Python Grundlagen

Julian Huber & Matthias Panny

Funktionen und Module

🎯 Lernziele

- Studierende können Code in eigene Funktionen gliedern
- Studierende können Type Hints einsetzen
- Studierende können Code in eigene Module gliedern

Funktionen

Beispielfunktion

- $f(x) = 2x$

Funktionsdefinition

```python
def f(x):
  """this function's name is f. It takes a value x and returns a value y"""
  [...]
  y = 2*x
  return y
```

Funktionsaufruf

```python
f(2) # Funktionsaufruf, führt die Funktion mit Argument aus und gibt Rückgabe aus
```

C++-Code

- für jeden Datentypen eine eigene Funktion → Überladung

```cpp
int f(int x) {
  return 2*x;
}
double f(double x) {
  return 2*x;
}
```

Funktionen

- Funktionen als zentrales Konzept in praktisch jeder Programmiersprache

Eigenschaften von Funktionen

- können eine Rückgabe haben → mit `Tupel` auch mehrere Werte
- werden erst beim Aufruf mit den übergebenen Parameter (z.B. `x`) ausgeführt

Eigenschaften von `guten` Funktionen

- sollten genau eine Sache tun
- sollten kürzer als 20 Codezeilen sein
- sind im besten Fall mittels Docstring beschrieben `"""`

Funktionen

Anatomie einer Python-Funktion

- `def` → Definition einer Funktion
- Funktionsname
- Argumente in runden Klammern
- Rückgabe nur durch `return`-Statement bzw. Type Hints ersichtlich

Beispiel

```python
# Beispiel: eine Funktion die das übergebene Argument um 1 erhöht
# arg1 stellt hierbei das Argument dar, das übergeben wird
def function_name(arg1):
  print(str(arg1) + " plus 1 is: ") # Innerhalb der Funktion
  # können beliebige Aktionen ausgeführt werden
  return (arg1) + 1  # Das return-Kennwort gibt an, was die Funktion,
  # wenn die Aufgerufen wird zurückgeben soll. Dies ist optional.

# Beispiel: eine Funktion die zwei beliebige Zahlen addiert
# Die beiden Zahlen werden als Argumente übergeben
def add_two_numbers(x : float, y : float) -> float:
  return x + y  # Diese Funktion gibt einen Wert zurück
  # und zwar die Summe der beiden Zahlen

add_two_numbers(1,2) # Gibt 3 zurück

z = add_two_numbers(2,3) # Hier wird die Rückgabe 5
# direkt in einer neuen Variable z gespeichert
```

Argumente von Funktionen

- Python unterstützt unterschiedliche Arten von Argumenten
  - positionale Argumente
  - Schlüsselwortargumente
  - variable Anzahl von Argumenten

Positionale Argumente

- Wie in C/C++ → Übergebene Parameter müssen mit erwarteten übereinstimmen → Reihenfolge

```
def describe_pet(animal_type, pet_name):
    """
    Display information about a pet.
    """
    print(f"I have a {animal_type}.")
    print(f"My {animal_type}'s name is {pet_name.title()}.")

describe_pet("hamster", "harry")
```

✍️ Aufgabe

- Geben Sie eine Liste von Personen `my_persons` an, schreiben Sie eine Funktion `look_for_person`, die die Position des ersten Auftretens der Person in der Liste herausfindet und diese ausgibt.
- Die Liste `persons` und der String `person_to_look_for` sind die Parameter der Funktion.

```
my_persons = ["Julian", "Andreas", "Philipp", "Manuel"]
my_person = "Andreas"

def look_for_person(persons, person_to_look_for):
  position = 0
  # Lösung hier fortsetzen
```

Musterlösung - `person_lookup.py`

- Liste wird iteriert
- Wenn Person gefunden wird, wird die Position ausgegeben

Argumente von Funktionen

Schlüsselwortargumente

- Schlüsselwortargumente sind Namen-Wert-Paare

```
my_persons = ["Tim", "Julian", "Andreas", "Philipp", "Manuel", "Jim"]
look_for_person(persons=my_persons, person_to_look_for="Tim")
```

- Reihenfolge ist egal, Zuordnung erfolgt ausschließlich über Namen des Arguments

🤓 Argumente von Funktionen

- Sowohl positionale als auch Schlüsselwortargumente können in variabler Anzahl übergeben werden
- Python bietet hierfür spezielle Syntax mit den unpacking Operatoren `*` bzw. `**`
- 🤓 kann auch zum unpacking von Listen und Dictionaries verwendet werden → `list_unpacking.py`

Variable Anzahl von positionalen Argumenten

- Mit `*args` können beliebig viele positionale Argumente übergeben werden → Asterisk (`*`) vor dem Argumentsnamen → als `Tupel`

```
def look_for_person(*args, person_to_look_for):
  for person in args:
    if person == person_to_look_for:
      return args.index(person)

look_for_person("Tim", "Julian", "Andreas", "Philipp", "Manuel", person_to_look_for="Tim")
```

Argumente von Funktionen

Variable Anzahl von Schlüsselwortargumenten

- Mit `**kwargs` können beliebig viele Schlüsselwortargumente übergeben werden → 2 Asterisk (`**`) vor dem Argumentsnamen → als `Dictionary`

```
def concatenate(**kwargs):
  result = ""
  for value in kwargs.values():
    result += value
  return result

concatenate(a="Hallo", b=" das ", c="ist", d=" ein ", e="Test")
```

⚠ Reihenfolge der Argumente

- Reihenfolge muss eingehalten werden:
  - normale Argumente
  - `*args`
  - `**kwargs`

```
def my_function(a, b, *args, **kwargs):
  pass
```

Rückgabewerte von Funktionen

- Funktionen können einen Wert oder eine Reihe von Werten zurückgeben → `return`
- Rückgabe z.B. von einzelnen Werten, Tupeln, Listen, etc.

Beispiel

```
def get_formatted_name(first_name, last_name):
    """
    Return a full name, neatly formatted.
    """
    full_name = first_name + ' ' + last_name
    return full_name.title()

person_1 = get_formatted_name('julian', 'huber')
print(person_1)
#> Julian Huber
```

Type Hints

Probleme - typehints_problem.py

```
def greeting(name):
  return 'Hello ' + name
greeting(123)
#> TypeError: can only concatenate str (not "int") to str
```

- Dynamic Typing in Python kann zu Fehlern führen, z.B. wenn einer Funktion ein falscher Datentyp übergeben wird

Mögliche Lösungsansätze

- Docstrings zur Dokumentation wie die Funktion genutzt werden soll
- Type Hints zur Angabe der erwarteten Datentypen
- → ⚠ Problem: Beides wird nicht vom Interpreter enforced

Type Hints - Lösung mit Docstrings

- Wird von Entwicklungsumgebung angezeigt, sobald der Funktionsname getippt wird
- Umfangreich und nicht standardisiert (es gibt verschiedene Standards für Docstrings, z.B. NumPy/SciPy-Stil, Google-Stil, etc.)

Beispiel - `typehints_docstring.py`

```python
def greeting(name):
    """
    This function can print a greeting to a name

    Parameters
    ----------
    name : string
        the name of the person to greet

    Returns
    -------
    string
        a greeting containing the name
    """
    return 'Hello ' + name
```

Type Hints - Lösung mit Type Hints

- Wird vom Interpreter nicht enforced
- Trotzem sinnvoll → bietet Notation, um andere Programmierer im Code über die erwarteten Datentypen zu informieren
- Linter können die Typen überprüfen

Beispiel - typehints.py

```
a_name: str

def greeting(name: str) -> str:
  return 'Hello ' + name

a_name = 42
greeting(a_name)
#> TypeError: can only concatenate str (not "int") to str
```

Module

Motivation

- Wir haben nun Funktionen implementiert die eine bestimmte Aufgabe erfüllen & diese sauber dokumentiert → Was nun?

Übertragbarkeit von Programmcode

- Wiederverwendbarkeit von Code soll forciert werden
- Code soll in übersichtliche, klar definierte Einheiten gegliedert werden
- Egal in welchem Projekt sie arbeiten, Sie können die Funktionen immer wieder verwenden
- In C/C++ haben wir hierfür Programmteile in Header-Dateien ausgelagert

Module

- Python nutzt hierfür Module → Unterprogramm als Sammlung von Funktionen, Klassen und Variablen

Module in Python

- Einige Module sind automatisch mit in der Python-Umgebung vorinstalliert → z.B. `math`, `random`, `os`, `datetime`, etc.
- Import des Moduls über `import`-Statement und Modulenamen:

```
import math
import os
print(math.pi)
```

- Module können selbst geschrieben werden oder auch von anderen Entwicklern bezogen werden → nächste Einheit

Eigene Module

- Python-Module sind ganz normale Python-Dateien (`*.py`)
- Üblicherweise sind dies Datein im Arbeitsverzeichnis des Projektes → i.d.R. der Ordner, der in VS Code geöffnet ist
- Jede dieser `*.py`-Dateien kann als Modul importiert werden → hierbei müssen einige Aspekte beachtet werden

Import von Modulen

- Als Beispiel legen wir nun in unserem Projekt zwei Skripte an

Unser Modul - my_module_without_main.py

```
def my_function(input):
  output = input * 2
  return output

print("Modul wurde geladen!")
```

Import des gesamtem Moduls - main_import_all.py

```
import my_module_without_main
# Importiert den gesamten Inhalt des Moduls

# Aufruf über Modulnamen
my_module_without_main.my_function(3)
#> 6
```

Import eine Funktion aus dem Modul - main_import_func.py

```
from my_module_without_main import my_function
# Importiert nur die genannte Funktion

my_function(3) # Direkter Aufruf
#> 6
```

Definition des Hauptprogramms

my_module_without_main.py

```
def my_function(input):
  output = input * 2
  return output

print(f"Ich stehe in der Datei: {__name__}")
```

main.py

```
import my_module_without_main # Importiert den gesamten Inhalt der
```

- Es wird der gesamte Programmcode aus `my_module_without_main` ausgeführt

Definition des Hauptprogramms

Folge

- Die Funktion `my_function(...)` wird definiert
- Der `print()` Befehl wird ebenfalls ausgeführt
  - `python main.py`:
    `Ich stehe in der Datei: my_module_without_main`
  - `python my_module_without_main.py`:
    `Ich stehe in der Datei: __main__`
- Dies ist nicht immer wünschenswert, da wir meist zunächst nur die Funktionen definieren möchten

⚠ Vorsicht!

- Die Variable `__name__` nimmt immer den Wert `__main__` an, wenn die Datei selbst direkt ausgeführt wird
- Dies is *unabhängig* vom eigentlichen Dateinamen

Hauptprogramm

Fallunterscheidung mittels `__name__=="__main__"`

- wird die Datei `my_module_with_main.py` selbst ausgeführt gilt:

```
__name__ == "__main__"
#> True
```

- wird die Datei als Modul importiert gilt:

```
__name__ == "__main__"
#> False
```

Anpassung von `my_module_with_main.py`

```
def my_function(input):
  output = input * 2
  return output

if __name__ == "__main__":
  print("Datei wurde direkt aufgerufen und die Main wird ausgeführt")
else:
  print("Datei wurde als Modul aufgerufen")

print(f"Ich stehe in der Datei: {__name__}")
```

Hauptprogramm

main.py

- Beste Practice: Hauptprogramm das als Einstiegspunkt dient `main.py` → für jeden Programmierer direkt klar
- Der Code des Hauptprogramms sollte in folgende `if`-Statement verpackt werden:

```
if __name__ == "__main__":
    ...
```

- Erlaubt Trennung zw. Import und Ausführung des Programms → wird bei automatisierten Softwaretests benötigt

🤓 Schnelle Tests von Module

- Wenn wir ein Modul schreiben, können wir es auch direkt testen → direktes ausführen des Codes
- Tests innerhalb des `if`-Statements (wie oben):

```
if __name__ == "__main__":
    ...
```

  verpacken → wird nur beim Testen ausgeführt

Sys-Path

- Grundsätzlich werden Module nur gefunden wenn sie im
  - Systempfad oder im
  - Arbeitsverzeichnis liegen

Standardverzeichnisse für Module

- Der Systempfad kann über `sys.path` abgefragt werden → `List` mit Pfaden als Strings

```
import sys
sys.path
#> ['', 'C:\\Users\\john\\Documents\\Python\\doc', 'C:\\Python36\\Lib\\idlelib',
#> 'C:\\Python36\\python36.zip', 'C:\\Python36\\DLLs', 'C:\\Python36\\lib',
#> 'C:\\Python36', 'C:\\Python36\\lib\\site-packages']
```

- Eigene Verzeichnisse können jedoch hinzugefügt werden

```
sys.path.append(r'C:\Users\john')
```

Verwendung von Underscores z.B. `__name__`

Single Underscore

- `_single_leading_underscore` bei Attributen (Variablen) und Methoden (Funktionen in Klassen), die nur für den internen Gebrauch bestimmt sind,
- aber dies nicht erzwingt

Double Underscore

- `__double_leading_underscore` bei Attributen (Variablen) und Methoden (Funktionen in Klassen), die nur für den internen Gebrauch bestimmt sind
- und von außen nicht sichtbar sind

Dunder

- `__name__` Variablen und Methoden, die eigentlich nur für den internen Gebrauch der Programmiersprache Python gedacht sind
- Werden i.d.R. nicht von uns als Programmierer:innen genutzt
- `dir(int)`