Eigenes Tooling

- comPYner
- spike-prime-connect

comPYner

Warum comPYner?

Wir benutzen SPIKE Legacy/v2 als Firmware für unseren Wettbewerbsroboter. Dies bringt viele Vorteile, aber auch einige Nachteile mit sich.

Eines der größten Probleme mit der offiziellen LEGO-Firmware ist, dass wir nur eine Datei hochladen können. Den Code tatsächlich in eine einzelne Datei zu schreiben, macht Wartung und Entwicklung sehr mühsam.

Bisher hatten wir die (inoffizielle) VSCode Extension für SPIKE Prime verwendet, und natürlich waren wir nicht die einzigen mit diesem Problem. So gab es zum Beispiel einen PR (#58) vom FLL-Team GreenSubMarine, indem die Funktion hinzugefügt wurde, mit `from <file> import *` den Inhalt von `<file>` an dieser Stelle einzufügen, sodass man den Code in mehrere Dateien aufteilen kann, die später zu einer einzelnen zusammengeflickt werden.

Während das die Situation bereits verbessert, wurden wir durch diesen PR dazu angeregt, das Ganze weiterzudenken. Wer schon länger Python programmiert, weiß, dass ein Start-Import vermieden werden sollte, damit man Funktionen nicht überschreibt und keine Namespace-Pollution betreibt. Was wäre also, wenn wir die anderen Import-Typen ermöglichen?

Wie funktioniert comPYner?

ComPYner zu Entwickeln hat mehrere Ansätze gebraucht. Die meisten haben in CPython wunderbar funktioniert, allerdings nicht auf dem Prime Hub.

Wir schauen uns die verschiedenen Wege an diesem Beispiel an:

```
# module.py


def greet(name: str):
    print("Hello", name)

# main.py
import module


module.greet("comPYner")
```

1. Funktionen

Eine Idee war es, den Inhalt jeder importierten Datei in eine Funktion zu schreiben. diese würde dann `locals()` zurückgeben. So könnte man dann zum importieren die Funktion aufrufen und das Ergebnis in einer Variable speichern. So würde das obenstehende Beispiel umgewandelt werden.

```
def _import_module():
  def greet(name):
      print("Hello", name)

  return locals()

def _import_main():
  module = module()
  module.greet("comPYner")

_import_main()
```

Sowohl in der Theorie, als auch in CPython funktioniert das — jedoch nicht in Micropython.
In Micropython gibt `locals()` immer `{}` zurück, da die Namen lokaler Variablen nicht gespeichert werden.

2. Zurücksetzen des globalen Namespace

Weiterhin war eine Überlegung, jede Datei hintereinander zu schreiben, `globals()` zu speichern und danach alle Variablen in `globals()` zu löschen. Das hätte so ausgesehen:

```
_modules = {}

def greet(name):
    print("Hello", name)

_module = globals()
for key in _module:
    if _key == "_modules":
        continue
    delete globals()[key]

_modules["module"] = _module

module = module()
module.greet("comPYner")
```

```
_module = globals()
for key in _module:
    if _key == "_modules":
        continue
    delete globals()[key]

_modules["main"] = _module
```

Dieser Ansatz funktioniert jedoch nicht einmal in CPython, da alle globalen Variablen einer jeden Datei nur beim initialisieren verfügbar sind. Zu dem Zeitpunkt, wo `greet()` aufgerufen wird, sind alle anderen Variablen in `module.py` bereits gelöscht. Würde `greet()` z.B. eine andere Funktion aufrufen, die ebenfalls in `module.py` liegt, wäre diese zum Zeitpunkt des Aufrufs bereits gelöscht und nurnoch unter `_modules["module"][name]` zu finden.

3. Folgende Versuche

Darauffolgende Versuche waren darauf konzentriert, die Probleme von 2. zu lösen, ohne viel grundlegend zu ändern. Das Problem war hier, das es mit solchen halbherzigen Lösungen wie in Funktionen globale Variablen durch etwas wie `_modules["module"][name]` auszutauschen sehr schwierig ist das Verhalten von CPython zu reproduzieren. Es gab hier viele Probleme, auf die ich jedoch nicht weiter eingehe.

4. Wie es jetzt funktioniert

ComPYner erstellt für jedes Modul ein Dictionary, auf dessen Werte mit Punkten zugegriffen werden kann.

```
class c_Module(dict):
    def __init__(self, name=None):
        super().__init__()
        self['__name__'] = name

    def __getattr__(self, key):
        return self[key]

    def __setattr__(self, key, value):
        self[key] = value

    def __delattr__(self, key):
        del self[key]
```

```
    def __repr__(self) -> str:
        return '<Module %s (comPyned)>' % self.get('__name__', 'unknown')


c_module_module = c_Module("module")
```

Dann werden mithilfe von `ast` alle globalen Variablen im Modul ermittelt. Globale Variablen werden dabei an diesen Merkmalen erkannt:

1. Eine globale Variable wird definiert.
2. Eine Variable wird mit dem `global` Keyword verwendet.
3. Eine Funktion wird auf äußerster Ebene definiert.
4. Eine Klasse wird auf äußerster Ebene deklariert.
5. Eine Variable heißt `__name__`

Als nächstes wird vor alle Verwendungen der gefundenen globalen Variablen ein `c_module_module` (je nach Name des Moduls) gesetzt. Bei der Definition von Funktionen und Klassen wird die Klasse mit einem temporären Namen definiert, im Modul gespeichert, und dann wieder aus dem globalen Namespace gelöscht. Das sieht für unser `greet()` z.B. so aus:

```
def c_func_greet(name):
    print("Hello", name)


c_module_module.greet = c_func_greet
delete c_func_greet
```

Zusätzliche Features

Nachdem wir jetzt bereits einen Schritt hinzugefügt haben, der einen "Umwandler" benötigt, können wir doch noch ein paar tolle Features hinzufügen, oder?

RAM sparen

Der Prime Hub hat echt nicht viel Memory, vorallem da das Programm beim Ausführen selbst auch noch im RAM liegt.
Wir wollen aber möglicht viel Code draufkriegen. Was können wir machen, um RAM zu sparen?

Wir wissen ja bereits, dass Namen von globalen Variablen gespeichert werden .(Aufgrund des Programm-im-RAM-Dings sogar doppelt.) Also machen wir die Namen einfach möglichst kurz. Statt Module und temporäre Variablen für Funktionen und Klassen `c_module_compyned_polyfill_typing` zu nennen, können wir doch einfach `c_8Zzi` nehmen, und schon haben wir 27B gespart. Das ist natürlich erstmal nicht viel, aber bei sehr großen Programmen macht das schon etwas aus. Wir sparen in unserem Wettbewerbsprogramm 2,5kB nur an kompilierter Datei (.mpy), was die globalen Variablen selbst ja noch gar nicht einschließt.

Typing und Polyfills

Um in unserem Programm problemlos Typehints benutzen zu können, entfernt comPYner diese. So können wir, was typing angeht, Python 3.13 Features benutzen, welche der Hub an sich gar nicht unterstützt.

Außerdem haben wir die Option hinzugefügt mithilfe von Polyfills CPython-Funktionen in Micropython nutzen zu können.

```
# src/compyned_polyfills/enum.py
Enum = object


# src/compyned_polyfills/abc.py
# (C) pycopy-lib
class ABCMeta:
    pass

class ABC:
    pass

def abstractmethod(f):
    return f
```

comPYner sucht für jedes importierte Modul zuerst nach einem, welches `compyned_polyfill.name` heißt. Ist es vorhanden, wird es statt dem Original verwendet.

Debugging-Unterstützung

Durch unsere "Behandlung" ist der Code jetzt natürlich komplett durcheinander. Woher soll man Wissen, was in der zusammengefügten Datei in Zeile 798 steht?

Daher haben wir folgendes hinzugefügt: comPYner fügt vor jeder Codezeile `##file_path:line_no##` ein. Wenn ein Fehler auftritt kann man entweder in der `.cpyd.py` Datei nachsehen, woher die Codezeile kommt, oder es mit `compyner.engine.get_lineno_map(module).get(798, "unknown")` programmatisch herausfinden. Das benutzen wir auch in spike-prime-connect, wo wir Fehlermehldungen automatisch umwandeln.

@compile

Mit @compile wird eine Funktion beim Kompilieren, also beim Ausführen von comPYner ausgeführt und ihr Rückgabewert in ihrem Namen gespeichert.

```
# Beispiel
```

```
@compile
def zehn_stunden_in_millisekuden(in_file):
    return 10 * 60 * 60 * 1000

# wird zu:

zehn_stunden_in_millisekunden = 36000000
```

Wofür das viel nützlicher ist, ist das Laden von nicht-Python-Dateien. Wir haben zum Beispiel eine `config.yaml`, die über @compile geladen wird:

```
@compile
def _config_dict(in_file) -> dict[str, Any]:
    from pathlib import Path  # pylint: disable=import-outside-toplevel
    import yaml  # pylint: disable=import-outside-toplevel

    file: Path = Path(in_file).absolute().parent / ".." / "config.yaml"
    with file.open("r", encoding="utf-8") as f:
        return yaml.load(f, yaml.Loader)
```

So können wir unsere PID-Werte und weiteres ganz einfach mit einer yaml-Datei einstellen.

__glob_import__

comPYner ermöglicht die Nutzung der Funktion `__glob_import__(glob)` welche alle Dateien importiert, die dem `glob` matchen und stellt sie als Liste zur Verfügung. Beispiel:

```
modules = __glob_import__("import/*.py")

# (es gibt import/a.py und import/b.py)
# wird zu:

# ... comPYner imports ...

modules = [c_module_a, c_module_b]
```

Das verwenden wir, um unsere Runs zu Laden. Wir haben einen Ordner `runs`, in dem für jeden Run eine Python-Datei liegt. Diese werden alle über __glob_import__ importiert. Wenn im Menü ein Run ausgewählt wurde wird die `.run()` Funktion des jeweiligen Moduls gestartet.

Ausprobieren

Um comPYner zu testen, kann es einfach per PIP installiert werden:

```
pip install compyner
```

Anschließend kann es wiefolgt aufgerufen werden:

```
comypner <input_file>
```

- Das Ergebnis wird in `<input_file>.cpyd.py` gespeichert, ein alternativer Speicherort kann mit `-o <output_file>` festgelegt werden.
- Um die zufälligen Namen zu aktivieren, kann `--random-name-length 4` (oder eine andere Zahl) verwendet werden.
- Um das Setzen von `__name__` zu deaktivieren, wenn es nicht verwendet wird, kann `--reduce-dunder-name` verwendet werden.
- Um Module auszuschießen, kann `--exclude os --exclude sys` etc. verwendet werden. Module die ausgeschlossen werden, werden nicht in die Ausgabedatei eingeschlossen, sondern die ursprünglichen imports bleiben bestehen.

Python-API

comPYner kann auch von Python aus aufgerufen werden. Hier ein Beispiel:

```
import ast
from compyner.engine import ComPYner

input_file = "test.py"
output_file = "test.cpyd.py"

compyner = ComPYner()

with open(input_file, "r") as f:
    module = ast.parse(f.read())

output = compyner.compyne_from_ast("__main__", module, origin=input_file)

with open(output_file, "w") as f:
    f.write(output)
```

Zur Konfiguration nimmt `ComPYner()` einige Keyword-Arguments.

```
ComPYner(
    exclude_modules=[],       # A list of string names of modules not to compyne/bundle.
```

```
                            ## eg. os, sys, itertools, etc.
    require_dunder_name=True, # Whether __name__ should be set even if no use is found in the
module
    random_name_length=0,     # How many random characters to append to generated names
    keep_names=True,          # Whether to keep the original names even if random_name_length
is set
    module_preprocessor=None, # A function that is called with a modules ast and name
                            ## for each imported module. It should return an ast that is
used
                            ## instead of the original one. Can be used to remove comments,
etc.
    pastprocessor=None,       # A function that is called with the ast before unparsing
                            ## and should return an ast that is used instead
)
```

Weitere Inhalte zu comPYner

- GitHub
- PyPI
- Website (Generell zu Programmierung)

spike-prime-connect

spike-prime-connect ist das Tool, mit dem wir Programme auf unseren Roboter hochladen.

Wir haben sehr lange die inoffizielle VSCode Extension für SPIKE Prime verwendet, allerdings war dies mit comPYner nicht mehr möglich. (Seit dem fix von #65 ist es möglich, wir sind allerdings nicht wieder gewechselt. Mittlerweile unterstützt die VSCode Extension SPIKE Legacy auch nicht mehr)
Daher haben wir spike-prime-connect entwickelt.

spike-prime-connect ist ein CLI-Tool, mit dem man SPIKE Prime und MINDSTORMS Robot Inventor Hubs steuern kann, sofern SPIKE Legacy/v2 installiert ist.
Man kann mit nur einem Befehl auf einem verbundenen Hub

- Programme hochladen, verschieben und löschen
- Programme starten und beenden
- Diverse Geräteinformationen auslesen
- Ausgaben lesen
- Die REPL starten
- Den Hub neustarten und ausschalten

Beim Hochladen von Programmen verwendet spike-prime-connect im Hintergrund comPYner, sodass importierte Dateien automatisch mit-hochgeladen werden. Mehr Informationen dazu, warum wir comPYner nutzen und wie es funktioniert sind hier zu finden.
spike-prime-connect stellt comPYner dabei so ein, dass alle auf SPIKE Prime Hubs vorhandenen Module standartmäßig ausgeschlossen sind und Variablennamen möglichst kurz gehalten werden. Das Umbennen der Variablen kann mit `--debug` verhindert werden.

Links

- GitHub
- PyPI
- Website (Generell zu Programmierung)