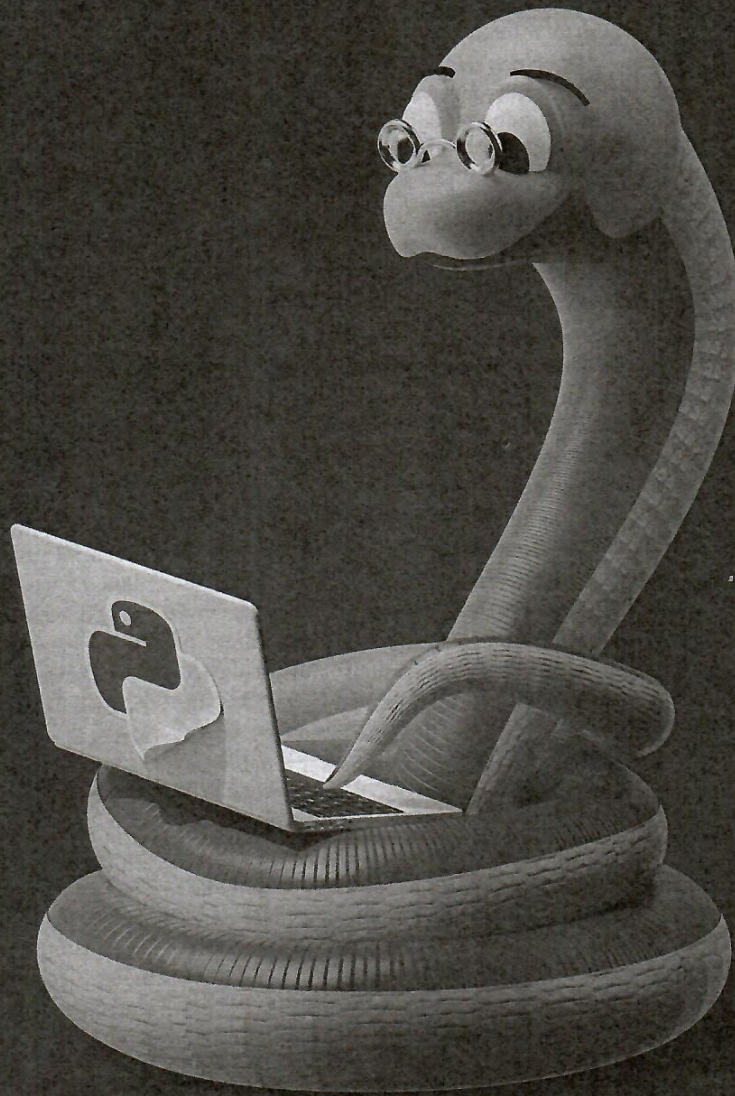


Python für alle

Gute Gründe für Python



Gute Gründe für Python	Seite 16
Python schnell und einfach einrichten	Seite 20
Python-Entwicklungsumgebungen	Seite 26

Python ist leicht lesbar und stößt daher gerade Anfängern die Tür in die Welt des Programmierens auf. Wir haben zusammengetragen, wo die Vorteile der Sprache liegen, welche Python-Edition Sie wie am schnellsten auf den Rechner bekommen und welche Entwicklungsumgebungen sich eignen.

Von Wilhelm Drehling und Pina Merkert

Eine Programmiersprache wie C erspart Entwicklern, Maschinencode zu tippen und der Compiler übersetzt auf jedem Rechner in den Dialekt der eingebauten CPU. Trotzdem nutzt so maschinennahes Programmieren noch jedes Byte – und der Code zerfällt in Speicherzugriffsfehler, wenn nur ein Bit falsch gesetzt ist.

Seit der Erfindung von C ist die Informatik glücklicherweise weit gekommen. Eine moderne Programmiersprache wie Python nimmt Entwicklern Fehlerquellen wie unsinnige Speicheradressen oder doppelt freigegebenes RAM. Programmieren geht mit Python nicht nur schneller und braucht weniger Zeilen Code; die Sprache erlaubt es, den Blick ganz auf den Algorithmus zu richten, ohne sich vorstellen zu müssen, was jeder Befehl mit den Registern der CPU macht.

Diese Bequemlichkeit hat ihren Preis: Python-Code braucht ungefähr 100-mal länger für dieselbe Aufgabe wie ein C-Programm. Was wie eine Rechenzeit-Katastrophe klingt, ist in der Praxis kaum eine Einschränkung. Wer nämlich große Datenmengen durch die Vektoreinheiten seiner vielkernigen CPU peitschen möchte, sollte ohnehin optimierte Bibliotheken einsetzen. Für Python wäre das beispielsweise NumPy, das hervorragend optimiert ist und mühelos den Code der allermeisten C-Entwickler überholt.

Beliebt und universell

Python-Code ist gut lesbar, leicht verständlich, ästhetisch und vor allem kurz. Diese Eigenschaften ziehen Entwickler an, weshalb die Sprache in den letzten zehn Jahren im Ranking von RedMonk mit

GitHub und Stack Overflow als Quelle durchweg unter den beliebtesten vier landete (siehe ct.de/y33e).

Ein wichtiger Faktor für den Erfolg ist aber auch die Vielseitigkeit. Durch die knappe und effiziente Syntax eignet sich Python für kurze Skripte und Spielereien auf der interaktiven Konsole. Da Python selbst in C programmiert ist, gibt es Python-Wrapper für die meisten in C geschriebenen Bibliotheken. Stehend auf den Schultern von Giganten gehen Programmierexperimente mit Python besonders schnell, weshalb Python in der Wissenschaft beliebt ist. Und da nichts so lange hält wie ein Provisorium, bleibt viel Python-Code aus dem Prototyp im späteren Produkt erhalten.

Beim maschinellen Lernen wird dies besonders deutlich: Alle großen KI-Frameworks bieten primär eine Python-Schnittstelle und Data Science wird überwiegend in Python gemacht. Jupyter-Notebooks (siehe S. 26) visualisieren die Experimente, die Rechenleistung für die Python-Zellen im Browser kommt aus der Cloud.

Am Raspi ist Python die Sprache der Wahl für Elektronikbasteleien, in Docker-Containern treibt die Sprache Webanwendungen an. Lediglich Spiele (abgesehen von Minispielen mit Pygame) und Smartphone-Apps werden nicht mit Python programmiert. Trotz der beiden Ausnahmen: So vielseitig ist keine andere Programmiersprache. Python ist eine Sprache für alle!

Schlüsselbund

Wie kurz und effizient Python-Code ist, zeigt unser gar nicht triviales Beispiel im Kasten: eine vereinfachte Implementierung der asymmetrischen Verschlüsselung RSA. Nachdem Sie Python installiert haben (siehe S. 20), fehlt nur PyCrypto, was Sie mit `pip install pycryptodome` installieren. Danach können Sie die wenigen

Zeilen aus dem Kasten abtippen (empfehlenswerte Editoren und IDEs stellen wir auf Seite 26 vor) und ausführen. Falls Sie nicht tippen wollen, finden Sie den Quellcode auch im Repository auf GitHub, zu finden über ct.de/y33e.

RSA nutzt aus, dass eine Primfaktorzerlegung großer Zahlen schwierig ist, während sich Multiplikationen leicht berechnen lassen [1]. Python macht die Rechnung besonders leicht, weil die Sprache automatisch mit beliebig langen Ganzzahlen rechnet. Eine Beschreibung der mathematischen Grundlagen des Verfahrens finden Sie über ct.de/y33e.

In den ersten beiden Zeilen definiert der Code die beiden Variablen `p` und `q`. Die beiden Primzahlen dienen als Grundlage für den öffentlichen und privaten RSA-Schlüssel. In einer realen Anwendung wären beides sehr große Primzahlen, das Beispiel zeigt aber, dass RSA auch mit kleinen Zahlen funktioniert. Mit den Variablen kann Python danach rechnen, beispielsweise um eine weitere Variable mit dem Ergebnis zu belegen:

```
phi = (p - 1) * (q - 1)
```

Unter dem Kommentar in Zeile 9 # Schlüsselgenerierung steht ein Aufruf der Funktion `inverse()`. Einer der Vorteile von Python sind mächtige Bibliotheken, die Sie mit `import` in Ihren Code laden. Die Funktion findet die modulare Inverse von `e`. Der erweiterte euklidische Algorithmus, der diese Zahl berechnet, ist nicht ganz trivial, glücklicherweise aber schon fertig im Python-Modul `crypto.Util.number` implementiert. Es folgen zwei Aufrufe der Funktion `print()`, die die generierten Schlüssel auf die Konsole schreiben. Die Funktion `format()` ersetzt die Platzhalter für Ganzzahlen `{d}` mit den Werten der Variablen, die `format()` als Parameter erhält.

Nachricht

Von der Konsole akzeptiert Python mit `input()` auch Eingaben. Das Beispiel erwartet hier, dass Benutzer eine Zahl eintippen, die dann verschlüsselt wird:

```
nachricht = int(input("Beliebige Zahl"
+ " zwischen 0 und 221 eintippen: "))
```

RSA verschlüsselt immer nur Zahlen und keine Texte. Die Python-Zeile gibt erst den String mit der Aufforderung aus, sammelt die eingegebenen Zeichen bis zum Enter ein

Implementierung der asymmetrischen Verschlüsselung RSA

```

01 from Crypto.Util.number import inverse
02
03 # Basiswerte
04 p = 13
05 q = 17
06 n = p * q
07 phi = (p - 1) * (q - 1)
08
09 # Schlüsselgenerierung
10 e = 11
11 d = inverse(e, phi)
12 print("Öffentlicher Schlüssel: ({:d},{:d})".format(e, n))
13 print("Privater Schlüssel: ({:d},{:d})".format(d, n))
14
15 # Nachricht
16 nachricht = int(input("Beliebige Zahl zwischen 0 und 221 eintippen: "))
17
18 # Verschlüsseln
19 cypher = pow(nachricht, e, n)
20
21 # Entschlüsseln
22 print(str(pow(cypher, d, n)))

```

Der Code generiert zuerst ein Schlüsselpaar. Anschließend benutzt er die Werte aus dem öffentlichen Schlüssel, um eine Benutzereingabe zu verschlüsseln. Mit dem privaten Schlüssel lässt sich daraus wieder die ursprüngliche Eingabe berechnen.

und konvertiert sie mit der Funktion `int()` in eine Ganzzahl. Tippt ein Benutzer etwas anderes als Ziffern, schlägt die Konvertierung fehl und Python wirft eine `Exception`, also eine Fehlermeldung, die auf der Konsole landet. Wenn die Konvertierung klappt, steht die Zahl in der Variablen `nachricht` zur Verfügung.

Beliebig große Potenzen berechnet `pow()`. Mit einem dritten Parameter rechnet die Funktion das Ergebnis modulo der dritten Variable. `pow(a, b)` ergibt also a^b , während `pow(a, b, c)` in einem Schritt $a^b \% c$ rechnet. Das Verschlüsseln und Entschlüsseln mit RSA erledigt Python also jeweils in einem einzigen Funktionsaufruf:

```


# Verschlüsseln
cypher = pow(nachricht, e, n)
# Entschlüsseln
print(str(pow(cypher, d, n)))

```

Bei der Entschlüsselung `pow(cypher, d, n)` kommt wieder eine Zahl heraus. Die `print()`-Funktion erwartet aber eine Zeichenkette als Parameter. Die Konvertierung übernimmt die Funktion `str()`, die in Python jeden Datentypen in einen String verwandeln kann. Wenn alles geklappt hat, steht am Ende die Zahl auf dem Bildschirm, die Sie vorher eingetippt haben.

Ausprobieren!

Probieren Sie unser RSA-Beispiel unbedingt mal aus! Verschlüsseln Sie dafür nicht nur verschiedene Zahlen, sondern machen Sie auch mal falsche Eingaben und bauen Sie auch Fehler in den Code ein. Dabei merken Sie, wie Python die Fehler meldet und versucht, Sie beim Debugging zur richtigen Zeile zu schicken. Sie dürfen das Beispiel auch gern erweitern, um beispielsweise Fehleingaben zu erkennen. Nur für ernsthafte Verschlüsselung sollten Sie unsere Python-Fingerübung nicht verwenden. Dafür gibt es gut getestete Implementierungen in Bibliotheken.

Wenn Sie Inspiration suchen, sollten Sie ein Blick auf die Kästen mit unterschiedlichen Anwendungszwecken werfen. Darin geben wir einen groben Überblick der Flexibilität, die Sie mit Python bekommen können: Sei es Server mit Python automatisieren, den Raspi aufpeppen, Daten in Jupyter Notebooks visualisieren oder mit Django Ihrem Webprojekt neues Leben einhauchen. (wid@ct.de) 

Literatur

- [1] Wilhelm Drehling, Reingefallen, Asymmetrische Verschlüsselung: Sicher durch Falltürfunktionen, c't 7/2021, S. 60

Quellcode und Mathematik: ct.de/y33e



Automatisierung mit Skripten

Zu automatisieren gibt es im Alltag (nicht nur als Admin) eine Menge – und Linux, Windows und macOS kann man mit jeweils eigenen Skripten steuern. Aber Bash für Linux ist nicht gerade intuitiv, BAT-Dateien unter Windows nicht gerade flexibel und die Syntax der PowerShell nicht jedermanns Sache. Python ist eine würdige Alternative für kleine und große Helferlein und verhält sich vor allem auf verschiedenen Betriebssystemen fast gleich. Wer zum Beispiel automatisch Dinge aus dem Web herunterladen will, schreibt in Python mit `http.client` ein übersichtliches und wartungsfreundliches Skript, statt mit elendig langen `curl`-Befehlen zu hantieren.

Steigen die Anforderungen, stehen die Bibliotheken `requests`, `json` und `yaml` bereit. Die setzen jeden noch so exotischen HTTP-Request ab und lesen und schreiben die wichtigsten Markup-Formate für Konfigurationsdateien. Mit so mächtigen Werkzeugen und der Flexibilität einer vollwertigen Programmiersprache ändert ein Skript bei Bedarf ruckzuck Dutzende von Config-Files. Und mit `subprocess` setzt Python ohne Probleme jeden Befehl ab, für den man sonst eine Konsole bräuchte.

Im Admin-Alltag kann man beim Automatisieren mit Python noch einen Schritt weitergehen: Das populäre Automationswerkzeug `Ansible` ist in Python geschrieben und steuert zum Beispiel Server anhand von wiederholbaren Rezepten. Mit dem Python-Code hinter den Kulissen hat man erst dann zu tun, wenn man eigene Module schreiben möchte. Dem Automatisierungswunsch sind dann keine Grenzen gesetzt. (jam@ct.de)

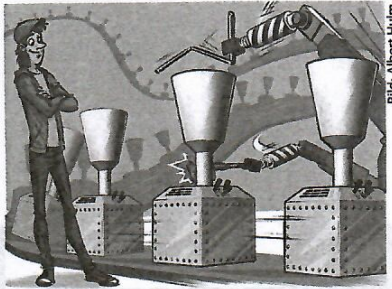


Bild: Albert Hulm

Große und kleine Raspis

Von Anfang an, also seit der erste Raspberry 2012 auf den Markt kam, gehen Python und der Himbeer-Computer Hand in Hand. Python steckt nicht nur in den Schaltkreisen, sondern auch im Namen: Pi für Python Interpreter.

Die ursprüngliche Idee hinter dem Raspi lag darin, in Schulen mehr Begeisterung für Informatik auszulösen. Obwohl der Raspberry als Schulcomputer scheiterte, war der Minicomputer trotzdem ein Markterfolg – und zwar bei den Bastlern.

Vom handlichen Desktop-PC bis hin zur überteuerten Steuerungseinheit konnte der Raspi die unterschiedlichsten Aufgaben erfüllen. Mit dem Raspi lassen sich erste Programmiererfahrungen in Python sammeln und schnell sichtbare Ergebnisse erzielen wie blinkende LEDs.

Ob auf dem großen Raspberry Pi, dem Raspberry Pi Zero oder dem winzigen Raspberry Pi Pico, Python ist das Mittel der Wahl, wenn es um hardware-nahe Programmierung geht. Das liegt vor allem an den zahllosen Bibliotheken: Die Python-Gemeinde, aber auch Komponentenhersteller wie Adafruit und SparkFun bieten sie fertig zum Download an, womit Sie Servocontroller, OLED-Displays, Schrittmotoren und vieles mehr ansteuern können.

Das erleichtert nicht nur die ersten Schritte in Python, auch erfahrene Programmierer profitieren von der Vielzahl an Bibliotheken, denn sie müssen sich nicht etwa um das korrekte Timing der GPIO-Pins kümmern, um einen Sensor anzusteuern, sondern können sich auf das Wesentliche konzentrieren und die Daten verarbeiten. Mit MicroPython auf dem Raspberry Pi Pico portieren Sie außerdem leicht Programme vom Raspberry Pi auf einen Mikrocontroller, ohne eine neue Programmiersprache erlernen zu müssen. (mid@ct.de)



Bild: Albert Hulm

Daten analysieren und visualisieren

Selbst größere Datenmengen lassen sich über Pythonskripte sammeln, verarbeiten, analysieren und visualisieren. So packt man die Corona-Rohdaten des RKI in ansehnliche Diagramme, wertet Serverlogs und Labordaten aus oder untersucht, welche Themen in einer Textsammlung besonders häufig vorkommen.

Mit den passenden Bibliotheken spricht Python ebenso mit SQL-Datenbanken wie mit REST-APIs und liest beliebige Datenformate ein – oder man kratzt sich Zahlen und Texte einfach selbst aus dem Netz zusammen. Im nächsten Schritt helfen die Bibliotheken NumPy und Pandas, die Daten zu bereinigen und zu analysieren. Matplotlib und Plotly generieren Grafiken, interaktive Diagramme und Live-Dashboards.

Viele Studenten und Forscher schreiben ihre Python-Skripte in der Programmierumgebung Jupyter Notebook. Darin mixt man ausführbare Codeschnipsel und visualisierte Ergebnisse mit Texten zur Dokumentation und Interpretation. Jupyter Notebooks eignen sich außerdem prima, um gemeinsam an einem Projekt zu arbeiten oder Kollegen seine Ergebnisse zu präsentieren.

Die meisten KI-Forscher veröffentlichen ihren Code als Jupyter-Notebooks. KI geht mit Python besonders gut, weil alle wichtigen Frameworks für die Entwicklung neuronaler Netze mit Python-Code laufen. Die Ergebnisse der Experimente befüllen Diagramme, die direkt im gleichen Notebook unter dem Code stehen. Die Notebooks stehen meist auf GitHub zum Download bereit, sodass sich viele KI-Experimente auch von Laien reproduzieren lassen. (acb@ct.de)



Bild: Albert Hulm

Webentwicklung mit Django

Für ziemlich jede C-Bibliothek hat schon jemand einen Python-Wrapper geschrieben. Dass Python über diesen Weg von über Jahrzehnte ausentwickelter Open-Source-Software profitiert, merkt man an großen Frameworks wie Django.

Das Web-Framework Django nutzt jede SQL-Datenbank, arbeitet mit allen Webservern zusammen und vertraut auf die verlässlichsten Crypto-Bibliotheken. Auf den Schultern von Giganten zu stehen, ist bei Django aber kein Balanceakt, da das Framework alles so einpackt, dass es perfekt in die Ästhetik und logische Struktur der Python-Syntax passt.

Django-Webanwendungen harmonisieren ausgezeichnet mit modernen Single-Page-Applikationen, mit Angular oder React. Für schnelle Erfolge gibt es aber auch eine integrierte Template-Engine. Besonders viel Zeit spart das automatisch generierte Admin-Backend, das Django mit minimalem Aufwand aus den Datenbankmodellen zusammenpuzzelt. Dank unproblematischer Python-Wrapper um Mongo-DB und Redis lässt sich auch die Performance leicht optimieren.

In der Summe bedeutet das, dass man mit Django rasend schnelle Webanwendungen entwickelt, die eine logische Struktur haben, deren Code verständlich bleibt und die sich modular testen lassen. Django lässt einem bei den Datenbankmodellen und den URLs maximale Freiheit, weshalb die Projekte letztlich doch umfangreiche Programme aus Dutzenden Dateien in mehreren Ordnern werden. Ein ausreichend großes Zeitbudget und eine gute Entwicklungsumgebung (siehe S. 26) sind daher Pflicht. (pmk@ct.de)