Inhaltsverzeichnis

Willkommen in Python

Unser erstes Programm

Unser erstes Programm könnte so aussehen:

```
# Ask the user for their name
name = input("What's your name? ")

# Greet the user
print(f"Hello {name}")
```

Nichts als Text

Ein **Programm** besteht aus einer Reihe von Anweisungen, die von einem Computer ausgeführt werden. Ein Programm wird als reiner Text geschrieben, und üblicherweise verwenden wir für jede neue Anweisung eine separate Zeile. Um ein Programm zu schreiben, benötigen wir lediglich einen einfachen Texteditor. Allerdings bevorzugen wir eine vollständige Entwicklungsumgebung wie Visual Studio Code, da sie viele hilfreiche Funktionen wie Syntaxhervorhebung, automatische Vervollständigung und Debugging-Tools bietet, die das Programmieren erleichtern.

Zeile für Zeile

Wenn ein Computer ein Programm ausführt, geht er Zeile für Zeile durch das Programm und führt aus, was jede Zeile - oder **Anweisung** - ihm vorgibt. Im Beispiel oben wird zuerst die Zeile

```
name = input("What's your name? ")
```

ausgeführt. Diese Zeile nutzt die Funktion `input()`, um eine Eingabe vom Benutzer abzufragen. Das Ergebnis wird in einer Variable namens `name` gespeichert. Anschließend geht der Computer zur nächsten Zeile, wo er die Anweisung

```
print(f"Hello {name}")
```

vorfindet. `print()` ist eine Funktion, die eine Nachricht auf der Konsole ausgibt.

Kommentare

Vielleicht fragst du dich, was es mit den beiden Zeilen auf sich hat, die mit einem #-Symbol beginnen?

```
# Ask the user for their name
# Greet the user
```

Diese Zeilen sind **Kommentare** und gehören nicht zum eigentlichen Programm. Der Computer überspringt sie – sie dienen ausschließlich uns Menschen als Hilfestellung. Ein gut platzierter Kommentar kann das Verständnis des Codes erheblich erleichtern.

Nicht nur sequenziell

Im Beispiel oben werden die Befehle Zeile für Zeile ausgeführt – das ist der Normalfall in einem Programm. Wir sprechen dann auch von einer sequenziellen Ausführung. Es gibt jedoch spezielle Anweisungen wie **Schleifen**, **Kontrollstrukturen** und **Funktionen**, die den Computer veranlassen, Befehle in einer anderen Reihenfolge auszuführen. Mit diesen Anweisungen kann der Computer bestimmte Aktionen wiederholen oder Codezeilen überspringen. Keine Sorge, wenn dir diese Konzepte noch nicht ganz klar sind – wir werden sie bald näher kennenlernen.

Ein Programm ausführen

Um ein Python-Programm auszuführen, benötigen wir einen Python-Interpreter – das Programm, das unseren Python-Code Zeile für Zeile liest und ausführt. Diesen musst du für dein Betriebssystem herunterladen und wie jedes andere Programm installieren. In Visual Studio Code können wir ein Python-Programm einfach mit einem Klick auf den „Play"-Button oder durch Drücken der Taste F5 starten. Ich persönlich bevorzuge jedoch den Weg über die Kommandozeile (oder Terminal), da wir so die volle Kontrolle über die Ausführung unseres Programms haben. Ein weiterer Vorteil: Wir lernen dabei wichtige Kommandozeilenbefehle kennen, die in der Softwareentwicklung unverzichtbar sind. Mach dich also mit dem Terminal vertraut.

Öffne direkt ein Terminal in Visual Studio Code über das Menü "Terminal" → "New Terminal". Bei Windows-Systemen solltest du darauf achten, dass es sich um ein "Command Prompt" (cmd) handelt. Bei Linux-Systemen heißt es "Bash", auf einem Mac-Rechner heisst es einfach "Shell".

Wenn unser Programm in einer Datei namens `my_first_program.py` im aktuellen Verzeichnis gespeichert ist, können wir es mit dem `python`-Befehl ausführen:

```
python my_first_program.py
```

Nach Eingabe des Namens und Drücken der Enter-Taste erscheint eine freundliche Begrüßung.

Befehle in Python

Eingebaute Befehle

In unserem ersten Programm haben wir bereits zwei wichtige Befehle kennengelernt, die wir in Python verwenden können:

- `input()`: Liest einen Text über die Tastatur ein.
- `print()`: Gibt einen Text auf der Konsole (Terminal) aus.

Beide Befehle stehen uns direkt zur Verfügung, da sie Teil der Standard-Python-Installation sind. Neben diesen beiden Befehlen gibt es noch viele weitere solche vordefinierten Funktionen.

Befehle aus mitgelieferten Modulen

Neben den standardmäßig verfügbaren Befehlen gibt es in Python auch spezialisierte Funktionen, die erst importiert werden müssen. Ein gutes Beispiel ist die Funktion `sqrt()` zur Berechnung von Quadratwurzeln. Diese Funktion ist im `math`-Modul enthalten, das zwar mit Python installiert wird, aber erst durch eine explizite Import-Anweisung in unserem Programm verfügbar wird. Dafür verwenden wir folgenden Befehl:

```
from math import sqrt
```

Anschließend können wir diese Funktion in unserem Programm nutzen, um eine Quadratwurzel zu berechnen:

```
square_root = sqrt(16)
```

Auf diese Weise laden wir aus dem Modul `math` nur die einzelne Funktion `sqrt`. Alternativ können wir auch alle Funktionen des Moduls `math` auf einmal laden:

```
import math
```

Wenn wir die Funktion auf diese Weise laden, müssen wir den qualifizierten Namen verwenden und den Modulnamen durch einen Punkt getrennt angeben:

```
square_root = math.sqrt(16)
```

In Abbildung 1 werden weitere Beispiele für Befehle aus sogenannten *built-in modules*, also in Python eingebauten Modulen, gezeigt. Außerdem sehen wir ganz rechts eine dritte Kategorie von Funktionen: solche, die über externe Module eingebunden werden.

Befehle aus externen Modulen

Die externen Module sind nicht Bestandteil der Python-Installation und müssen daher getrennt installiert werden. Dafür verwenden wir den mitgelieferten Paketmanager Pip:

```
pip install tinkerforge
```

Das Beispiel oben installiert, wenn wir den Befehl im Terminal eingeben, das Modul mit dem Bezeichner `tinkerforge`. Dabei handelt es sich um eine Bibliothek, die es uns ermöglicht, mit der Tinkerforge-Hardware zu kommunizieren, die wir im Rahmen des LiFi-Projektes verwenden. Nach der Installation können wir die Funktionen des Moduls in unserem Programm verwenden, genauso wie wir es bei den eingebauten Modulen getan haben. Solche externen Module erweitern die Funktionalität von Python erheblich und ermöglichen uns den Zugriff auf verschiedenste Hardware und Dienste.

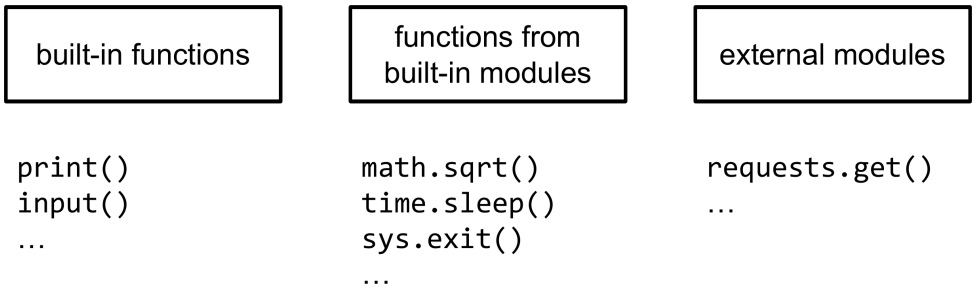

Abbildung 1: Unterschiedliche Arten von Befehlen in Python.

Argumente eines Befehls

Ein Befehl (oder eine Funktion) in Python kann Argumente entgegennehmen – das sind Werte, die wir dem Befehl zur Verarbeitung übergeben. Diese Argumente werden in runden Klammern nach dem Befehlsnamen angegeben. Bei der Funktion `print()` ist das zum Beispiel der Text, den wir ausgeben möchten, und bei `sqrt()` ist es die Zahl, deren Quadratwurzel wir berechnen wollen.

Es gibt auch Befehle, die kein Argument benötigen. Die `input()`-Funktion ist ein Beispiel dafür – obwohl wir ihr bisher immer einen Text in den Klammern mitgegeben haben, damit der Benutzer weiß, was er eingeben soll:

```
name = input("What's your name? ")
```

Auf der Konsole erscheint dann der Text "What's your name?", gefolgt von einem blinkenden Cursor, der die erwartete Eingabe signalisiert.

Wir können die Funktion auch ohne Argument aufrufen:

```
name = input()
```

In diesem Fall erscheint nur der blinkende Cursor ohne Text. Wichtig: Auch wenn wir kein Argument übergeben, müssen wir die runden Klammern angeben. Sie sind das universelle Zeichen für einen Funktionsaufruf.

Rückgabewerte eines Befehls

Programme erzeugen nach dem EVA-Modell (Eingabe-Verarbeitung-Ausgabe) Ergebnisse. In Programmiersprachen wie Python können nicht nur Programme, sondern auch einzelne Befehle oder Funktionen Ergebnisse produzieren. Diese Ergebnisse nennen wir Rückgabewerte.

Ein typisches Beispiel ist die `input`-Funktion:

```
name = input("What's your name?")
```

Diese Funktion wartet auf eine Benutzereingabe und gibt den eingegebenen Text als Rückgabewert zurück. Mit dem Zuweisungsoperator (`=`) speichern wir diesen Rückgabewert in der Variable `name`, um ihn später im Programm weiterverwenden zu können. Das Konzept der Rückgabewerte und Variablen werden wir in den folgenden Kapiteln noch ausführlicher behandeln.

Fehler

Wenn wir ein Python-Programm ausführen, können verschiedene Fehler auftreten. Die häufigsten sind **Syntaxfehler**, die entstehen, wenn wir uns nicht an die Regeln der Python-Sprache halten, zum Beispiel wenn wir vergessen, Klammern zu schließen oder Einrückungen falsch setzen. **Laufzeitfehler** treten dagegen erst während der Programmausführung auf, etwa wenn wir versuchen, durch Null zu teilen oder auf nicht vorhandene Dateien zuzugreifen.