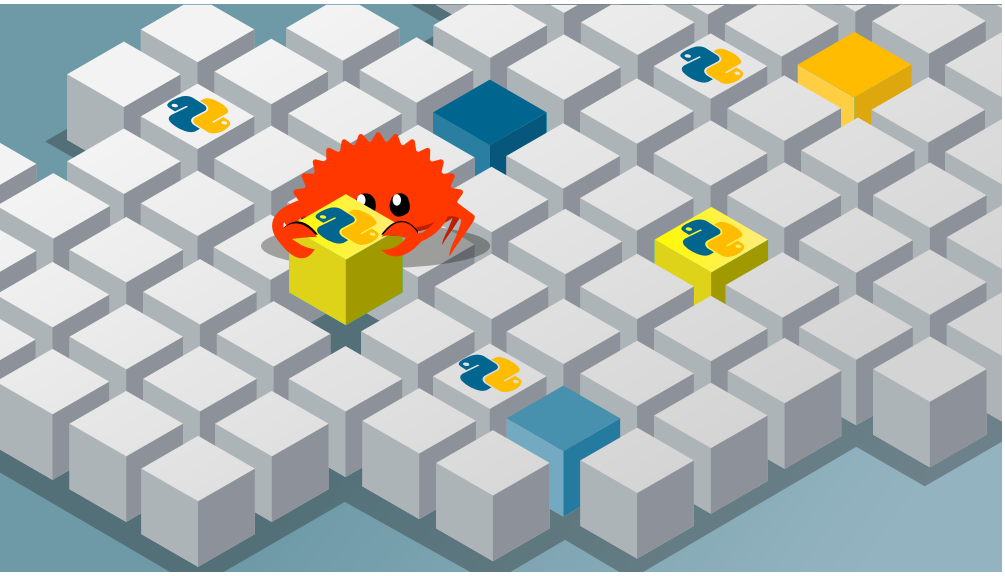


Python-Module mit Rust erstellen

Python fehlt es bei rechenintensiven Aufgaben an Performance. Rust hingegen liefert schnellen, kompilierten und sicheren Code. Die besten Eigenschaften aus beiden Welten zu kombinieren, ist jedoch eine Herausforderung.

Von Gerhard Völkl

Wer in Python aufwendige Berechnungen durchführt, für die es keine fertigen Module gibt, kann den betroffenen Algorithmus in ein mit C++ oder C geschriebenes Modul auslagern und kompilieren. Rechenmodule wie NumPy gehen diesen Weg. Man muss sich dabei jedoch um vieles selbst kümmern, was sonst Python erledigt, beispielsweise Referenzzähler und Speicherbelegung. Eine Alternative ist die Rust-Bibliothek PyO3. Sie erstellt eine komfortable Verbindung zu Rust, der speichersicheren Systemsprache, die mittlerweile in einigen Projekten C und C++ ablöst. Der Artikel gibt Python-Entwicklerinnen und -Entwicklern ohne Rust-Vorkenntnisse die Möglichkeit, die Kombination aus Python, Rust und PyO3 kennenzulernen.

Speicher automatisch verwalten ohne Garbage Collection

Rust ist eine systemnahe Programmiersprache, die von C++, Erlang, Haskell und anderen beeinflusst ist. An den vielen geschweiften Klammern ist die Abstammung von C++ und die Verwandtschaft zu Java und C# erkennbar. Rust verwendet eine eigenständige, nahezu automatische Speicherverwaltung ohne Garbage Collection. Es gibt feste Regeln, wer was im Speicher besitzt. Ist ein Speicher ohne Besitzer, gibt Rust den Speicher frei. Dadurch ist ein sicherer Speicherzugriff garantiert.

Um schnelle Programme zu erzeugen, verlagert Rust die Komplexität und das Prüfen der Korrektheit des Codes in Richtung Compiler. Dadurch sind die Übersetzungszeiten teilweise länger als bei anderen Programmiersprachen und der Compiler muss zur Übersetzungszeit schon viel wissen.

Da Rust manche Dinge etwas anders angeht als andere Programmiersprachen – beispielsweise kennt es weder Vererbung noch den Datentyp `None` –, ist der Einstieg nicht einfach. Es ist jedoch alles logisch aufgebaut. Rust erlaubt schnelles und zuverlässiges Multithreading. Viele Werkzeuge und Bibliotheken erleichtern Entwicklern die Arbeit.

Mit Python als Interpreter-Sprache kann man schnell neue innovative Software entwickeln, nicht zuletzt dank des gut entwickelten Ökosystems mit vielen Standardmodulen, vor allem in den Bereichen Data Science und maschinelles Lernen. Einerseits vereinfachen der Interpreter und das dynamische Typsystem die Programmierung, andererseits verlangsamen sie die Ausführungsgeschwindigkeit. Darüber hinaus tut sich Python durch die interne Struktur schwer: Der Global Interpreter Lock verhindert automatisches Multithreading. Deshalb bietet es sich an, die beiden Sprachen zu kombinieren: Python für das schnelle und interaktive Entwickeln und Rust für Performance und Multithreading.

iX-TRACT

- Der Interpreter und das dynamische Typsystem vereinfachen das Programmieren mit Python, machen Programme aber langsam. Rust hingegen ist schnell, sytemnah und erlaubt einfaches Multithreading.
- Die Rust-Bibliothek PyO3 liefert Rust-Bindings für Python.
- Mit PyO3 kann man native Python-Module in Rust erstellen und Python-Code aus Rust heraus aufrufen.

Rust, Python und PyO3: wie alles zusammenspielt

Das Framework PyO3 vereinfacht mit Werkzeugen und Bibliotheken das gemeinsame Verwenden von Rust und Python (Abbildung 2). Dabei gibt es unterschiedliche Herangehensweisen. In diesem Artikel ist das Ziel, ein Python-Modul durch ein in Rust geschriebenes und kompiliertes Modul zu ersetzen.

Rust ist einfach und kostenlos auf den Rechner zu bekommen. Bei macOS und Linux genügt der Befehl

```
curl --proto '=https' --tlsv1.2 ↵
      -sSf https://sh.rustup.rs | sh
```

Für Windows lädt man das Programm rustup-init.exe von der Rust-Forge-Webseite und startet es (siehe ix.de/ztd4). Damit sind Compiler und alle Werkzeuge auf dem Rechner verfügbar. Eine ausführliche Beschreibung der Installation und mehr liefert das frei verfügbare Rust-Einsteiger-Buch (siehe ix.de/ztd4).

Mit dem in Rust enthaltenen Paketmanager Cargo können Entwickler schnell ein neues Projekt erstellen. Im hier verwendeten Beispiel ist es die Bibliothek python-rust:

```
cargo new --lib python-rust
```

Im Verzeichnis python-rust findet man jetzt die Datei Cargo.toml, analog zur Make-Datei anderer Sprachen. Sie enthält die Beschreibung, wie Cargo vom Quellcode zum fertigen Programm kommt. Der Abschnitt `package` listet die Einträge des aktuellen Projekts:

```
[package]
name = "python-rust"
version = "0.1.0"
edition = "2021"
```

Der Abschnitt `lib` enthält die Einstellungen für die kompilierte Bibliothek, also das externe Python-Modul. Als Erstes steht der Name, den das `import`-Statement in Python verwendet. Die Bezeichnung `cdylib` erzeugt eine Python-kompatible Shared Library:

```
[lib]
name = "python_rust"
crate-type = ["cdylib"]
```

Unter `dependencies` findet man die Rust-Bibliotheken des Projekts:

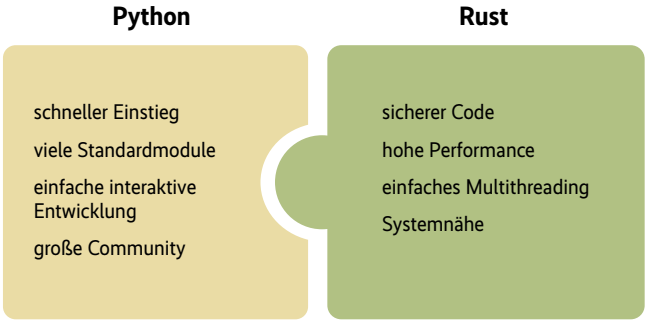
```
[dependencies]
pyo3 = { version = "0.16.4", ↵
    features = ["extension-module"] }
```

Als hilfreich beim Entwickeln von Rust mit Python erweist sich das Werkzeug Maturin. Es kompiliert Rust-Code zu einer Python-Bibliothek und legt sie so ab, dass Python sie importieren kann.

Primzahlensuche mit dem Sieb des Eratosthenes

Weil sich Rust vor allem für Python-Module mit vielen Berechnungen und komplexen Algorithmen empfiehlt, widmet sich das Beispielprogramm der Primzahlensuche. Es ermittelt die Anzahl der Primzahlen bis zu einer bestimmten Obergrenze. Ein Array von 2 bis zur Obergrenze enthält die Information, ob eine Zahl eine Primzahl ist oder nicht. Alle Elemente des Arrays haben zu Beginn den Wert `True`, da davon auszugehen ist, dass jede Zahl eine Primzahl ist. Der unter dem Namen „Das Sieb des Eratosthenes“ bekannte Algorithmus nimmt die erste Zahl mit dem Wert `True` und setzt alle durch die Primzahl teilbare Zahlen auf den Wert `False`. Danach folgt die nächste Zahl, die noch den Wert `True` hat. Das geht so weiter, bis das Array abgearbeitet ist.

Die äquivalente Python-Funktion `count_prime_python` enthält eine Liste mit True/False-Werten und Schleifen. Sie ist im Modul python_python.py untergebracht. Die Rust-Funktion `count_prime_rust` im Modul python_rust unterscheidet sich auf den ersten Blick nicht allzu sehr (Listing 1). Sie ähnelt Pythons Typangaben. Rust arbeitet für die Programmstruktur mit geschweiften Klammern, Python hingegen mit Einrückungen.



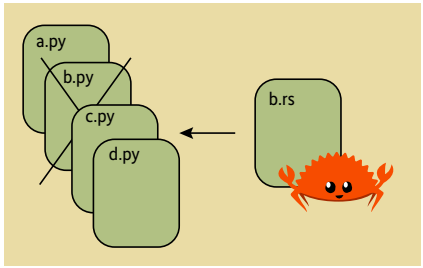
Python und Rust ergänzen sich: Python ist einfach und Rust performant (Abb. 1).

Will man die Rust-Funktion `count_prime_rust` in Python verwenden, muss man sie mit PyO3 in ein Python-Modul einbetten. Der Sourcecode liegt im bereits erzeugten Projekt python_rust der Datei /source/lib.rs (Listing 2).

Die erste Zeile bindet alle Elemente der Bibliothek PyO3 ein, die die fertige Bibliothek braucht. Sie entspricht einem Import-Statement in Python. Den Funktionen, auf die Python zugreifen soll, stellt man die Zeichenkette `#[pyfunction]` voran (Code-Annotation). Den Rest erledigen Rust und der Compiler. Ein prozedurales Makro erzeugt automatisch den Code, den Python von einer Funktion in einem externen Modul erwartet.

Damit Rust weiß, welche Funktionen und andere Programmobjekte in einem Python-Modul weitergegeben werden sollen, ist eine Funktion, die das beschreibt, mit der Zeichenkette `#[pymodule]` zu kennzeichnen. Der erste Parameter `_py` verbindet Rust mit Python (Python-Token) und der zweite Parameter `m` bezeichnet das neue Modul. Der Rückgabewert `PyResult` ist das Ergebnis der Verarbeitung: Steht er für einen aufgetretenen Fehler, löst er in Python eine Exception aus.

Die Befehle der Funktion `python_rust` definieren die Inhalte des Moduls. So fügt die Methode `add_function` dem Modul `m` die Funktion `count_prime_rust` hinzu. Verläuft alles gut, ist der Rückgabewert `OK(())`.



Mit der Bibliothek PyO3 und ihren Werkzeugen lässt sich ein externes Python-Modul in Rust erzeugen (Abb. 2).

Zeitvergleich: Welches Modul ist schneller?

Das in Python geschriebene Modul braucht auf dem Testrechner 24,9 Sekunden für die Primzahlenermittlung bei einer Obergrenze von 10 Millionen. Um es mit Rust vergleichen zu können, ist erst ein Compilerlauf und dann das Übertragen an Python erforderlich. Das erledigt der Befehl

```
maturin develop
```

Verwendet man nun in Python das Rust-Modul, beträgt die Zeit für die Primzahlen (Grenze 10 Millionen) 36,9 Sekunden – und damit 10 Sekunden mehr als beim Python-Modul. Eigentlich sollte es schneller sein. Rust schleppt aber sehr viele Debug-Informationen mit. Zudem verzichtet der Compiler auf die letzte mögliche Optimierung, um die Zeiten für das Kompilieren möglichst gering zu halten. Daher ist es sinnvoll, vor dem Ausliefern die Option `--release` zu aktivieren. Dann fällt jeglicher Ballast weg und der Compiler optimiert alles:

```
maturin develop --release
```

Jetzt braucht das Rust-Modul nur noch 0,6 Sekunden – ein Vierzigstel der 24,9 Sekunden in Python.

Datentypen sind mit PyO3 frei wählbar

Rust- und Python-Datentypen haben Ähnlichkeiten, unterscheiden sich aber in einigen Fällen (siehe Tabelle). PyO3 lässt Entwicklerinnen und Entwicklern bei der Angabe von Datentypen die Wahl, sich für die von Rust oder Python zu entscheiden. Verwendet man Python-Datentypen in Rust, ist zwar keine Konvertierung erforderlich, PyO3 greift aber auf die Python-Runtime zurück. Bei den Rust-Datentypen setzt PyO3 hingegen die Python-Inhalte in Rust um. Entwickler haben dann Zugriff auf alle Funktionen des Rust-Ökosystems.

Ein weiterer Unterschied zwischen Rust und Python ist das Error Handling.

Listing 1: Die Funktionen count_prime_python und count_prime_rust zum Ermitteln der Primzahl im Vergleich: der Python-Code links enthält Einrückungen, Rust auf der rechten Seite geschweifte Klammern

```python
def count_prime_python(
    limit: int
) -> int:

    prime = [True for _ in range(limit + 1)]

    i = 2

    while (i*i <= limit):
        if prime[i] == True:
            for j in range(i*i,limit+1,i):

                prime[j] = False


        i += 1


    count = 0

    for i in range(2, limit+1):
        if prime[i]:
            count += 1

    return count
```

```rust
fn count_prime_rust(
    limit: usize
) -> usize {

    let mut prime = vec![true; limit + 1];

    let mut i = 2;

    while i*i <= limit {
        if prime[i] == true {
            for j in ((i * i)..(limit + 1)).step_by(i) {

                prime[j] = false
            }
        };
        i += 1;
    }

    let mut count = 0;

    for i in 2..(limit+1) {
        if prime[i] {
            count += 1
        }
    }
    count
}
```

Rust erledigt das konsequent über Rückgabewerte und kennt keine Exceptions. PyO3 wandelt Fehlerwerte automatisch in Python-Exceptions um, sie sind aber genauso explizit verwendbar, wie das Beispiel für eine Teilen-durch-null-Exception zeigt (Listing 3).

Rust-Konzepte wie Ownership, Borrowing und Lifetime erlauben es zu bestimmen, wann ein bestimmter Speicherbereich wieder freigegeben ist. In Python verweist hingegen eine Variable auf einen Speicher. Mehrere Variablen gleichzeitig können ihn verwenden und ändern. Der Garbage Collector kümmert sich darum, dass der Speicher wieder frei ist.

Neben Funktionen lassen sich in PyO3 auch Klassen und deren Methoden in Rust auslagern (Listing 4). Die Python-Beispielklasse `Player` steht dabei für eine Sportlerin oder einen Sportler mit einer bestimmten Punktzahl.

```
c = Player("susi",100)
```

erzeugt ein neues Objekt und

```
>print(c.name)
`susi`
```

liefert den Wert zurück.

PyO3 verwendet Strukturen statt Klassen

In Rust gibt es keine Klassen. Deshalb verwendet PyO3 an dieser Stelle Strukturen (`struct`), die Attribute und Methoden enthalten können (Listing 5). Damit Python auf die Strukturen zugreifen kann, müssen Entwicklerinnen und Entwickler sie mit `#[pyclass]` kennzeichnen. Vor den Feldern `name` und `points` geben sie dabei an, ob Python die Strukturen lesen (`get`) oder ändern (`set`) kann. Damit ist eine explizite Zugriffssteuerung möglich.

Was fehlt, ist noch eine Methode, die die Struktur initialisiert. In Rust definiert man daher innerhalb des `impl`-

Listing 2: Rust-Modul lib.rs für Primzahlen

```rust
use pyo3::prelude::*;  // Elemente von PyO3 einbinden

#[pyfunction]
fn count_prime_rust(
    limit: usize
) -> usize {
...
}

#[pymodule]
fn python_rust(_py:Python, m: &PyModule) -> PyResult<()> {
    m.add_function(wrap_pyfunction!(count_prime_rust, m)?)?;  ↵
                                                    // Befehle für Modulinhalt
    Ok(())
}
```

PyO3 konvertiert Python-Datentypen automatisch in Rust und erlaubt auch, mit den ursprünglichen Datentypen weiterzuarbeiten

Python	Rust	PyO3 Python-nativ (Rust)
Object	-	&PyAny
str	String, Cow<str>, &str, OsString, PathBuf	&PyUnicode
bool	bool	&PyBool
int	jeder Integer-Typ (i32, u32, usize etc.)	&PyLong
float	f32, f64	&PyFloat
list[T]	Vec<T>	&PyList
dict[K, V]	HashMap<K, V>, BTreeMap<K, V>, hashbrown::HashMap<K, V>2, indexmap::IndexMap<K, V>3	&PyDict
tuple[T, U]	(T, U), Vec<T>	&PyTuple

Codeabschnitts eine Methode (hier `new`), die in Python `__init__` entspricht. In der Moduldefinition (`python_rust`) fügt man mit der Methode `add_class` Klassen hinzu, ähnlich wie bei den veröffentlichten Funktionen:

```
m.add_class::<PlayerRust>()?;
```

In Python kann dabei eine Klasse spezielle Methoden besitzen, die mit einem doppelten Unterstrich _ beginnen. Sie haben ebenfalls ihre Entsprechung in PyO3, etwa bei der Methode `__repr__` zur Ausgabe als Zeichenkette oder `__eq__` zum Vergleich von Objekten. Darüber hinaus implementiert PyO3 Klassenmethoden und -attribute respektive statische Methoden.

Die Tücken beim Ausliefern des Moduls

Solange Entwickler das Python-Modul auf demselben Rechner erstellen und verwenden, lässt es sich leicht mit den gezeigten Werkzeugen ausliefern. Etwas schwieriger verhält es sich, wenn das Modul auf eine andere Plattform soll. Dann sind drei Aspekte ausschlaggebend: die Python-Version auf der Anwendungsplattform, deren Betriebssystem (macOS, Windows, Linux) und die vorhandene Prozessorarchitektur (ARM, Intel, Apple Silicon). Kombiniert man die einzelnen Aspekte, beispielsweise drei Python-Versionen auf drei verschiedenen Betriebssystemen und zwei unterschiedlichen Prozessorarchitekturen, kommt man auf achtzehn Auslieferungen (Python-Wheels).

Das Python-Verschlüsselungsmodul cryptography von Paul Kehrer und Alex Gaynor, das täglich 3,5 Millionen Downloads hat und Rust verwendet, stellt derzeit über zwanzig vorkompilierte Auslieferungen zur Verfügung. Zusätzlich gibt es eine Sourcecode-Variante, die es erlaubt, das Modul auf weiteren Plattformen selbst zu kompilieren. Ein weiteres Werkzeug von PyO3 ist setuptools-rust. Es erweitert die Python-Konfigurationsdatei setup.py um zusätzliche Schlüsselwörter, um Module in Rust auszuliefern. Dies ist zwar aufwendiger als die bisher gezeigten Werkzeuge, bietet aber mehr Möglichkeiten, Rust in den Auslieferungsweg des Python-Moduls einzubeziehen.

Ausblick

Rust erweist sich als interessante Ergänzung zu Python und könnte in den nächsten Jahren C und C++ ablösen. Wer nicht mit Extension-Modulen arbeiten möchte, kann die Kommunikation mit Python und Rust weiterdenken. Mit PyO3 lässt sich Rust in Python einbinden und vice versa auch Python in Rust. (nb@ix.de)

Quellen

Downloadmöglichkeiten, Links zu Tools und Rust-Buch: ix.de/ztd4

Listing 3: Exception: Teilen durch null

```
#[pyfunction]
fn teilen(arg_1: i32, arg_2: i32) -> PyResult<i32> {
    match arg_1.checked_div(arg_2) {
        Some(result) => Ok(result),
        None => Err(PyZeroDivisionError::new_err("division by zero"))
    }
}
```

Listing 4: Python-Klasse Player

```
from functools import total_ordering

@total_ordering
class Player:
    def __init__(self,name:str, points:int)->None:
        self.name = name
        self.points = points

    def __lt__(self, other:Player)->bool:
        return self.points < other.points

    def __eq__(self, other:Player)->bool:
        return self.points == other.points

    def __repr(self)->str:
        return f'Player: name={self.name} points={self.points}'
```

Listing 5: Rust-Klasse PlayerRust

```
#[pyclass]
struct PlayerRust {
    #[pyo3(get,set)]
    name: String,
    #[pyo3(get,set)]
    points: i32
}

#[pymethods]  // Struktur initialisieren
impl PlayerRust {
    #[new]
    fn new(name:String, points:i32) -> PlayerRust {
        PlayerRust{name,points}
    }

    fn __repr__(&self) -> String {
        format!("PlayerRust(name={}, points={})", self.name, self.points)
    }

    fn __richcmp__(&self, other: &Self, op: CompareOp) -> PyResult<bool> {
        match op {
            CompareOp::Lt => Ok(self.points < other.points),
            CompareOp::Le => Ok(self.points <= other.points),
            CompareOp::Eq => Ok(self.points == other.points),
            CompareOp::Ne => Ok(self.points != other.points),
            CompareOp::Gt => Ok(self.points > other.points),
            CompareOp::Ge => Ok(self.points >= other.points),
        }
    }
}
```

GERHARD VÖLKL

ist Fachjournalist für Softwareentwicklung, Data Science, Spieleprogrammierung und Computergrafik.