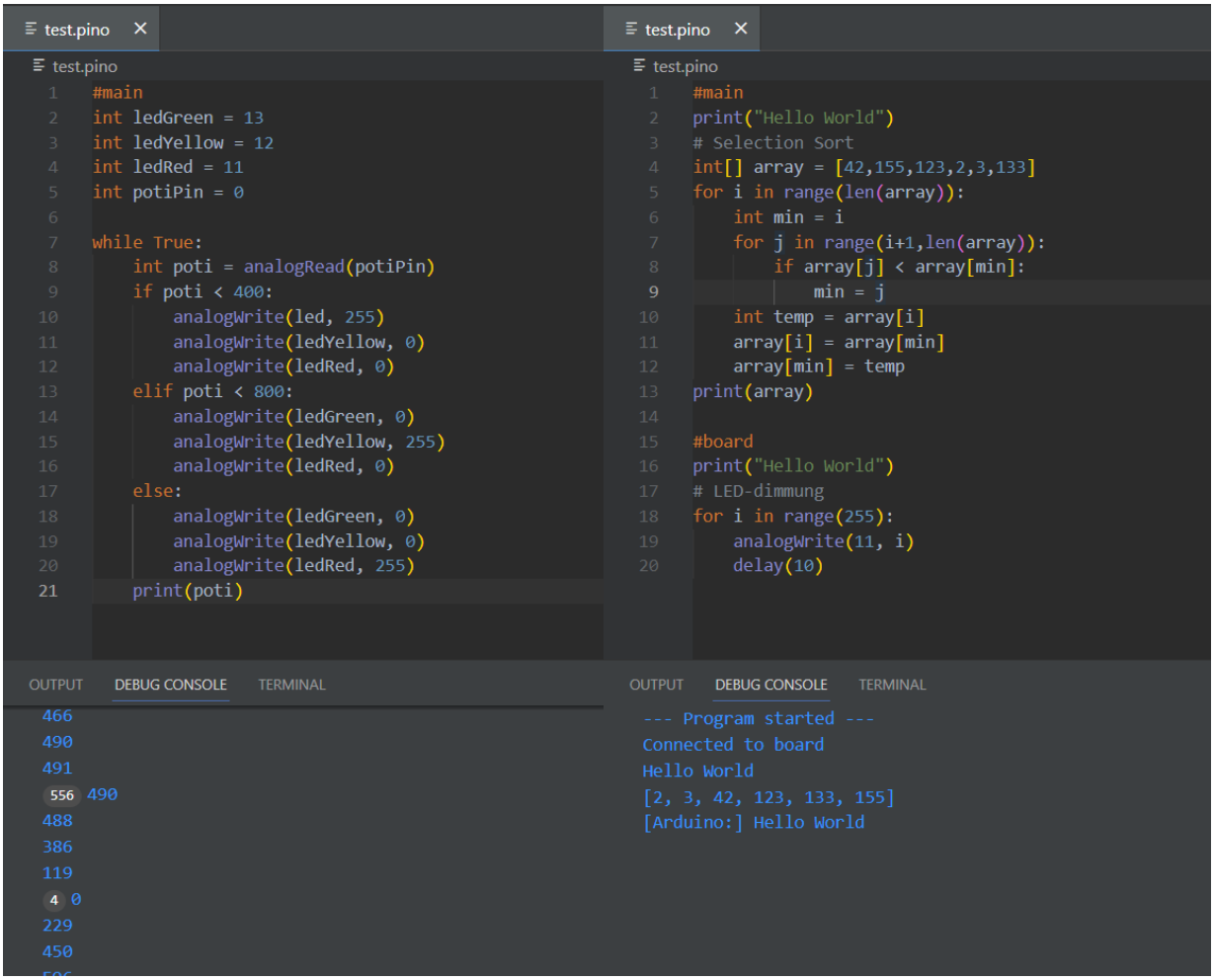
# Pyduino

Christian Krause

Schülerforschungszentrum Südwürttemberg (Standort Ochsenhausen)

Jugend Forscht 2023



# Inhalt



# Kurzfassung

In meinem Projekt habe ich eine einfachere Programmiersprache für den Microcontroller Arduino entwickelt. Sie soll mit einer Python-ähnlichen Syntax den Einstieg ins Programmieren erleichtern, da für den Arduino sonst die schwierigere Programmiersprache C verwendet werden muss. Um dies möglich zu machen, habe ich einen Transpiler in Python programmiert, der die Pyduino Syntax in C Syntax übersetzt und auf Fehler überprüft. Dieses C Programm wird dann für den PC und den Arduino kompiliert. So kann dieselbe Syntax auf beiden Plattformen ausgeführt werden. Wenn der PC mit dem Arduino verbunden ist, kann der PC auf die Ports des Arduino zugreifen und der Arduino kann Text in der Konsole des PCs ausgeben, auf Dateien zugreifen und Funktionen aufrufen. Mit Pyduino kann der Arduino vom PC aus gesteuert werden, ohne das Programm jedes Mal hochladen zu müssen und selbst auf die Rechenleistung eines PCs zugreifen, was die Einsatzmöglichkeiten des verbreiteten Microcontrollers stark erweitert.

# Problem

Der Arduino ist ein sehr beliebter Microcontroller, mit dem sehr viele die Grundlagen des Programmierens lernen. Im NWT-Unterricht in Baden-Württemberg, sowie in Werkrealschulen und Realschulen im Fach Technik ist er ein wichtiger Bestandteil [Q1]. Außerdem wird der Arduino in vielen anderen Bundesländern für Lehrer verbreitet fortgebildet, in einigen Hochschulen zum Start in das technische Studium genutzt und er findet sich als Empfehlung sogar in der offiziellen Abschlussprüfung der IHKs [Q2]. Die Popularität des Arduino hängt damit zusammen, dass er einfach, benutzerfreundlich und universell einsetzbar ist. Auch der Einstieg ist sehr einfach: Innerhalb von wenigen Minuten kann selbst ein Anfänger die ersten Ergebnisse, wie zum Beispiel eine blinkende LED, sehen. Der Arduino ist aber trotzdem auch für fortgeschrittene Anwendungen geeignet. Dabei ist er aber immer noch sehr benutzerfreundlich, vor allem durch die neue Arduino IDE 2.x, die Features wie Syntax-Highlighting und Autovervollständigung für die Arduino-Programmiersprache zur Verfügung stellt. Diese Programmiersprache basiert auf C++ und ist mit einigen Hilfsfunktionen, mit denen zum Beispiel die Ein- und Ausgänge des Arduino angesteuert werden können, ausgestattet [Q3].

Da der Arduino oft von Anfängern verwendet wird, um in die Programmierung einzusteigen, ist die Programmiersprache allerdings auch ein Problem. C++ ist eine low-level Programmiersprache, bei der dem Nutzer weniger „abgenommen“ wird als bei high-level Programmiersprachen wie zum Beispiel Python. Das führt dazu, dass Programmieranfänger nicht nur grundlegende Programmierkonzepte, sondern auch spezielle Eigenheiten von C++ lernen müssen, die den Einstieg in die Informatik unnötig kompliziert machen.

In einer high-level Programmiersprache wie Python ist es zum Beispiel möglich, verkettete Listen, bei denen einfach Elemente angehängt und entfernt werden können, zu verwenden oder Funktionen zu programmieren, die Arrays zurückgeben können. Außerdem sind viele hilfreiche Funktionen, wie die **`len()`** Funktion oder die **`sum()`** Funktion, mit denen die Länge und Summe von Listen bestimmt werden können, in Python bereits enthalten.

Solche einfachen Dinge sind in C++ aber nur mit großem Aufwand umsetzbar. Verkettete Listen sind in C++ nicht implementiert, d.h. entweder müssen sie selbst programmiert werden oder es müssen Arrays verwendet werden. Arrays können aber nur mit einer festen Größe und Datentyp definiert werden, es können also zum Beispiel keine Elemente eingefügt oder entfernt werden.

Für Anfänger wäre es daher deutlich komfortabler und intuitiver, mit Python in die Informatik einzusteigen. Aufgrund der vielen Vereinfachungen benötigt Python aber deutlich mehr Ressourcen und ist langsamer als C++. Deshalb ist es bisher nicht ohne weiteres möglich Python auf dem Arduino auszuführen.

Die Arduino IDE basiert bisher auf dem Konzept, Programme zu kompilieren und auf den Arduino hochzuladen [Q4]. Diese Funktionen stellt das Arduino CLI zur Verfügung, sie sind aber sehr zeitaufwändig, was das Programmieren erschwert:

| Plattform *(siehe Quellen)* | Kompilieren | Hochladen | Gesamt |
|---|---|---|---|
| PC | 0.74s | 1.85s | 3.59s |
| Laptop | 4.43s | 2.78s | 7.21s |

*(Die Zeiten wurden mit diesen Programmen ermittelt: [https://github.com/Bergschaf/Arduino-Benchmark](https://github.com/Bergschaf/Arduino-Benchmark))*

Man kann feststellen, dass die Zeit, die für das Kompilieren benötigt wird, bei größeren Programmen nur leicht steigt. Die Zeit, die zum Hochladen benötigt wird, steigt dagegen etwas bei größeren Programmen. Diese Zeiten sind aber trotzdem sehr lang, da der Nutzer jedes Mal bis zu 7s warten muss, nur um den Effekt einer kleinen Änderung im Programm zu sehen.

Dieses Problem lässt sich mit der bestehenden Methode aber nicht lösen, da die Arduino-Programme kompiliert und auf den Arduino Microcontroller hochgeladen werden müssen. Python-Programme werden dagegen von einem Interpreter ausgeführt, was fast ohne Zeitverzögerung funktioniert.

# Lösungsidee

Um diese Probleme zu lösen, könnte man beim Einstieg in die Informatik an den Schulen natürlich einfach mit einer high-level Programmiersprache wie zum Beispiel Python beginnen. Dabei bliebe aber ein großer Vorteil

des Arduino auf der Strecke, der für den Siegeszug des Microcontrollers an den Schulen und in der Ausbildung ausschlaggebend war. Er bietet die Möglichkeit, nicht nur theoretisch zu programmieren, sondern auch mit Elektronik zu experimentieren. Es können zum Beispiel LEDs und Sensoren angesteuert werden. Es gibt zwar Ansätze diese zwei Aspekte der Elektronik und der einfachen Programmierung zusammenzuführen. Da Python aber deutlich mehr Ressourcen benötigt als C++ sind Kleinrechner wie der Raspberry Pi deutlich teurer als der Arduino und Microcontroller wie der ESP, der ebenfalls mit einer vereinfachten Python-Version programmiert werden kann, viel weniger etabliert. Durch die große Community und die zahlreichen, speziell auf den Arduino angepassten Sensoren kommen Anfänger am populären Microcontroller kaum vorbei.

Deshalb ist meine Idee, eine Programmiersprache zu entwickeln, deren Syntax von Python inspiriert, also möglichst einfach ist, die aber trotzdem so wenig Ressourcen verbraucht, dass sie auch auf dem Arduino ausgeführt werden kann. Um dies zu ermöglichen, müssen zwar einige Vereinfachungen gegenüber Python unternommen werden, die die Programmiererfahrung aber nicht signifikant einschränken. Diese Vereinfachungen machen es möglich, dass die Sprache von einem Transpiler in C++ Syntax übersetzt und danach für den Arduino kompiliert werden kann. Ein weiterer Vorteil der Sprache ist, dass sie nicht nur auf dem Arduino, sondern auch auf einem herkömmlichen Computer läuft. Programme können somit auf dem PC und auf dem Arduino parallel ausgeführt werden. Der Programmteil, der auf dem PC ausgeführt wird, hat dabei, sofern der Arduino verbunden ist, die Möglichkeit, die Pins auf dem Arduino anzusteuern. Der Programmteil auf dem Arduino kann über die Verbindung zum Beispiel Ausgaben an den PC senden, um sie dann in der Konsole anzuzeigen.

Diese Verbindung gibt dem Programmierer die Möglichkeit, den Arduino auch nur als Steuerungseinheit für die Pins zu verwenden und sie direkt vom PC aus zu steuern. Da so das Programm nicht jedes Mal von dem langsamen Arduino CLI kompiliert und hochgeladen werden, muss können so die Wartezeiten minimiert werden. Das Programm müsste nur von einem Transpiler in C++ übersetzt werden und dann von einem C++ Compiler für den PC kompiliert werden, was deutlich schneller als der Arduino Compiler ist. Weil mein Ansatz beide Welten von Arduino und Python verbindet, habe ich meine Programmiersprache „Pyduino“ genannt.

Pyduino soll mit einer IDE, die mit modernen Features wie Syntax-Highlighting, Autovervollständigung und automatischer Fehlererkennung eine möglichst gute Programmiererfahrung bieten. Da es sehr schwierig ist, eine eigene IDE zu entwickeln, war meine Idee, ein Plugin für die bekannte IDE Visual Studio Code von Microsoft zu entwickeln. Sie ist frei verfügbar und bietet eine gute API, die es ermöglicht, mit geringem Aufwand neue Programmiersprachen einzubinden.

## Umsetzung

Als Grundlage verwende ich die IDE Visual Studio Code [Q5]. Sie stellt eine API für Erweiterungen zur Verfügung, mit der auch Unterstützung für neue Programmiersprachen implementiert werden kann. Um erweiterte Funktionen für diese Sprachen zur Verfügung zu stellen, gibt es die Möglichkeit, einen Language Server mithilfe des Language Server Protocols (LSP) zu verwenden [Q6]. Die Idee hinter dem LSP ist, die Entwicklung von Features wie Autovervollständigung und Fehlererkennung für verschiedene Texteditoren so einfach wie möglich zu machen [Q7]. Deshalb wird die Implementierung dieser Features von den Texteditoren und IDEs getrennt, indem nur ein Language Server für jede Programmiersprache entwickelt werden muss. Dieser Language Server kann dann über das Language Server Protocol mit einem Texteditor kommunizieren und so Features wie Autovervollständigung, Fehlererkennung und Dokumentation zur Verfügung stellen. Das bringt auch den Vorteil, dass der Language Server in einer beliebigen Programmiersprache entwickelt werden kann, unabhängig von der API des Texteditors oder der IDE.

Daher war meine Idee, mithilfe der Bibliothek pygls [Q8] einen Language Server für Pyduino in Python programmieren. Die implementierten Features können dann mit einer Erweiterung in Visual Studio Code verwendet werden.

Um Pyduino Programme auszuführen, wird aber auch ein Transpiler benötigt, der den Pyduino-Code in C++ übersetzt. Der Code muss beim Übersetzten in C++ auch auf Fehler überprüft werden, was eng mit den Aufgaben des Language Servers zusammenhängt. Deshalb habe ich mich entschieden, den Language Server und den Transpiler in einem Python Modul umzusetzen. Wenn ein Programm auf Fehler überprüft werden soll,

wird es transpiliert und die Fehler werden gespeichert. Die Daten, die dabei über das Programm gesammelt werden, könnten auch dafür verwendet werden, um Autovervollständigung zur Verfügung zu stellen.

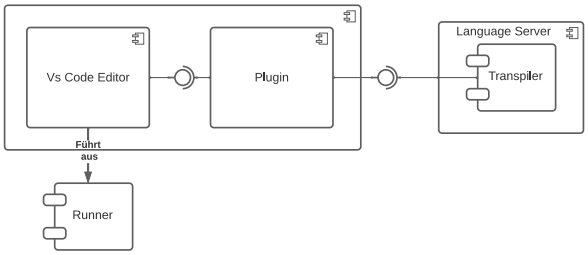



*Abbildung 1: Überblick der Architektur von Pyduino*

## Transpiler

Ich habe mich dafür entschieden, den Transpiler in Python zu schreiben, da Python eine einfache und übersichtliche Programmiersprache ist, die es einfach macht, zum Beispiel mit Strings zu arbeiten. Außerdem ist Python sehr weit verbreitet und bekannt, was es für andere einfach macht, sich in das Projekt einzuarbeiten.

Python ist zwar langsamer als kompilierte Sprachen wie C++, die Performance reicht für den Language Server aber aus. Er kann ein Pyduino Programm innerhalb von wenigen Millisekunden übersetzen. Das folgende Diagramm zeigt den Aufbau des Transpilers:

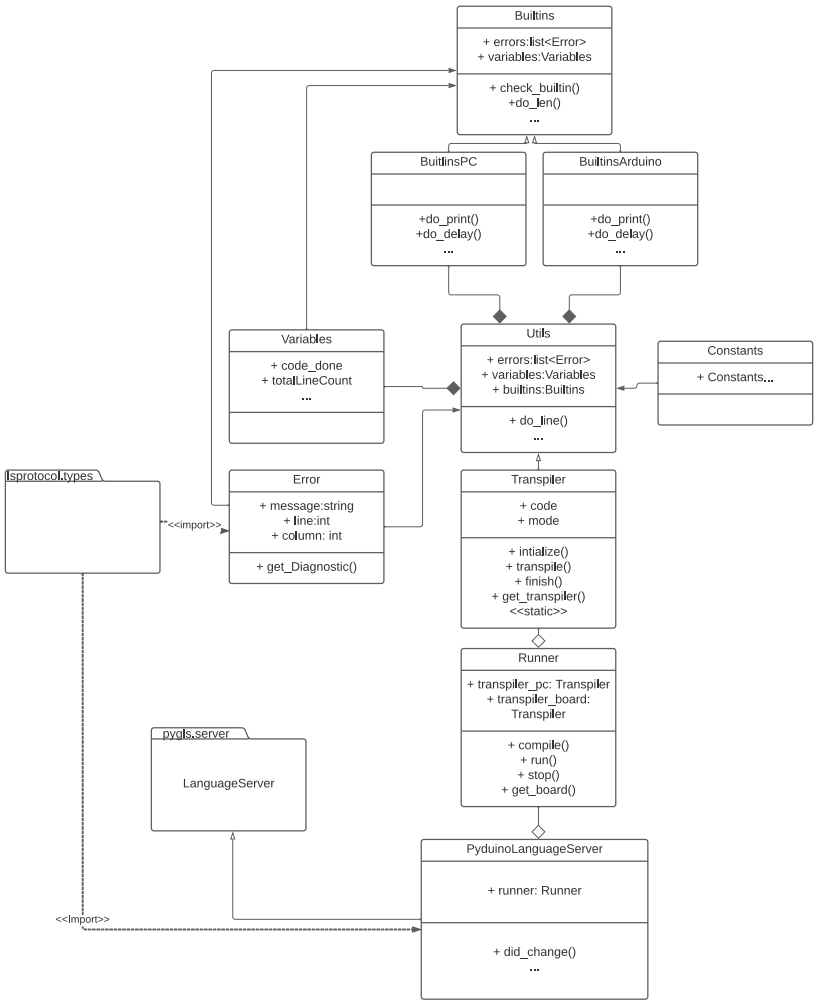



*Abbildung 2: Klassendiagramm für den Transpiler*

Für das Transpilieren von einem Pyduino Programm zu C++ ist die Klasse **`Transpiler`** verantwortlich. Sie wird jeweils für den **`#main`** und **`#board`** Teil des Programms instanziiert. Dann erstellt jede Instanz der Klasse ein **`Variables`** Objekt, in dem wichtige Daten über das Programm gespeichert sind. Es wird zum Beispiel eine Liste für den fertig übersetzten C++-Code angelegt. Außerdem wird im **`Variables`** Objekt ein Iterator angelegt, mit dem der Code in der richtigen Reihenfolge Zeile für Zeile übersetzt werden kann und auf den von allen Klassen aus zugegriffen werden kann. Das ist wichtig, da der Transpiler rekursiv arbeitet. Für das Übersetzten des Programms ist nach der Initialisierung die **`transpile()`** Funktion zuständig. Sie funktioniert nach folgendem Prinzip:

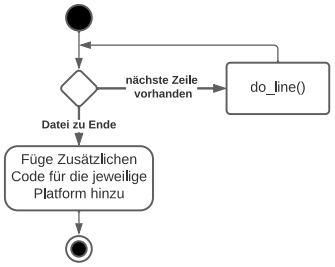



*Abbildung 3: transpile() Funktion*

Der Code wird mithilfe der **`do_line()`** Funktion, die die **`Transpiler`** Klasse von der **`Utils`** Klasse erbt, zeilenweise übersetzt. Anschließend wird mit der **`finish()`** Funktion der **`Transpiler`** Klasse der plattformspezifische Code, der zum Beispiel für die Verbindung zwischen PC und Arduino zuständig ist, hinzugefügt. Dieses Diagramm zeigt das grundlegende Prinzip der **`do_line()`** Funktion:

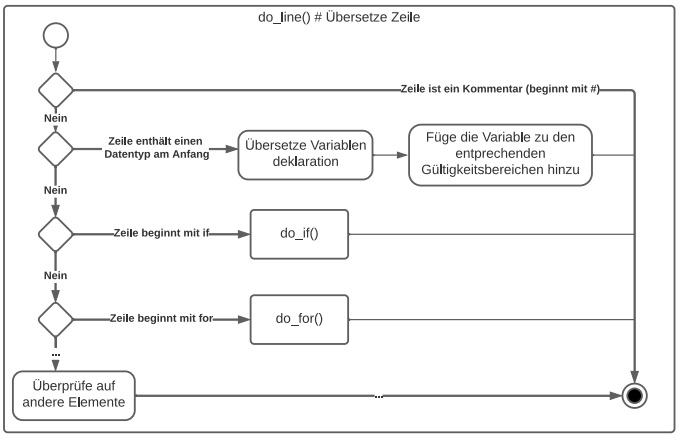



*Abbildung 4: do_line() Funktion*

Die **`do_line()`** Funktion überprüft, welches Syntax-Element oder welche Art von Anweisung am Anfang der Zeile steht. Dementsprechend wird dann die passende Funktion aus der **`Utils`** Klasse aufgerufen. Ein Beispiel dafür ist eine **`if`** Bedingung. Wenn eine Zeile mit **`if`** beginnt, dann wird die **`do_if()`** Funktion aufgerufen, die nach folgendem Prinzip arbeitet:

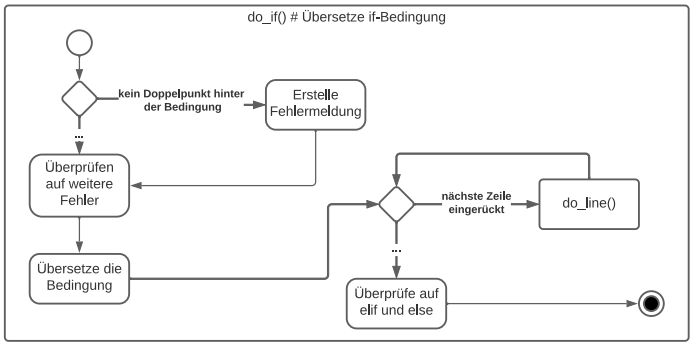



*Abbildung 5: do_if() Funktion*

Zuerst wird die Bedingung in C++ übersetzt. Anschließend werden alle Zeilen hinter der **`if`** Bedingung, die eingerückt sind, mit der **`do_line()`** Funktion übersetzt. Danach wird überprüft, ob nach dem **`if`** Block ein **`elif`** oder **`else`** Segment zu finden ist. Für diese Elemente wird dann ggf. auch die Bedingung und der eingerückte Teil übersetzt.

Falls beim Transpilieren Fehler entdeckt werden, wie zum Beispiel ein fehlender Doppelpunkt nach einer **`if`** Bedingung, werden diese gespeichert. Dafür wird ein **`Error`** Objekt erstellt, in dem die Fehlermeldung, die Zeile und die Spalte des Fehlers gespeichert sind. Wenn ein Fehler entdeckt wird, wird das Programm nicht abgebrochen, sondern der Transpilierungsprozess wird fortgeführt. Das ist wichtig, da so alle Fehlermeldungen im Texteditor angezeigt werden können.

In der **`Utils`** Klasse sind Funktionen definiert, die für das Transpilieren der grundlegenden Programmstruktur wichtig sind. Außerdem enthält sie ein **`Variables`** Objekt, in dem die Daten, die während dem Transpilieren wichtig sind, gespeichert sind und ein **`Builtins`** Objekt.

Die **`Builtins`** Klasse ist die Oberklasse für die Funktionen, die in Pyduino standardmäßig implementiert sind. Sie enthält dabei die Funktionen, deren C++ äquivalent auf dem PC und dem Arduino gleich sind. Ein Beispiel dafür ist die **`len()`** Funktion, die die Länge eines Arrays ermittelt.

Die **`BuiltinsPC`** und die **`BuiltinsArduino`** Klasse erben von der **`Builtins`** Klasse. Sie sind für die Funktionen zuständig, deren Implementation sich zwischen PC und Arduino unterscheidet. Ein Beispiel dafür ist die **`print()`** Funktion. In der PC-Version werden einfach die Argumente in der Konsole ausgegeben, während in der Arduino Version über die serielle Verbindung eine Anfrage an den PC geschickt werden muss. Dieser gibt dann die Werte in der Konsole aus.

Um ein Pyduino Programm auszuführen, wird die **`Runner`** Klasse instanziiert. Sie erhält für den **`#main`** und den **`#board`** Teil des Programms jeweils eine Instanz der **`Transpiler`** Klasse. Dieses Diagramm zeigt den Ablauf, um ein Pyduino Programm auszuführen:

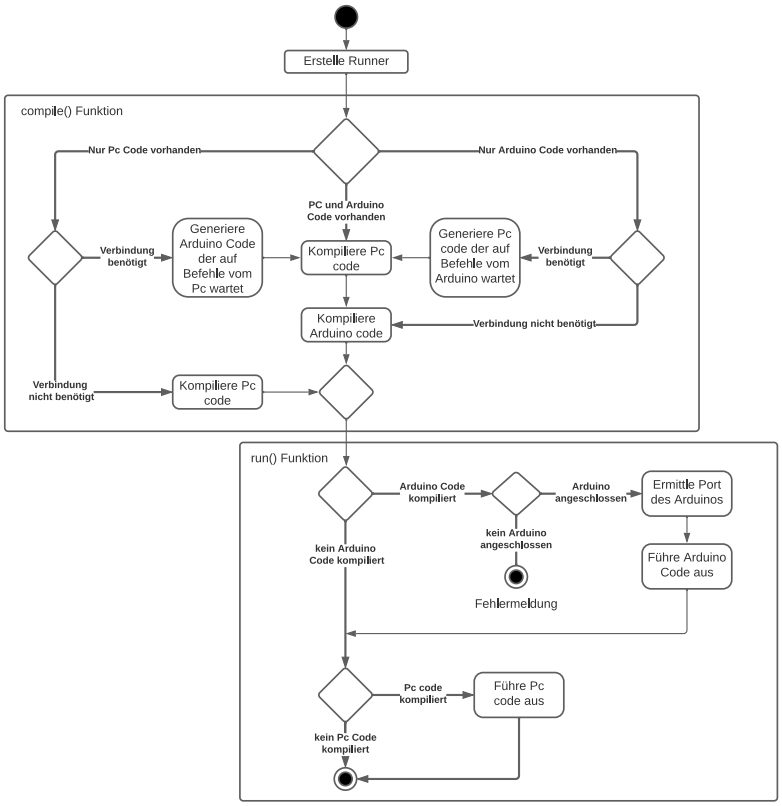



*Abbildung 6: Ablauf, um ein Pyduino Programm auszuführen*

Zuerst wird eine Instanz der **`Runner`** Klasse erstellt. Sie enthält jeweils für den **`#main`** und **`#board`** Programmteil ein **`Transpiler`** Objekt, falls der jeweilige Teil vorhanden ist. Mit der **`transpile()`** Funktion dieser **`Transpiler`** Instanzen können dann die Programmteile in die Syntax für die jeweilige Plattform übersetzt werden. In der **`compile()`** Funktion der **`Runner`** Klasse wird zuerst überprüft, welche Programmteile vorhanden sind. Wenn ein **`#main`** und ein **`#board`** Teil existiert, werden beide für die jeweilige Plattform kompiliert. Wenn nur ein Programmteil vorhanden ist, wird überprüft, ob die Verbindung zum PC oder zum Arduino benötigt wird. Dies wäre zum Beispiel der Fall, wenn im **`#main`** Teil auf die Pins des Arduino zugegriffen wird oder wenn im **`#board`** Teil die **`print()`** Funktion aufgerufen wird. Wenn dies der Fall ist, dann wird für die Plattform, die keinen Programmteil im Pyduino-Programm hat, Code generiert, der auf Befehle der anderen Plattform wartet. Anschließend werden die Programme für beide Plattformen kompiliert. Falls die Verbindung nicht benötigt wird, wird nur der vorhandene Programmteil kompiliert.

In der **`run()`** Funktion wird zuerst überprüft, ob ein kompilierter Arduino Programmteil vorhanden ist. Wenn dies der Fall ist, wird mithilfe des Arduino CLI ermittelt, an welchem Port der Arduino angeschlossen ist. Falls kein Arduino angeschlossen ist, bricht das Programm mit einer Fehlermeldung ab, sonst wird das Programm über den entsprechenden Port hochgeladen und auf dem Arduino ausgeführt. Dasselbe wird für den **`#main`** Teil wiederholt. Falls ein kompiliertes PC-Programm vorhanden ist, wird dieses ausgeführt.

## Verbindung zwischen PC und Arduino

Dieses Diagramm zeigt die Funktionsweise der Verbindung zwischen PC und Arduino:

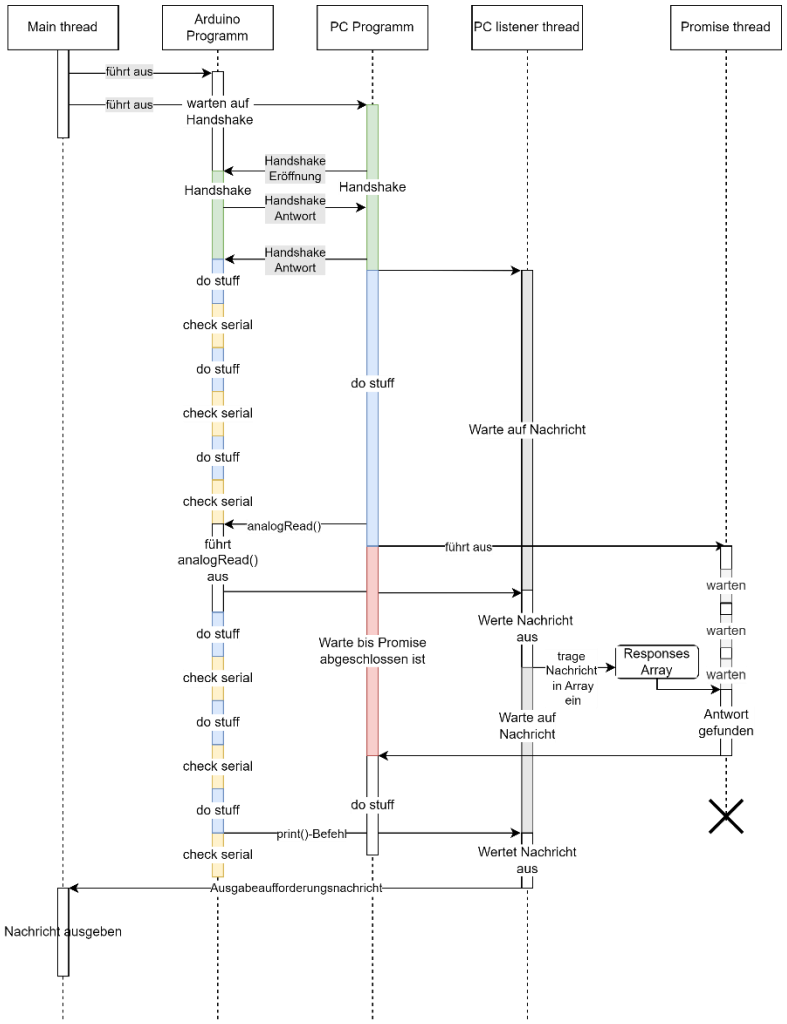



*Abbildung 7: Verbindung zwischen PC und Arduino*

Wenn ein PC- und ein Arduino-Programmteil vorhanden sind und die Verbindung benötigt wird, dann verbinden sich PC und Arduino zuerst über einen Handshake. Dabei werden vorher festgelegte Werte ausgetauscht, um die Verbindung zu überprüfen. Erst wenn der Handshake abgeschlossen ist, starten beide Programme. Dabei wird auf dem PC ein **`listener`** Thread gestartet, der auf Nachrichten des Arduino über die serielle Verbindung wartet. Da der Arduino kein Multithreading unterstützt, wird nach jedem Befehl überprüft, ob Daten im seriellen Buffer [Q9] vorhanden sind. Diese werden dann verarbeitet und ggf. werden die Befehle ausgeführt.

Wenn in dem PC-Programmteil ein Befehl auf dem Arduino ausgeführt werden soll, dann wird der Befehl über die serielle Verbindung an den Arduino geschickt. Dieser führt dann den Befehl aus und schickt die Antwort zurück an den PC. Sie wird dann von dem **`listener`** Thread ausgewertet und anhand der Nachrichten-ID in das **`Responses`** Array geschrieben. Um den Rückgabewert des Befehls zu erhalten, hat der PC-Programmteil neuen Promise, also eine Instanz der **`Promise`** Klasse, erstellt, nachdem er den **`analogRead()`** Befehl an den Arduino geschickt hat:

```
template<typename T>
class Promise {
public:
    typedef T(bytesToType)(char* bytes);
    thread *t;
    static void resolve(int requestID, bytesToType bytesToType, T* targetVariable, char
        Responses[MaxRequests][MaxDataLength]) {
        int millisecondsWaited = 0;
        while (Responses[requestID][0] == 0) {
            sleep_for(milliseconds(1));
            millisecondsWaited++;
            if (millisecondsWaited > MaxMillisecondsToWaitForData) {
```

```
                cout << "Error: Timeout while waiting for response" << endl;
                return;
            }
        }
        if(targetVariable == nullptr || bytesToType == nullptr) {
            Responses[requestID][0] = 0;
            return;
        }
        *targetVariable = bytesToType(Responses[requestID]);
        Responses[requestID][0] = 0;
    }

    Promise(T* targetVariable,bytesToType bytesToType, int requestID, char
        Responses[MaxRequests][MaxDataLength]) {
        // start a thread with the resolving function
        t = new thread(resolve, requestID, bytesToType, targetVariable, Responses);
    }

    // Destructor
    ~Promise() {
        // join the thread
        t->join();
    }
};
```

Die **`Promise`** Klasse wird mit einem Pointer zur Zielvariable, einer Funktion, die die Antwort zu dem entsprechenden Datentyp der Zielvariable konvertiert, einer Nachrichten ID und dem **`Responses`** Array initialisiert. Dabei wird ein neuer Thread mit der **`resolve()`** Funktion gestartet. Diese Funktion wartet, bis für die Nachricht mit der Nachrichten ID eine Antwort im **`Responses`** Array vorhanden ist, oder das Zeitlimit überschritten ist. Anschließend wird die Antwort zu dem entsprechenden Datentyp konvertiert und in die Zielvariable geschrieben. Um sicherzustellen, dass eine Antwort eingegangen ist und ein Wert in die Zielvariable geschrieben wurde, muss der Promise gelöscht werden, was den Destructor aufruft. In diesem wird gewartet, bis der Thread mit der **`resolve()`** Funktion abgeschlossen ist, was bedeutet, dass ein Wert eingegangen ist.

Um die Antwort auf eine synchrone Anfrage, bei der das Programm wartet, bis eine Antwort vom Arduino eingegangen ist, zu erhalten, wird der Promise direkt nach seiner Initialisierung wieder gelöscht, was den Destructor aufruft und den **`main`** Thread damit anhält, bis der Promise aufgelöst ist. Diese Methode wird auch in der zurzeit implementierten **`analogRead()`** Pyduino-Funktion angewendet. Das bedeutet, dass das Pyduino-Programm anhält, bis der Rückgabewert der **`analogRead()`** Funktion eingegangen ist.

Mit der Promise Klasse ist es aber auch möglich, asynchrone Anfragen, bei denen das Pyduino-Programm nicht angehalten wird, bis eine Antwort eingegangen ist, zu implementieren. Dabei würde der Promise gestartet werden, ohne ihn direkt wieder zu löschen. Da der Promise einen separaten Thread mit der **`resolve()`** Funktion startet, wird das Pyduino Programm nicht angehalten. Wenn der Rückgabewert dann gebraucht wird, kann der Promise gelöscht werden, was sicherstellt, dass der Wert vorhanden ist.

## VS Code Erweiterung

Dieses Diagramm zeigt die wichtigsten Dateien der VS Code Erweiterung, die auch aus dem Extension Marketplace heruntergeladen werden kann (https://github.com/Bergschaf/Pyduino-Plugin):

```
D:.
|   language-configuration.json
|   main.py
|   package-lock.json
|   package.json
```

In der **`language-configuration.json`** Datei befindet sich die grundlegende Konfiguration für die Pyduino-Sprache, die mit der Erweiterung als neue Programmiersprache registriert sind. Es wird zum Beispiel festgelegt, dass Kommentare mit **#** beginnen oder dass der Editor Klammern beim Tippen automatisch schließen soll.

Die **`main.py`** Datei kann in der Konsole mit dem Befehl **`env/Scripts/python.exe main.py [Pyduino Datei]`** aufgerufen werden, um eine Pyduino Datei zu kompilieren und auszuführen.

In der **package.jso**n Datei sind die Eckdaten der Erweiterung wie Name, Version und Veröffentlicher festgelegt. Außerdem ist beschrieben, dass die Erweiterung die Sprache „Pyduino“ zur Verfügung stellt und immer dann aktiviert wird, wenn eine **.pino** Datei geöffnet ist.

```
+---client
|   \---src
|       |   extension.ts
|
+---env
|   |   pyvenv.cfg
|   +---Include
|   +---Lib
|   +---Scripts
|   |   |    python.exe
```

Der **env** Ordner enthält die virtuelle Umgebung mit allen Bibliotheken, die benötigt werden, um den in Python implementierten Language Server zu starten. Dabei kann die **python.exe** Datei im **Scripts** Ordner verwendet werden, um Python Dateien auszuführen. Um die insgesamte Größe der Datei kleiner zu halten, ist in der Erweiterung kein Python Interpreter enthalten. Daher muss dieser auf dem System installiert sein. Der Ort, an dem der Interpreter auf dem System zu finden ist, ist in der **pyvenv.cfg** Datei festgelegt. Da sich dieser Ort aber von System zu System unterscheidet, wird er bei jedem System neu bestimmt.

Die **activate()** Funktion der **extension.ts** Datei wird von VS Code aufgerufen, wenn die Erweiterung aktiviert wird, d.h. es wurde eine **.pino** Datei geöffnet. Sie ist dafür zuständig den Language Server zu starten. Dafür wird die **python.exe** Datei in der virtuellen Python-Umgebung verwendet. Dafür muss aber zuerst der Pfad des Python Interpreters auf dem jeweiligen System bestimmt werden. Dieser wird dann in die **pyvenv.cfg** Datei geschrieben. Anschließend kann der Language Server ausgeführt werden.

```
+---mingw
|   |   MinGW.7z
|   \---MinGW
|       +---bin
|           c++.exe
|
+---node_modules
|
```

Da die Programme für den PC in C++ übersetzt werden, muss ein C++ Compiler verwendet werden, um die Programme zu ausführbaren **.exe** Dateien zu kompilieren. Da nur wenige Programmieranfänger einen C++ Compiler installiert haben, ist in der Extension der mingw C++ Compiler enthalten. Um die Dateigröße zu reduzieren ist nur die **.7z** Datei enthalten, die bei Bedarf entpackt wird. In dem **mingw/MinGW/bin** Ordner ist die **c++.exe** Datei zu finden, mit der die C++ Programme kompiliert werden können.

```
+---server
|   |   server.py
|   +---transpiler
|   |   |   arduino-cli.exe
|   |   |   builtin_functions.py
|   |   |   constants.py
|   |   |   error.py
|   |   |   runner.py
|   |   |   transpiler.py
|   |   |   utils.py
|   |   |   variables.py
|   |   +---SerialCommunication
|   |   |       ArduinoSerial.ino
|   |   |       Serial.cpp
|   |   |       Serial.txt
|   |   |       SerialClass.h
|   |   |       SerialPc.cpp
```

```
|   |   |
|   |   +---test
|   |   |       test.py
|
+---syntaxes
|       pyduino.tmLanguage.json
```

Der Language Server, der von der **extension.ts** Datei gestartet wird, ist in der **server.py** Datei implementiert. Bei jeder Änderung der **.pino** Datei wird die **did_change()** Funktion aufgerufen, die mithilfe der **Transpiler** Klasse das Programm auf Fehler überprüft. Die vorher genannten Klassen wie **Transpiler** oder **Runner** sind in den entsprechenden Python Dateien implementiert.

Um Arduino Programme zu kompilieren, den Port des Arduino zu ermitteln und die Programme auf den Arduino hochzuladen wird die **arduino-cli.exe** Datei verwendet.

Die Programme im **SerialCommunication** Ordner sind für die serielle Verbindung zwischen Arduino und PC zuständig.

Im **test** Ordner sind Unit-Tests zu finden, die verschiedene Features des Transpilers testen. Dabei werden Pyduino-Programme in C++ übersetzt, kompiliert und ausgeführt. Die Ausgaben werden dann mit den erwarteten Ausgaben verglichen. Da das Kompilieren der Programme auf nur einem Prozessorkern aber sehr zeitaufwändig ist, werden die Tests mithilfe der multiprocessing Bibliothek parallel ausgeführt, was die Geschwindigkeit signifikant erhöht.

In der **pyduino.tmLanguage.json** Datei sind die Regeln für das Syntax Highlighting, wenn eine Pyduino Datei in VS Code geöffnet ist, festgelegt.

Um Programme in VS Code auszuführen können in der **.vscode/launch.json** Datei Run-Konfigurationen festgelegt werden. Daher wird von dem Language Server eine „Pyduino“ Konfiguration angelegt. Sie startet die **main.py** Datei mit der Python Installation im **env/Scripts** Ordner und der entsprechenden **.pino** Datei als Argument. Die **main.py** Datei kompiliert und startet das Programm, wenn die Pyduino Run-Konfiguration ausgeführt wird.

# Features

## Programmstruktur

Pyduino-Programme sind in zwei Teile aufgeteilt, den **main** Teil und den **board** Teil. Die einzelnen Teile werden mit **#main** und **#board** eingeleitet.

```
#main
print("Hello World Pc")

#board
print("Hello World Arduino")
```

Ein Programm kann auch nur einen der Teile enthalten.

Wenn die Funktionen des anderen Teils gebraucht werden, dann wird automatisch ein Programm auf der jeweiligen Plattform gestartet, das zwar kein Pyduino Programm ausführt, aber trotzdem die Funktionen der jeweiligen Plattform zur Verfügung stellt.

```
#board
analogWrite(11,255)
```

Hier wird nur ein Programm auf dem Arduino gestartet, da keine Funktionen auf dem PC gebraucht werden.

```
#main
analogWrite(11,255)
```

Hier läuft das eigentliche Programm auf dem PC, es wird aber trotzdem ein Programm auf dem Arduino gestartet, das auf Befehle vom PC (hier **`analogWrite()`**, also „schalte LED an“) wartet.

```
# Das ist ein Kommentar
```

Kommentare werden mit **#** eingeleitet und werden vom Transpiler nicht berücksichtigt.

## Variablen

Variablen werden mit dem Datentyp und einem Namen definiert und ihnen kann ein Wert zugewiesen werden.

```
int x = 42
float y = 2.0
float z = x / y
string s = "Hello World"
```

Um Datentypen ineinander umzuwandeln, können folgende Funktionen verwendet werden:

```
float y = 3.14
int z = int(y)
float w = float(z)
```

## Arrays

Für alle dieser Datentypen können mit folgender Syntax auch Arrays erstellt werden.

```
int[] a = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
float[] b = [3.14,4.2]
string[] c = ["hello", "world"]
```

Arrays können auch nur mit der Länge ohne Werte initialisiert werden.

```
int[10] a  # integer Array mit der Länge 10
float[4] b
string[2] c
```

Auf einzelne Elemente von Arrays kann mithilfe von Indices, die bei 0 beginnen, zugegriffen werden.

```
int[] array = [42, 15, 4, 8, 23, 16]
print(array[0]) # -> 42
print(array[3]) # -> 8
```

## Builtins

### print

Die **`print()`** Funktion gibt wie in Python Werte aus. Mehrere Werte können mit einem Komma getrennt werden. Bei Arrays und Listen werden die einzelnen Werte ausgegeben.

```
print("Hello World") # Hello World
print("Hello", "World") # Hello World
int[] a = [1, 2, 3]
print("Array", a) # Array [1, 2, 3]
```

Wenn diese Funktion auf dem Arduino ausgeführt wird und der Arduino mit dem PC verbunden ist, dann wird die Ausgabe in der Konsole auf dem PC angezeigt.

### len

Mit der **`len()`** Funktion kann die Länge von Arrays bestimmt werden.

```
int[] array = [42, 15, 4, 8, 23, 16]
print(len(array)) # 6
```

### delay

Die **`delay()`** Funktion hält das Programm für eine bestimmte Zeit in Millisekunden an. Dabei wird aber trotzdem auf Anweisungen von jeweils Arduino oder PC gewartet, die dann während dieser Zeit auch ausgeführt werden können.

```
delay(1000) # warte 1 Sekunde
```

### analogWrite

Steuert die Pins am Arduino mit einem Wert von 0 - 255 an. Der Arduino, der eigentlich nur digitale Pins besitzt kann die Spannung bestimmter Pins, die mit einer Welle markiert sind, durch Pulsweitenmodulation (PWM) steuern. Dadurch kann zum Beispiel die Helligkeit von LEDs gesteuert werden.

```
for i in range(255):
    analogWrite(11,i)
    delay(10)
```

Die LED an Port 11 wird langsam heller, egal ob das Programm auf dem PC oder auf dem Arduino ausgeführt wird.

### analogRead

Liest die Spannung an den analogen Ports des Arduino in einem Wertebereich von 0 – 1023 aus.

```
print(analogRead(0))
```

## Kontrollstrukturen

**`if`** Bedingungen, **`while`** Schleifen und **`for`** Schleifen funktionieren gleich wie in Python.

```
if i > 5:
    print("i ist größer als 5.")
elif i >= 0:
    print("i ist größer oder gleich wie 0")
else:
    print("i ist kleiner als O")
```

Wenn die erste Bedingung wahr ist, dann wird der erste Teil hinter dem **`if`** Befehl ausgeführt. Wenn dies nicht der Fall ist werden nacheinander die Bedingungen der **`elif`** Segmente überprüft. Wenn eine der Bedingungen wahr ist, wird der entsprechende Programmteil ausgeführt. Ansonsten wird der **`else`** Teil ausgeführt.

Eine **`while`** Schleife wird ausgeführt, solange eine Bedingung wahr ist.

```
int x = 0
while x < 10:
    x = x + 1
    print(x)
```

Diese Schleife gibt die Zahlen von 1 bis 10 aus.

Die **`for`** Schleife iteriert mit einer Zählvariable über einen Bereich von Zahlen, der mit **`range`** festgelegt wurde. Das Grundlegende Schema sieht folgendermaßen aus:

```
for Zählvariable in range(start,end,step):
    #code
```

Beispiele:

```
for i in range(4,20,3):
    print(i)
```

Diese **`for`** Schleife gibt die Zahlen von 4 bis 20 in 3er Schritten aus.

**`for`** Schleifen können auch über Arrays iterieren.

```
int[] array = [42, 15, 314, 69, 20]
for i in array:
    print(i)
```

Diese **`for`** Schleife gibt die Elemente aus dem Array einzeln aus.

# Getting Started

Um Pyduino Programme zu schreiben kann die VS Code Erweiterung verwendet werden. Diese ist zurzeit nur mit Windows kompatibel.

Als erstes muss VS Code installiert werden. Die Installationsdatei kann unter https://code.visualstudio.com/download heruntergeladen werden.

Danach kann die Pyduino-Erweiterung installiert werden. Dafür muss zuerst an der Seitenleiste der „Extensions“ Tab ausgewählt werden. Mit der Suchfunktion kann die Erweiterung unter „pyduino“ gefunden werden. Mit einem Klick auf „Install“ wird die neuste Version der Erweiterung installiert.

Um ein Pyduino-Programm zu schreiben, muss dann in VS Code eine Datei mit der Endung „.pino“ erstellt werden. Der Pyduino-Code wird dann automatisch auf Fehler überprüft, die dann rot unterstrichen sind, und es wird Syntax-Highlighting angewendet. Um das Programm auszuführen kann dann im Menu „Run and Debug“ die „Pyduino“ Konfiguration gestartet werden. In der Debug-Konsole sind dann die Ausgaben von PC und Arduino zu sehen.

## Zusammenfassung

Als Ergebnis habe ich nach einigen Monaten Entwicklung die Programmiersprache Pyduino in einer ersten Version so weit umgesetzt, dass sie lauffähig ist und grundlegende Funktionen bereits implementiert sind. Die Syntax ist von Python inspiriert und so für Anfänger deutlich einfacher zu erlernen als C++.

Ein Pyduino Programm ist in zwei Teile, den **`#main`** und den **`#board`** Teil, aufgeteilt. Der **`#main`** Teil wird auf dem PC ausgeführt, der **`#board`** Teil wird auf dem Arduino ausgeführt. Ein wichtiger Vorteil von Pyduino ist, dass die beiden Programmteile über die serielle Verbindung miteinander kommunizieren können. Dadurch kann der PC auf die Pins des Arduino zugreifen und der Arduino kann Werte in der Konsole am PC ausgeben.

Pyduino kann auf jedem Windows-System verwendet werden, auf dem VS Code und Python installiert sind. Es muss lediglich die VS Code Erweiterung für Pyduino installiert werden. Dann können Pyduino-Programme erstellt werden, bei denen die Erweiterung Syntax-Highlighting und Fehlererkennung zur Verfügung stellt.

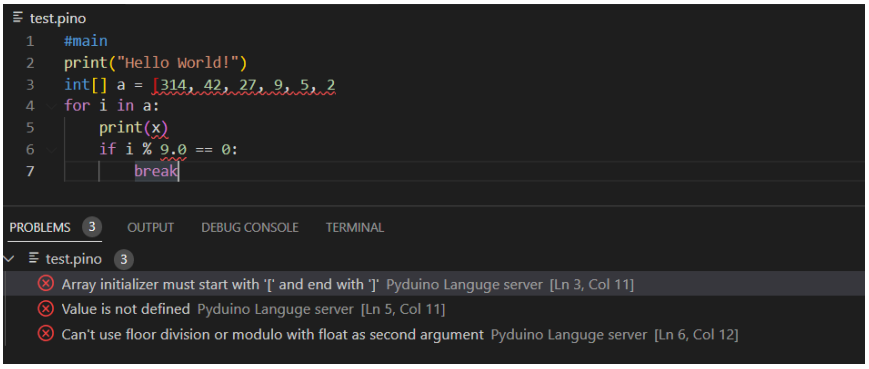



*Abbildung 8 Pyduino Datei in VS Code*

Diese Programme lassen sich mit der automatisch erstellten Run-Konfiguration einfach ausführen.

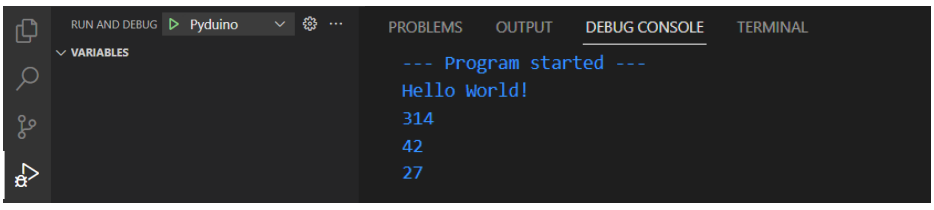



*Abbildung 9: Pyduino Run Konfiguration* *Abbildung 10: Ausgabe in der Konsole*

Der Quellcode der Sprache ist auf Github unter https://github.com/Bergschaf/Pyduino-Plugin veröffentlicht.

## Ausblick

Ein nächster Schritt für Pyduino ist es, neue Features zu implementieren. Dabei kommen zum Beispiel weitere Datentypen wie **`char`**, **`short`**, **`long`**, **`double`** und **`string`** in Frage. Für diese Datentypen können, wie in Python, dann auch verschiedene Builtin-Funktionen implementiert werden. Beispiele dafür wären

**`string.find(value)`**, **`string.strip()`** oder **`array.sort()`**. Als weiteren Datentyp sind verkettete Listen geplant. Diese würden es, im Gegensatz zu Arrays, einfach machen Elemente anzuhängen oder zu entfernen. Um Datentypen ineinander umzuwandeln, sollen auch Builtin-Funktionen wie zum Beispiel **`int(wert)`** oder **`string(wert)`** implementiert werden. Ein weiteres wichtiges Ziel sind Funktionen. Diese sollen, wie in Python, Argumente und optionale Keyword-Argumente erhalten und in der Lage sein, Werte zurückzugeben. Im Gegensatz zu C++ soll es auch ohne weiteres möglich sein, Datenstrukturen wie Arrays und Listen in Funktionen als Rückgabewert zu verwenden.

Ein weiterer Schritt ist es, die Kommunikation zwischen dem Arduino und dem PC weiter auszubauen. Dafür sollen zum Beispiel Befehle wie **`digitalRead()`** und **`digitalWrite()`** umgesetzt werden. Wünschenswert sind außerdem Funktionsaufrufe zwischen den Plattformen. Mit ihnen könnte zum Beispiel der Arduino Funktionen auf dem PC aufrufen, um die Leistung des PCs zu nutzen aber dabei trotzdem in der Lage zu sein, schnell auf Änderungen an den Pins zu reagieren. Zusätzlich dazu könnte man auch Builtin-Funktionen implementieren, die den Zugriff auf die Variablen der jeweils anderen Plattform ermöglichen.

Der Austausch zwischen PC und Arduino könnte durch asynchrone Anfragen effizienter gemacht werden. Dabei würden Befehle oder Funktionsaufrufe an die jeweils andere Plattform geschickt, ohne auf die Antwort zu warten. Das Programm würde dann weiterlaufen, bis der Wert wirklich gebraucht wird. Erst dann würde die Antwort abgewartet werden, falls sie noch nicht da ist.

Ein wichtiger Vorteil des Arduino ist, dass viele Bibliotheken existieren, die in die Programme eingebunden werden können. Sie stellen zum Beispiel Funktionen für LCD-Displays oder spezielle Sensoren zur Verfügung. Da die Pyduino-Programme für den Arduino in C++ übersetzt werden, könnte auch Unterstützung für Bibliotheken, die in C++ geschrieben sind, eingebaut werden. Damit könnte man alle Arduino-Bibliotheken in Pyduino Programme einbinden und ihre Funktionen nutzen.

In der Zukunft könnte man sich auch damit beschäftigen, Unterstützung für Objektorientierung, also zum Beispiel Klassen und Vererbung, zu implementieren. Das hätte den Vorteil, dass es so möglich wäre, Objektorientierung direkt mit Pyduino zu lernen. Diese Fähigkeiten könnten dann reibungslos zu Python übertragen und dort weiter genutzt werden.

Um das Problem der Wartezeit, bis Arduino Programme kompiliert und hochgeladen sind, vollständig zu lösen könnte der Arduino beim Debugging auch nur als Steuereinheit für die Pins verwendet werden. Dabei würde am Anfang ein Programm auf den Arduino hochgeladen werden, in dem nur auf Befehle über die serielle Verbindung gewartet wird. Auf dem PC könnten dann Pyduino Programme ausgeführt werden, die auf den Arduino zugreifen, ohne jedes Mal beim Ausführen eines Programms einen Teil auf den Arduino hochladen zu müssen.

Es gibt aber Situationen, in denen das Hochladen der Programme auf den Arduino nicht zu vermeiden ist. Um die Wartezeit in solchen Situationen, in denen auch PC-Programme kompiliert werden müssen, möglichst zu reduzieren, wäre es möglich die Aufgaben mithilfe von Multiprocessing auf verschiedene Prozessorkerne zu verteilen und so parallel auszuführen.

Um das Programmieren mit Pyduino weiter zu erleichtern, wäre es möglich in dem VS Code Plugin neben Syntax Highlighting und Fehlererkennung Autovervollständigung zu implementieren. Das könnte mithilfe der Daten, die der Transpiler über das Programm sammelt, ebenfalls über den Language Server implementiert werden.

Um Pyduino robuster und weniger Fehleranfällig zu machen, wäre eine Möglichkeit, mehr Unit Tests zu schreiben, die dann möglichst alle Features des Transpilers abdecken.

Außerdem besteht die Möglichkeit, die Pyduino VS Code Erweiterung als separate IDE zu veröffentlichen, die mit einer einzigen „.exe“ Datei installiert werden kann. Dafür würde die Theia Plattform [Q10] in Frage kommen, mit der auch die Arduino IDE 2.x implementiert ist. Das würde die Installation weiter vereinfachen.

# Dank

An dieser Stelle will ich Matthias Ruf und Benno Hölz danken, die mir am SFZ immer meine Informatik-Fragen beantwortet haben und mich auch bei der Langfassung auf gute Ideen gebracht haben. Außerdem gilt mein Dank meinem Betreuer, Herrn Beck, der mich bei meinem Projekt begleitet hat.

# Quellen

[Q1] https://www.schule-bw.de/service-und-tools/bildungsplaene/allgemein-bildende-schulen/bildungsplan-2016/beispielcurricula/gymnasium/BP2016BW_ALLG_GYM_NWT_BC_8-10_BSP_1.pdf

[Q2] https://www.ihk.de/blueprint/servlet/resource/blob/5556990/a9943739149bb694dba56dd7d272b678/h22-3280-b1-data.pdf

[Q3] https://en.wikipedia.org/wiki/Arduino#Sketch

[Q4] https://github.com/arduino/arduino-ide

[Q5] https://code.visualstudio.com

[Q6] https://code.visualstudio.com/api/language-extensions/overview

[Q7] https://microsoft.github.io/language-server-protocol/overviews/lsp/overview/

[Q8] https://github.com/openlawlibrary/pygls)

[Q9] https://www.arduino.cc/reference/en/language/functions/communication/serial/available/

[Q10] https://theia-ide.org

Abbildung 1-6: selbst erstellt mit Lucidchart (https://www.lucidchart.com/pages/de)

Abbildung 7: selbst erstellt mit Drawio (https://app.diagrams.net)

Abbildung 8, 9, 10: Screenshots aus VS Code

PC: CPU: AMD Ryzen 5 5600X @3,7GHz 6-Cores 12-Threads; RAM: 16gb; OS: Windows 11

Laptop: CPU: Intel Pentium N3710 @1,6GHz 4-Cores 4-Threads; RAM: 4gb; OS: Windows 10