Python im Web

Python im Web

Dynamisches HTML im Browser mit PyScript statt JavaScript

Browser führen nur JavaScript aus? Nicht mehr! PyScript tritt als Alternative zu JavaScript auf. Wir erklären, wie das möglich ist, und programmieren als Beispiel ein Spiel mit der neuen Technik.

Von Pina Merkert

kompakt

- PyScript nutzt die maschinennahe Sprache WebAssembly, um Python mit der JavaScript-Engine eines Browsers auszuführen.
- Über PyScript-Funktionen kann der Python-Code auf das DOM zugreifen, was dynamisches HTML ermöglicht.
- PyScript kann JavaScript ersetzen, es lädt aber wesentlich langsamer.

Beim Versuch, auf eine Objekteigenschaft zuzugreifen, wird die Eigenschaft nicht nur in dem Objekt selbst, sondern auch in seinem Prototyp und dem Prototyp des Prototyps gesucht. Wenn Ihnen Sätze wie dieser auch rätselhaft vorkommen, ist JavaScript wohl auch nicht Ihre Muttersprache. Python ist da oft zugänglicher, der Code hat weniger Zeilen und das Sprachdesign ist auf Lesbarkeit optimiert. Nur leider muss Python immer lokal installiert sein, die allgegenwärtigen Webbrowser verarbeiten nur JavaScript. Oder etwa nicht?

Dass Python nicht im Browser läuft, stimmt nicht mehr: Mit einer trickreichen Software namens „Pyodide“ interpretieren Webseiten auch Python-Code. Python-Entwickler können auf diesem Weg JavaScript durch Python ersetzen. Sie programmieren dann in Python mit PyScript-Funktionen statt in JavaScript. Wir zeigen, wie Sie mit PyScript loslegen.

Als Beispiel haben wir uns von dem Worträtsel „Wordle“ inspirieren lassen und ein Rätsel für Nerds mit dem Namen „Nerdle“ programmiert. Die Spielregeln sind einfach: PyScript wählt aus einer langen Liste mit Begriffen aus der Technikwelt mit fünf Zeichen (Nerdle ist schwieriger als Wordle, weil unser Rätsel Wörter mit Ziffern und Sonderzeichen enthält) einen zufälligen aus, den Sie erraten müssen. Dafür tippen Sie über die Bildschirmtastatur zunächst einen Begriff. Jeder geratene Begriff muss in der Liste stehen, damit das Spiel die Eingabe akzeptiert. Wenn ein Zeichen des geratenen Begriffs an der gleichen Stelle steht wie im gesuchten Begriff, wird es grün. Kommt ein Zeichen irgendwo im gesuchten Begriff vor, wird es gelb. Zeichen, die gar nicht vorkommen, werden grau. Die Tasten der Bildschirmtastatur verfärben sich genauso. Diese Farben geben Hinweise für den nächsten Begriff, sodass Sie mit zusätzlichen Versuchen immer mehr über den gesuchten Begriff erfahren. Wenn Sie spätestens beim sechsten Versuch richtig raten, gewinnen Sie das Spiel. Wenn Sie zu oft falsch raten, sollten Sie mehr c't lesen *zwinkersmiley*. Ohne selbst zu programmieren, können Sie das Spiel sofort unter nerdle.pinae.net ausprobieren; den Code finden Sie als Open Source (GPLv3) über ct.de/ynk9.

WebAssembly

Browser integrieren weiterhin keinen Python-Interpreter. Sie führen aber in ihrer virtuellen Maschine schon länger WebAssembly aus. WebAssembly ist eine maschinennahe Sprache,

die der Browser innerhalb von Millisekunden in Maschinencode übersetzt. Das funktioniert so gut, dass WebAssembly meist nur wenige Prozent langsamer läuft als gleicher Code, den ein Compiler direkt in Maschinencode übersetzt hat. Da der Code aber in der Browser-Sandbox läuft, ist nicht jedes Programm WebAssembly-tauglich. Beispielsweise verbieten Browser direkten Zugriff aufs Dateisystem oder Hardware wie Netzwerkkarten.

Das Pyodide-Projekt hat die Python-Referenzimplementierung CPython so modifiziert, dass sie nach WebAssembly kompiliert. Damit läuft Python zwar im Browser, man sieht davon aber noch nichts. An dieser Stelle kommt PyScript ins Spiel: PyScript bringt Funktionen mit, um vom Browser-Python auf den DOM-Tree zuzugreifen, also auf die HTML-Struktur der Webseite. Außerdem stellt es eigene HTML-Tags bereit, die den Code aufnehmen, Module nachladen und Eingabefelder bereitstellen. Das Projekt steht noch am Anfang: Release-Nummern sind im Datumsformat, die Versionen auf GitHub als „Pre-release" getaggt. Manchen Funktionen sieht man den frühen Entwicklungsstand noch an. Beispielsweise muss man per Hand Funktionen einkapseln, um sie ins Ereignissystem von JavaScript einzuklinken. Trotzdem funktioniert der Code bereits gut genug für ein Browserspiel.

Einbinden

PyScript bindet man wie ein JavaScript-Framework ein, indem man zwei Tags im <head> der Webseite ergänzt:

```
<link rel="stylesheet"
    href="https://pyscript.net/alpha/pyscript.css" />
<script defer
src="https://pyscript.net/alpha/pyscript.js"></script>
```

Den Code lädt der Browser dann aus dem Content Delivery Network (CDN) der PyScript-Entwickler, man bekommt also immer die aktuellste Version. Um Probleme bei Updates zu umgehen, könnte man PyScript auch selbst übersetzen und hosten. Momentan raten wir aber noch davon ab, PyScript produktiv

einzusetzen, weil sich in den kommenden Monaten sicherlich noch einiges an den Funktionssignaturen ändern kann.

Danach kann man einfach irgendwo im HTML der Seite den <py-script>-Tag einfügen und in dem Python-Code platzieren. Für ein Hallo-Welt-Programm braucht man nur drei Zeilen:

```
<py-script>
  print("Hallo Welt.")
</py-script>
```

Module

Python funktioniert im Browser wie gewohnt. Das bezieht sich auch auf Module, die man wie üblich mit import ins Programm einbindet:

```
import random
wordlist = ["CT.DE", "RULEZ"]
word = random.choice(wordlist)
```

Da dem Browser der Python-Paketmanager pip fehlt, packt man externe Module mit Spiegelstrichen in den <py-env>-Tag und PyScript kümmert sich ums Nachladen:

```
<py-env>
  - numpy
  - matplotlib
</py-env>
```

Es stehen schon einige beliebte Module wie numpy und matplotlib zur Verfügung, solche mit Hardwarezugriff wie requests können aber nicht funktionieren. Gibt man ein Modul im <py-env>-Tag an, muss man es trotzdem im Code mit einem import einbinden.

Wortlisten per Ajax asynchron laden

Damit wir die Begriffe zentral verwalten können, wollten wir sie nicht als ellenlange Liste hardcoden, sondern lieber mit Ajax nachladen. In normalem Python-Code würde man für so etwas

eine Bibliothek wie requests nehmen. Die läuft aber im Browser nicht. Der kann jedoch von sich aus Daten laden, was die Funktion pyfetch() aus dem pyodide-Modul anstößt.

Ajax-Anfragen sind von Natur aus asynchron, weshalb pyfetch() nur in asynchronen Funktionen funktioniert. Die erzeugt man mit async def statt def. Da sie dem normalen Code nicht im Weg stehen, kann man in so einer Funktion problemlos mit await auf Antworten warten, ohne das ganze Programm auszubremsen:

```
from pyodide.http import pyfetch
from pyodide import JsException
from js import console

async def load_wordlist():
  try:
    response = await pyfetch(
url="https://raw.githubusercontent.com/pinae/Nerdle/main/nerdl
e-begriffe.txt",
      method="GET",
      headers={"Content-Type":
               "text/plain"})
    if response.ok:
      data = await response.string()
      console.log(data.split())
      return data.split()
  except JsException:
    return None
```

Die JsException ist die Basisklasse aller JavaScript-Fehler, die PyScript automatisch kapselt. Man könnte hier auch spezifische Exceptions fangen, um aussagekräftige Fehlermeldungen anzuzeigen.

Das Speichern der Liste und das Auswählen des zufälligen Worts übernimmt die Funktion pick_word(). Auch sie ist asynchron, weil sie mit await auf die Rückgabe von load_wordlist() warten muss:

```
async def pick_word():
  global wordlist, word, accept_input
```

```
  wordlist = await load_wordlist()
  word = random.choice(wordlist)
  console.log("Wort: ",
              " ".join(list(word)))
  accept_input = True
```

Um die asynchronen Funktionen so aufzurufen, dass sie den Code nicht blockieren, kann man einen alten JavaScript-Trick benutzen:

```
setTimeout(create_proxy(pick_word), 0)
```

Der Timeout von 0 zwingt den Code nicht zum Warten, bei 0 Millisekunden Verzögerung gibt es aber auch keine Wartezeit. Die Timeout-Funktion sorgt dabei automatisch für eine nebenläufige Ausführung.

Nerdle-HTML

Nerdle benutzt ein Spielbrett aus 30 Quadraten (6 Zeilen mit je 5), die je ein Zeichen aufnehmen. Das geht hervorragend mit einem Grid-Layout. Und da alle Felder gleich aussehen, kann man sie bequem im Quellcode erzeugen. Der fügt alle hintereinander in <div id="board"></div> ein und merkt sich die Divs in einer Liste aus Listen (eine pro Zeile):

```
tiles=[]
board=document.getElementById("board")
for row in range(6):
  tiles.append([])
  for col in range(5):
    tile=document.createElement("div")
    board.appendChild(tile)
    tiles[-1].append(tile)
```

Form und Farbe legt die Datei nerdle-styles.css fest, die Sie zusammen mit dem Rest des Codes im Repository über [ct.de/ynk9](ct.de/ynk9) finden.

Die Funktionen getElementById() und createElement() funktionieren genau wie in JavaScript, sodass Sie dort die

Dokumentation konsultieren können, solange PyScript noch keine eigene dafür hat. Die Funktionen gehören zum document-Objekt, das Sie mit from js import document laden.

Die Tasten der Bildschirmtastatur sind ganz ähnliche <div>, allerdings von Anfang an im HTML. Es gibt einen Unterschied: Das umrahmende <div> hat die ID "keyboard". Der Python-Code kann sich anhand der Beschriftung der Tasten dann selbst ein Dictionary zusammenbauen, das jedem Zeichen das passende DOM-Objekt zuordnet:

```
key_objects = {}
for row in document.getElementById(
               "keyboard").childNodes:
  for key in row.childNodes:
    key.addEventListener("click",
                       js_key_clicked)
    if len(key.textContent) == 1:
      key_objects[
                key.textContent] = key
```

childNodes ist dabei die JavaScript-Datenstruktur NodeList, die aber das Iterator-Interface implementiert, sodass sich damit fast wie mit einer Python-Liste arbeiten lässt. addEventListener() ist die von Python aufrufbare JavaScript-Funktion, die den JavaScript-Function-Pointer js_key_clicked annimmt. Den muss man allerdings etwas umständlich erzeugen.

Verpackte Funktionen

Funktionen definiert man im Python-Code wie üblich mit def. Heraus kommt dabei eine Python-Funktion, die im Python-Code ganz normal funktioniert. Will man aber mit dem DOM interagieren, muss man die von PyScript gekapselten JavaScript-Funktionen benutzen, die man an den Namen in CamelCase erkennt. Diese JavaScript-Funktionen sehen die Python-Funktionen nicht, weshalb man eine Funktionsreferenz beispielsweise nicht einfach an addEventListener() übergeben kann.

Die Sprachbarriere überwindet das pyodide-Modul, das PyScript standardmäßig mitbringt (kein Eintrag in <py-env> nötig).

```
from pyodide import create_proxy
def key_clicked(e):
  print(e.target.textContent)

js_key_clicked = create_proxy(
                          key_clicked)
```

Mit create_proxy() packt man die Python-Funktion so ein, dass JavaScript sie sehen kann. Die so erzeugte Referenz kann man wie eine JavaScript-Funktion an addEventListener() übergeben.

Dabei funktionieren wiederum alle von JavaScript bekannten Datenstrukturen, sodass die Python-Funktion über den Parameter e das Event-Objekt bekommt. Das verweist unter e.target auf das DOM-Objekt, von dem das Ereignis ausging, und das verrät mit e.target.textContent, was innerhalb des HTML-Tags steht.

Noch ein Hinweis auf eine mögliche Fehlerquelle beim Konvertieren von JavaScript-Code von StackOverflow: Die Python-Funktion muss alle Parameter nennen, die JavaScript übergibt. JavaScript erlaubt es, Parameter still und heimlich wegzulassen, während Python mindestens einen _ verlangt. Man muss die übergebenen Parameter in der Funktion aber nicht benutzen, wenn man sie nicht braucht.

Raten

Nachdem der Code schon ein Wort zum Erraten ausgewählt und das Dictionary mit den Tasten initialisiert ist, muss er nur noch die Eingaben verarbeiten und bei „Enter“ den geratenen Begriff auswerten.

Die aktuelle Eingabe speichert das Programm in der Variable guess. Fürs Verarbeiten der Eingaben macht sich die Funktion key_clicked() den Umstand zunutze, dass alle <div> mit einem Zeichen einen textContent mit Länge 1 haben. Das Backspace-

Symbol ist ein SVG, weshalb textContent bei dieser Taste ein leerer String ist. Die Enter-Taste dagegen ist mit „Enter“ beschriftet, also mit fünf Zeichen. Die Fallunterscheidung ist eine gute Gelegenheit, das mit Python 3.10 eingeführte Pattern-Matching einzusetzen:

```
match len(e.target.textContent):
  case 0:
    guess = guess[:-1]
  case 1:
    guess += e.target.textContent
  case _:
    check_enter()
display_guess()
```

match…case funktioniert so ähnlich wie switch…case in C, erlaubt aber beispielsweise auch reguläre Ausdrücke hinter case. Statt default: gibt es case _:, der zum Tragen kommt, wenn keiner der anderen Fälle eintrifft. In allen Fällen kümmert sie die Funktion display_guess() darum, den Inhalt von guess auch anzuzeigen. Pattern Matching funktioniert auch mit Objekten, beispielsweise mit re (das Modul für reguläre Ausdrücke) erzeugte Tupel. Ein Beispiel dafür finden Sie im Code auf GitHub.

Die Funktion display_guess() konsultiert die Variable guess_no, die speichert, in welcher Zeile Spieler gerade raten. Zuerst füllt die Funktion überall Leerzeichen ein, um vorherige Eingaben zu löschen:

```
for i in range(5):
  tiles[guess_no][i].textContent = ""
```

Das Eintragen aller Buchstaben aus guess geht fast genauso einfach, weil Python klaglos mit list() Strings in Listen aus Einzelzeichen verwandelt:

```
for pos, c in enumerate(list(guess)):
  tiles[guess_no][pos].textContent = c
```

Python kann Tupel automatisch auspacken, wenn man der Anzahl

entsprechend viele Variablen mit Komma getrennt hintereinander schreibt. enumerate() gibt für jeden Iterator, also für alles, was sich wie eine Liste behandeln lässt, ein 2-Tupel aus der Nummer und dem Element zurück. Im Beispiel landet in pos also die Nummer des Zeichens und in char das Zeichen. Die Schreibweise, die dabei herauskommt, versteht man ganz intuitiv.

Vergleichen

Bei einem „Enter“ muss das Spiel zunächst prüfen, ob der geratene Begriff fünf Zeichen hat und ob er in der Wortliste steht. Wenn nicht, schreibt die Funktion in das <div> mit der ID "info" eine Fehlermeldung:

```
pyscript.write("info", f"„{guess}“ " +
      "steht nicht in der Wortliste.")
```

Die Funktion pyscript.write() nimmt einem dabei die Arbeit ab, das DOM-Element mit der ID "info" herauszusuchen und seinen textContent zu ändern. Wir vermuten, dass PyScript in Zukunft noch weitere Funktionen ähnlicher Art bekommt, die DOM-Zugriffe mit weniger Code erlauben.

Außerdem nutzt die Zeile Pythons seit 3.6 verfügbare f-Strings. Die definiert man mit dem Buchstaben f vor den Anführungszeichen und Python ersetzt dann alle in geschweiften Klammern angegebenen Variablen durch deren Werte. Man kann die Ausgabe mit den gleichen Filtern wie in format() beeinflussen.

Steht in guess ein Begriff aus der Wortliste, vergleicht die Funktion evaluate_guess() den geratenen Begriff mit word, dem zu erratenden Begriff. Dafür macht sie den Begriff wieder zu einer Liste, holt sich mit enumerate() die Zeichennummer dazu, die sie dann benutzt, um das Zeichen aus dem zu erratenden Begriff zu ziehen und die Zeichen zu vergleichen:

```
for p, char in enumerate(list(guess)):
  if char == word[p]:
```

```
    tiles[guess_no][p].classList.add(
                           "nailedit")
    key_objects[char].classList.add(
                           "nailedit")
    correct_counter += 1
```

Damit Spieler Grün und Gelb sehen, ergänzt die Funktion über classList.add() die passende CSS-Klasse, die die Farbe festlegt. Das passiert sowohl bei den Quadraten im Spielfeld als auch bei der Bildschirmtastatur. Dass die Tasten ihre Farbe verändern, ist eine willkommene Hilfe beim Sinnieren über den nächsten Begriff, den man raten könnte.

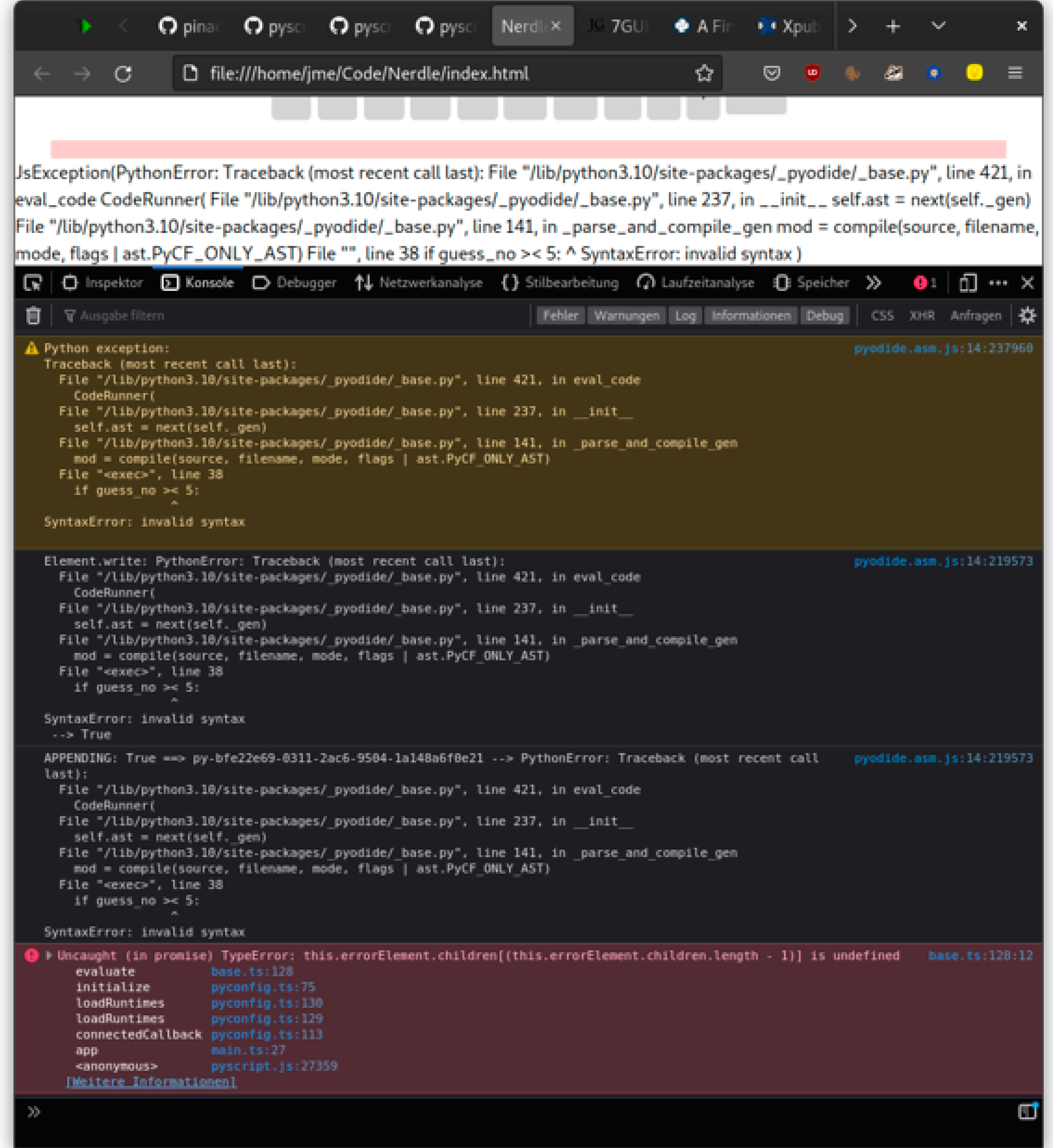
Bei richtig geratenen Zeichen zählt die Funktion auch die lokale Variable correct_counter hoch. Steigt die nämlich beim Einfärben aller Zeichen-Quadrate auf 5, ist das Ratespiel gewonnen. Ist die Variable kleiner und zusätzlich guess_no >= 5, ist das Spiel verloren.

Usability

Der vollständige Code, den Sie über das Repository über [ct.de/ynk9](ct.de/ynk9) finden, enthält noch einige zusätzliche Zeilen. Die dienen der besseren Bedienbarkeit, damit man beispielsweise statt per Bildschirmtastatur auch mit der normalen Tastatur tippen kann. Das geht mit einer key_down()-Funktion, die den onkeydown-Event-Handler von document überschreibt:

```
js_key_down = create_proxy(key_down)
document.onkeydown = js_key_down
```

Der Code ist mitsamt HTML kaum mehr als 200 Zeilen lang, die bis auf einen regulären Ausdruck leicht zu lesen sind.



Exceptions sind in der Entwicklerkonsole sichtbar, die Browser mit F12 öffnen. Die Darstellung ist leider nicht so übersichtlich, weil PyScript alles in JavaScript-Objekte einpacken muss.

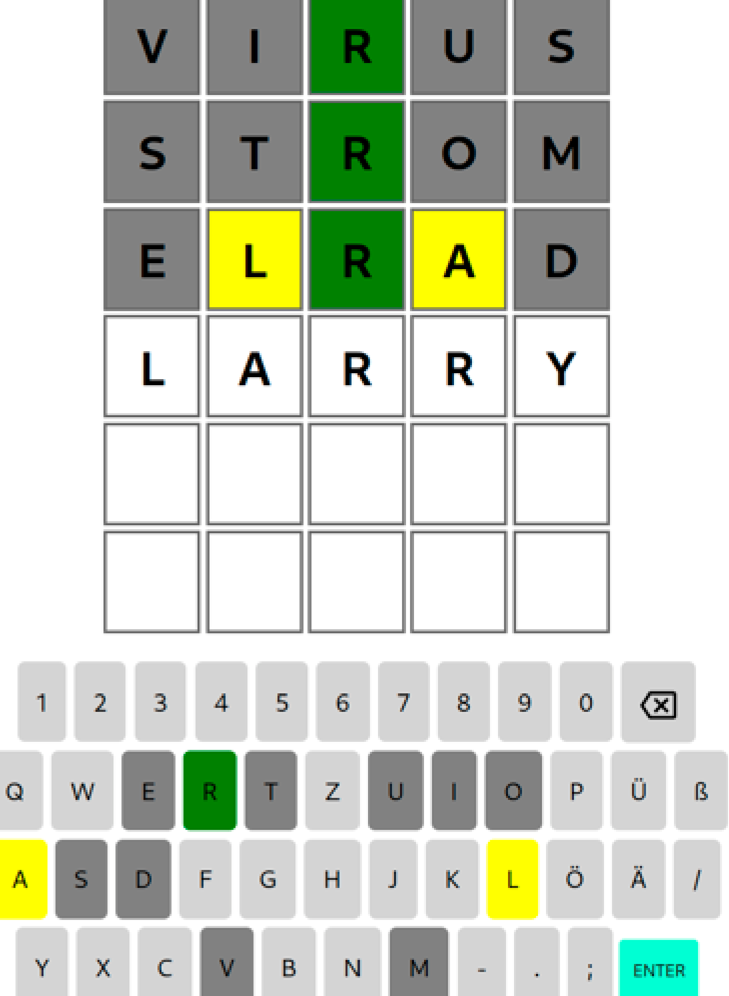
Fürs Debugging lohnt es sich, mit F12 die Entwickler-Konsole zu öffnen. Dort landet auch alles, was Sie mit console.log() ausgeben. Die Fehlermeldungen sind leider nicht gut lesbar, weil es in JavaScript-Exceptions verpackte Python-Exceptions sind. Das müssen die Entwickler noch stark nachbessern.

Eine weitere Baustelle sind die Entwicklungsumgebungen. Beispielsweise erkannte die Entwicklungsumgebung PyCharm den Code in <py-script>-Tags nicht als Python-Quelltext, sodass

weder Farben noch Code-Vervollständigung funktionierten.

Spielen

Das selbst programmierte Nerdle starten Sie, indem Sie einfach die Datei index.html im Browser öffnen. Ein Webserver ist dafür nicht notwendig, zum Nachladen des PyScript-Codes und der Wortliste aber eine Internetverbindung. Installieren müssen Sie gar nichts. Falls Sie das Spiel doch mit einem Webserver hosten wollen, muss der nur eine statische Seite ausliefern können.



Unser Worträtsel „Nerdle“ ist wegen der Tech-Begriffe mit Abkürzungen deutlich schwerer als „Wordle“ von der New York Times.
Unsere Liste mit Begriffen finden Sie im Repository in der Datei nerdle-begriffe.txt. Momentan stehen da schon mehr als 1500 Begriffe zum Raten bereit. Mit dem Skript word_list_filter.py können Sie die Liste aber bequem erweitern. Es liest die Datei neue-begriffe.txt, filtert nach den richtigen Zeichen und fragt für alle neuen Begriffe einzeln, ob Sie die hinzufügen wollen. So können Sie mit wenig Arbeit ganze Listen aus Kreuzworträtsel-Datenbanken ergänzen.

Kritik

Wir sind von PyScript begeistert. Python-Code ohne Installation im Browser ausprobieren? Grandios!

Noch ist PyScript aber nicht an dem Punkt angelangt, darauf bestehenden Code zu portieren und über Experimente hinausgehende Projekte damit umsetzen zu wollen. Außerdem muss eine Webseite mit PyScript mehr als 14 Megabyte an Code laden, bevor sie überhaupt starten kann. Danach muss die JavaScript-Engine den WebAssembly-Code zunächst in Bytecode für die Prozessorarchitektur übersetzen, was selbst auf einer schnellen Desktop-CPU fast eine Sekunde dauert. Erst danach läuft der Code der Seite los. Bei normalem JavaScript lädt der Browser nur die winzige Skriptdatei und führt diese sofort aus, weil die JavaScript-Engine längst warm gelaufen ist.

PyScript hat trotz allem sinnvolle Anwendungen: Hat beispielsweise eine Statistikerin ihre Daten schon mit Python ausgewertet und mit Matplotlib ein Diagramm gezeichnet, müsste sie normalerweise alles in JavaScript nachprogrammieren, um das Diagramm in eine Webseite einzubinden. PyScript senkt die Hürde für Pythonisten, mal eben schnell aus einer Idee eine Webanwendung zu basteln – wie unser Beispiel zeigt. (*pmk@ct.de*)

Code bei GitHub, PyScript-Dokumentation: ct.de/ynk9