

Sicheres Programmieren in Python

Tipps und Tricks

Inhaltsverzeichnis

| | |
|--|-----------|
| Einleitung | 3 |
| 1 Skripts | 4 |
| 1.a Empfehlungen | 6 |
| 1.b Benutzbarkeit | 8 |
| 1.c Passwörter | 10 |
| 1.d Eingabenüberprüfung | 12 |
| 1.e Debugging | 14 |
| 2 Softwareprojekte | 15 |
| 2.a Code Repository | 16 |
| 2.b Bibliotheken | 18 |
| 2.c Code Qualität | 19 |
| 2.d Projektstruktur | 20 |
| 3 Ausgewählte Lösungen | 21 |
| 3.a Client-Server Kommunikation | 22 |
| 3.b Web Services Nutzen | 25 |
| 3.c Implementierung Server | 26 |
| 3.d Excel | 27 |
| 3.e Datenanalyse und -visualisierung | 28 |
| Anhang | 30 |
| Styleguides | 30 |
| Cheat Sheets | 30 |

Einleitung

Die zentrale Frage ist: **Wann und wie macht Coden Spaß?** Dazu ist gut zu wissen: Programmierer:innen lieben effizient und selbstbestimmt zu arbeiten.

1. Sie wollen sich bei typischen Problemen schnell und effektiv **selbst helfen** können. Das verleiht Sicherheit, bereitet Freude und macht Lust auf mehr.
2. Sie pflegen **produktive Faulheit**, indem sie sich wiederholende Abläufe automatisieren.
3. Sie wollen aber auch nicht unbedingt alles neu selbst erfinden. So nutzen sie **existierenden Code** von Libraries, Packages oder Code-Snippets.
4. Sie stellen an sich den Anspruch Code zu **verstehen** bevor sie ihn weiterverwenden.
5. Selbst bei kleineren Aufgaben ist **Lesbarkeit und Wartbarkeit** von Code für Programmierer:innen immer ein wichtiges Thema.
6. Sie **lieben Fehler** die sie machen. Warum? Weil sie auftretende Fehler selbst beheben können, da die Python Fehlermeldungen leicht zu verstehen, hilfreich und konstruktiv sind.
7. Sie entwickeln Software **in kleinen Schritten**. Das bringt viele Erfolgserlebnisse und verringert ganz nebenbei die Gefahr von „*Overengineering*“.
8. Sie wissen, dass für Skripts oft **wenige Zeilen Code genügen**. Sie nutzen umfangreichere Modellierung und objektorientierte Programmierung (OOP) erst bei größeren Projekten.
9. Je nach vorliegendem Problem greifen sie auf unterschiedliche Konzepte der Programmierung zurück. Zum Beispiel auf deklarative Programmierung beim Pattern-Matching mit regulären Ausdrücken „*regex*“ und auf funktionale Programmierung mit „*Thinking in Lists*“ bei umfangreichen textuellen Eingabedaten.
10. Sie legen auch Wert auf **Performance** und optimieren bzw. parallelisieren **bei Bedarf**.

Zum Aufbau des Workshops

Im Workshop wird „*hands-on*“ gearbeitet. Im ersten Schritt versucht man als Teilnehmer:in existierenden Code zu **lesen**. Da der vorgestellte Code nicht optimal und teilweise fehlerhaft ist, stellt man Vermutungen über Probleme und deren Auswirkungen auf. Dann werden gemeinsam **Lösungen diskutiert** und Alternativen bewertet. Eine mögliche Lösung wird **ausprogrammiert**.

Hinweis: Fehler sind wertvoll! Fehler sind produktiv! Eine Empfehlung lautet, während des Workshops möglichst viele unterschiedliche Fehler zu machen. Aber anstatt schnell Anpassungen „auf Verdacht“ vorzunehmen, versucht man **zuerst** die Fehlermeldungen im Kontext zu verstehen und erst **daraufhin** zu beheben. Dadurch kann der größtmögliche Lerneffekt erzielt werden. Erfahrungsgemäß sinkt dadurch die Zeit für die Fehlerbehandlung dramatisch.

1 Skripts

Fast alle Entwickler:innen stehen anfangs vor einigen der folgenden Probleme und stellen sich die Fragen:

- **„How-to?“ oder „Wie soll ich beginnen?“** Wie kommt man von einem Problem zur Lösungsidee bis hin zum funktionierenden Code? Eine Möglichkeit ist, zuerst Demo-Daten vorzubereiten, dann den Ablauf in einem Skript als Kommentar zu beschreiben. Dann Schritt für Schritt die Kommentare durch echten Code zu ersetzen. Dabei hilft es möglichst nützliche Ausgaben über die aktuellen inneren Zustände während des Programmablaufes zu produzieren oder mittels „*Debugging Breakpoints*“ den Code anzuhalten und den Zustand von Variablen zu inspizieren.
- **Wer benötigt Wartbarkeit bei nur wenigen Zeilen Code?** Im Sinne der Wartbarkeit ist es von Anfang an besser den Code und auch die Kommentare immer auf englisch zu verfassen. Der Source-Code wird für Menschen geschrieben, somit ist verständliche, sinnvolle Namensgebung extrem wichtig. Schöner Code macht Freude und ist lesbarer. Strukturierter Aufbau und selbst-erklärender Code ersparen umfangreiche und mühsame Dokumentation.
- **Was bedeutet Usability bei einem Kommandozeilenprogramm?** „*User Experience*“ und „*Usability*“ zählen auf mehreren Ebenen! Die Nutzer:innen des Codes sind einerseits die Anwender:innen, aber andererseits auch die Entwickler:innen. Für jene die den Code weiterentwickeln zählt die Lesbarkeit von Programmen, für Endnutzer:innen zählen einfaches Starten, sprechender Output und hilfreiche Fehlermeldungen, sowie auch gut gewählte Default-Werte und einfache Konfiguration.
- **Wieviel Flexibilität ist nötig?** Nutzer:innen wünschen sich unterschiedliche teilweise widersprüchliche Funktionalitäten. Für größtmögliche Flexibilität ist es sinnvoll vorhandene „*default values*“ durch Angabe von „*Command-Line Parameter*“ überschreiben zu können. Diese Parameter werden beim Start des Skripts angegeben. Daher hat es sich bewährt in einer eigenen Startup Datei – oft ein einzeliges Bash Skript – die Verwendung exemplarisch zu demonstrieren und damit auch zu dokumentieren.
- **Wer sollte den Code wiederverwenden?** Weiterverwendung – „*copy paste*“ – von existierenden Programmteilen („*Code Snippets*“) ist durchaus wünschens- und empfehlenswert; allerdings nur von Code, den man selbst gut versteht und somit anpassen bzw. abändern kann.

1 Skripts

Ein paar Gedanken zu **Datensicherheit, Einbruchssicherheit, Nachvollziehbarkeit** und **Privatsphäre**:

- **Security (Datensicherheit)**: Ist Backup was für Feiglinge? Ein Backup muss nicht mühsam sein. Auch kleine Skripts sollten an einer zentraler Stelle verwaltet und abgelegt werden. So ist jede Änderung nachvollziehbar und ein Programm kann leichter verteilt und aktualisiert werden, bzw. im Falle eines lokalen Datenverlusts wiederhergestellt werden. Use „*Use Git code repositories*“!
- **Security (Vertraulichkeit)**: Hard-coding von Passwörtern, Tokens und API Keys lässt sich durch Command-line Parameter und Konfigurationsdateien leicht vermeiden. „*Never hard code credentials*“!
- **Security (Hacking)**: Als unbedingte Notwendigkeit wird heute weiters das Prüfen von Benutzer-eingaben – Validierung – angesehen. Durch Vermeidung von gefährlichen API Aufrufen muss die unbeabsichtigte Ausführung von Schadcode verhindert werden. „*Never trust user data*“! „*Use Whitelisting for input validation*“!
- **Privacy**: Wer kann und darf Zugriff auf **sensible Daten** bekommen? Besser gar keine sensiblen Daten erheben. Wer liest die Logfiles? Oftmals kann man auch sehr gut mit anonymisierten Daten arbeiten! „*Follow the principle of data minimization*“! „*Use anonymisation*“!

1.a Empfehlungen

Summary: Grundlegende Empfehlungen („good practice“) für das Verfassen von Skripts zur Automatisierung von Abläufen: Sinnvolle Demo Daten, schrittweises Vorgehen, geringe Verschachtelungstiefe des Source Codes, sowie defensives Programmieren und mit fehlerhaftem Input rechnen.

Problem: Leider sind Programme manchmal wirklich schwer anzupassen und weiter zu verwenden. Wenn die Lesbarkeit mangelhaft ist, oder das Nachvollziehen des Algorithmus schwerfällt, dann getraut man sich nicht fremden Code zu ändern. Schlecht lesbarer Code und nichtssagende Fehlermeldungen machen keinen Spaß.

```
1 import sys, os
2 print("Wir starten jetzt!")
3 def diedatennunversenden(alle):
4     print("Versuch2; da bin ich noch nicht ganz fertig damit.")
5     k= sys.argv[-1]
6     diedatennunversenden( open("data-"+k+".txt").read() )
```

Please Improve



- **Empfehlung A: Start mit „guten“ Daten.** Das Generieren einer Liste von einigen „echten“ Demodaten – also auch mit üblichen Fehlern und menschlichen Ungenauigkeiten – hilft beim Durchdenken des Lösungswegs.

```
http://cloud.fh-joanneum.at:8080/images/sunflower.png
#...
http://cloud.fh-joanneum.at:8080/images/sunflower.svg
https://cloud.fh-joanneum.at:8080/images/sunflower.png
```

urls.txt

- **Empfehlung B: Schritt-für Schritt Umsetzung.** Man startet mit der Entwicklung des Ablaufes, des Algorithmus als Kommentar. Schrittweise können dann die Kommentare Zeile für Zeile durch echten Code ersetzt werden. Hilfreich ist das Skript möglichst oft – auch schon nach kleinen Änderungen – auszuführen, denn so treten jeweils nur wenige Fehler auf. Die auftretenden Fehler können auch leichter lokalisiert werden.

```
1 File name from command line arg (default to "urls.txt")
2 # Read image urls from file
3 # ...
```



- **Empfehlung C: Tiefe Verschachtelung vermeiden** durch Prüfen der Voraussetzung und Eingangsdaten am Beginn jeder Funktion. Bei nicht behebbaren Fehlern sofortiges Beenden des Programmes. „Exit early“!

```
1 if len(sys.argv) < 2:
2     exit("Usage down <filename>")
```



1.a Skripts

- **Empfehlung D: Sichere Abläufe durch Fehlervermeidung.** Beispielsweise empfiehlt es sich einen `with` Block zu verwenden um Ressourcen automatisch zu schließen. „*Defensive coding*“!

```
1 with open(fn) as f:
2     print( f.read())
```



Weitere Empfehlungen:

- Für die Übersichtlichkeit und das Lesen ist das Aufteilen von Code auf mehrere Funktionen und auch auf mehrere Dateien nützlich.
- Funktionen sollten nicht zu lang sein. Programmierer:innen können Funktionen im Umfang von bis zu einer halben Seite (bis maximal eine Seite) gut erfassen.
- Hilfreich ist ein klarer Einstieg ins Programm mit einer sehr kurzen, übersichtlichen Startroutine.

```
1  #!/usr/bin/env python3
2  import sys
3
4  def parse_the_args():
5      if len(sys.argv) > 1:
6          return sys.argv[1:]
7      return ["cities.csv"]
8
9  def process(file,idx,cnt):
10     print(f" {idx}/{cnt} Processing '{file}'...")
11     # much more code...
12
13  def download(inputfiles):
14     print(f"We use data from file '{inputfiles}'.")
15     for idx,file in enumerate(inputfiles,start=1):
16         process(file,idx,len(inputfiles))
17
18
19  if __name__ == "__main__":
20     infiles = parse_the_args()
21     download( inputfiles=infiles )
```



1.b Benutzbarkeit

*Summary: Benutzbarkeit (Usability) ist auch im Terminal für Konsolenapplikationen wichtig. User erwarten **hilfreiches Feedback** über die Art und Umfang der Erwarteten Eingaben, über den Systemzustand, über den aktuellen Fortschritt oder eines auftretenden Fehlers.*

Problem: Programme sind leider oft schwer oder umständlich zu bedienen. Vor allem, wenn die Verwendung nicht einfach, schnell und intuitiv ist. Zum Beispiel kann die Benutzbarkeit (Usability) durch fehlende Grundeinstellungen oder unverständliche Rückmeldungen für die Nutzer:innen niedrig sein. Dadurch werden diese Programme dann nicht gerne benutzt.

```
1 # Better optimise user feedback:
2 #   give context,
3 #   tell what is happening.
4 print("Wir inkrementieren den Zähler jetzt!")
```

Please Improve

- **Empfehlung A: Sprechende Namensgebung** Der Name eines Skripts (einer Applikation) sollte selbsterklärend sein.

```
> ./image_downloader.py urls.txt
```

- **Empfehlung B: Bereitstellen eines Start Skripts.** Ein Shell Start Skript soll vorhanden sein, damit die Verwendung klar ersichtlich ist. Vor allem, wenn Kommandozeilen Parameter nötig sind haben die User ein funktionierendes Beispiel zur Hand. Weiters ist das Bereitstellen von geeigneten Default-Werten hilfreich, weil es ebenfalls die Verwendung vereinfacht.

```
1 #!/bin/bash
2 echo "Starting up the image downlaoder with file 'urls.txt'..."
3 ./image_downloader.py urls.txt
```

- **Empfehlung C: README.** Ein – durchaus kurzes – Markdown Textfile mit Informationen über den Zweck und die Verwendung eines Programmes (README.md) wird auf Git direkt dargestellt. Bereitgestellte Links können angeklickt werden.

Image Downloader

README.md

The script `image_downloader.py` tries to download images with urls given in file `urls.txt` into a subfolder.

* Requirements: *Python3*

* Startup:

Run script `./startup.sh`.

Image Downloader

The script `image_downloader.py` tries to download images with urls given in file `urls.txt` into a subfolder.

- Requirements: *Python3*

- Startup:

Run script `./startup.sh`.

1.b Skripts

- **Empfehlung D: Feedback** über Zustand und über Fehler durch Ausgabe des aktuellen Status. Der aktuelle Zustand kann durch einfache Print Ausgaben mitgeteilt werden. Fehlerzustände durch konstruktive, lösungsorientierte Fehlermeldungen.

```
1 if os.path.exists(outdir) and os.path.isdir(outdir):
2     print(f"Downloading images into '{outdir}'.")
3 else:
4     print(f"Please create an output directory {outdir}. Then try again.")
```

- **Empfehlung E: Abbruch bei Fehlern.** Wenn nach dem Auftreten eines Fehlers nicht sinnvoll weitergearbeitet werden kann, ist es ok, bei solchen Fehlern einfach ein Programm abzubrechen. „*Let it crash approach*“! **Erwartbare Fehler abfangen:** jene – erwartbaren – Fehler, die durch fehlerhafter Daten (z.B.: ungültige URL) oder falsche Eingaben (z.B.: Tippfehler) entstehen, müssen aber sehr wohl abgefangen werden. Nach der Fehlerbehandlung (z.B.: Fehler in ein Logfile schreiben) kann das Programm sinnvoll weiterlaufen.

```
1 for idx, url in enumerate(urls, start=1):
2     print(f"\t{idx}/{len(urls)} downloading '{url}'", end="...")
3     try:
4         urlretrieve(url, f"{out_dir}/image.{suffix}")
5         print("OK")
6     except Exception as err:
7         print("NOT OK.")
8         custom_log(err)
```

Problem: Die Benutzbarkeit, die *Usability* sollte für Entwickler:innen immer möglichst hoch sein.

- **Empfehlung A: Sprechende Namensgebung.** Bei Variablen, Funktionen, Klassen und Methoden sollen sprechende, selbsterklärende Namen auf englisch verwendet werden. Dabei ist die Python übliche Schreibweise laut „*styleguides*“¹ zu beachten.

```
1 def pretty_print(data, indent = 3, symbol = " "):
2     print(f"{symbol*indent}{data}")
3
4 point_of_interests = {"home": "Kapfenberg", "work": "Telework"}
5
6 pretty_print(point_of_interests, symbol = "_")
```

- **Empfehlung B: Effizienter durch Python-artigen Code.** Das Verwenden von Python-typischem Code hilft kurze, prägnante und wartbare Programme zu schreiben. „*Write idiomatic Python code*“! Beispielsweise könnte man umfangreiche Daten in Listen `list` und Dictionaries `dict` verwalten, sortieren und filtern, oder mit einem Set `set` Duplikate entfernen. „*Thinking in lists*“!

```
1 for idx, city in enumerate(cities):
2     print(f"{idx+1}/{len(cities)}: {city}")
```

¹„StyleGuides“ <https://pep8.org>

1.c Passwörter

*Summary: Idealerweise erhalten Python Skripts ihre Daten von extern, sind also nicht fest in den Source Code hineinkodiert. Skripts sind ebenfalls über externe Settings konfigurierbar. Sensiblen Daten wie Passwörter landen auf keinen Fall im Code. „**Never hard code credentials**“!*

Problem: Sensible Daten (wie Passwörter oder auch API keys) finden sich als Teil des Source Codes. Schlimmstenfalls werden sensible Informationen sogar ausgegeben und geheime Passwörter tauchen später in Log Dateien auf.

```
1 pwd="very-secure"
2 print(f"We log in with password '{pwd}' at the webservice.")
```

Please Improve

- **Lösung A: Kommandozeilenargumente.** Übergabe von Konfigurationen als *command-line parameters*.

```
1 parser=argparse.ArgumentParser(description='''Image Loader''')
2 parser.add_argument("-p", "--password",
3     metavar="<password>",
4     help='specify the password used to login at the server')
5 pwd = parser.parse_args().password
```



- **Lösung B: Umgebungsvariable.** Übergabe von Konfigurationen mittels *environment variables* (Dies ist vor allem im Umfeld von CI/CD – siehe Abschnitt 2.a Git – üblich).

```
1 outdir=os.environ['OUTDIR']
```



- **Lösung C: Konfigurationsfile.** Festlegen von Konfigurationen in einem *config-file*. Hier zum Beispiel *config.py* mit dem Inhalt `ip="10.0.0.7"`.

```
1 import config
2 server = config.ip
```



- **Lösung D: Standardwerte.** Das Festlegen von gut gewählten *default values* erübrigt in vielen Fällen die explizite Übergabe von Startwerten.

```
1 outdir=os.environ.get('OUTDIR', "/tmp")
```



1.c Skripts

- **Lösung E: Pipes.** Werte können auch über eine sogenannte *Pipe* an ein Programm übergeben werden.

```
data_processing.py
1 import sys
2 input_from_other_prog=sys.stdin.read().strip()
3 print(f"Data piped into this script: '{input_from_other_prog}'.") ✓
```

bzw.

```
startup.sh
1 echo "server_url=http://data.imgs.com/" | ./data_processing.py
2 # Data piped into this script: 'server_url=http://data.imgs.com/'. ✓
```

Weitere Empfehlungen:

- Bei fehlenden Kommandozeilen Parametern sollte man User zur Angabe gültiger Werte durch **hilfreiche Fehlermeldungen** hinleiten, z.B. Usage ./downloader.py <urls> <target_dir>.
- Bei Konfigurationsfiles ist es günstig eine **Templatedatei**, z.B.: config_template.yaml, als Muster mit Demo-Daten bereitzustellen, welches vor bei der Verwendung umbenannt und mit konkreten Werten versehen wird.
- Im Git Repository sollten nur Vorlagen gespeichert werden, die jeweiligen lokalen Konfigurationsdateien werden mittels .gitignore vom **Repo ausgenommen**.

1.d Eingabenüberprüfung

*Summary: Validierung des Inputs ist bei Benutzereingaben immer nötig, da den Eingaben von Usern prinzipiell nicht vertraut werden darf. **Never trust user data!***

Problem: Bei der Verwendung von Funktionen wie `eval`, `exec`, oder `os.system` kann beliebiger Python Code ausgeführt werden. Dadurch ist die Verwendung auch extrem gefährlich!

```
1 sum = 0
2 for line in open('data.txt').readlines():
3     exec(f"sum+={line.strip()}")
4 print(sum)
```

Please Improve



- **Lösung A Checks:** Jeden (User-)Input vor der Verwendung überprüfen

Idealerweise „Whitelisting“.

```
1 bad_character = [';', '&', '|']
2
3 sum = 0
4 for line in open(filename).readlines():
5     stripped_line = line.strip()
6     for char in bad_character:
7         if char in stripped_line:
8             sys.exit(f"ERR: invalid sequence '{stripped_line}' in data.")
9     sum += eval(stripped_line)
10
11 print(sum)
```



1.d Skripts

- **Lösung B Keine gefährlichen Funktionen** verwenden, da den Input-Daten niemals vertraut werden kann. Es erscheint zwar besser `eval` statt `exec` zu verwenden, aber empfohlen wird gefährliche Funktionen - welche dynamisches Ausführen von beliebigen Code ermöglichen - überhaupt gar nicht zu verwenden!

Hier ein Beispiel mit explizitem Parsing durch Regular Expressions „*regex*“ und verrechnen der Daten mit den entsprechenden mathematischen Operationen (anstatt `eval`). „*Use regex*“!

```
1 import re
2
3 input_lines = open(filename).readlines() # lines, such as: "3+13" or "14*8"
4
5
6 # Note: only integer numbers and operators +, -, * are supported.
7 def extract_with_regex(input):
8     calc_match = re.match(r"^(\d+\s*)([/*-+])\s*(\d+)$", input) # see below
9     if not calc_match:
10         raise Exception("Unallowed input. Calculations + - * are supported")
11     (left, op, right) = calc_match.groups()
12     return int(left), op, int(right)
13
14
15 sum = 0
16 for idx, line in enumerate(input_lines, start=1):
17     try: # parsing might fail on erroneous input
18         left, opr, right = extract_with_regex(line.strip())
19     except Exception as err:
20         print(f"\tSorry, unsure about '{line}', so we skip it. Err: '{err}'")
21         continue
22     if opr == "+":
23         sum += left + right # no eval/exec.. but explicit math operation
24     # elif opr == "-": ...same for - and *
25     else:
26         print("WARN: Unexpected operator '{opr}'. Not implemented.")
27
28 print(f"Final result: Sum = {sum}.")
```

Erklärung zur „*regex*“ von oben (Ersatz für Zeile 8 aus dem oberen Listing):

```
8 calc_match = re.match(
9     r"""
10     ^           # For patterns we use Python "raw" strings with r""
11     (           # From the beginning of the line
12         \d+      # first match group = LEFT number
13         \d+      # one or more digits (i.e. an arbitrary number)
14     )
15     \s*         # zero or more whitespaces (ie. optional)
16     (           # second match group = math OPERATOR
17         [/*-+]   # one of '*' or '-' or '+' (i.e. operators)
18         \s*      # Note: we escaped special chars like * with \*
19     )
20     \s*         # zero or more whitespaces (ie. optional)
21     (           # third match group = RIGHT number
22         \d+      # one or more digits (i.e. an arbitrary number)
23     )
24     $           # To end of line
25     """
26     user_input, # e.g.: "0 *33"
27     re.VERBOSE, # all whitespaces and comments are ignored
28 )
```

1.e Debugging

*Summary: Zur Inspektion des Programmablaufes können mit einerseits einfache **print Ausgaben** oder besser Ausgaben über einen **Logger** hilfreich sein. Im Zweifel ist aber der Einsatz eines **Debuggers** unumgänglich.*

Problem: Der effektive Ablauf, der Flow durch das Programm kann leider im Source Code nicht immer gut und einfach ersichtlich sein.

- **Lösung A Print Statements:** Ausgabe von Log Informationen. Um den Ablauf eines Programmen nachzuvollziehen ist die Ausgabe von Statusinformationen hilfreich. „Use poor man’s debugging“!

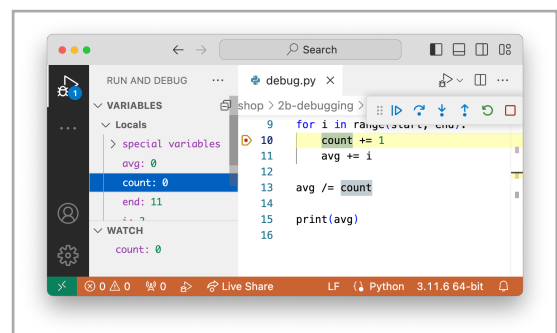
```
1 for i in range(count, start, end):
2     print(f" Dbg before: cnt='{count}')"
3     count += 1
4     print(f" Dbg after: cnt='{count}')
```

- **Lösung B Logging:** Umfangreiche Ausgaben mittels *Logger*. Bei vielen Outputs ist die Verwendung einer Logging Library empfehlenswert. Da kann man den Umfang der Ausgabe sehr gut über die Angabe eines **Log-Levels** wie info, debug oder error steuern.

```
1 import logging as l
2 import sys
3 l.basicConfig(filename='download.log', encoding='utf-8',
4               level=l.INFO, format='%(asctime)s %(message)s', datefmt='%H:%M:%S')
5 l.info(f"We process command line args '{sys.argv[1:]}'")
```

- **Lösung C Debugger:** Zur gründlichen Analyse ist die Verwendung eines Debuggers noch viel mächtiger. Bei Live **debugging** kann nach jedem Ausführungsschritt der gesamte Programmstatus inspiziert werden. „Use breakpoints“!

```
1 # Add a breakpoint
2 #     in VS Code by clicking
3 #     on the line numbers on the left
4 #
5 # start the debugger, it stops at the
6 #     breakpoint, then step over/into
7 #
8 # inspect the values of any variables
9 #
10 # add watches
```



2 Softwareprojekte

Folgende Strategien helfen um gemeinsam im Team größere Softwareprojekte mit guter Qualität zu schreiben: Distributed Code Repository Git, Virtual Environments, Automated Testing und der Einsatz von externen Formatierung-, Linter- und Security- Tools.

- (2a) Das Verwenden eines **Source Code Repository** garantiert Nachvollziehbarkeit und ermöglicht verteilte Zusammenarbeit im Entwicklungsteam. Abgesehen von „Gratis-Backups“ ermöglicht Git weiters die Entwicklung in Branches, zum Beispiel für *development spikes*. Use „distributed code repo Git. Develop in feature branches“!
- (2b) **Virtuelle Umgebungen** (venv) ermöglichen es externe Libraries zu verwenden, verhindern dabei aber Versionskonflikte. „Use virtual environments“!
- (2c) **Automatisiert Testen** ist nötig, damit sich neue Entwickler:innen überhaupt erst getrauen Änderungen an der Software durchzuführen. „Use unit tests. Test driven development“!
- (2d) **Check-Tools** für das Format, die Struktur und die Security erhöhen die Qualität und die Sicherheit des Source Codes. „Use a formatter and linter“!

Oft erweist sich ein kleines Skript als sehr nützlich und soll dann um neue Funktionalitäten erweitert werden. Software wächst. Nun steht man vor neuen Herausforderungen:

- **Die Struktur** für ein größeres Softwaresystem unterscheidet sich drastisch von einem simplen Skript. Vor dem Wachsen ist es nötig eine Struktur zu finden die Erweiterungen gut zulässt. Die Frage der Art der Strukturierung – nach Funktionalität, als Komponente, in Schichten – ist nicht immer eindeutig und leicht zu beantworten. „Structure by modules, layers, components“!

2.a Code Repository

*Summary: Source Code Verwaltung mit Git gibt Sicherheit durch **Backup** am Server und ermöglicht nachvollziehbare **Zusammenarbeit** unter Entwickler:innen*

Problem: Anfangs denkt man, kleine Skripts werde man eh nur selbst verwenden. Aber oft wächst der Umfang und die Funktionalität der Software. Dann kommt auch noch jemand und meint, diese Lösung würde man gerne ebenfalls einsetzen oder sogar weiterentwickeln.

- **Lösung: Source Code Verwaltung mit Git**

- Schritt A: Über die Web-Oberfläche ein neues Repository **Repo anlegen** und aus der Weboberfläche die URL zum „clonen“ des Repositories (für Schritt B) kopieren.
- Schritt B: Lokal am Rechner eine vollständige **Kopie** des Repos mittels `git clone` erstellen.

```
> git clone git@mode.fh-joanneum.at:secure-code/workshop.git
```

Hinweis: Im Dateisystem werden alle (nicht leeren) Unterverzeichnisse und Dateien angelegt. Die Metadaten (vorige Versionen der Dateien, URL des Git Servers usw.) finden sich im versteckten Verzeichnis `.git/`.

- Schritt C: Verzeichnisse und Dateien von der Source Code Verwaltung durch Erstellen einer Datei mit Namen `.gitignore` **ausnehmen**. Zum Beispiel um das `tmp` Verzeichnis auszunehmen:

```
> echo "tmp" >> .gitignore
```

```
tmp
.venv
__pycache__
config.py
*.pdf
```

`.gitignore`

Bzw. Dateien zum Repository mit `git add` **hinzufügen**.

```
> git add .
```

- Schritt D: Änderungen an den Dateien durchführen, diese mit `status` (optional auch mit `diff`) prüfen. Danach können die Änderungen mit `commit` übernommen werden. Dabei wird ein – die Änderung erklärender – Kommentar angegeben:

```
> git status
```

Den Grund bzw. Inhalt der Änderung durch eine so genannte **commit message** beschreiben. Diese ist dann jederzeit im **log** sichtbar:

```
> git commit -am "Fixed typo in README.md"
> git log
```


2.a Softwareprojekte

- Schritt E: Das Aufrufen von pull nicht vergessen um alle zwischenzeitlichen Änderungen am Source durch andere Personen vom Git Repository zu übernehmen. Dann erst weiter arbeiten mit Schritt B.

```
> git pull
```



- Schritt F: Abschließend (oder auch gerne Zwischendurch) die eigenen Änderungen mit push auf den Server übertragen, zu „pushen“.

```
> git push
```



Weitere Empfehlungen:

- Automatische Anmeldung auf Servern (hier am Git Repository) durch SSH mit Schlüsselaustausch. Dazu wird zuerst ein **SSH Key Pair** erstellt (ohne „passphrase“).

```
> ssh-keygen -t ed25519 -f ~/.ssh/key-for-git
```

Den *public key* ausgeben und den Plain Text mit Copy/Paste über das Web Interface am Server hinterlegen.

```
> cat ~/.ssh/key-for-git.pub  
> # ssh-ed25519 AA.....KT60m
```

Wenn nicht der Standard Name des Schlüsselpaares (wie `id_rsa` oder `id_ed25519`) verwendet wird, dann kann in der Datei `~/.ssh/config` konfiguriert werden, für welchen Server welche Keys verwendet werden sollen. Beispielsweise:

```
Host mode mode.fh-joaanneum.at gitlab  
  Hostname mode.fh-joaanneum.at  
  IdentityFile ~/.ssh/key-for-git  
  User git
```

~/.ssh/config

- Änderungen ohne zu speichern zu **verwerfen** funktioniert mit `git stash`. Log information kann mit `git log --oneline` sehr kompakt betrachtet werden.
- Änderungen an den Dateien mittels `git diff` oder in einer IDE betrachten. In modernen IDEs können auch **Konflikte** interaktiv aufgelöst werden. „*Solve merge conflicts*“!
- Für Zusammenarbeit in (verteilten) Teams empfiehlt es sich, sich über Entwicklung in **Feature Branches** zu informieren. „*Select your branching strategy*“!

2.b Bibliotheken

*Summary: Bibliotheken (**Libraries**) ersparen viel Code selbst zu schreiben. Die Installation sollte bezüglich verwendeter **Versionen** transparent und nachvollziehbar sein.*

Problem: Um Code aus anderen Quellen wie Bibliotheken (Libraries) zu verwenden, müssen diese lokal installiert werden.

- **Schritt A: Python Virtual Environment erstellen.** Hier wird zum Beispiel ein verstecktes Verzeichnis `.venv` angelegt, um dort später Libraries zu installieren.

```
> python3 -m venv .venv
```

- **Schritt B: Environment aktivieren.**

- Unter Linux/Mac mit `source .venv/bin/activate`.

```
> source .venv/bin/activate
```

- Bzw. unter Windows mit `.venv\Scripts\activate`.

```
> .\.venv\Scripts\activate
```

- **Schritt C: Bibliotheken installieren.** In der Datei `requirements.txt` werden benötigte Libraries gelistet.

```
requests
bs4
```

requirements.txt

Bei aktivem Python Virtual Environment werden die Bibliotheken lokal ins Verzeichnis `.venv` installiert.

```
> pip install -r ./requirements.txt
```

- Schritt D: Libraries kann man im Python Skript durch ein `import` Statement verwenden.

```
1 import requests
2 from bs4 import BeautifulSoup
```

- Schritt E: Schlussendlich kann man das Python Virtual Environment durch `deactivate` verlassen.

```
> deactivate
```

2.c Code Qualität

*Summary: Automatisiertes **Testen** und die Verwendung von Checker-Tools wie **Linter** oder **Formatter** erhöhen die Lesbarkeit, Sicherheit und Qualität von Programmen enorm.*

Problem: Wenn man sich nicht sicher sein kann, dass Code nach Änderungen weiterhin noch wie gewollt funktioniert, getraut man sich nicht Code abzuändern. Je nach Stil und Entwicklungsumgebung erstellen Entwickler:innen unterschiedlich aussehenden Code, welcher dadurch schwerer zu lesen und warten ist.

```
1 def pretty_output(msg, indent=3):
2     print(f"{' ' * indent}{msg}")
3
4 print("Please comment this later, we just test if it works")
5 pretty_output("Hi", 3)
```

Please Improve



- **Lösung A: Automatisiertes Testen**². „Use unit testing“!

- Source code so gestalten, dass er einfach testbar ist:

```
1 def pretty_format(msg, indent=3):
2     return f"{' ' * indent}{msg}"
3
4 def pretty_output(msg, indent=3):
5     res = pretty_format(msg, indent)
6     print(res)
7
8 if __name__ == "__main__":
9     pretty_output("This is visible only if the script is run directly.")
```

pretty.py



- Testfälle bereitstellen

```
1 import unittest
2 from solution import pretty_format
3
4 class TestPrettyPrintFunctions(unittest.TestCase):
5     def test_edge_case_negative_indent(self):
6         result = pretty_format("Hi", indent=-2)
7         self.assertEqual(result, "Hi")
8
9 if __name__ == "__main__":
10     unittest.main()
```



```
> python3 -m unittest unittests4solution.py
```



- **Lösung B: Source Code formatieren** „formatter“ bzw. Code **überprüfen** „linter“

```
> ruff format solution.py
> ruff check solution.py
```



²<https://docs.python.org/3/library/unittest.html>

2.d Projektstruktur

*Summary: Die **Projektstruktur** für größere Projekte soll logisch gut aufgeteilt sein, damit sich Programmierer:innen schnell zurechtfinden. Typischerweise kann die Modularisierung nach **Funktionalität**, in **Schichten** oder nach **Komponenten** vorgenommen werden. „Structure in layers, components, or features“!*

Problem: Wenn Skripts/Programme anwachsen kommt der Punkt, wo diese schwer lesbar werden. Leichtes Verständnis, Erweiterbarkeit und Wartbarkeit sind nicht mehr gegeben.

- **Lösung A: Objektorientierte Programmierung (OOP)** vorziehen. Komplexe Aufgabenstellungen lassen sich durch *Klassen* modellieren und der Code ist leichter verständlich bzw. besser test- und wartbar.
- **Lösung B: Einstiegspunkt** festlegen. Durch das Überprüfen auf `__main__` wird Code nur beim direkten Start des Skripts aufgerufen. Beim Testen und bei der Verwendung als Library wird Code aber nicht unabsichtlich aufgerufen.

```
1 def main():
2     # insert program logic here
3
4 if __name__ == "__main__":
5     main()
```

- **Lösung C: Eigene Packages** erstellen. Unterverzeichnisse kann man als **Packages** strukturieren.
 - Zum Beispiel im Unterverzeichnis `dv` die Datei `vis.py` anlegen.

dv/vis.py

```
1 class DataVisualisation():
2     # your code here
```

- Die eigenen **Packages** können dann im restlichen Code mittels `import` verwendet werden.

```
1 from dv.vis import DataVisualisation
2
3 cty_vis = DataVisualisation(csv_input_file)
```

- **Lösung D: Typsicher programmieren.** Mit „*type hints*“ kann die IDE (PyCharm, VSCode) den Typ der Übergabe- und Rückgabeparameter überprüfen und beim Schreiben Hinweise geben. Zum Beispiel kann angezeigt werden, welche Parameter mit welchem Typ an eine Methode übergeben werden muss.

```
1 class Message:
2     # ...
3
4 def compose_message(email: str, token: str) -> Message:
5     body = # ...
6     return Message(body)
```

3 Ausgewählte Lösungen

Spezialthemen

Das Bereitstellen eigener Daten als **ReSTful Web Service** ist für Systeme relevant, wo unterschiedlichste Clients die Informationen zentral abrufen wollen. Das API Design für Webserver/Webservices erfordert gute Dokumentation, und umfasst bei der Implementierung das Prüfen der Daten bezüglich Struktur, Datentyp und Wertebereich. In der Planung muss auf zukünftige Änderungen einer API durch Vergabe von API Versionsnummern Rücksicht genommen werden.

- (3a) Client- Server Kommunikation
- (3b) WebServices
- (3c) Ein eigener Server

Datenmanagement, **Datenanalyse und -visualisierung**. „*Data processing*“ bedeutet Daten anhand einer „*pipeline*“ abzuarbeiten. Daten werden eingelesen, bereinigt und verarbeitet, um schließlich als Grafiken exportiert zu werden. Zur Interpretation von Daten ist „*visualisation*“ nützlich: Grafische Aufbereitung erleichtert die Analyse und Interpretation der Daten. Zum besseren Verständnis von großen Daten eignet sich aber auch „*interactive exploration*“.

- (3d) Datenmanagement mit Excel
- (3e) Datenanalysen und Datenvisualisierung

Generelle Anmerkungen bezüglich Security bei der Softwareentwicklung: Algorithmen bezüglich Kryptografie und Security dürfen keineswegs selbst implementiert werden. Die wenigsten Entwickler:innen haben das theoretische und praktische Backgroundwissen um Aspekte wie *random*, *hashing*, *encrypt/decrypt* oder *certificates* selbst auf eine „kryptografisch sichere“(!) Art zu implementieren. „*Use well tested, reliable and proven cryptographic libraries*“!

3.a Client-Server Kommunikation

Wie schon im Abschnitt über Passwörter angeführt, dürfen sensible Daten nie im Source Code hardcodiert gespeichert werden. Clients benötigen für die Anmeldung am Server aber genau solche sensible Daten wie Zugangsdaten (z.B. *Benutzername+Passwort*). Verschiedene Arten der Anmeldung sind möglich.

3.a.1 Anmelden am Server (Authentication)

*Summary: Zur Anmeldung an Netzwerkdiensten sollten bevorzugt **Tokens** verwendet werden. Im Speziellen sind **username/password** besser interaktiv einzugeben. **API-tokens** werden nur lokal am Rechner hinterlegt und nicht ins Git Repository gespeichert. Bei sicherheitskritischen Anwendungen ist eine **multi-faktor-authentication** unumgänglich.*

Problem: Wenn Skripts Daten von einem Server holen oder an ein Web-Service Endpoint senden, dann ist meist eine Anmeldung nötig.

Unterschiedliche Arten der Anmeldung sind möglich.

- **Lösung A: Username und Passwort**

- Vorteil: Einfach.
- Nachteil: es müssen die Passwörter im Klartext bereitgestellt werden.

- **Lösung B: API Tokens**

- Vorteil: Kein Passwort nötig. Vom (random) Token kann nicht auf ein ursprüngliches Passwort zurückgeschlossen werden. Bei Bedarf ist ein Entziehen der Zugriffserlaubnis in modernen Systemen recht einfach.
- Nachteil: One-time Tokens sind fallweise aufwendig in der Umsetzung.

- **Lösung C: SSH Autorisierung**

- Vorteil: Maschine-zu-Maschine Kommunikation ist bequem.
- Nachteil: Eine *public key infrastructure* ist nötig.

- **Lösung D: Autorisierung mittels mehreren Faktoren**

- Vorteil: Sehr sicher, da neben einem Passwort (sogar mehrere) weitere Faktoren (z.B. biometrischer Fingerabdruck, SMS-Bestätigung) nötig sind.
- Nachteil: In der Umsetzung sehr aufwendig.

Dringende Empfehlungen:

- Credentials (wie Username und Passwort, API-Tokens, SSH-Keys) niemals im Source Code hardcodiert hinterlegen.
- Konfigurationsdateien mit sensibler Information immer mittels `.gitignore` vom Git Source Code Management ausnehmen.

3.a.1 Ausgewählte Lösungen

3.a.2 Kommunikation mit einem Server

*Summary: Bei der Übertragung von Anmeldedaten sind **POST Requests** den GET requests vorzuziehen, da POST Daten nicht als Bookmarks oder in Log-files landen. Tokens werden im **Header** mitgesendet. Typischerweise handelt es sich um ReST Aufrufe und die Daten werden als JSON übertragen.*

Problem: Clients senden sensitive Daten über das Netzwerk. Bei Anfragen (requests) mittels GET sind die Anmeldedaten direkt in der URL sichtbar und sensible Daten können dadurch auch in Log Files am Server landen.

Über das HTTP-Protokoll können Daten als **GET** oder **POST** Parameter übertragen werden. Weiters ist es möglich Daten im Body oder auch direkt im Header zu übermitteln. Es ist besser auf POST (wie bei HTML-Formularen) Requests zurückzugreifen, da hier die Daten im Body – und somit nicht Teil der URL – übertragen werden. Beim Senden von Anmeldedaten via **Basic Authentication** oder via **Bearer**-Information werden die Daten im Header übermittelt.

Import der nötigen Libraries.

```
1 import http.client
2 import os
```

Festlegen von Variablen für Username, Passwort, API Keys usw.

```
1 username=os.environ['USER'] # admin
2 password=os.environ['PASSWD'] # VerySecure!WhatElse?
3 api_key=s.environ['APIKEY'] # d0b5d600-33c7-4d8f-b54f-d340375242e8
4 resource="api/v3/login"
```

Anfrage (*request*) absenden.

```
1 conn = http.client.HTTPConnection(server_ip, port)
2 # conn.request(...) # see below
3 res = conn.getresponse()
4 if res.status == 200:
5     print(f"Server responded with '{res.read().decode('utf-8')}'")
```

- **Lösung A GET** Nicht empfehlenswert, da sensible Anmeldedaten in die URL hinein kodiert sind und somit beim Bookmarks und in den Log-Files sichtbar werden.

```
1 conn.request("GET", f"/{resource}?user={username}&pwd={password}")
```

- **Lösung B POST:** Senden von Daten im Body (via POST) und nicht in der URL.

```
1 headers = { "Content-Type": "application/json" }
2 data = { "username": username, "password": password }
3 conn.request("POST", f"/{resource}", body=json.dumps(data), headers=headers)
```

3.a.2 Ausgewählte Lösungen

- **Lösung C: Basic Auth:** Bei Basic Authentication erfragt der Browser die Benutzerdaten (Username and Password) in einem Browser-spezifischen Eingabefenster.

```
1 from base64 import b64encode
2 user_pass = f"{username}:{password}".encode('utf8')
3 basic = b64encode(user_pass).decode('utf8')
4 headers = {"Authorization": f"Basic {basic}"}
5 conn.request("GET", f"/{resource}", headers=headers)
```

- **Lösung D: Bearer:** Im Header wird ein Token oder API key übertragen.

```
1 headers = {"Authorization": f"Bearer {api_key}"}
2 conn.request("GET", f"/{resource}", headers=headers)
```

Weitere Empfehlungen:

Verbindungen müssen vertrauensvoll sein.

- Eine verschlüsselte Verbindung sollte standardmäßig verwendet werden. Auch Man-in-the-Middle (MitM) Attacken sind so viel schwerer durchzuführen. „Use *https with Transport Layer Security; TLS*“!
- **Ende zu Ende Verschlüsselung.** Wenn hochsensible Daten übertragen werden, ist eine zusätzliche Verschlüsselung der Daten selbst nötig, denn am Ende einer sicheren Verbindung werden Daten typischerweise am Server entpackt und sind am Server (in der Datenbank, im Filesystem) wieder im Klartext lesbar. „Use *end-to-end encryption*“!

3.b Web Services Nutzen

*Summary: Viele Services/Server bieten ein **JSON Rest API** an, um Daten automatisiert erstellen oder abrufen zu können.*

Problem: Automatische, wiederholte Auswertung von Daten oder wiederholte manuelle Erstellung von Git Issues über das Web Interface benötigt viele Klicks und könnte mühsam sein.

Folgender Abschnitt zeigt beispielhaft die Verwendung von ReST Web Services anhand von GitLab³. Bei *GitLab* muss über das Web Interface ein API Token erstellt werden. Mit Hilfe dieses Tokens kann eine Anmeldung stattfinden und nur dann können auch Daten von privaten Projekten abgerufen werden. Hier ausgewählte Beispiele der GitLab API Nutzung.

- Aufruf *public* verfügbarer Information. Hier mit `curl`.

```
> curl "https://mode.fh-joaanneum.at/api/v4/projects?search=example"
```

- **IDs der Projekte** mittels Python Skript abrufen.

request_project_id.py

```
7 api_endpoint = f'https://mode.fh-joaanneum.at/api/v4/projects/?search={search}'
8 headers = { 'Authorization': f'Bearer {TOKEN}' }
9 request = urllib.request.Request(api_endpoint, headers=headers)
10 with urllib.request.urlopen(request) as response:
11     response_data = response.read().decode('utf8')
12     json_data = json.loads(response_data)
13     print(json.dumps(
14         [ {'id': j['id'], 'name': j['name'], 'description': j['description']}
15           for j in json_data ],
16         indent=2))
```

- Mittels Python Skript ein **Issue erstellen**, für ein konkretes Projekt. Die unique ID muss dazu bekannt sein.

create_issue.py

```
7 api_endpoint = f'https://mode.fh-joaanneum.at/api/v4/projects/{id}/issues'
8 headers = { 'Authorization': f'Bearer {TOKEN}',
9             'Content-Type': 'application/json' }
10 data = {'title': 'this is a test by jf',
11         'labels': 'example,test',
12         'description': 'this is an example to demonstrate the API'}
13 json_data = json.dumps(data).encode('utf8')
14 request = urllib.request.Request(api_endpoint, data=json_data,
15                                 headers=headers, method='POST')
16 with urllib.request.urlopen(request) as response:
17     response_data = response.read().decode('utf8')
18     print(json.dumps(json.loads(response_data), indent=2))
```

³GitLab API Dokumentation: <https://docs.gitlab.com/ee/api/rest/>.

3.c Implementierung Server

*Summary: Zur Bereitstellung von Daten im Netzwerk - als **Web-Service** - ist es üblich ein maschinenlesbares Format wie **JSON** zu verwenden.*

Problem: Normale Webseiten sind zwar gut menschenlesbar, aber für automatisierte Verarbeitung schlecht geeignet.

- Python3 Server mit Flask Framework um ein (ReSTful) Web Service bereitzustellen.

```
1 from flask import Flask, request
2 imgapi = Flask(__name__)
3
4 @imgapi.route('/')
5 def welcome():
6     return """This is a web service endpoint."""
7
8 if __name__ == '__main__':
9     imgapi.run()
```

- Definition eines API Aufrufs (web service endpoints) für das Web Service.

```
1 @imgapi.route('/store', methods=['POST'])
2 def add_image():
3     return """{"Info": "Not implemented yet"}"""
```

- Typischerweise können mittels ReSTful web services Daten im JSON Format über definierte API Aufrufe von beliebigen Clients (JavaScript im Browser, mobile Apps, auf der Kommandozeile mit CURL) abgerufen werden.

Beispielhafte Verwendung des Kommandozeilen Tools curl.

```
> curl -H "Authorization: Bearer 65cac19d8d1fb17ef44c9f" \
>     http://122.40.5.7:8080/api/v3/image/store \
>     --data "imagename=sunset&imageformat=svg"
```

Weitere Empfehlungen:

- Je nach Bedarf auch weitere Frameworks (*FastAPI*,...) in Erwägung ziehen.
- Produktive Server über das Server-Team bereitstellen und die Server härten (**hardening**) lassen.

3.d Excel

*Summary: Wenn gesammelte und verarbeitete Daten weitergegeben werden, dann bietet sich – je nach Zielgruppe – oftmals **Excel** an.*

Problem: Viele Personen (z.B. im Management) wollen Daten interaktiv in Excel bearbeiten und Visualisieren. CSV – „comma seperated value“ – Dateien sind so einfach aufgebaut, dass man diese direkt, ohne die Verwendung von Libraries, erstellen kann. Allerdings ist das bald unflexibel, fehleranfällig und mühsam.

```

1 data = [
2     ["Apfel",    "1,2", "10", "12,0"],
3     ["Birne",    "1,4", "15", "21,0"],
4     ["Kirsche",  "2,5",  "7", "14,5"],
5 ]
6 with open("umsatz.csv", "w", encoding="utf-8") as f:
7     for row_elements in data:
8         f.write(';' .join(row_elements)+"\n")

```

Please Improve

✗

• Lösung:

- Eine Excel Datei (*.xlsx) kann mithilfe geeigneter Bibliotheken⁴ erstellt werden.

```

1 from openpyxl import Workbook
2 wb = Workbook()
3
4 ws = wb.active
5 ws.title = "Umsatz"
6
7 ws.append(
8     ["Produkt", "€", "Anz.", "Ges.:"])
9 d = [
10    ["Apfel",    1.2,  10 ],
11    ["Birne",    1.4,  15 ],
12    ["Kirsche",  2.5,   7 ]]
13
14 for idx, row in enumerate(d, start=2):
15     # append => column D e.g. '=B2*C2'
16     row.append(f"B{idx}*C{idx}")
17     # append current row to worksheet
18     ws.append(row)
19
20 ws["D5"] = "=SUM(D2:D4)"
21 wb.save("umsatz.xlsx")

```

Ansicht in MS Excel:

| D5 | | | | |
|----------------|---------|-------|-------|-------------|
| fx =SUM(D2:D4) | | | | |
| | A | B | C | D |
| 1 | Produkt | Preis | Menge | Gesamtpreis |
| 2 | Apfel | 1,2 | 10 | 12 |
| 3 | Birne | 1,4 | 15 | 21 |
| 4 | Kirsche | 2,5 | 7 | 17,5 |
| 5 | | | | 50,5 |
| 6 | | | | |

Es werden die Werte in Spalte D durch die Formel berechnet. Auch die Summenformel für das Feld D5 wurde in Python erstellt.

⁴OpenPyXL Dokumentation auf <https://openpyxl.readthedocs.io/>.

3.e Datenanalyse und -visualisierung

Um Aussagen über die Information in große Datensets zu erhalten müssen diese aufbereitet bzw. visualisiert werden.

3.e.1 Data processing pipeline

*Summary: Für automatisierte Analyse und -interpretation werden Schritt für Schritt Daten entlang einer **Visualisation Pipeline** aufbereitet und visualisiert.*

Problem: Große Mengen an Daten sind oft schwer zu verstehen und zu interpretieren.

In einer **Data processing pipeline** können Daten aufbereitet und visualisiert werden. Dazu sind typischerweise Libraries wie Pandas, Matplotlib oder Seaborn nötig.

```
1 import pandas as pd
2 import matplotlib.pyplot as plt
3 import seaborn as sns
```

- **Schritt A: Daten einlesen** und im Speicher verwalten. „Data Frames“

```
1 wine = pd.read_csv(fn, index_col=0)
```

- **Schritt B: Filtern:** Man verwendet Befehle wie `info()` oder `head(40)` für das Inspizieren und Filtern der Daten.

```
1 wine.info()
2 top_df = wine["variety"].value_counts().head(3)
```

- **Schritt C: Korrekturen vornehmen:** Die Daten durch Ergänzen, Entfernen, Korrekturen bereinigen. Hier ein Beispiel um 4-stellige Zahlen im Text der Spalte „title“ zu finden und als neue Spalte „year“ zu übernehmen. „data cleansing“

```
1 def add_year_vintage(dat):
2     dat["year"] = dat["title"].apply(lambda x: re.findall(r'\d{4}', x)[0])
3     return dat
4 wine = add_year_vintage(wine)
```

- **Schritt D: Berechnungen:** Nützliche Werte wie Summen können über die Daten berechnet werden.

```
1 top_wines = wine[wine['variety'].isin(top_df.index)].copy()
2 top_italy = top_wines[top_wines["country"] == "Italy"]
```

- **Schritt E: Visualisierungen:** Daten werden für besseres Verständnis als Grafiken visualisiert und die Plots („figures“) abspeichern.

```
1 sns.heatmap(italy_ptsvsvariety, cmap="viridis", annot=True)
2 plt.savefig("heatmap.png")
```

3.e.1 Ausgewählte Lösungen

3.e.2 Interaktive Datenexploration

*Summary: **Interaktive** Jupyter Notebooks⁵ ermöglichen es jederzeit Zwischenstände zu speichern und erleichtern das **Experimentieren**. Zum Beispiel wenn für eine Optimierung unterschiedlichste Parameter durchgetestet werden.*

Problem: Große Mengen an Daten sind oft schwer zu verstehen und zu interpretieren.

In Jupyter Notebooks kann man interaktiv die Daten bearbeiten und visualisieren. Dabei bleibt der letzte Zustand intern gespeichert und eine Session kann später jederzeit wieder aufgenommen werden.

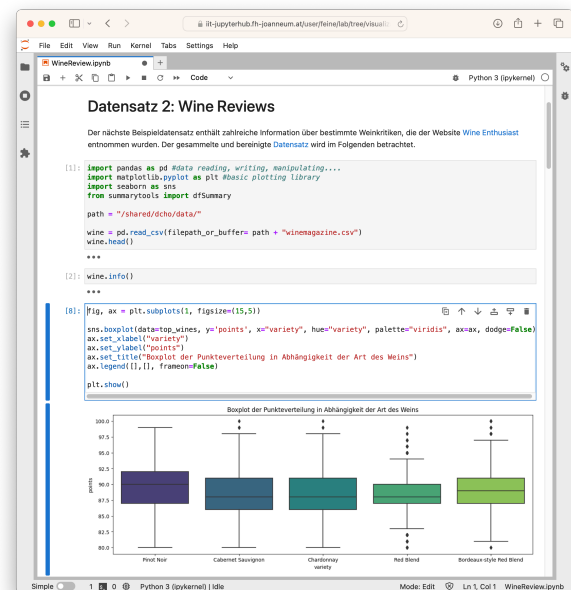
- **Schritt A Jupyter Notebooks:** Es kann eine lokale Installation vorgenommen werden oder Jupyter Notebooks können über das Web-Interface verwendet werden. Auch VS Code bietet Unterstützung für Jupyter Notebooks⁶.
- **Schritt B Blockweises Ausführen von Code:** Python Source Code wird in kleinere Blöcke gruppiert und bei Bedarf auch mit nützlichen Kommentaren im Markdown Format versehen.

```
1 import pandas as pd
```

```
1 wine = pd.read_csv("wine.csv")
```

```
1 wine.info()  
2 wine.head(13)
```

Tipp: Wichtige Tastenkürzel einlernen: Zum Beispiel CTRL-RETURN zum Ausführen des aktuellen Blocks.



Weitere Empfehlungen:

- Export des Notebooks als Python Skript, um es wiederholt und automatisiert ablaufen zu lassen.

⁵Jupyter Notebooks: <https://jupyter.org>. Auch die Installation von JupyterLab ist da zu finden.

⁶Jupyter Notebooks in VS Code <https://code.visualstudio.com/docs/datascience/jupyter-notebooks>

Anhang

Welche Ressourcen nutzen Python Entwickler:innen typischerweise? Neben Stackoverflow, ChatGPT und CoPilot wird empfohlen, die originalen Dokumentationen, Community Styleguides aber auch Cheatsheets zu verwenden.

Styleguides

Richtlinien für Python Code werden im Team verteilt, um sich beim Programmieren an die üblichen Schreibweisen zu halten. Das erleichtert das Lesen von Code und die Kommunikation unter Kolleg:innen.

- Official Python Styleguides „PEP 8“: <https://pep8.org>.
- Google Style für Open Source Projekte: <https://google.github.io/styleguide/pyguide.html>.

Cheat Sheets

Cheatsheets sind nützlich, um auf ein oder zwei Seiten die Basics der Programmiersprache immer bereit zu haben. Das geht oft schneller als eine Internet-Suche zu starten.

- Alt aber gut ist folgendes **Python3** „Schwindelzettel“ https://perso.limsi.fr/pointal/_media/python:cours:mementopython3-german.pdf.
- Für die **Datenvisualisierung**:
 - Große Datenmengen importieren und verarbeiten mit **Numpy**, *DataFrames* mit **Pandas**:
<https://s3.amazonaws.com/dq-blog-files/numpy-cheat-sheet.pdf> by Dataquest 2017 https://pandas.pydata.org/Pandas_Cheat_Sheet.pdf by Pandas.org
https://images.datacamp.com/image/upload/v1676302204/Marketing/Blog/Pandas_Cheat_Sheet.pdf by DataCamp
 - Daten einfach visualisieren mit **Seaborn** oder **Matplotlib**:
https://images.datacamp.com/image/upload/v1676302629/Marketing/Blog/Seaborn_Cheat_Sheet.pdf by DataCamp
<https://matplotlib.org/cheatsheets/> by DataCamp
 - Aufwändige und wirklich schöne Plots mit **ggplot2**:
<https://ggplot2.tidyverse.org> by TidyVerse.org bzw. auf GitHub.

Source Code: Die vollständigen Python Skripts zu den in diesem Dokument präsentierten Code Snippets findet sich unter <https://mode.fh-joanneum.at/secure-code/workshop/>. Happy Coding!