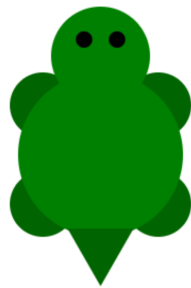


PYNTURTLE - EINE TURTLE-KLASSE FÜR JUPYTER NOTEBOOKS



Individuelles Projekt erstellt im Rahmen der GymInf-Ausbildung

Reto Schwander

27. August 2025

Betreuung: Dr. Patrick Schnider
Universität Basel

Inhaltsverzeichnis

1	Einleitung	3
2	Didaktische Prinzipien beim Programmierenlernen	4
3	Anforderungen	5
4	Installation	6
4.1	Voraussetzungen	6
4.2	Installation via Pip	6
5	Paketstruktur	7
6	Umsetzung	11
6.1	Die Turtle	11
6.2	Der Canvas und die Steuerbefehle	11
6.2.1	Initialisierung	11
6.2.2	Die <code>drawScene()</code> -Funktion	14
6.2.3	Die Steuerbuttons	17
6.2.4	Die Steuerung der Turtle	17
6.3	Die <code>showcode</code> -Implementierung in <code>turtle.py</code>	20
6.4	Die <code>showcode</code> -Implementierung in <code>codeControl.js</code>	22
7	Abhängigkeiten und Performance	24
7.1	Abhängigkeiten	24
7.2	Performance	24
8	Nutzung	26
8.1	Import und Initialisierung	26
8.2	Beispiele	26
9	Tutorial	29
10	Fazit und Ausblick	30
	Literatur	31
	Abbildungsverzeichnis	31

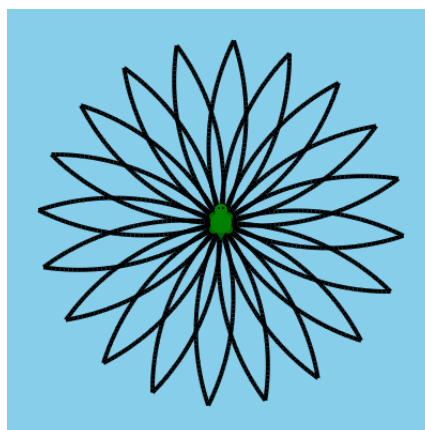


Abb. 1: Mit *pynbTurtle* erstellte Graphik.

Vorwort

Diese Arbeit entstand im Rahmen der Informatikausbildung für Gymnasiallehrkräfte (GymInf). Sie dokumentiert die Entwicklung eines Python-Pakets, mit dem Zeichnungen direkt in ein Jupyter Notebook erstellt werden können.

Zunächst wird die Motivation für das Paket erläutert, gefolgt von den funktionalen Anforderungen. Im Anschluss erklärt ein kurzes Kapitel die Installation, bevor die Paketstruktur übersichtlich vorgestellt wird. Daraufhin wird die technische Umsetzung beschrieben und ein Blick auf Abhängigkeiten sowie Performance geworfen. Ein weiteres Kapitel zeigt anhand von Codebeispielen die praktische Nutzung des Pakets. Ergänzend dazu wurde ein kleines Tutorial erstellt, dessen Aufbau kurz skizziert wird. Abschliessend fasst die Arbeit die Ergebnisse zusammen und gibt einen Ausblick auf mögliche Erweiterungen.

Mein Dank gilt allen, die mich bei dieser Arbeit unterstützt haben. Besonderer Dank gebührt meiner Familie, die während der Entwicklungsphase auf vieles verzichtet hat. Zuletzt möchte ich Dr. Patrick Schnider für die Betreuung dieser Arbeit herzlich danken.

Reto Schwander

1 Einleitung

Im digitalen Zeitalter ist Programmieren eine Schlüsselkompetenz: Es fördert logisches Denken, kreative Problemlösung und eröffnet vielfältige Möglichkeiten in Wirtschaft, Wissenschaft und Alltag. Je nach Altersstufe und Vorkenntnissen kommen dabei unterschiedliche Sprachen zum Einsatz – von einfachen, einsteigerfreundlichen Umgebungen bis hin zu komplexen Frameworks. Doch häufig bleibt verborgen, was hinter dem Code tatsächlich passiert: Genau hier setzt Turtle-Graphics an, indem es das unmittelbare Zeichnen und Erleben von Programmabläufen ermöglicht.

In meinem Unterricht arbeite ich seit jeher mit Jupyter Notebooks und der Programmiersprache Python – obwohl es zahlreiche Alternativen gibt. Ergänzend verwenden wir *VS Code* [1] im Unterricht, um auch ausserhalb der Notebook-Welt in einer professionellen Entwicklungsumgebung vertraut zu werden. Die klassische Python-Turtle öffnet dabei jedoch stets ein separates Fenster, das nach jedem Durchlauf geschlossen werden muss. Um diese Ablenkung zu reduzieren, möchte ich die Zeichenfläche direkt ins Notebook integrieren.

Es existieren bereits Ansätze wie *mobilechelonian* [2] (oder ähnliche Bibliotheken), die eine Einbettung in Jupyter Notebooks ermöglichen. Leider laufen diese Lösungen nicht immer reibungslos oder überzeugen in der Darstellung nur bedingt. Gerade die Lösung von *mobilechelonian* beruht auf älteren Versionen von zusätzlichen Paketen und läuft nur nach einem Downgrade von diesen, was möglicherweise im Konflikt mit dem Einsatz von neueren Versionen steht.

Das hier vorgestellte Paket „pynbTurtle“ orientiert sich an der Architektur von *mobilechelonian*, erweitert sie aber um zusätzliche Funktionen und Verbesserungen – für ein nahtloses, anschauliches und flexibel erweiterbares Turtle-Erlebnis direkt im Notebook.

```
1 import turtle
2
3 t = turtle.Turtle()
4 t.forward(100)
5 t.left(90)
```

Listing 1: Der simple Aufruf von Turtle-Graphics liefert ein Bild, wie es in Abbildung 2 dargestellt wird.

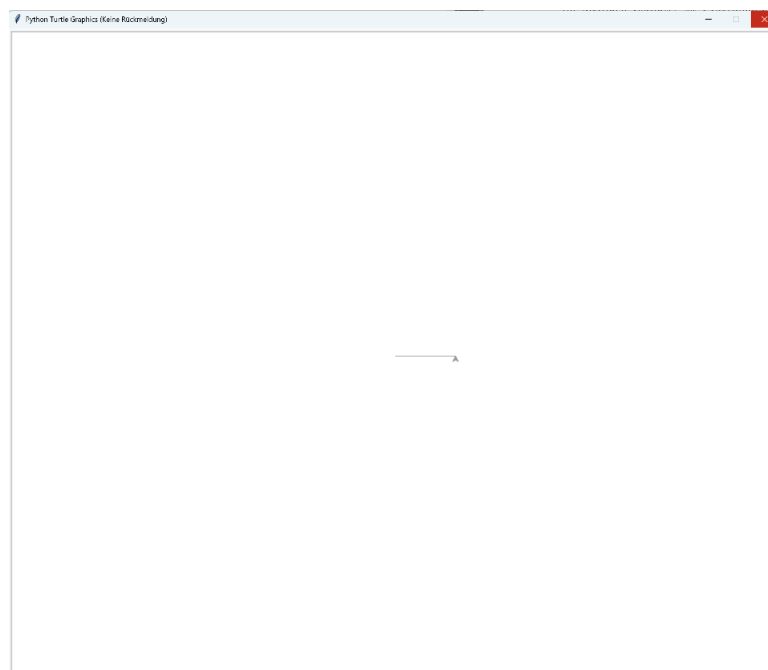


Abb. 2: Das mit dem Code in Listing 1 erzeugte Bild.

2 Didaktische Prinzipien beim Programmierenlernen

Beim Lernen des Programmierens werden folgende zentrale didaktische Prinzipien nach Wiater [3] verfolgt:

1. **Schrittweise Progression**

Die Lernenden werden systematisch von einfachen Programmen (z. B. „Hello World“) zu komplexeren Projekten geführt.

2. **Handlungsorientierung**

Lernen durch aktives Tun: Kleine, unmittelbar ausführbare Beispiele ermöglichen einen direkten Zugang zum Programmieren.

3. **Fehlerfreundlichkeit**

Fehler werden nicht als Scheitern, sondern als integraler Bestandteil des Lernprozesses behandelt.

4. **Spiralcurriculum**

Konzepte werden mehrfach und zunehmend differenziert wieder aufgenommen — etwa Schleifen zunächst in Scratch, später in Python.

5. **Anschaulichkeit**

Durch Live-Coding, Simulationen oder Visualisierungen werden Abläufe sichtbar gemacht und Verständnis gefördert.

Die Punkte 1 bis 4 werden durch die entsprechende Planung des Unterrichts abgedeckt. Mit dem hier vorgestellten Paket „pynbTurtle“ wird auf die Anschaulichkeit durch die technische Umsetzung eingegangen. Dabei soll als Grundsatz der Fokus auf dem Programmieren liegen und nicht auf der Umgebung.

Klassische Python-Turtle-Implementierungen weisen jedoch folgende Einschränkungen (vgl. Einleitung) auf:

- Zwei-Fenster-Ansatz: Die entstehenden Fenster lenken ab und blockieren gegebenenfalls den Ablauf, weil sie hinter dem eigentlichen Fenster geöffnet bleiben.
- Begrenzte Zeichenfläche: Programmierende müssen zusätzlich die Positionierung und Dimensionierung des Bildes bedenken.
- Fehlende Verzahnung von Code und Animation: Der Zusammenhang zwischen geschriebenem Code und visueller Ausgabe bleibt weniger offensichtlich.

Das hier vorgestellte Paket `pynbTurtle` adressiert diese Mängel:

- Es bietet eine **integrierte Turtle-Grafik**, die direkt unterhalb des Codes im Notebook dargestellt wird, ohne separate Fenster.
- Die Zeichenfläche ist **unbegrenzt**, mit Funktionen für Zoomen und Verschieben, damit alles sichtbar gemacht werden kann.
- Das Paket unterstützt eine **Live-Anzeige des ausgeführten Codes**, wodurch das Prinzip „Teile und Herrsche“ (Divide & Conquer) im Lernprozess greifbar wird.

3 Anforderungen

Der Funktionsumfang soll über die üblichen Befehle für die Steuerung der Turtle wie **forward(s)**, **backward(s)**, **left(d)** und **right(d)** verfügen. Ebenso soll gesteuert werden können ob die Turtle bei der Bewegung zeichnet (**pendown()**) oder nicht (**penup()**). Auch die Stiftfarbe soll gewählt werden können (**pencolor(color)**). Des weiteren soll die Turtle an eine beliebige Position mit **setposition(x,y)** gebracht werden können. Auch die Steuerung der Animationsgeschwindigkeit soll mittels **speed(v)** möglich sein, sofern gewünscht.

In *mobilechelonian* ist die Zeichenfläche auf 400×400 Pixel mit einem offset von 20 Pixel begrenzt. Sobald die Turtle den Randbereich erreicht, geht sie nicht mehr weiter. Diese Beschränkung soll nicht gelten, so dass theoretisch eine unbegrenzte Zeichnungsfläche zur Verfügung steht. Um trotzdem sehen zu können, was die Turtle zeichnet, wenn sie aus dem dargestellten Bereich läuft, soll die Zeichenfläche verschiebbar sein. Darüber soll es auch eine Zoomfunktion und Zentrierfunktion geben. Die Verschiebung, Zoomfunktion und Zentrierfunktion soll über Buttons gesteuert werden.

Oftmals verdeckt die Turtle einen vielleicht wichtigen Bereich oder gar das ganze Bild. Deshalb soll sie über einen Button aus- und wieder eingeblendet werden können. Das Einblenden eines Gitters soll die Möglichkeit bieten, die Bewegung der Turtle besser nachvollziehen zu können.

Die Abhängigkeiten zu anderen Paketen soll möglichst gering gehalten werden, so dass bei Upgrades der Pakete keine Konflikte auftreten sollen.

Gerade bei grösseren Zeichnungen, insbesondere mit Teilkreisen, zeigt die Version von *mobilechelonian* keine gute Performance auf. Dies soll mit dem erstellten Paket deutlich verbessert werden, obwohl es weiterhin Unterschiede zwischen den einzelnen Rechnern aufgrund der unterschiedlichen Leistungen der Prozessoren geben wird.

Als zusätzliche Funktion soll Programmzeile für Programmzeile die gerade ausgeführte Codezeile angezeigt werden. Diese Funktion muss zusätzlich aufgerufen werden. Wenn möglich soll sie über die Option verfügen, dass man die Animation stoppen und dann Schritt für Schritt abarbeiten kann.

Bei der Umsetzung sollte auch der pädagogische Gedanke eine zentrale Rolle einnehmen. Daher sollte die Darstellung ansprechend sein und die Bedienung unkompliziert.

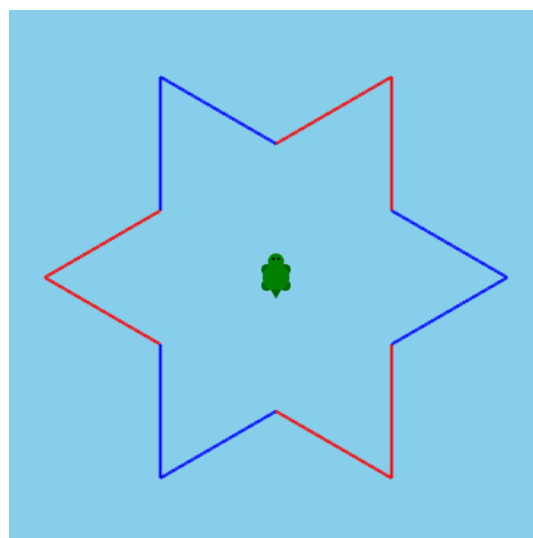


Abb. 3: Mit *pynbTurtle* erstellter Stern.

4 Installation

Der Einsatz des Pakets ist für Jupyter Notebooks gedacht. Es lässt sich aber wie jedes andere Paket installieren.

4.1 Voraussetzungen

Das Paket setzt Python 3.8 [4] oder neuer voraus. Eventuell müssen zusätzlich benötigte Paket (siehe Abschnitt 7.1) nachinstalliert werden.

4.2 Installation via Pip

Das Paket kann über `pip` wie in Listing 2 installiert werden und steht auf <https://test.pypi.org/project/pynbTurtle/> zur Verfügung. Eine generelle Veröffentlichung ist angestrebt, womit `-i https://test.pypi.org/simple/` bei einer Installation entfallen wird.

```
1 pip install -i https://test.pypi.org/simple/ pynbTurtle
```

Listing 2: *Installation des Pakets*

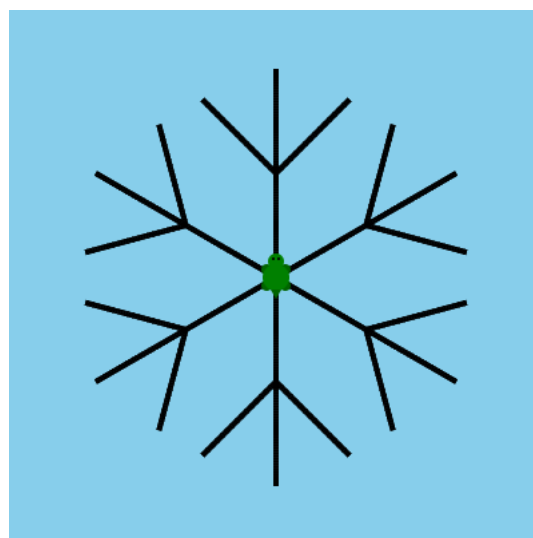


Abb. 4: *Mit `pynbTurtle` erstellte Schneeflocke.*

5 Paketstruktur

Die Paketstruktur wurde gemäss dem Python Packaging User Guide [5] erstellt. Für die Veröffentlichung wurde das Paket mit `pip -m build` verpackt und mit `py -m twine upload --repository testpypi dist/*` auf die Plattform <https://test.pypi.org/> hochgeladen.

Die Ordnerstruktur des Pakets im Überblick:

```
1 pynbTurtle-master/  
2 |-- LICENSE  
3 |-- pyproject.toml  
4 |-- README.md  
5 |-- example  
6     |-- example.ipynb  
7     |-- pynbTurtle_tutorial.ipynb  
8 |-- pynbTurtle/  
9     |-- __init__.py  
10    |-- turtle.py  
11    |-- static/  
12        |-- buttons.js  
13        |-- codeControl.js  
14        |-- turtleCanvas.html  
15        |-- turtleGraphics.js
```

Listing 3: *Ordnerstruktur*

Die Dateien können auch unter <https://github.com/screto/pynbTurtle-master.git> eingesehen werden.

Im nachfolgenden wird auf die einzelnen Bestandteile eingegangen.

LICENSE

Die Datei enthält das Copyright sowie die ausdrückliche Erlaubnis, die Software und zugehörige Dokumentation kostenlos zu nutzen, zu kopieren, zu ändern, zusammenzuführen, zu veröffentlichen, zu verbreiten, zu unterlizenzieren und/oder zu verkaufen. Weiter wird auch auf einen Haftungsausschluss hingewiesen.

pyproject.toml

Die Datei `pyproject.toml` legt die Metadaten und den Build-Workflow für das Paket fest. Damit wird definiert, wie das Paket benannt ist, welche Abhängigkeiten und Lizenzbedingungen gelten und wie es beim Bauen und Installieren inkl. aller statischen Dateien korrekt verarbeitet wird.

README.md

Die `README.md`-Datei gibt eine Übersicht über das Paket und liefert alle nötigen Infos an den Nutzer.

example.ipynb und pynbTurtle_tutorial.ipynb

Diese Jupyter Notebooks werden als Beispiele mitgeliefert. In der ersten Datei sind alle in dieser Arbeit verwendeten Abbildungen enthalten. Die zweite Datei enthält ein Tutorial zu `pynbTurtle` (vgl. Kapitel 9).

`__init__.py`












Die Datei `__init__.py` enthält nur eine einzige Zeile Code (zuzüglich eines Kommentars). Durch diese Datei wird das Verzeichnis als Python-Paket erkannt, und zugleich ermöglicht sie es, die Turtle-Klasse direkt beim Paketimport verfügbar zu machen (z.B. `from pynbTurtle import Turtle`). Somit dient `__init__.py` sowohl der Paketinitialisierung als auch der komfortablen Neuorganisation der öffentlichen API.

`turtle.py`

In der Datei `turtle.py` wird die zentrale Klasse `Turtle` definiert, die alle Methoden zur Steuerung der Turtle-Grafik (z. B. `forward`, `backward`, `left`, `right`, `penup`, `pendown`) kapselt und über eine interne JavaScript-Queue mit dem gerenderten Canvas kommuniziert. Beim Erzeugen einer neuen Instanz wird automatisch ein eindeutiges HTML-Canvas-Element in das Jupyter Notebook eingefügt, alle benötigten JavaScript-Dateien geladen und Schaltflächen für Zoom, Pan und Code-Steuerung initialisiert, sodass Zeichnungsbefehle in Python direkt visuell umgesetzt werden. Die Klasse verwaltet ausserdem Animationstempo und Pausenfunktionen, ermöglicht das Hervorheben der zuletzt ausgeführten Codezeile im Notebook und sorgt am Ende jeder Zelle für das automatische „Flushen“ aller gesammelten Befehle ins Frontend. Zusätzlich bietet sie mit `showcode` eine Funktion, um den Python-Quellcode der letzten Zelle einzublenden und interaktiv abzuspielen oder schrittweise auszuführen.

`button.js`

Die Datei `buttons.js` definiert eine einzige Funktion `window.initButtonFunctions`, die beim Laden des Turtle-Widgets alle interaktiven Schaltflächen („Buttons“) mit ihren zugehörigen Klick-Handlern verbindet. Zunächst enthält sie eine Hilfsfunktion `addListener(prefix, event, handler)`, die für jedes Element mit einer ID, die mit `prefix_` beginnt, genau einmal einen Event-Listener hinzufügt. Anschliessend werden damit Klick-Events für folgende Schaltflächen registriert:

- Zoom-In/Zoom-Out (, ): Vergrössern bzw. Verkleinern des Zeichenbereichs und Neuzeichnen der Szene.
- Pan (, , , ): Verschiebung des Ansichtsbereichs um feste Schritte in alle vier Richtungen.
- Center (): Automatisches Zentrieren aller bisher gezeichneten Linien im Canvas.
- Toggle Turtle/Grid (, ): Ein- und Ausblenden der Turtle oder des Gitternetzes.
- Export (): Exportiert den aktuellen Canvas-Inhalt als PNG-Datei.
- Reset (): Stoppt laufende Animationen, leert die Befehlsschlange, setzt die Turtle-Position zurück und zeichnet die Szene neu.

Durch diese Struktur wird sichergestellt, dass alle Bedienelemente der Benutzeroberfläche automatisch mit den entsprechenden Funktionen im Frontend verknüpft werden, ohne dass in HTML oder Python mehrfach dieselben Event-Handler angehängt werden müssen.

`codeControl.js`

In `codeControl.js` finden sich alle JavaScript-Funktionen, die das `showcode`-Feature der Turtle-API in Jupyter Notebooks realisieren und den interaktiven Ablauf (Play, Step, Stop) steuern. Zu Beginn werden in globalen Variablen die gemeinsame Befehlswarteschlange und das Ausführungs-Timing angelegt. Anschliessend wird das Hervorheben der aktuell ausgeführten Codezeile im gerenderten Code-Block definiert.

Mit `window._turtleFlush(cmds_list, delay)` wird die Python-seitige Befehlssammlung in die globale Queue `window._turtleCommands` gespielt, ohne sie sofort abzuspielen. Über `initTurtleCodeBlock(uid, raw)` wird der HTML-Container für den Code aufbereitet: Der rohe Zellinhalt wird in ein `<pre>`-Element mit nummerierten ``-Zeilen aufgeteilt und angezeigt. Direkt danach sorgt `initTurtleCodeControls(uid, commandDelay)` dafür, dass über dem Code-Block drei Buttons (`Stop`, `Step`, `Play`) generiert und mit den entsprechenden Event-Handletern (`window._turtleStop`, `window._turtleStep`, `window._turtlePlay`) verknüpft werden.

So ermöglicht `codeControl.js` eine flüssige, interaktive Wiedergabe und schrittweise Ausführung von Turtle-Befehlen direkt im Notebook, inklusive Code-Hervorhebung und eigener UI-Steuerung.

`turtleCanvas.html`

In `turtleCanvas.html` befindet sich die HTML- und Inline-JavaScript-Vorlage, die das Turtle-Widget in ein Jupyter Notebook einbettet und initialisiert. Konkret umfasst die Datei:

- **Kontroll-Buttons**
Ein `<div>` mit dynamischen IDs (`zoomInButton_UID`, `panUpButton_UID` usw.), in dem alle Schaltflächen zum Zoomen, Verschieben, Ein-/Ausblenden von Grid und Turtle, Exportieren und Resetten gruppiert sind. Daneben gibt es einen separaten Container für die `showcode`-Steuerung („Stop“, „Step“, „Play“).
- **Styles**
Ein Inline-`<style>` definiert das Aussehen des Code-Blocks: hellblauer Hintergrund, Monospace-Schrift, Zeilennummerierung via CSS-`counter`, und Hervorhebung der aktuellen Zeile über die Klasse `.highlight`.
- **Layout**
Ein flexibles Container-Layout (`display: flex`) teilt den Bereich in zwei Spalten auf: links das `<canvas>`-Element (400×400 px) für die Turtle-Grafik, rechts die Code-Spalte mit Platz für die externen Controls und den ausgeblendeten Code-Container (`<div id="codeContainer_UID">`), in den `showcod()` später den Python-Quelltext als `<pre>` einfügt.
- **Globale Variablen & Funktionen**
Direkt im `<script>` werden sämtliche Zustandsvariablen (`zoomLevel`, `panX`, `turtleX`, `turtleSegments`, `penIsDown` etc.) angelegt und die grundlegenden Turtle-Funktionen (`moveTurtle`, `rotateTurtle`, `penup`, `pendown`, `resetTurtle`, `setStrokeColor`) definiert.
- **Zeichenschleife**
Im abschliessenden Inline-Script enthält `drawScene()` die komplette Logik zum Aktualisieren des Canvas: Hintergrundfarben, optionales Gitter, alle bisher gespeicherten Linien-segmente sowie das Aufrufen von `drawTurtle()` aus `turtleGraphics.js`.

Die Datei bildet somit das Gerüst, das sowohl die Benutzer-Interface-Elemente bereitstellt als auch die Kern-State-Machine und Render-Pipeline für die interaktive Turtle-Grafik in Jupyter Notebooks implementiert.

`turtleGraphics.js`

In `turtleGraphics.js` wird die Funktion `window.drawTurtle(x, y, angle)` definiert, die das eigentliche „Turtle“-Icon auf dem Canvas rendert. Die Funktion speichert zunächst den aktuellen Zeichenkontext (`ctx.save()`), verschiebt und rotiert ihn auf die übergebenen Koordinaten und den Winkel und zeichnet dann den Körper der Schildkröte:

1. Beine: Vier dunkelgrüne Kreise an den Ecken des Körpers.
2. Körper: Ein grosser, grüner Kreis in der Mitte.
3. Schwanz: Ein dreieckiges, dunkelgrünes Polygon am hinteren Ende.
4. Kopf: Ein kleinerer, grüner Kreis vorne.
5. Augen: Zwei winzige, schwarze Kreise auf dem Kopf.

Am Ende wird der ursprüngliche Canvas-Zustand wiederhergestellt (`ctx.restore()`), sodass nachfolgende Zeichnungen nicht von dieser Transformation beeinflusst werden.

Die Aufgabe von `turtleGraphics.js` ist also allein, das Turtle-Symbol in der Szene darzustellen; alle weiteren Zeichen- und Steuerungsfunktionen (Linien, Animation, Buttons, Code-Highlighting) werden in den anderen Skripten (`turtleCanvas.html`, `buttons.js`, `codeControl.js`) umgesetzt.



Abb. 5: Die Turtle wie sie in `turtleGraphics.js` erzeugt wird.

6 Umsetzung

Die Umsetzung in Kapitel 3 gestellten Anforderung wird hier beschrieben.

6.1 Die Turtle

Das Aussehen der Turtle wird in Abbildung 5 dargestellt. Im Prinzip ist das Aussehen zweit-rangig soll aber für den Nutzer ansprechend sein (siehe Kap. 3). Daher wurde ein schlichtes Design gewählt, das aber einer Schildkröte sehr ähnlich sieht. Die Schildkröte sollte stets in die Bewegungsrichtung schauen, daher muss sie eine Drehung mit machen und natürlich auch die Bewegungen müssen mitgeführt werden.

```
1 // Datei: pynbTurtle/static/turtleGraphics.js
2
3 window.drawTurtle= function(x, y, angle) {
4     ctx.save();
5     ctx.translate(x, y);
6     ctx.rotate(angle * Math.PI / 180);
7
8     // Beine ...
23    // Koerper ...
29    // Schwanz ...
38    // Kopf ...
44    // Augen ...
52
53     ctx.restore();
54 }
```

Listing 4: Auszug aus *turtleGraphics.js*

Die Funktion `window.drawTurtle` enthält die Parameter `x`, `y`, `angle` welche die Position und Ausrichtung der Turtle übergeben. Die Methode `ctx.save()` (siehe Listing 4) speichert den aktuellen Zeichenzustand des Canvas – also insbesondere die Transformationsmatrix (Translation, Rotation, Skalierung), aber auch Füll- und Strichstile, etc. – auf einen internen Stapel. Anschliessend wird `ctx.translate(x,y)` und `ctx.rotate(angle...)` aufgerufen, um die Turtle am entsprechenden Ort und mit entsprechender Ausrichtung gezeichnet werden kann. Der Wert von `angle` wird im Gradmass angegeben und muss ins Bogenmass umgerechnet werden. Nach dem Zeichnen wird mit `ctx.restore()` zum ursprünglichen Koordinatensystem und Stil zurückgekehrt. So bleiben alle folgenden Zeichnungen unberührt von den temporären Verschiebungen und Drehungen, die nur für die Turtle-Darstellung gelten.

6.2 Der Canvas und die Steuerbefehle

Die Initialisierung und das Rendern der Animation laufen in mehreren Schritten ab, die eng verzahnt und in `turtle.py`, `turtleCanvas.html` und `button.js` definiert sind.

6.2.1 Initialisierung

Der Aufruf der Turtle-Klasse initialisiert eine neue Turtle-Instanz mit einem eindeutigen Canvas. Damit wird garantiert, dass in einer neuen Zelle des Jupyter Notebooks beim erneuten Aufruf der Turtle-Klasse auch eine neuer Canvas erzeugt wird. Das `<canvas>`-Element ist 400×400 Pixel gross. In diesem `<canvas>`-Element wird die Animation dargestellt.

```

1  # Datei: pynbTurtle/turtle.py
11
12 class Turtle:
15
16     def __init__(self, command_delay=50):
29
30         # 1) Generiert pro Instanz eine eindeutige ID, um
           Canvas-Kollisionen zu vermeiden
31         self._uid      = uuid.uuid4().hex[:8]
32         canvas_id      = f"turtleCanvas_{self._uid}"
33         controls_id     = f"controls_{self._uid}"

```

Listing 5: Generierung einer eindeutigen ID, um Canvas-Kollisionen zu vermeiden

Um das Code-Highlighting darstellen zu können, wird ein zusätzlicher Bereich neben dem `<canvas>`-Element reserviert, wo der Code falls gewünscht hingeschrieben wird. Dies wird mit einem Flex-Container ermöglicht. Der Code-Block soll sich direkt neben der Animation befinden, damit beides gleichzeitig sichtbar ist.

```

1  <!-- Datei: pynbTurtle/static/turtleCanvas.html -->
74
75 <!-- 2) Flex-Container für Canvas links / Code rechts -->
76 <div style="display: flex; align-items: flex-start; gap: 10px;">
77
78     <!-- 2a) Zeichen-Canvas auf der linken Seite -->
79     <div style="flex: 1;">
80         <canvas id="{{CANVAS_ID}}" width="400" height="400"
81             style="border:1px solid black; display: block;">
82     </canvas>
83 </div>
84
85 <!-- 2b) Code-Spalte: Controls-Wrapper + Code-Container -->
86 <div style="flex: 1; display: flex; flex-direction: column; gap:
87     4px;">
88     <!-- Platzhalter für externe Controls (Stop/Step/Play) -->
89     <div id="codeControlsWrapper_{{UID}}"></div>
90
91     <!-- eigentlicher Code-Container -->
92     <div id="codeContainer_{{UID}}"
93         class="code-container"
94         style="display: none;
95             border:1px solid #404040;
96             padding:8px;
97             overflow:auto;
98             max-height:400px;">
99         <!-- showcode() schreibt hier sein <pre>...</pre> hin -->
100     </div>
101 </div>
102

```

Listing 6: Auszug aus `turtleCanvas.html` zur Erzeugung des Zeichnungsbereichs und Codeblocks

Direkt danach setzt ein Inline-Skript in `turtleCanvas.html` die globalen Zustands-Variablen auf, darunter `zoomLevel`, `panX`, `panY`, `turtleX`, `turtleY`, `turtleAngle` sowie das leere Array `turtleSegments`. Die Standard-Werte lassen sich aus Listing 7 entnehmen. Aus pädagogischen Überlegungen wird die Startausrichtung der Turtle auf dem Bildschirm nach oben gesetzt (Standard nach rechts gerichtet). Darum wird `turtleAngle= -90` gesetzt. Als Startposition wird

die Turtle in die Mitte des Canvas an die Koordinaten (200/200) gesetzt (Listing 7 Zeilen 117-118).

Hier wird auch das Canvas-Element

```
window.canvas = document.getElementById("{CANVAS_ID}");
```

und der 2D-Zeichnungskontext

```
window.ctx = window.canvas.getContext("2d");
```

geholt. Somit stehen alle Zustände und Zeichenwerkzeuge zur Verfügung.

```

1  <!-- Datei: pynbTurtle/static/turtleCanvas.html -->
103
104 <!-- 3) Globals & State -->
105 <script>
106     window.zoomLevel      = 1;
107     window.panX           = 0;
108     window.panY           = 0;
109     window.turtleVisible  = true;
110     window.gridVisible    = false;
111     window.strokeColor    = "black";
112     window.gridSpacing    = 20;
113
114     window.canvas         = document.getElementById("{CANVAS_ID}");
115     window.ctx            = window.canvas.getContext("2d");
116
117     window.turtleX        = 200;
118     window.turtleY        = 200;
119     window.turtleAngle    = -90;
120     window.turtleSegments = [];
121     window.penIsDown      = true;
122 </script>

```

Listing 7: Die globalen Zustandsvariablen und deren Initialwerte

Mit Listing 8 werden die drei JavaScript-Dateien `turtleGraphics.js`, `buttons.js` und `codeControl.js` in den Notebook-Kernel geladen und ausgeführt.

```

1  # Datei: pynbTurtle/turtle.py
11
12 class Turtle:
13
14     def __init__(self, command_delay=50):
15
16         # 3) JS-Dateien inline laden
17         graphics_js = resources.read_text(__package__ + ".static",
18                                           "turtleGraphics.js")
19         buttons_js  = resources.read_text(__package__ + ".static",
20                                           "buttons.js")
21         code_control_js = resources.read_text(__package__ + ".static",
22                                               "codeControl.js")
23         display(Javascript(graphics_js + "\n\n" + buttons_js + "\n\n" +
24                             code_control_js))
25
26         # 4) Reset + Init + Draw in einem einzigen, garantiert nach dem
27           HTML-Inject laufenden Block
28         js = f"""
29         (function() {{
30             resetTurtle();           // leert turtleSegments, setzt
31             turtleX/Y/Angle

```

```

53     initButtonFunctions();    // bindet die Buttons
54     drawScene();             // rendert komplett neu ohne
                               Rest-Spuren
55     })());
56     """
57     display(Javascript(js))

```

Listing 8: Laden der statischen JavaScript-Dateien, zurücksetzen auf die Initialwerte, initialisieren der Buttons und rendern der Szene

Direkt im Anschluss wird mit den Befehlen `resetTurtle()`, `initButtonFunctions()` und `drawscene()` (Listing 8 Zeilen 53-55) die Turtle zurückgesetzt und die Standardwerte aus Listing 7 werden erneut gesetzt, die Buttons gesetzt und die Szene gerendert.

6.2.2 Die drawScene()-Funktion

Die Funktion `drawScene()` selbst führt folgende Abläufe in dieser Reihenfolge aus:

1. Hintergrund löschen und Himmel malen

Dadurch wird der Canvas zurückgesetzt und mit einem hellblauen Rechteck gefüllt.

```

1  <!-- Datei: pynbTurtle/static/turtleCanvas.html -->
148
149 <!-- 5) drawScene() -->
150 <script>
151     window.drawScene = function() {
152         // Himmel
153         ctx.clearRect(0,0,canvas.width,canvas.height);
154         ctx.fillStyle = "#87CEEB";
155         ctx.fillRect(0,0,canvas.width,canvas.height);
221     };
222 </script>

```

Listing 9: Einfärben des Canvas in hellblau

2. Pan- und Zoom-Transformation und falls aktiviert Gitternetz zeichnen

Mittels `ctx.save()` und `ctx.translate(panX, panY)` wird der Ursprung verschoben und somit die Zeichnung verschoben. Bei aktiviertem Grid wird darüber hinaus zentriert und skaliert (`ctx.scale(zoomLevel, zoomLevel)`), bevor das Gitternetz gezeichnet wird.

Innerhalb einer weiteren `ctx.save()-/ctx.restore()`-Blockstruktur wird basierend auf `gridSpacing` ein rechteckiges Liniennetz über die gesamte Zeichenfläche gelegt. Der Wert von `gridSpacing` ist konstant und ändert sich durch die Zoom-Funktion nicht, sprich beim Hineinzoomen werden die Linienabstände somit grösser.

Für das Legen des Gitters werden die sichtbaren Grenzen bestimmt und die Abstände der Linien entsprechend berechnet und anschliessend in horizontaler und vertikaler Richtung gezeichnet.

```

1  <!-- Datei: pynbTurtle/static/turtleCanvas.html -->
148
149 <!-- 5) drawScene() -->
150 <script>
151     window.drawScene = function() {
156
157         // Pan & Grid
158         ctx.save();
159         ctx.translate(panX, panY);
160         if (gridVisible) {

```

```

161     ctx.save();
162     ctx.translate(canvas.width/2, canvas.height/2);
163     ctx.scale(zoomLevel, zoomLevel);
164     ctx.translate(-canvas.width/2, -canvas.height/2);
165     ctx.beginPath();
166     ctx.strokeStyle = "#666";
167
168     // 1) Sichtbare Welt-Grenzen berechnen (invertiere die
169         Canvas-Transformation)
170     var halfW    = canvas.width / 2;
171     var halfH    = canvas.height / 2;
172     var invZ     = 1 / zoomLevel;
173     var wMinX    = (0 - panX - halfW) * invZ + halfW;
174     var wMaxX    = (canvas.width - panX - halfW) * invZ +
175         halfW;
176     var wMinY    = (0 - panY - halfH) * invZ + halfH;
177     var wMaxY    = (canvas.height - panY - halfH) * invZ +
178         halfH;
179
180     // 2) Auf gridSpacing runden
181     var xStart = Math.floor(wMinX / gridSpacing) *
182         gridSpacing;
183     var xEnd   = Math.ceil (wMaxX / gridSpacing) *
184         gridSpacing;
185     var yStart = Math.floor(wMinY / gridSpacing) *
186         gridSpacing;
187     var yEnd   = Math.ceil (wMaxY / gridSpacing) *
188         gridSpacing;
189
190     // 3) Vertikale Linien über die ganze Canvas-Höhe
191     for (var x = xStart; x <= xEnd; x += gridSpacing) {
192         ctx.moveTo(x, yStart);
193         ctx.lineTo(x, yEnd);
194     }
195
196     // 4) Horizontale Linien über die ganze Canvas-Breite
197     for (var y = yStart; y <= yEnd; y += gridSpacing) {
198         ctx.moveTo(xStart, y);
199         ctx.lineTo(xEnd, y);
200     }
201
202     ctx.stroke();
203     ctx.restore();
204 }
205 ctx.restore();
206 };
207 </script>

```

Listing 10: Einfärben des Canvas in hellblau

3. Liniensegmente einzeichnen

Erneut zentriert, skaliert und iteriert `drawScene()` über alle in `turtleSegments` gespeicherten Liniestücke. Für jedes Segment werden Pfadbefehle (`ctx.moveTo`, `ctx.lineTo`) ausgeführt und anschliessend mit `ctx.stroke()` in der zuletzt gesetzten `strokeStyle`-Farbe dargestellt.


```

1  <!-- Datei: pynbTurtle/static/turtleCanvas.html -->
148
149 <!-- 5) drawScene() -->
150 <script>
151     window.drawScene = function() {
152
153         // Pan & Grid
154         ctx.save();
155
156         // Linie aus turtleSegments
157         ctx.save();
158         ctx.translate(canvas.width/2, canvas.height/2);
159         ctx.scale(zoomLevel, zoomLevel);
160         ctx.translate(-canvas.width/2, -canvas.height/2);
161         turtleSegments.forEach(function(seg){
162             ctx.beginPath();
163             ctx.moveTo(seg.points[0].x, seg.points[0].y);
164             seg.points.slice(1).forEach(function(pt){
165                 ctx.lineTo(pt.x, pt.y); });
166             ctx.strokeStyle = seg.color;
167             ctx.lineWidth = 2;
168             ctx.stroke();
169         });
170         ctx.restore();
171         ctx.restore();
172     };
173 </script>

```

Listing 11: Liniensegmente einsetzen

4. Turtle-Icon setzen

Nachdem alle Linien gezeichnet sind, prüft `drawScene()` das Flag `turtleVisible`. Ist es wahr, so werden die Bildschirmkoordinaten der Turtle (aus `turtleX`, `turtleY`, `zoomLevel` und `Offsets`) berechnet und die Funktion `drawTurtle(tx, ty, turtleAngle)` aufgerufen. `drawTurtle` selbst speichert mit `ctx.save()`, verschiebt und rotiert das Koordinatensystem, zeichnet dann in der richtigen Reihenfolge Beine, Körper, Schwanz, Kopf und Augen und ruft zum Schluss `ctx.restore()` auf, um den ursprünglichen Zeichenzustand wiederherzustellen.

```

1  <!-- Datei: pynbTurtle/static/turtleCanvas.html -->
148
149 <!-- 5) drawScene() -->
150 <script>
151     window.drawScene = function() {
152
153         // Turtle-Icon
154         if (turtleVisible) {
155             var tx = (turtleX - 200)*zoomLevel + 200 + panX;
156             var ty = (turtleY - 200)*zoomLevel + 200 + panY;
157             drawTurtle(tx, ty, turtleAngle);
158         }
159     };
160 </script>



```


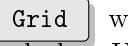
Listing 12: Turtle auf Canvas platzieren


Beim Aufruf von `drawScene()` entsteht eine vollständige Neurenderung der gesamten Szene, in der zunächst der Hintergrund, dann das Gitternetz, die gezeichneten Linien (sofern überhaupt Linien gezeichnet wurden) und schliesslich die Turtle-Grafik eingeblendet werden.





6.2.3 Die Steuerbuttons


Die Steuerbuttons werden komplett in `buttons.js` (auf den Abdruck einzelner Codeabschnitte wird in diesem Kapitel verzichtet) durch die Funktion `window.initButtonFunctions` dynamisch mit ihren jeweiligen Klick-Handlern verknüpft. Dabei sucht `initButtonFunctions` für jeden Button-Typ alle Elemente mit der entsprechenden ID-Präfix-Notation (`<prefix>_{{UID}}`) und hängt ihnen genau einen Event-Listener an.


Sobald der Nutzer auf die Buttons  oder  klickt, ruft der zugewiesene Handler die Funktion auf, die den globalen Wert `zoomLevel` mit 1.1 multipliziert oder durch 1.1 dividiert. Im Anschluss wird über einen erneuten Aufruf von `drawScene()` die gesamte Szene mit der neuen Skalierung gerendert. Dabei ist die Darstellung der Turtle von den Zoom-Buttons nicht betroffen und bleibt in jedem Zoomlevel gleich gross.

Mit den Buttons  und  werden die Boolean-Flags `turtleVisible` bzw. `gridVisible` jeweils umgeschaltet. Nach dem Umschalten startet ebenfalls `drawScene()`, sodass entweder die Schildkröte oder das Gitternetz (oder beides) sichtbar bzw. unsichtbar gemacht wird.

Der  setzt Pan-Offsets (`panX`, `panY`) so zurück, dass die gesamte Zeichnung mittig im Canvas steht. Dabei werden die Werte aus `turtleSegments` evaluiert und die entsprechende Mitte berechnet. Auch hier folgt unmittelbar ein `drawScene()`, um die Szene mit der neuen Verschiebung zu aktualisieren.

Die vier Pan-Buttons (, , , ) passen die Werte von `panY` bzw. `panX` um festgelegte Schritte an (z.B. jeweils ± 20 Pixel) und zeichnen anschliessend neu, sodass sich die Ansicht in die gewünschte Richtung verschiebt.

Beim Klick auf  wird mit Hilfe von `canvas.toDataURL("image/png")` ein Data-URL erzeugt, in ein unsichtbares `<a>`-Element als Download-Link (`download="turtle_drawing.png"`) eingefügt und programmatisch ausgelöst, sodass die aktuelle Canvas-Darstellung als PNG-Datei heruntergeladen wird.

Schliesslich leert der  alle gespeicherten Linien in `turtleSegments`, setzt Position und Winkel der Schildkröte auf die Startwerte zurück und – nachdem `drawScene()` aufgerufen wurde – ist der Canvas wieder in seinem Ausgangszustand.

Durch diese Verknüpfung von DOM-Elementen und JavaScript-Funktionen bietet `initButtonFunctions` eine vollständig interaktive UI-Steuerung, bei der jeder Klick unmittelbar in einer neuen Darstellung der Turtle-Szene resultiert.

6.2.4 Die Steuerung der Turtle

Die Steuerung der Turtle erfolgt primär über Methoden der Python-Klasse `Turtle` in `turtle.py`, die intern Zeichenbefehle in eine Warteschlange legen, und anschliessender Ausführung dieser Befehle mittels JavaScript.

1. Python-Seite (`turtle.py`)

Jedes Mal, wenn ein Nutzer in einer Notebook-Zelle einen Befehl wie `t.forward(50)`, `t.left(90)`, `t.penup()` oder `t.pendown()` aufruft, erzeugt die Methode einen entsprechenden Datensatz und fügt ihn der internen Liste `self._cmds` hinzu. Am Ende der Zelle ruft der Konstruktor automatisch `self._flush()` auf, wodurch über die Funktion `window._turtleFlush(cmds_list, delay)` alle gesammelten Befehle nebst gewünschtem Zeitintervall ins JavaScript-Frontend übertragen werden, ohne sie sofort auszuführen.

In Listing 13 wird die `forward`-Methode dargestellt. Analog ist auch die `backward`-Methode implementiert. Die Zeilen 70 und 71 sind für die Methode `showcode()`. Hier wird die aktuelle Zeilennummer des Codes ausgelesen und an JavaScript übertragen. In Zeile 72 wird überprüft, ob ein negativer Wert für `steps` eingegeben wurde. Fall ja, dann entspricht dies einer Rückwärtsbewegung, deshalb wird in diesem Fall `t.backward()` mit inversem Vorzeichen von `Steps` (Zeile 73) aufgerufen. In Zeilen 76-79 wird die Strecke von `t.forward()` in Teilstrecken zerlegt, damit die Turtle nicht von einem Punkt zum anderen springt, sondern sich flüssig über den Bildschirm bewegt. Mit `self._send_js` wird die Bewegung ebenfalls in JavaScript übertragen.

```

1  # Datei: pynbTurtle/turtle.py
11
12 class Turtle:
13
14     def forward(self, steps):
15         """
16         Bewege die Turtle schrittweise vorwärts.
17
18         t.forward(100)
19
20         """
21         lineno = inspect.currentframe().f_back.f_lineno
22         self._send_js(f"window.highlightLine('{self._uid}',
23             {lineno});")
24         if steps < 0:
25             self.backward(-steps)
26         else:
27             remaining = int(steps)
28             while remaining > 0:
29                 chunk = min(self._step_size, remaining)
30                 self._send_js(f"moveTurtle({chunk},
31                     {self.angle});")
32                 remaining -= chunk

```

Listing 13: Die `forward`-Methode: Die Bewegung wird in Teilstücke zerlegt, damit sich die Turtle flüssig über den Bildschirm bewegt.

In Listing 14 wird die `left`-Methode dargestellt. Die `right`-Methode ist analog. Die Drehung wird nicht in Teildrehungen unterteilt, sondern die Turtle dreht sich direkt um den entsprechenden Winkel. In Zeile 108 wird der neue Winkel gespeichert und sichergestellt, dass der Wert zwischen 0 und 360 Grad annimmt. Da eine Linksdrehung mit einem negativen Winkel einhergeht, wird in Zeile 109 der Wert mit negativem Vorzeichen an Javascript übergeben (Rechtsdrehung = positiv).

```

1  # Datei: pynbTurtle/turtle.py
11
12 class Turtle:
13
14     def left(self, deg):
15         """
16         Drehe die Turtle schrittweise nach links.
17
18         t.left(90)
19
20         """
21         lineno = inspect.currentframe().f_back.f_lineno
22         self._send_js(f"window.highlightLine('{self._uid}',
23             {lineno});")

```

```

108     self.angle = (self.angle - deg) % 360
109     self._send_js(f"rotateTurtle({-deg});")

```

Listing 14: Die *left*-Methode: Die Drehung erfolgt direkt ohne Unterteilung. Die neue Ausrichtung der Turtle wird gespeichert.

Hervorzuheben ist noch die *speed*-Methode. Hier kann der Nutzer optional die Geschwindigkeit der Bewegung der Turtle steuern. Über die Werte 1-10 wird der Wert von *command_delay* gesteuert. Dieser kann auch über bei der Initialisierung der Turtle (z.B. *t = Turtle(200)*) gesetzt werden, sollte aber über diese Methode gemacht werden, da dies intuitiver erscheint.

```

1  # Datei: pynbTurtle/turtle.py
11
12 class Turtle:
167     def speed(self, s):
168         """
169         Setzt Animationstempo 1-10 (1 = langsam, 10 = schnell)
170
171         t.speed(5)
172
173         """
174         s = max(1, min(10, int(s)))
175         # mappe linear auf delay-Bereich [200..2]
176         self._step_size = s
177         max_d, min_d = 200, 2
178         span = max_d - min_d
179         self.command_delay = int(max_d - (s - 1) * span / 9)

```

Listing 15: Die *speed*-Methode: Hier wird die Bewegungsgeschwindigkeit der Turtle festgelegt: 1=langsam, 10 = schnell

Die restlichen Steuermethoden *penup*, *pendown*, *pencolor*, *setposition* werden in *turtle.py* in analoger Weise implementiert.

Die *flush*-Methode nimmt alle bislang gesammelten Zeichenbefehle der aktuellen Turtle-Instanz und überträgt sie gebündelt an JavaScript, wo sie ausgeführt werden.

2. JavaScript-Seite (*codeControl.js*)

Im Browser nimmt *window._turtleFlush* die Liste *cmds_list* und den Parameter *delay* und schiebt sie in das globale Array *window._turtleCommands*. Die eigentliche Abarbeitung erfolgt sofort, sofern nicht in *showcode* pausiert wird.

```

1  // Datei: pynbTurtle/static/codeControl.js
23
24 // 3) abspielen der Turtle-Befehle
25 window._turtleFlush = function(cmds_list, delay) {
26     window._turtleCommands = cmds_list.slice();
27     window._turtleCommandDelay = delay;
28     window._turtlePlay();
29 };

```

Listing 16: Das Abspielen der Turtle-Befehle in *codeControl.js*

3. Canvas-Rendering (*turtleCanvas.html* & *turtleGraphics.js*)

Die Canvas-Funktionen wie *moveTurtle(x,y)*, *rotateTurtle(angle)*, *penUp()*, *penDown()* (vgl. Listing 17 ändern den globalen Zustand (*turtleX*, *turtleY*, *turtleAngle*, *penIsDown* usw.) und fügen neue Liniensegmente zu *turtleSegments* hinzu. Anschliessend sorgt *drawScene()* (siehe

Abschnitt 6.2.2) dafür, dass alle hinterlegten Segmente in der richtigen Reihenfolge gezeichnet, das Gitter optional gerendert und zuletzt das Turtle-Icon an der aktuellen Position mit korrekter Rotation dargestellt wird.

```

1  <!-- Datei: pynbTurtle/static/turtleCanvas.html -->
123
124 <!-- 4) Turtle-Kommandos inline -->
125 <script>
126     window.moveTurtle      = function(dist, angle) {
127         var rad = angle * Math.PI/180;
128         var newX = turtleX + Math.cos(rad)*dist;
129         var newY = turtleY + Math.sin(rad)*dist;
130         if (penIsDown) {
131             turtleSegments.push({
132                 points: [{x: turtleX, y: turtleY}, {x: newX, y: newY}],
133                 color: strokeColor
134             });
135         }
136         turtleX = newX; turtleY = newY;
137     };
138     window.rotateTurtle    = function(deg) { turtleAngle += deg; };
139     window.resetTurtle     = function() {
140         turtleX = 200; turtleY = 200; turtleAngle = -90;
141         turtleSegments = [];
142     };
143     window.penup           = function() { penIsDown = false; };
144     window.pendown         = function() { penIsDown = true; };
145     window.setPosition     = function(x,y) { turtleX = x; turtleY =
146         y; };
146     window.setStrokeColor = function(c) { strokeColor = c; };
147 </script>

```

Listing 17: Canvas-Rendering in *turtleCanvas.html*

Durch dieses Zusammenspiel aus Python-Methoden, Befehls-Queue und JavaScript-Steuerung entsteht eine interaktive Steuerung der Turtle, bei der Nutzer sowohl gezeichnete Linien als auch animierte Bewegungen in beliebiger Geschwindigkeit im Jupyter Notebook sehen können.

6.3 Die showcode-Implementierung in *turtle.py*

Die Methode `showcode` (Listing 18) in der Klasse `Turtle` stellt den zuletzt ausgeführten Notebook-Code neben dem Zeichen-Canvas dar, hebt die aktuell ausgeführte Zeile hervor und fügt Steuerungs-Buttons (`Stop`, `Step`, `Play`) hinzu. Mit dem Parameter `mode` lassen sich die Zustände 'play' (Standard) und 'pause' einstellen.

```

1  # Datei: pynbTurtle/turtle.py
11
12 class Turtle:
208     def showcode(self, mode='play'):
209         """
210         Stelle die zuletzt ausgeführte Notebook-Zelle als Code-Block
211         dar und
212         hebe den aktuellen Befehl hervor. Mit 'pause' wird die
213         Animation pausiert,
214         mit 'play' läuft sie weiter.
215
216         t.showcode('pause') oder t.showcode('play') oder t.showcode()

```

```

216     """
217     self.speed(1)
218
219     if mode == 'pause':
220         js_mode_cmd = "window._turtleStop();"
221     elif mode == 'play':
222         js_mode_cmd = "window._turtlePlay();"
223
224     # Lade codeControl.js erneut
225     code_control_js = resources.read_text(__package__ + ".static",
226                                           "codeControl.js")
227     display(Javascript(code_control_js))
228
229     if self._history:
230         cmds = ', '.join(repr(c) for c in self._history)
231         js_flush = f"window._turtleFlush([{cmds}],
232                               {self.command_delay});"
233         display(Javascript(js_flush))
234
235     # In[]-History holen (letzte Zelle)
236     try:
237         shell = get_ipython()
238         history = shell.user_ns.get('In', [])
239         raw = history[-1] if history else ''
240     except NameError:
241         raw = ''
242
243     # HTML-escape und split in Zeilen
244     raw = raw.replace('\r\n', '\n').replace('\r', '\n')
245     escaped = html.escape(raw)
246
247     # JS-Aufruf: Code block + Controls initialisieren
248     js = f"initTurtleCodeBlock('{self._uid}', '{escaped}');"
249     js += f"initTurtleCodeControls('{self._uid}',
250                                   {self.command_delay});"
251     js += js_mode_cmd
252     display(Javascript(js))

```

Listing 18: Die `showcode`-Methode: Beim Aufruf der Methode wird der gesamte Code-Block einer Notebook-Zelle neben dem Zeichen-Canvas dargestellt und die aktuell ausgeführte Zeile hervorgehoben

Im Einzelnen läuft sie folgendermassen ab (vgl. Listing 18):

1. Temporäre Geschwindigkeit einstellen (Zeile 217):
Zu Beginn wird das Animationstempo auf den langsamsten Wert (Speed 1) gesetzt, um die Code-Darstellung schrittweise nachvollziehen zu können.
2. Modus umschalten (Zeilen 219-222):
Je nach übergebenem Parameter `mode` ('pause' oder 'play') wird die Turtle entweder angehalten (`window._turtleStop()`) oder gestartet (`window._turtlePlay()`). Wird kein Parameter angegeben, läuft die Animation weiter (play).
3. JS-Steuerung laden (Zeilen 225-226):
Das Frontend-Skript `codeControl.js` wird erneut geladen und ins Notebook eingefügt. Dadurch stehen die Funktionen zum Aufbau des Code-Blocks und der Buttons zur Verfügung.

4. Bisherige Turtle-Befehle anzeigen Zeilen (Zeilen 228-231):
Befinden sich bereits ausgeführte Befehle in der internen Historie (`self._history`), so werden diese per JavaScript-Aufruf `window._turtleFlush(..., delay)` sofort im Code-Block vorbereitet und direkt abgespielt.
5. Quelltext der letzten Zelle auslesen (Zeilen 234-243):
Über den IPython-Hook wird der rohe Python-Code der zuletzt ausgeführten Notebook-Zelle aus `In[-1]` bezogen. Er wird HTML-escaped und in einzelne Zeilen aufgesplittet, um später im `<pre>`-Block dargestellt zu werden.
6. Code-Block und Controls initialisieren (Zeilen 245-248):
Mit den beiden Funktionen

```
initTurtleCodeBlock(uid, raw) und initTurtleCodeControls(uid, delay)
```

aus `codeControl.js` wird der formatierte Code zusammen mit den Steuerungs-Buttons in den dafür vorgesehenen Container im Notebook-DOM eingefügt. Anschliessend erfolgt ein initialer Aufruf von `window._turtlePlay()` oder `window._turtleStop()`, um den Ausgangszustand zu setzen.

6.4 Die showcode-Implementierung in `codeControl.js`

In `codeControl.js` finden sich fünf zentrale Bestandteile, die gemeinsam die Anzeige und Steuerung des im Notebook ausgeführten Turtle-Codes ermöglichen. Die Funktionen werden durch den Aufruf von `t.showcode()` aktiviert (siehe auch Abschnitt 6.3).

1. Globale Variablen:
Ganz oben werden drei Fenster-Variablen angelegt, die eine zentrale Befehls-Queue und das Timing der Animation verwalten:
 - `window._turtleCommands` enthält die auszuführenden JavaScript-Befehle.
 - `window._turtleInterval` hält die ID des aktuellen `setInterval`, um Animationen starten und stoppen zu können.
 - `window._turtleCommandDelay` speichert die Millisekunden-Pause zwischen zwei einzelnen Turtle-Schritten.
2. Zeilenhervorhebung
Mit `window.highlightLine(uid, n)` wird jeweils eine bestimmte Codezeile im rechts angezeigten Code-Block gelb unterlegt. Dabei werden zunächst werden alle bisherigen Hervorhebungen entfernt. Dann sucht das Script das `` im Container mit der gegebenen `uid` und fügt diesem die CSS-Klasse `highlight` (siehe dazu `turtleCanvas.html`) hinzu.
3. Queue-Initialisierung (`_turtleFlush`)
Die Methode `window._turtleFlush(cmds_list, delay)` wird aus Python heraus aufgerufen, sobald eine Zelle ausgeführt ist. Sie kopiert die übergebenen Befehle in `window._turtleCommands` und setzt `window._turtleCommandDelay`. Der eigentliche Start der Animation erfolgt erst, wenn später `window._turtlePlay()` aufgerufen wird.
4. Animation steuern: Play, Stop, Step
 - Play (`window._turtlePlay()`): Legt mit `setInterval` eine Schleife an, die in jeder Runde den nächsten Befehl aus der Queue holt, per `eval()` ausführt und dann `drawScene()` aufruft. Sobald keine Befehle mehr übrig sind, wird das Intervall automatisch gelöscht.

- `Stop (window._turtleStop())`: Bricht bei laufendem Intervall die Wiederholungen ab und setzt `window._turtleInterval` zurück.
- `Step (window._turtleStep())`: Ruft zuerst `window._turtleStop()` auf, um sicherzustellen, dass keine automatische Schleife läuft, und führt dann genau einen Befehl aus der Queue aus, gefolgt von einem `drawScene()`-Aufruf.

5. Erzeugen der Steuer-Buttons

Die Funktion `window.initTurtleCodeControls(uid, commandDelay)` baut oberhalb des Code-Blocks drei Buttons (`Stop`, `Step`, `Play`) zusammen, verknüpft ihre Klick-Events mit den oben beschriebenen Funktionen und ruft direkt danach `window._turtleStop()`, um eventuell noch laufende Animationen zu unterbrechen. So lassen sich Anwenderinteraktivität und Automations-Modus nahtlos kombinieren.

Zusammen ermöglichen diese Routinen, dass nach Ausführung einer Notebook-Zelle nicht nur die Zeichnungsbefehle ans Canvas geschickt, sondern auch der entsprechende Quellcode daneben angezeigt, Zeile für Zeile hervorgehoben und sowohl automatisch abgespielt als auch schrittweise durchgegangen werden kann.

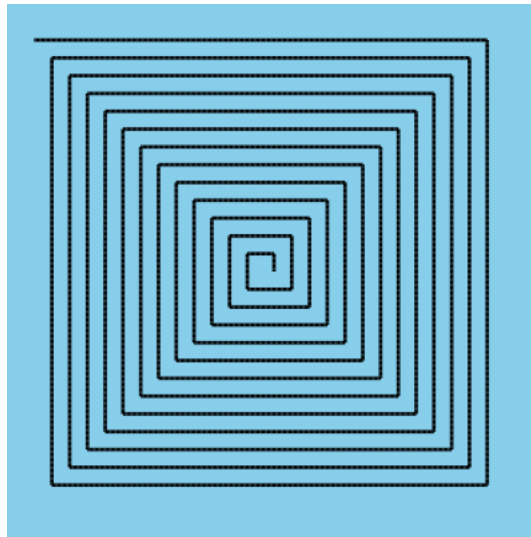


Abb. 6: Mit *pynbTurtle* erstellte Spirale.

7 Abhängigkeiten und Performance

In diesem Abschnitt werden zunächst die externen Pakete und Systemvoraussetzungen aufgeführt, die für den Betrieb von `pynbTurtle` benötigt werden. Anschliessend wird auf Performance-Eigenschaften und -Optimierungsmöglichkeiten eingegangen.

7.1 Abhängigkeiten

- **Python [4]:** Das Paket benötigt Python in Version ≥ 3.8 . Dies ist in `pyproject.toml` unter `requires-python = ">=3.8"` definiert.
- **IPython [6]:** Zur Anzeige von HTML- und JavaScript-Widgets in Jupyter Notebooks ist mindestens IPython 9.0 erforderlich. Im Build-System ist dies als `IPython>=9.0` hinterlegt. Zudem wird auf die Bibliothek `IPython.display` (HTML, Javascript, display) für das Rendering von Canvas und Steuer-Buttons zugegriffen.
- **Standardbibliothek:**
 - `uuid` [7], `inspect` [8], `html` [9] für interne Identifier-Erzeugung, Quellcode-Inspektion und HTML-Escaping.
 - `importlib.resources` [10] zur Einbettung der statischen Dateien (`.html`, `.js`) ins Notebook.

7.2 Performance

Befehls-Dispatch

Jeder Turtle-Befehl (z.B. `forward()`, `left()`) wird als JavaScript-String in eine Queue eingereiht. Standardmässig ist in der `Turtle`-Klasse ein `step_size` von 1 Pixel definiert, sodass für eine Vorwärtsbewegung von 100 Pixeln hundert Einzelschritte erzeugt werden. Dies kann bei grossen Zeichnungen zu einer langen Ausführungszeit führen.

Verzögerung zwischen Schritten

Das Attribut `command_delay` (Standard: 50 ms) steuert das Intervall, in dem die Befehle aus der Queue geholt und gezeichnet werden. Ein kleinerer Wert erhöht die Zeichengeschwindigkeit, führt aber zu höherer CPU-Last und kann in Notebook-Umgebungen zu Verzögerungen im UI führen.

```
1 from pynbTurtle import Turtle
2
3 # Grössere Schritte und schnellere Abarbeitung:
4 t = Turtle(command_delay=20)
5 t._step_size = 5
6 t.pen_down()
7 t.forward(200)
```

Listing 19: Anpassen von `step_size` und `command_delay`

Batching und Flush

Am Ende jeder Zelle oder nach manueller Änderung wird durch `t.flush()` die gesamte Queue auf einmal an die JavaScript-Umgebung übergeben. Durch diese Bündelung verringert sich der Overhead einzelner Python-zu-JS-Übergaben.

Rendering im Canvas

Die JavaScript-Funktion `drawScene()` rendert das gesamte Canvas bei jedem Schritt neu. Bei komplexen Szenen mit vielen Segmenten kann dies spürbar langsam werden. In solchen Fällen empfiehlt es sich,

- das Canvas vorübergehend auszublenden oder `drawScene()` nur nach jedem `n`-ten Schritt aufzurufen,
- den Zoom-/Pan-Modus zu minimieren, um Redraw-Overhead zu senken.

Empfehlung

Für interaktive Lehr- und Lernzwecke ist die voreingestellte Verzögerung und Schrittgrösse meist ausreichend. Bei automatisierten oder grossflächigen Zeichnungen lohnt es sich, `_step_size` und `command_delay` zu erhöhen, um die Performance zu optimieren.

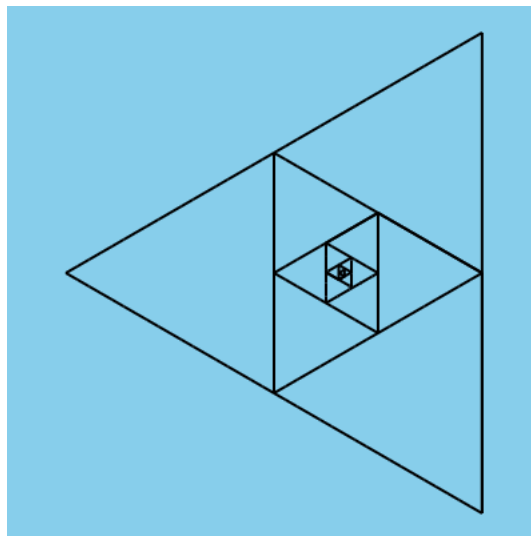


Abb. 7: Mit *pynbTurtle* erstelltes Dreiecksmuster.

8 Nutzung

In diesem Kapitel wird gezeigt, wie das `pynbTurtle`-Paket in einem Jupyter Notebook eingebunden und verwendet wird. Anhand von Codebeispielen wird gezeigt, wie eine Turtle-Instanz erstellt, Befehle abgesetzt und eine Zeichnung gestartet wird.

8.1 Import und Initialisierung

Nach der Installation (siehe Kapitel 4) kann die Turtle Klasse wie in Listing 20 importiert und initialisiert werden.

```
1 from pynbTurtle import Turtle
2
3 # Erzeugt eine neue Turtle mit Standard-Verzögerung (100 ms)
4 t = Turtle()
```

Listing 20: Import des Moduls und Initialisierung einer Turtle-Instanz

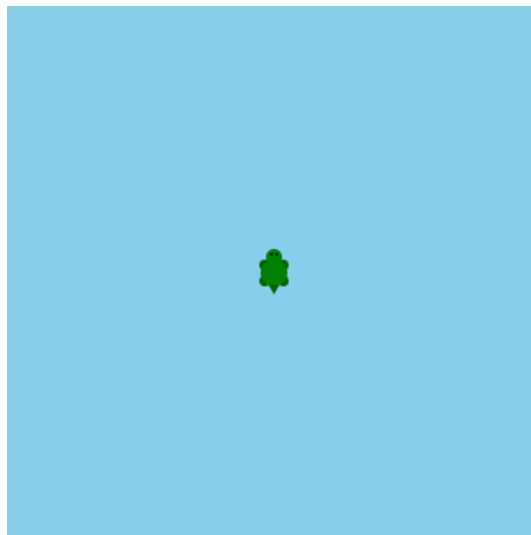


Abb. 8: Der Canvas nach der Initialisierung in einem Jupyter Notebook.

8.2 Beispiele

Alle aufgeführten Beispiele lassen sich auch in der Datei `example.ipynb` im Ordner `example` unter <https://github.com/screto/pynbTurtle-master.git> einsehen.

Beispiel 1

Im folgenden Beispieldokument wird in einer Jupyter Notebook-Zelle der folgende Code ausgeführt. Dabei wurden alle Methoden (ausser `showcode()`) der Turtle Klasse verwendet.

```
1 from pynbTurtle import Turtle
2
3 t = Turtle()
4 t.speed(10) # Setze die Geschwindigkeit auf 10 (schnellste)
5 t.forward(100) # Bewege die Turtle 100 Einheiten vorwärts
6 t.left(90) # Drehe die Turtle um 90 Grad nach links
7 t.backward(50) # Bewege die Turtle 50 Einheiten rückwärts
8 t.right(60) # Drehe die Turtle um 60 Grad nach rechts
9 t.penup() # Hebe den Stift an, um keine Linie zu zeichnen
10 t.forward(200) # Bewege die Turtle 200 Einheiten vorwärts
11 t.pendown() # Setze den Stift ab, um eine Linie zu zeichnen
12 t.pencolor("yellow") # Setze die Stiftfarbe auf rot
```

```

13 t.backward(100) # Bewege die Turtle 100 Einheiten rückwärts
14 t.left(-90) # Drehe die Turtle um -90 Grad nach links -> Rechtsdrehung
15 t.pencolor("green") # Setze die Stiftfarbe auf rot
16 t.backward(50) # Bewege die Turtle 50 Einheiten rückwärts
17 t.setposition(200,200) # Verschiebe die Turtle an die Position (200,200)

```

Listing 21: Beispielcode, der alle Methoden der Turtle-Klasse enthält

In Abbildung 9 ist links die Ausgabe im Canvas zu sehen. Hier wurde das Gitter eingeschaltet um die Bewegung besser nachvollziehen zu können. Weiter fällt auf, dass die Turtle den Canvas verlässt. In Abbildung 9 wurde rechts mittels der Zentrier- und Zoom-Buttons das Bild in das Zentrum des Canvas gerückt. Hier ist deutlich zu erkennen, dass der Abstand zwischen den Gitternetzlinien konstant bleibt. Weiter ist auch zu sehen, dass der Befehl `setposition(x,y)` eine Verschiebung der Turtle ohne zeichnen (`penup`) erfolgt und die Ausrichtung der Turtle beibehalten wird. Beide Bilder wurden mit dem Export-Button exportiert.

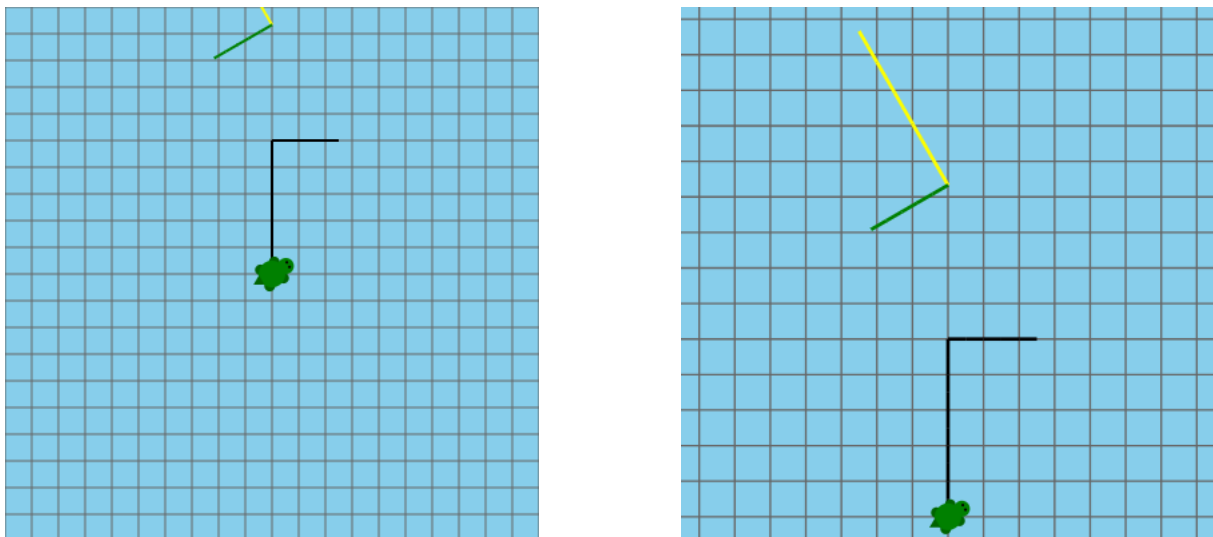


Abb. 9: *links:* Das aus Listing 21 erzeugte Bild mit gezeichnetem Skript. Dabei ist zu sehen, dass nicht alles im Canvas zu sehen ist. *rechts:* Das Bild wurde nun mittels Zentrier- und Zoom-Button entsprechend verschoben.

Beispiel 2

Im nachfolgenden Code wird der Befehl `circle(steps)` definiert. Nach der Initialisierung der Turtle wird `showcode('play')` aufgerufen. Damit wird neben dem Zeichen-Canvas der Code dargestellt. Durch `'play'` wird die Zelle automatisch geflusht und die Animation gestartet. Beim Aufruf von `circle(2)` werden die Zeilen 5 und 6 im Wechsel hervorgehoben, bis der Kreis vollständig gezeichnet ist. Anschliessend werden die Zeilen 12-14 hervorgehoben und mit dem Aufruf von `circle(1)` geht es wieder zu den Zeilen 5 und 6. Am Schluss bleibt die Hervorhebung bei Zeile 6 stehen, da `t.right(1)` die letzte Bewegung der Turtle ist (vgl. Abbildung 10).

```

1 from pynbTurtle import Turtle
2
3 def circle(steps): # Hier wird der Befehl für einen Kreis festgelegt
4     for i in range(360):
5         t.forward(steps) # Bewege die Turtle um steps Einheiten vorwärts
6         t.right(1) # Drehe die Turtle um 90 Grad nach rechts
7
8 t = Turtle()
9 t.speed(10)
10 t.showcode('play') # showcode wird im Modus 'play' aufgerufen
11 circle(2) # Zeichne einen Kreis mit Schrittgrösse 2

```

```

12 t.left(90) # Drehe die Turtle um 90 Grad nach links
13 t.forward(100) # Bewege die Turtle 100 Einheiten vorwärts
14 t.left(90) # Drehe die Turtle um 90 Grad nach links
15 circle(1) # Zeichne einen Kreis mit Schrittgrösse 1

```

Listing 22: Beispielcode, der die Methode `showcode('play')` aufruft.

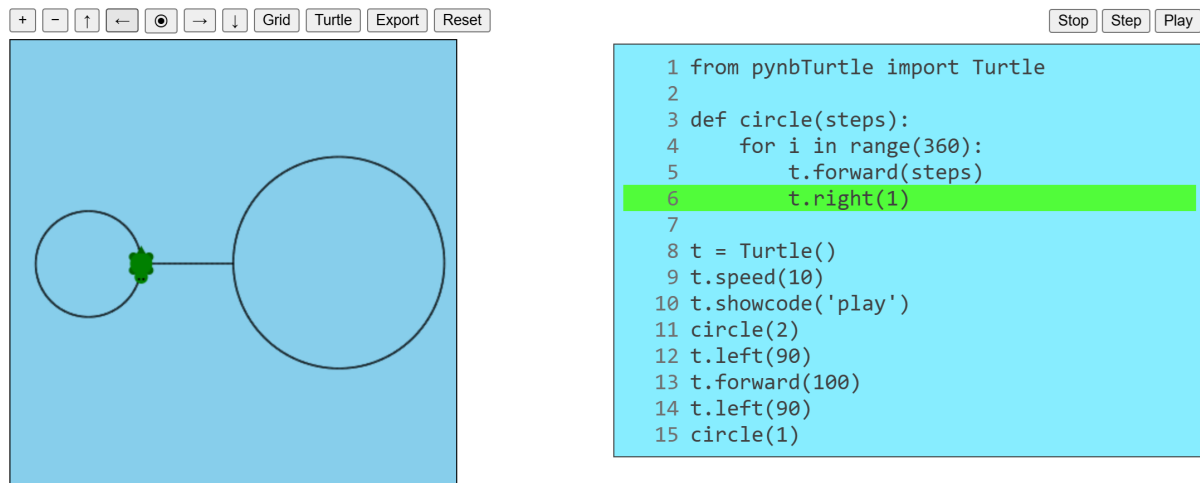


Abb. 10: Screenshot der in Listing 22 erzeugten Ausgabe. Der Zeichen-Canvas wurde durch Rauszoomen bearbeitet. Die Codezeile 6 wird hervorgehoben, da dies der letzte Befehl ist, den die Turtle ausübt.

Beispiel 3

Mite dem nachfolgenden Code werden verschiedenfarbige Quadrate unterschiedlicher Grösse gezeichnet. Die Länge des Codes ist deutlich grösser als im Code-Block angezeigt werden kann. Sobald dies eintritt, wird der Code-Block scrollbar (vgl. Abbildung 11).

```

1 from pynbTurtle import Turtle
2
3 def quadrat(seite, farbe):
4     t.pencolor(farbe)
5     for i in range(4):
6         t.forward(seite)
7         t.right(90)
8
9 def fenster(seite, farbe):
10    for i in range(4):
11        quadrat(seite, farbe)
12        t.right(90)
13
14 def mosaik(seite, farbe1, farbe2):
15    fenster(2*seite, farbe1)
16    t.forward(seite)
17    t.left(90)
18    t.forward(seite)
19    fenster(seite, farbe2)
20    t.backward(seite)
21    t.right(90)
22    t.backward(2*seite)
23    t.right(90)
24    t.forward(seite)
25    fenster(seite, farbe2)
26

```

```

27 t=Turtle()
28 t.speed(10)
29 t.showcode()
30 mosaik(80, "blue", "red")

```

Listing 23: Beispielcode, der die Methode `showcode('play')` aufruft.

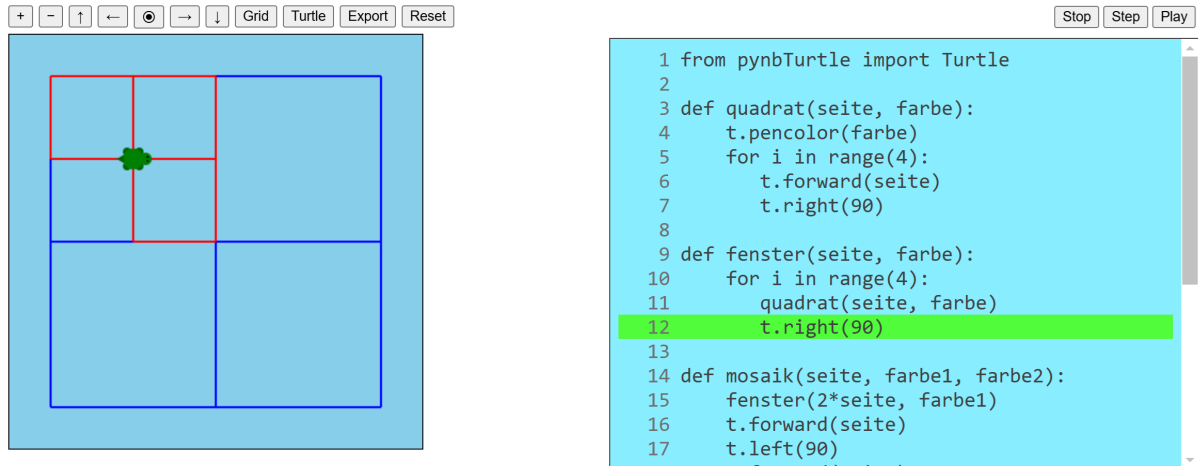


Abb. 11: Screenshot der in Listing 23 erzeugten Ausgabe. Die Animation wurde innerhalb der `for`-Schleife im Befehl `fenster(seite, farbe)` gestoppt. Der Code-Block wird aufgrund der Codelänge scrollbar, so dass die Zeilen unterhalb von Zeile 17 sichtbar werden.

9 Tutorial

Im Ordner `example` auf <https://github.com/screto/pynbTurtle-master.git> befindet sich die Datei `pynbTurtle-Tutorial.iypnb`. Hierbei handelt es sich um ein Jupyter Notebook, dass ein Tutorial zum Paket anbietet.

Das Tutorial ist wie folgt aufgebaut:

1. Erzeugen einer Turtle-Instanz
2. Einfache Zeichenbefehle
3. Die Turtle verlässt den Bereich
4. Farbige Bilder
5. Anpassung der Geschwindigkeit und Abbruch
6. Die Ausführung des Codes beobachten

In diesem Tutorial geht es keineswegs um das Erlernen des Programmierens, sondern vielmehr um das Kennenlernen der verschiedenen Befehle. Der Nutzer soll durch Aufträge dazu angeregt werden selber auszuprobieren und zu beobachten, was die Befehle machen.

10 Fazit und Ausblick

Die vorliegende Arbeit zeigt, dass sich mit dem **pynbTurtle**-Paket eine intuitive und leistungsfähige Umgebung zur Visualisierung von Turtle-Grafiken in Jupyter Notebooks realisieren lässt. Eine Vorversion wurde mit Schülerinnen und Schülern getestet, die bereits Erfahrung mit dem Paket *mobilechelonian* hatten. Auf Basis ihres Feedbacks wurden verschiedene optische Anpassungen vorgenommen (schlechte Sichtbarkeit im Darkmode). Insgesamt fiel das Echo jedoch durchwegs positiv aus. Besonders gelobt wurden die deutlich schnellere Zeichengeschwindigkeit und die wegfallende Begrenzung der Zeichenfläche.

Das Paket arbeitet derzeit noch nicht in allen Situationen zuverlässig – hier besteht weiterer Untersuchungsbedarf. Insbesondere die Funktionen der Stop-, Step- und Play-Buttons funktionieren noch nicht fehlerfrei und sollen in einem nächsten Update überarbeitet werden. Sobald diese Probleme behoben wurden, wird eine Veröffentlichung auf <https://pypi.org/> angestrebt. Weitere mögliche Verbesserungen umfassen:

- Eine Option, den animierten Zeichenprozess zu überspringen, sodass das Endresultat sofort dargestellt wird.
- Die Implementierung zusätzlicher Methoden. Z.B. `repeat()` wie sie in Tigerjython verwendet wird.
- Die Möglichkeit der farblichen Anpassung der Turtle und des Hintergrunds.
- Die Möglichkeit die Stiftstärke einzustellen und umrahmte Bereiche einzufärben.
- Bessere Sichtbarmachung der Sprünge innerhalb eines Programms.
- Anzeige des aktuellen Durchlaufs in einer `for`-Schleife.
- Die Möglichkeit der Erstellung einer Bildfolge als Kurzfilm (setzt das Überspringen des Zeichnungsprozesses voraus).

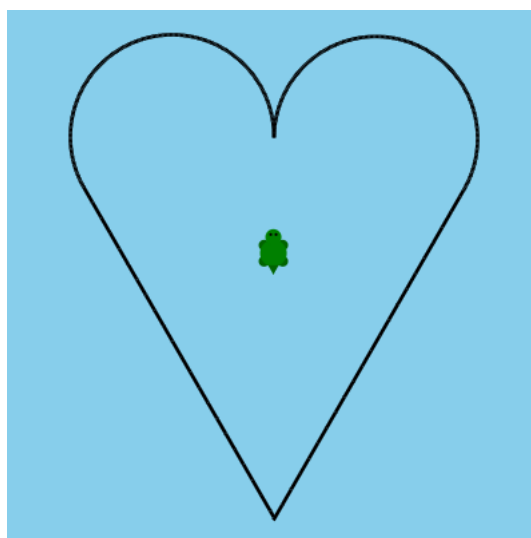


Abb. 12: Mit *pynbTurtle* erstelltes Herz.

Literatur

- [1] Visual Studio Code - Code Editing. Redefined. URL <https://code.visualstudio.com/>.
- [2] Thomas Kluyver. Turtles in the jupyter notebook, 2018. URL <https://pypi.org/project/mobilechelonian/>.
- [3] Werner Wiater. *Unterrichtsprinzipien. 7. Auflage, Neubearbeitung*. Prüfungswissen, Basiswissen Schulpädagogik; Didaktik. Auer, Augsburg, 2018. ISBN 978-3-403-03617-3. Type: gedruckt.
- [4] Welcome to Python.org, June 2025. URL <https://www.python.org/>.
- [5] Packaging Python Projects - Python Packaging User Guide. URL <https://packaging.python.org/en/latest/tutorials/packaging-projects/>.
- [6] Jupyter and the future of IPython — IPython. URL <https://ipython.org/>.
- [7] uuid — UUID objects according to RFC 4122. URL <https://docs.python.org/3/library/uuid.html>.
- [8] inspect — Inspect live objects. URL <https://docs.python.org/3/library/inspect.html>.
- [9] html — HyperText Markup Language support. URL <https://docs.python.org/3/library/html.html>.
- [10] importlib.resources – Package resource reading, opening and access. URL <https://docs.python.org/3/library/importlib.resources.html>.

Abbildungsverzeichnis

Alle hier verwendeten Abbildungen wurden selbst erzeugt und können in der Datei `example.ipynb` nachvollzogen werden.