` erhält.

```
>>> roundState.getUnits()
[<classes.Unit object at 0x7fa2d48528d0>, <classes.Unit object at 0
    x7fa2d48520b8>, <classes.Unit object at 0x7fa2d484bef0>]
>>> roundState.getOwnUnits()
[<classes.Unit object at 0x7fa2d48528d0>, <classes.Unit object at 0
    x7fa2d48520b8>, <classes.Unit object at 0x7fa2d484bef0>]
```

- `getOwnName()`
  Gibt den eigenen Spielernamen als `String` zurück.

```
>>> roundState.getOwnName()
'yourAI'
```

- `getOwnResources()`
  Gibt die Anzahl der Resourcen in der Basis als `int` zurück.

```
>>> roundState.getOwnResources()
200
```

- `getRoundNumber()`
  Gibt die aktuelle Rundenzahl als `int` zurück.

```
>>> roundState.getRoundNumber()
1
```

5.5 FieldType

Diese Klasse enthält nur die `Strings` der Feldtypen und hat keine Methoden. Für weitere Informationen über `fieldType.RESOURCE` siehe 6.1.

```
>>> FieldType.LAND
'LAND'
>>> FieldType.WATER
'WATER'
>>> FieldType.WALL
'WALL'
>>> FieldType.RESOURCE
'RESOURCE'
>>> FieldType.BASE
'BASE'
```

5.6 Field

Die Klasse Field enthält Informationen über ein Feld auf dem Spielfeld.
Verwendet wird diese Klasse in `roundState.getMap()`.
`roundState.getMap()` ist eine Liste von Listen von Objekten vom Typ Field. Dabei entspricht jedes dieser Objekte einem Feld auf dem Spielfeld. Zum Beispiel wird das Feld mit den Koordinaten $(2, 3)$ folgendermaßen aufgerufen:

```
roundState.getMap()[3][4]
```

Bemerkung. Man beachte, dass der k-te Eintrag einer Liste `liste` durch `liste[k-1]` aufgerufen wird.

Beachte die Warnung zur Methode `roundState.getMap()` in 5.4!

- `getFieldType()`
  Gibt den Feldtyp des Feldes als `String` zurück.

```
>>> type(field)
<class 'classes.Field'>
>>> field.getFieldType()
'BASE'
```

- `getPosition()`
  Gibt die Position des Feldes auf dem Spielfeld als Liste zweier positiven `int`-Werte, die den Koordinaten entsprechen, zurück.

```
>>> field.getPosition()
[3,3]
```

- `getBase()`
  Falls sich eine Basis auf dem Feld befindet, wird die Basis als Objekt vom Typ BASE zurückgegeben. Ansonsten wird `None` zurückgegeben.

```
>>> field1.getBase()            # Basis auf dem Feld
<classes.Base object at 0x7fa2d48730f0>
>>> field2.getBase() == None    # keine Basis auf dem Feld
True
```

- `getUnitOnField()`
  Wenn das Feld nicht vom Typ BASE ist und sich eine Einheit auf dem Feld befindet, wird die Einheit als Objekt vom Typ UNIT zurückgegeben. Befindet sich keine Einheit auf dem Feld, so wird `None` zurückgegeben.
  Ist das Feld vom Typ BASE, dann wird eine Liste aller Einheiten als `list`-Objekt zurückgegeben, die sich in der Basis befinden. Diese Liste ist leer (`[]`), wenn sich keine Einheit in der Basis befindet.

```
>>> field1.getBase()                    # Basis auf dem Feld
<classes.Base object at 0x7fa2d48730f0>
>>> field1.getUnitOnField()
[<classes.Unit object at 0x7ff19c8f2ba8>]
>>> field2.getBase()                    # Keine Basis auf dem Feld
>>> field2.getUnitOnField()             # Einheit auf dem Feld
<classes.Unit object at 0x7fa2d484bef0>
>>> field2.getUnitOnField() == None     # keine Einheit auf dem Feld
True
```

- `getResources()/getResourcesOnField()`
  Gibt die Anzahl der Resourcen auf dem Feld als `int`-Objekt zurück.

```
>>> field.getResources()
0
>>> field.getResourcesOnField()
0
```

- `isCrossable(unit)`
  Das Argument muss vom Typ UNIT sein.
  Gibt einen Wahrheitswert (`True` oder `False`) zurück, je nachdem ob das Feld theoretisch von der Einheit `unit` betreten werden kann. Das muss jedoch nicht unbedingt bedeuten, dass die Einheit das Feld auch tatsächlich betritt. Zum Beispiel könnte das Feld vorher von einer anderen befreundeten Einheit besetzt werden oder das Feld ist zu weit entfernt. Für weiteres siehe 6.4!
  Konkret gilt:
  Ist das Feld vom Typ `FieldType.LAND` oder `FieldType.RESOURCE` und befindet sich keine andere Einheit desselben Spielers auf dem Feld, so wird `True` zurückgegeben.
  Ist das Feld vom Typ `FieldType.BASE` und ist die Basis desselben Spielers wie `unit`, so wird `True` zurückgegeben.
  Ist das Feld vom Typ `FieldType.WATER` oder `FieldType.WALL` oder besetzt durch eine andere Einheit desselben Spielers, so wird `False` zurückgegeben.
  Ist das Feld vom Typ `FieldType.BASE` und ist die Basis eines anderen Spielers, so wird `False` zurückgegeben.
  Ist `unit` schon auf dem Feld, so wird `True` zurückgegeben.

```
>>> (type(field),type(unit1),type(unit2))
(<class 'classes.Field'>, <class 'classes.Unit'>, <class 'classes.Unit'>)
>>> field.getPosition()
[3, 3]
>>> (unit1.getPosition(),unit2.getPosition())
```

```
([3, 3], [2, 4])
>>> unit1.getOwner() == unit2.getOwner() # units gehören demselben Spieler
True
>>> field.isCrossable(unit1)
True
>>> field.isCrossable(unit2)
False
```

- `isReachable(unit)`
  Das Argument muss vom Typ UNIT sein.
  Gibt einen Wahrheitswert (`True` oder `False`) zurück, je nachdem ob das Feld theoretisch von der Einheit `unit` im nächsten Zug **erreicht** werden kann, d.h. es wird genau dann `True` zurückgegeben, wenn sich die Positionen von `unit` und dem Feld nur in einer Koordinate um maximal 1 unterscheiden.
  Hier wird nicht geprüft, ob das Feld theoretisch von der Einheit **betreten** werden kann. Das geschieht durch `isCrossable(unit)`.

```
>>> field.getPosition()
[3, 4]
>>> (unit1.getPosition(),unit2.getPosition(),unit3.getPosition(),unit4.
    getPosition())
([3, 4], [2, 4], [3,6], [4,3])
>>> field.isCrossable(unit1)
True
>>> field.isCrossable(unit2)
True
>>> field.isCrossable(unit3)
False
>>> field.isCrossable(unit4)
False
```

5.7 Base

Die Klasse BASE enthält Informationen zur Basis eines Spielers inklusive der aktuellen Anzahl der Resourcen und eine Liste der Einheiten des Spielers. Weiter lassen sich in der Basis neue Einheiten erzeugen.

- `getResources()`
  Gibt die Resourcen des Spielers als `int`-Objekt zurück.

```
>>> base
<classes.Base object at 0x7fa2d48730f0>
>>> base.getResources()
200
```

- `getHiddenUnits()`
  Gibt eine Liste der Einheiten zurück, die sich in der Basis befinden.

```
>>> base.getHiddenUnits()
[<classes.Unit object at 0x7ff19c8f2ba8>]
```

- `getUnits()`
  Gibt eine Liste aller Einheiten des Spielers zurück. Das ist exakt dieselbe Liste, die man durch `RoundState.getUnits()` bzw. `RoundState.getOwnUnits()` erhält.

```
>>> base.getUnits()
[<classes.Unit object at 0x7fa2d48528d0>, <classes.Unit object at 0
    x7fa2d48520b8>, <classes.Unit object at 0x7fa2d484bef0>]
```

- `getPosition()`
  Gibt die Position der Basis auf dem Spielfeld als Liste zweier positiven `int`-Werte, die den Koordinaten entsprechen, zurück.

```
>>> base.getPosition()
[3,3]
```

- `getOwner()`
  Gibt den Namen des Spielers, dem die Basis gehört, als `String` zurück.

```
>>> base.getOwner()
'yourAI'
```

- `createUnitWarrior()`
  Erzeugt eine Einheit vom Typ `UnitType.WARRIOR`, solange genug Resourcen vorhanden sind. Diese Methode gibt bei Erfolg `None` zurück. Die neuen Einheiten werden ans Ende der Liste eingefügt, die man durch `base.getUnits()` oder `roundState.getUnits()` erhält.
  Sind nicht genügend Resourcen vorhanden, wird der String `'Not enough resources for warrior!'` in die Log-Datei ausgegeben und `False` zurückgegeben.

```
>>> base.getResources()
80
>>> base.getUnits()
[]
>>> base.createUnitWarrior()
>>> base.getUnits()
[<classes.Unit object at 0x7fa2d48528d0>]
>>> base.getResources()
20
>>> base.createUnitWarrior() # Nicht genügend Resources
Not enough resources for warrior!
False
>>> base.createUnitWarrior() == False
Not enough resources for warrior!
True
```

- `createUnitGatherer()`
  Wie `createUnitWarrior()` entsprechend für GATHERER. Hier wird bei nicht genügend Resourcen der String `'Not enough resources for gatherer!'` in die Log-Datei ausgegeben und `False` zurückgegeben.

5.8 GameConfiguration

Diese Klasse enthält Informationen zum Spiel wie die Rundenanzahl und Status der Einheitentypen. Für weitere Informationen siehe 5.2!

- `getRounds()`
  Gibt die Gesamtrundenanzahl des Spiels als `int` zurück.

```
>>> gameConfiguration.getRounds()
1000
```

- `getUnitCostWarrior()`
  Gibt die Kosten des Einheitentyps `UnitType.WARRIOR` als `int` zurück.

```
>>> gameConfiguration.getUnitCostWarrior()
60
```

- `getUnitCostGatherer()`
  Gibt die Kosten des Einheitentyps `UnitType.GATHERER` als `int` zurück.

```
>>> gameConfiguration.getUnitCostGatherer()
50
```

- `getUnitStrengthWarrior()`
  Gibt die Stärke des Einheitentyps `UnitType.WARRIOR` als `float`-Wert zurück.

```
>>> gameConfiguration.getUnitStrengthWarrior()
0.75
```

- `getUnitStrengthGatherer()`
  Gibt die Stärke des Einheitentyps `UnitType.GATHERER` als `float`-Wert zurück.

```
>>> gameConfiguration.getUnitStrengthGatherer()
0.5
```

- `getUnitMaxResourcesWarrior()`
  Gibt die Anzahl der Resourcen, die der Einheitentyp `UnitType.WARRIOR` maximal tragen kann, als `int` zurück.

```
>>> gameConfiguration.getMaxResourcesWarrior()
30
```

- `getUnitMaxResourcesGatherer()`
  Gibt die Anzahl der Resourcen, die der Einheitentyp `UnitType.GATHERER` maximal tragen kann, als `int` zurück.

```
>>> gameConfiguration.getMaxResourcesGatherer()
100
```

6 Die Spielmechanik

In diesem Kapitel werden einige Aspekte der Spielmechanik näher beleuchtet. Nach jedem Zug werden die Einheiten, die den entsprechenden Befehl erhalten haben, bewegt, Kämpfe ausgeführt und neue Resourcen erzeugt.

6.1 Resourcenfelder

Felder vom Typ `UnitType.RESOURCE` sind die einzigen Felder, auf denen Resourcen erscheinen können, ohne dass eine Einheit diese verlieren muss. Ansonsten unterscheiden sie sich nicht von Feldern vom Typ `UnitType.LAND`.
Nach jedem Zug wird für jedes Feld gewürfelt, ob dort Resourcen erscheinen oder nicht. Der Vollständigkeit halber wird im Folgenden dargestellt, wie die Wahrscheinlichkeit dafür berechnet wird. Da das für das Programmieren der KI praktisch irrelevant ist, sollte man sich davon nicht verwirren lassen!
Sei N die Gesamtanzahl der Felder vom Typ `UnitType.RESOURCE` auf dem Spielfeld, n die Anzahl der Felder auf dem Spielfeld, auf denen schon eine Resource liegt und P die Wahrscheinlichkeit, dass auf einem Feld vom Typ `UnitType.RESOURCE` eine Resource erscheint. Dann gilt:

$$P(n/N) = \frac{1}{40} \cdot 10^{-4 \cdot \frac{n}{N}}$$

Resourcen können nur auf den Feldern vom Typ `UnitType.RESOURCE` erzeugt werden, auf denen nicht schon Resourcen liegen und auf dem sich keine Einheit befindet.

6.2 Resourcen sammeln

Resourcen werden gesammelt, indem man Einheiten auf ein Feld bewegt, auf dem eine Resource liegt. Die Resource wird automatisch von der Einheit aufgehoben. Dabei wird so viel der Resource aufgehoben, wie viel die Einheit noch tragen kann. Die Differenz wird liegen gelassen.
Betritt eine Einheit, die Resourcen bei sich trägt, die Basis, so werden die gesamten Resourcen automatisch in die Basis übertragen.

6.3 Kampf

Betritt eine Einheit ein Feld (Angreifer), auf dem schon eine gegnerische Einheit (Verteidiger) steht, so kommt es zum Kampf zwischen den beiden. Die Einheit, die den Kampf verliert, verschwindet vom Spielfeld und ist dem Spieler in der nächsten Runde nicht mehr verfügbar. Die verlierende Einheit lässt alle bei sich tragenden Resourcen fallen. Diese liegen auf dem Feld und werden, soweit Möglich, von der gewinnenden Einheit eingesammelt.
Der Ausgang des Kampfes wird folgendermaßen berechnet:
Haben Angreifer und Verteidiger dieselbe Kampfstärke (`UnitType.getStrength()`), so ist die Wahrscheinlichkeit P_A, dass der Angreifer den Kampf gewinnt,

$$P_A = 2/3.$$

Dies ist unabhängig von dem Wert der Kampfstärke.
Sind nun $F_A \neq F_V$ die Kampfstärken des Angreifers und des Verteidigers, dann ist die Wahrscheinlichkeit P_A, dass der Angreifer den Kampf gewinnt,

$$P_A = F_A.$$

Dies ist unabhängig von dem Wert der Kampfstärke des Verteidigers.
Entsprechend ist die Wahrscheinlichkeit, dass der Verteidiger den Kampf gewinnt, immer

$$P_V = 1 - P_A.$$

In der folgenden Tabelle ist nochmal eine Übersicht über die Wahrscheinlichkeiten gegeben.

P_A		Angreifer	
		Warrior	Gatherer
Verteidiger	Warrior	$\frac{2}{3}$	$\frac{1}{2}$
	Gatherer	$\frac{3}{4}$	$\frac{2}{3}$

6.4 Bewegen der Einheiten

Um Einheiten zu bewegen, verwendet man die Methode `moveUnit(direction)` aus 5.3. Dabei kann eine Einheit nur in die Himmelsrichtungen bewegt oder stehengelassen werden. Wird bei einem Objekt `unit` vom Typ Unit die Methode `moveUnit(direction)` nicht verwendet, so bleibt die Einheit stehen und `getLastMove()` gibt dann in der nächsten Runde `Direction.STAY` zurück.
Die Einheiten werden in der Reihenfolge bewegt, in der sie erzeugt wurden, d.h. die Einheit mit der niedrigeren UnitID (siehe 5.3) wird zuerst gezogen.

Beispiel. *Zwei Einheiten wollen auf dasselbe Feld ziehen:*

```
>>> unit1.getUnitID()
1
>>> unit2.getUnitID()
2
>>> unit1.getPosition()
[2,3]
>>> unit2.getPosition()
[3,2]
>>> unit1.moveUnit(Direction.NORTH)
>>> unit2.moveUnit(Direction.WEST)
```

Beiden Einheiten wurde also der Befehl gegeben, das Feld `[2,2]` zu betreten. In diesem Fall betritt `unit1` das Feld `[2,2]` und `unit2` bleibt stehen.
In der nächsten Runde haben die Einheiten folgenden Status:

```
>>> unit1.getPosition()
[2,2]
>>> unit2.getPosition()
[3,2]
>>> unit1.getLastMove()
'NORTH'
>>> unit2.getLastMove()
'STAY'
```

Haben zwei Einheiten den Befehl erhalten ein Feld zu betreten, auf dem sich ein Gegner befindet, so betritt die Einheit mit der niedrigeren UnitID das Feld. Der Kampf wird ausgeführt wie in 6.3 beschrieben. Sollte der Gegner gewonnen haben, betritt die zweite Einheit das Feld und kämpft auch gegen die gegnerische Einheit. Sollte der erste Kampf zu Gunsten der eigenen Einheit ausgegangen sein, so bleibt die zweite Einheit stehen.

7 Anhang

7.1 Einführung in Python

In diesem Kapitel werden insbesondere die Sachen eingeführt, die in Warriors & Gatherers häufig verwendet werden und sich unter Umständen stark von entsprechenden Dingen in anderen Programmiersprachen unterscheiden können. Generell empfiehlt es sich einen Interpreter zur Hand zu nehmen und darin etwas zu spielen.
Für weiteres wird auf das Internet verwiesen.

Variablen und Strings

Variablen müssen in Python nicht deklariert werden. Möchte man zum Beispiel eine Zahl als `String` ausgeben oder andersrum, verwendet man folgende Befehle:

```
>>> str(42)
'42'
>>> int('42')
42
```

`Strings` werden entweder durch ' oder " eingeklammert:

```
>>> 'hallo'
'hallo'
>>> "hallo"
'hallo'
```

Listen

Die Listen sind in PYTHON das, was in C oder JAVA die Arrays sind. Dabei kann jeder Eintrag einer Liste ein Objekt unterschiedlichen Typs beinhalten.

```
>>> Liste = [ 42 , 'string', 2.3, ['noch eine Liste']]
>>> Liste
[42, 'string', 2.3, ['noch eine Liste']]
>>> len(Liste)  # len gibt die Länge der Liste an
4
```

Einzelne Einträge von Listen können geändert werden.

```
>>> Liste[0] = 3
>>> Liste
[3, 'string', 2.3, ['noch eine Liste']]
```

Man kann Listen erweitern oder Elemente daraus löschen.

```
>>> Liste.append(2)
>>> Liste
[3, 'string', 2.3, ['noch eine Liste'], 2]
>>> Liste.remove(2.3)
>>> Liste
[3, 'string', ['noch eine Liste'], 2]
```

Hierbei ist zu beachten, dass diese Operationen nicht sonderlich performant sind. Hat man Schleifen im Code, in denen diese Operationen häufig gebraucht werden, könnte es sich lohnen sich zu überlegen, ob man das Problem nicht auf "konservative" Art lösen kann.
Werden Listen kopiert, wird nur die Referenz kopiert. Das heißt ändert man die Kopie, so wird auch das Original geändert.

```
>>> Liste
[3, 'string', ['noch eine Liste'], 2]
>>> kopie = Liste
>>> kopie
[3, 'string', ['noch eine Liste'], 2]
>>> kopie.remove(3)
>>> kopie
['string', ['noch eine Liste'], 2]
>>> Liste
['string', ['noch eine Liste'], 2]
>>> kopie[1] = 0
>>> kopie
['string', 0, 2]
>>> Liste
['string', 0, 2]
```

Weiter lassen sich Listen noch auf weitere Art und Weisen manipulieren. Hier ein paar Beispiele.

```
>>> Liste = [1,2,3,4,5]
>>> Liste
[1, 2, 3, 4, 5]
>>> Liste[0:2]
[1, 2]
>>> Liste[0:3]
[1, 2, 3]
>>> Liste[3:5]
[4, 5]
>>> Liste[3]
4
>>> Liste[3:4]
```

```
[4]
>>> Liste[:2]
[1, 2]
>>> Liste[2:]
[3, 4, 5]
>>> Liste[::2]
[1, 3, 5]
>>> Liste[::-1]
[5, 4, 3, 2, 1]
>>> Liste[::-1][:2]
[5, 4]
```

for-Schleifen, while-Schleifen und if-Abfragen

Hierbei ist die Einrückungsregel zu beachten: Blöcke werden nicht mit Klammern markiert, sondern durch Einrückung. Sollen also mehrere Zeilen in denselben Block einer Schleife, müssen alle Zeilen dieselbe Einrückung vom linken Rand aus haben. Dabei ist es wichtig Tabulator und Space nicht zu vermischen, da man sonst schnell den Überblick verliert. In Warriors & Gatherers wurde durchgehend eine 4-Leerzeichen-Einrückung für Blöcke verwendet. Es wird empfohlen diese beizubehalten.
Beispiele für Schleifen:

```
for k in range(4):
    print(k)
```

Ausgabe:

```
0
1
2
3
```

Dass die k hier nur bis 3 geht hat folgenden Grund.

```
Liste = [0,1,2,3]
for k in range(len(Liste)):
    print(k)
```

Das hat dieselbe Ausgabe wie der Code oben. Die Liste hat die Länge 4.

```
k = 0
while k<4:
    print(k)
    k += 1      # entspricht k = k + 1
```

Ausgabe:

```
0
1
2
3
```

Eine if-Abfrage sieht folgendermaßen aus:

```
if k == 2:
        print('k ist 2')
elif k == 3:
    print('k ist 3')
else:
    print('k ist was anderes')
```

Die Wahrheitswerte heißen `True` und `False`.

Funktionen

Funktionen müssen wie Variablen nicht deklariert werden. Jedoch müssen sie, bevor sie verwendet werden können, definiert werden. Das heißt wird eine Funktion an einer Stelle im Code verwendet, muss sie an einer Stelle oberhalb definiert werden.

```
def foo(Variable1, Variable2):
        # hier passiert etwas
        return Wert
def bar():
        # hier kann foo aufgerufen werden, nicht andersrum
        return  # Hier wird None zurückgegeben
```