Eigene Funktionen

by Woche 2

```
import numpy as np
import pandas as pd
```

Pythons vordefinierte Funktionen und importierte Funktionen z.B. aus `numpy` und `pandas` haben wir ja bereits kennengelernt. Es ist aber natürlich auch möglich sich selbst neue Funktionen zu definieren. Der Hauptgrund dafür ist eins der bekanntesten Prinzipien der Programmierung:

Don't Repeat Yourself – Wiederhole dich nicht

Funktionen ermöglichen die Erstellung von übersichtlichen, wartbaren und wiederverwendbaren Programmen bzw. Workflows.

Syntax

Für die Definition einer Funktion muss der Funktionsname und die Funktionsparameter ("Argumente") vorgegeben werden. Die grundlegende Syntax dafür sieht so aus:

```
# Definiere Funktion
def gruess(name):
    print("Hallo " + name + "!")
```

```
# Rufe Funktion auf
gruess("Lukas")
gruess("Mila")
gruess("Kerstin")
```

```
Hallo Lukas!
Hallo Mila!
Hallo Kerstin!
```

Das Argument `name` ist in der Funktion also eine Variable, die mit dem Wert, den wir im Funktionsaufruf übergeben, definiert wird. Wie bei Schleifen und anderen logischen Blöcken wird der Kontext der Funktion über die Einrückung definiert.

Wir sparen es uns also, jedes mal die String-Verkettung und den `print`-Befehl auszuschreiben, und man kann direkt im Code erkennen, dass drei Personen beim Namen gegrüßt werden sollen. Bei diesem einfachen Beispiel ist der Vorteil zwar noch gering, aber sobald komplexere Funktionen definiert werden und gar andere Personen den Code lesen und ändern sollen, werden sie bald unerlässlich.

Wir könnten die Funktion zum Beispiel so erweitern, dass sie in mehreren Sprachen grüßen kann, welche durch ein weiteres Argument definiert werden:

```
def localized_greeting(name, language):
    if language == 'de':
        print("Hallo " + name + "!")
    elif language == 'en':
        print("Hello " + name + "!")
    elif language == 'es':
```

```
        print("¡Hola " + name + "!")
    else:
        print("👋 " + name)
```

```
localized_greeting("Lukas", "fr")
localized_greeting("Mila", "es")
localized_greeting("Kerstin", "jp")
```

```
👋 Lukas
¡Hola Mila!
👋 Kerstin
```

> **i** Namenskonventionen
>
> Um das Lesen von Python-Code zu vereinfachen, hat sich die Python-Community auf einige Konventionen für die Benennung von verschiedenen Dingen geeinigt. Die wichtigsten für uns sind:
>
> - Variablen und Funktionen: im sogenannten **Snake-Case**, also kleingeschrieben und einzelne Wörter mit einem Unterstrich getrennt, z.B. `localized_greeting`
> - Konstanten, also Werte, die sich nicht ändern: in Großbuchstaben, z.B. `PI = 3.14159`
>
> Diese und weitere Richtlinien können in dem PEP 8 Style guide nachgelesen werden.

Standardargumente

Wir können ein Argument optional machen, indem wir einen Standardwert vorgeben, den es einnimmt, wenn kein anderer Wert beim Aufruf spezifiziert wird:

```
def localized_greeting(name, language = 'de'):
    if language == 'de':
        print("Hallo " + name + "!")
    elif language == 'en':
        print("Hello " + name + "!")
    elif language == 'es':
        print("¡Hola " + name + "!")
    else:
        print("👋 " + name)
```

```
localized_greeting("Brigitte")
```

```
Hallo Brigitte!
```

Rückgabewerte

Eine weitere Eigenschaft von Funktionen ist das “Zurückgeben” von Werten, welche in der Funktion berechnet wurden. Das wird über das Schlüsselwort `return` erreicht. `return` beendet die Funktion und gibt den nachfolgenden Wert zurück.

```
PI = 3.14159

def berechne_zylindervolumen(radius, hoehe):
    grundflaeche = PI * radius ** 2
```

```
    volumen = grundflaeche * hoehe
    return volumen

r = 5
h = 20
vol = berechne_zylindervolumen(r, h)
print(f"Ein Zylinder mit Radius {r} und Höhe {h} hat ein Volumen von ungefähr {vol}.")
```

```
Ein Zylinder mit Radius 5 und Höhe 20 hat ein Volumen von ungefähr 1570.795.
```

Nur wenn eine Funktion explizit einen Wert zurückgibt, kann dieser auch außerhalb der Funktion verwendet, also z.B. in eine Variable gespeichert werden. Wenn keine `return`-Anweisung vorhanden ist, gibt die Funktion `None` zurück. So wäre es beispielsweise nicht möglich das Ergebnis der `gruess()`-Funktion von vorhin in eine Variable zu speichern:

```
eingruss = gruess("Peter")
```

```
Hallo Peter!
```

```
print(eingruss)
```

```
None
```

Eine Funktion kann natürlich auch mehrere Werte bzw. “Dinge” gleichzeitig zurückgeben. Python ermöglicht es, diese direkt an mehrere Variablen zuzuweisen:

```
def berechne_kreis(radius):
    umfang = 2 * PI * radius
    flaeche = PI * radius ** 2
    return umfang, flaeche

# Beide Rückgabewerte direkt verschiedenen Variablen zuweisen
u, a = berechne_kreis(5)
print(f"Umfang: {u:.2f}, Fläche: {a:.2f}")
```

```
Umfang: 31.42, Fläche: 78.54
```

Dieses “Auspacken” der Rückgabewerte macht den Code übersichtlicher und ist eine elegante Methode, um mit mehreren Ergebnissen zu arbeiten.

Scoping

Vielleicht ist euch im Beispiel oben etwas aufgefallen: Warum wird eine neue Variable `vol` definiert, die den Rückgabewert der Funktion enthält, wo doch in der Funktion selbst eine Variable `volumen` definiert wurde? Die Antwort darauf ist der **Scope** (dt. Nutzbarkeitsbereich/Sichtbarkeitsbereich) der Variablen.

Variablen, die in einer Funktion definierten werden, sind *nur* innerhalb dieser Funktion verwendbar. Das muss auch so sein, da man sich sonst aus Versehen durch die Verwendung einer Funktion eigene Variablen überschreiben könnte. Das bedeutet also auch, dass man Variablennamen, die in Funktionen verwendet werden, ohne weiteres wiederverwendet werden können.

In Funktionen können jedoch trotzdem Variablen (oder Konstanten) von außerhalb der Funktion verwendet werden, wie hier zum Beispiel `PI`. Eine so verwendete Variable wird *globale Variable* genannt.

> ⚠ Vorsicht
>
> Die Verwendung von globalen Variablen sollte in den meisten Fällen vermieden werden. Sie verringern die Lesbarkeit des Codes, da nicht offensichtlich ist, wo die verwendeten Werte in einer Funktion herkommen. Außerdem können globale Variablen an verschiedenen Stellen modifiziert werden, was die eventuelle Fehlersuche erschwert.

Im folgenden Beispiel tritt ein Fehler auf, da die Variable `volumen` außerhalb der Funktion nicht definiert ist. Stattdessen gibt es `volumen` nur innerhalb der Funktion und das Ergebnis wird in `vol` gespeichert. Die letzte Zeile hätte also `print(vol)` lauten müssen:

```
PI = 3.14159

def berechne_zylindervolumen(radius, hoehe):
    grundflaeche = PI * radius ** 2
    volumen = grundflaeche * hoehe
    return volumen

r = 5
h = 20
vol = berechne_zylindervolumen(r, h)

print(volumen)
```

```
NameError: name 'volumen' is not defined
```

Docstrings

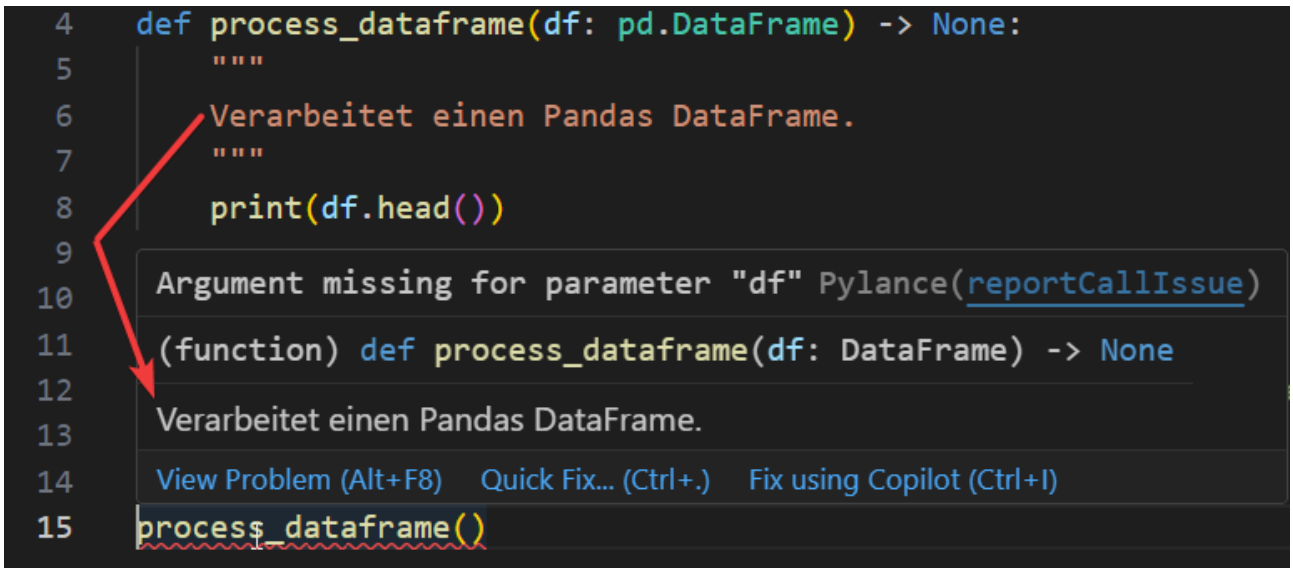
Eine gute Praxis beim Schreiben von Funktionen ist die Verwendung von Docstrings. Ein Docstring ist ein mehrzeiliger String, der direkt unterhalb der Funktionsdefinition steht und die Funktion mit ihren Parametern und Rückgabewerten beschreibt. Im Endeffekt handelt es sich dabei um einen Kommentar, der die Funktion in keinster Weise beeinflusst, sondern nur dokumentiert. Dies erleichtert das Verständnis der Funktion für andere Menschen und für einen selbst, wenn man sich den Code später erneut ansieht.

Ein Docstring wird mit dreifachen Anführungszeichen (“““) oder (‘‘‘) geschrieben und kann mehrere Zeilen umfassen. Hier ist ein Beispiel, das die `berechne_zylindervolumen`-Funktion mit einem Docstring ergänzt:

```
PI = 3.14159

def berechne_zylindervolumen(radius, hoehe):
    """
    Berechnet das Volumen eines Zylinders.

    Parameter:
    radius: Der Radius der Zylinderbasis.
    hoehe: Die Höhe des Zylinders.

    Rückgabewert:
```

```
    Das berechnete Volumen des Zylinders.
    """
    grundflaeche = PI * radius ** 2
    volumen = grundflaeche * hoehe
    return volumen
```

Durch die Verwendung eines Docstrings wird klar definiert, welche Parameter die Funktion erwartet und welchen Wert sie zurückgibt. Dies verbessert die Lesbarkeit und Wartbarkeit des Codes erheblich. Viele Entwicklungsumgebungen zeigen euch auch automatisch den Docstring an, wenn ihr die Funktion benutzen wollt. Hier ein Beispiel aus VSCode, in welchem erste eine Funktion mit Docstring definiert wird und dann beim Hovern über die Funktion ein Tooltip[1] mit dem Docstring erscheint:



Typ-Annotationen

Python ist eine dynamisch typisierte Sprache, was bedeutet, dass man nicht angeben muss, welchen Typen (float, int, bool, etc.) eine Variable haben soll. Allerdings helfen Typ-Annotationen dabei, Fehler frühzeitig zu erkennen und den Code verständlicher zu machen. Sie sind vor allem für größere Projekte nützlich.

Hier ist die `berechne_zylindervolumen`-Funktion mit Typ-Annotationen:

```
PI: float = 3.14159 # Auch bei der Erstellung einer Variable kann der Typ vorgegeben
werden

def berechne_zylindervolumen(radius: float, hoehe: float) -> float:
    """
    Berechnet das Volumen eines Zylinders mit Typ-Annotationen.

    Parameter:
    radius (float): Der Radius der Zylinderbasis.
    hoehe (float): Die Höhe des Zylinders.

    Rückgabewert:
    float: Das berechnete Volumen des Zylinders.
    """
    grundflaeche: float = PI * radius ** 2
```

[1] Ein Tooltip ist ein kleines Informationsfenster, das erscheint, wenn man mit der Maus über ein bestimmtes Element fährt.

```
    volumen: float = grundflaeche * hoehe
    return volumen
```

Die Syntax ist also `<Variablenname>: <Typ>`, sowohl bei Variablendeklaration als auch bei der Angabe von Funktionsparametern. Mit `-> <Typ>` Nach der Angabe der Parameter wird vorgegeben, welchen Typ der Rückgabewert der Funktion hat.

> **i** Statische Typisierung
>
> In statisch typisierten Sprachen wie C++ oder Java *müssen* die Typen von jeder Variable explizit angegeben werden, da der Compiler diese Information für die Generierung des ausführbaren Maschinencodes benötigt. Das mag zuerst wie ein eindeutiger Nachteil klingen, aber es ermöglicht die Erzeugung von etwas effizienteren Programmen und wirft viele Fehler schon frühzeitig auf.

Warum Typ-Annotationen hilfreich sind

Obwohl Python weiterhin dynamisch bleibt und die Typen nicht erzwungen werden, helfen sie beim Debuggen und in modernen Entwicklungsumgebungen, indem sie Fehler frühzeitig aufzeigen. So gut wie alle gängigen Python-Pakete sind nahezu vollständig mit Typ-Annotationen versehen.

Beispiel: Fehlererkennung durch Typ-Annotationen

Hier deifinieren wir eine Funktion, die einen Pandas DataFrame erwartet und ausgibt. Dann übergeben wir allerdings ein Numpy Array an die Funktion.

```
def process_dataframe(df: pd.DataFrame) -> None:
    """
    Verarbeitet einen Pandas DataFrame.
    """
    print(df.head())
```

```
arr = np.array([[1, 2, 3], [4, 5, 6]])

process_dataframe(arr)
```

```
AttributeError: 'numpy.ndarray' object has no attribute 'head'
```

In einigen Entwicklungsumgebungen würde nun schon vor Ausführen des Codes eine Warnung ausgegeben werden. Beispielsweise würde in VSCode eine rote, gewellte Linie unter `arr` erscheinen:

```python
import pandas as pd
import numpy as np

def process_dataframe(df: pd.DataFrame) -> None:
    """
    Verarbeitet einen Pandas DataFrame.
    """
    print(df.head())

# Hier passiert ein Fehler, da ein numpy Array übergeben wird
arr = np.array([[1, 2, 3], [4, 5, 6]])
process_dataframe(arr) # Entwicklungsumgebung oder Type-Checker warnt hier
```

> **TypeChecking in VSCode**
>
> Falls ihr auch VSCode nutzt um Python Code auszuführen, aber keine solchen Warnungen erhaltet, liegt es vermutlich daran, dass ihr folgendes noch nicht getan habt:
>
> 1. Das Pylance Plugin installieren
> 2. TypeChecking aktivieren in den Einstellungen: `File > Preferences > Settings`
>
> Siehe z.B. auch diese Anleitung

Zusammenfassung

Python-Funktionen ermöglichen die modularisierung und wiederverwendung von Code, wobei richtige Dokumentation mit Docstrings und Typ-Annotationen die Codequalität und -verständlichkeit maßgeblich verbessern.

> **Weitere Ressourcen**
>
> - 5 Ways To Improve Any Function in Python

Übungen

Übung 1

Welcher Funktionsaufruf wird einen Fehler verursachen?

```python
def function_1(arg1):
    print(arg1)

def function_2(arg1, arg2, arg3: int = 5):
    return arg1 + arg2 + arg3

def function_3():
    print(arg3)

def function_4(arg1: str = 5):
    print(arg1)

function_1(42)
function_2(5.0, arg2 = -15)
function_3()
function_4()
```

- (A) Funktionsaufruf 1
- (B) Funktionsaufruf 2
- (C) Funktionsaufruf 3
- (D) Funktionsaufruf 4

Übung 2

In dieser Übung sollst du eine Funktion erstellen, die den Mittelwert einer numerischen Spalte berechnet und das pro Gruppe in mit Daten aus einem DataFrame. Schreibe also eine Funktion `mittelwert_pro_gruppe()`, die:

1. Einen DataFrame, eine Gruppen-Spalte und eine Werte-Spalte als Eingabe nimmt
2. Den Mittelwert der Werte-Spalte für jede Gruppe berechnet
3. Das Ergebnis via `return` ausgibt.

```
import pandas as pd

# Beispieldatensatz
verkaufsdaten = pd.DataFrame({
    'Produkt': ['Laptop', 'Maus', 'Tastatur', 'Monitor', 'Laptop', 'Maus', 'Tastatur',
'Monitor'],
    'Kategorie': ['Computer', 'Zubehör', 'Zubehör', 'Computer', 'Computer', 'Zubehör',
'Zubehör', 'Computer'],
    'Verkäufer': ['Anna', 'Ben', 'Anna', 'Ben', 'Chris', 'Chris', 'Anna', 'Ben'],
    'Preis': [1200, 25, 45, 150, 999, 20, 50, 180],
    'Anzahl': [5, 10, 8, 3, 2, 15, 5, 4]
})

# Hier deine Funktion erstellen:
def mittelwert_pro_gruppe(df, gruppen_spalte, werte_spalte):
    # Dein Code hier
    pass
```

- (A) Geschafft

Übung 3

Überlege dir für folgende Funktion einen sinnvollen Funktionsnamen, Namen für die Argumente und ergänze Docstring und Typ-Annotationen:

```
import pandas as pd

def function(df, arg2, arg3, arg4):
    df_filtered = df[df[arg2] >= arg3]
    df_sorted = df_filtered.sort_values(by=arg4)
    return df_sorted

# Beispielaufruf
data = {
    "name": ["A", "B", "C", "D"],
    "value": [5, 15, 25, 8],
    "score": [50, 80, 90, 60]
}

df = pd.DataFrame(data)
function(df, "value", 10, "score")
```

- (A) Geschafft

Hinweis: Lösungen zu den meisten Übungsaufgaben findet ihr, indem ihr ganz oben rechts im Kapitel auf den `</> Code` Button klickt und dann entsprechend nach ganz unten scrollt. Es ist Absicht, dass dies etwas umständlich ist.