Programmiervorkurs 2025

Schleifen & Funktionen

Vorkurs-Schnupsis

30. September 2025

TU Darmstadt

while

Wiederholung

```
1 i = 0
2 s = 'Lorem'
3 while i < len(s):
4     print(f'Buchstabe an Stelle {}: {s[i]}')
5     i += 1
```

Tuple

Tuple

- Listen sind praktisch, aber nicht immer braucht man Datenstrukturen mit veränderlicher Größe
- Ist nur eine feste Anzahl an Werten erwünscht, können Tuple verwendet werden
- Tuple werden mittels `t1 = (wert1, wert2, ..., wertN)` definiert
- Zugriffe auf einzelne Einträge erfolgen mit der gleichen Syntax wie bei Listen mit `t1[0]`

Tuple

- Ebenfalls verhalten sich Slices von Tuples analog zu Listen
- Sie können zu Listen mit `list()` konvertiert werden und vice versa mit `tuple()`
- Wichtiger Unterschied zu Listen: Tuple können nicht modifziert werden, nur überschrieben
- Aber: Tuple haben bessere Performance

Dictionaries

dict

Erstellung

- Dictionaries erlauben einen freien Zugriff mit beliebigen Schlüsselwerten

dict

Erstellung

- Dictionaries erlauben einen freien Zugriff mit beliebigen Schlüsselwerten
- Schlüssel dürfen unterschiedliche Typen haben

dict

Erstellung

- Dictionaries erlauben einen freien Zugriff mit beliebigen Schlüsselwerten
- Schlüssel dürfen unterschiedliche Typen haben
- Werte können ebenfalls unterschiedliche Typen haben

dict

Erstellung

- Dictionaries erlauben einen freien Zugriff mit beliebigen Schlüsselwerten
- Schlüssel dürfen unterschiedliche Typen haben
- Werte können ebenfalls unterschiedliche Typen haben
-

```
d = dict()
d = {
    'a': 5,
    'x': 'lorem'
}
```

dict

Zugriffe

```
d['a']         # => 5
d['b']         # => Fehler
d.get('b', 0) # => 0
d.get('a', 0) # => 5
```

dict

Zugriffe

```
d['a']         # => 5
d['b']         # => Fehler
d.get('b', 0) # => 0
d.get('a', 0) # => 5
```

- Mit `'a' in d` kann man überprüfen, ob `'a'` im Dictionary ist

dict

Zugriffe

```
d['a']         # => 5
d['b']         # => Fehler
d.get('b', 0) # => 0
d.get('a', 0) # => 5
```

- Mit `'a' in d` kann man überprüfen, ob `'a'` im Dictionary ist
- Elemente aus Dictionary entfernen: `del d['a']`

Schleifen II

while

Wiederholi

- Erinnern wir uns zurück an unsere while-Schleife
- Häufig wollen wir über eine Liste oder ähnliches iterieren
- Manuelles Hochzählen ist unpraktisch und ein Vergessen resultiert in Endlosschleifen

```
1 i = 0
2 s = 'Lorem'
3 while i < len(s):
4     print(f'Buchstabe an Stelle {}: {s[i]}')
5     i += 1
```

for

- Neben `while`-Schleifen gibt es auch noch `for`-Schleifen
- Diese

Grundsätzlich sieht der Aufbau wie folgt aus:

```
1 for var in <iterable>:
2     do_something(var)
```

Merksatz: ”Führe etwas für jedes <X> in <Y> aus.”

for

- Die Nutzung kann, anders als bei while, etwas unterschiedlich ausfallen

for

- Die Nutzung kann, anders als bei while, etwas unterschiedlich ausfallen
- Dafür gibt es mehrere Hilfsfunktionen, welche wir im Folgenden erläutern wollen:

for

- Die Nutzung kann, anders als bei while, etwas unterschiedlich ausfallen
- Dafür gibt es mehrere Hilfsfunktionen, welche wir im Folgenden erläutern wollen:
  - for-each

for

- Die Nutzung kann, anders als bei while, etwas unterschiedlich ausfallen
- Dafür gibt es mehrere Hilfsfunktionen, welche wir im Folgenden erläutern wollen:
  - for-each
  - range

for

- Die Nutzung kann, anders als bei while, etwas unterschiedlich ausfallen
- Dafür gibt es mehrere Hilfsfunktionen, welche wir im Folgenden erläutern wollen:
  - for-each
  - range
  - enumerate

for

- Die Nutzung kann, anders als bei while, etwas unterschiedlich ausfallen
- Dafür gibt es mehrere Hilfsfunktionen, welche wir im Folgenden erläutern wollen:
  - for-each
  - range
  - enumerate
  - zip

for

each

- Bei Listen oder Strings möchte man häufig eine Aktion mit jedem Element machen
- Hierbei helfen ”for-each”-Schleifen

Beispiel:

```
1 s = 'Lorem Ipsum'
2 for c in s:
3     action(c)
```

for

range

- Erinnern wir uns an die letzte Vorlesung und die `while`-Schleife:

```
1 i = 0
2 while i < num:
3     action(s[i])
4     i += 1
```

for

range

- Erinnern wir uns an die letzte Vorlesung und die `while`-Schleife:

```
1 i = 0
2 while i < num:
3     action(s[i])
4     i += 1
```

- Dies kann man kürzer schreiben mit einer ”for-range”-Schleife

for

range

- Erinnern wir uns an die letzte Vorlesung und die `while`-Schleife:

```
1 i = 0
2 while i < num:
3     action(s[i])
4     i += 1
```

- Dies kann man kürzer schreiben mit einer ”for-range”-Schleife

  $\Rightarrow$ Hierbei kann man auch nicht das Inkrementieren des Index vergessen

```
1 for i in range(len(s)):
2     action(s[i])
```

for

range

- Aber wie funktioniert `range()`?

for

range

- Aber wie funktioniert `range()`?
- Im Prinzip wie Slices:

```
1 lst = [9,1,8,6,7,2,1,8]
2 lst[1:7:2]
3 # => [1, 6, 2]
4 # => Werte an den Indizes 1, 3, 5
5 list(range(1,7,2))
6 # => Ergibt [1, 3, 5]
```

for

range

- Aber wie funktioniert `range()`?
- Im Prinzip wie Slices:

```
1 lst = [9,1,8,6,7,2,1,8]
2 lst[1:7:2]
3 # => [1, 6, 2]
4 # => Werte an den Indizes 1, 3, 5
5 list(range(1,7,2))
6 # => Ergibt [1, 3, 5]
```

- Allgemeine Funktionsweise: `range(start, end, step)`

for

enumerate

Teilweise möchte man zum Element auch den zugehörigen Index haben, aber keine range-Schleife schreiben.
Dafür kann man eine Schleife mit enumerate umsetzen:

```
1 for i,c in enumerate(s):
2     action(i, c)
```

for

zip

Für den Fall, dass man über zwei gleich lange Listen iterieren möchte, gibt es die Funktion ”zip”:

```
1 for c1,c2 in zip(s1, s2):
2     action(c1, c2)
```

for

dict

Auch Dictionaries können in Schleifen verwendet werden:

```
for key in dictionary:
    value = dictionary[key]
```

Pakete importieren

Paketimport

- Häufig möchte man Funktionalitäten nicht mehrfach erfinden müssen

Paketimport

- Häufig möchte man Funktionalitäten nicht mehrfach erfinden müssen
- Deshalb bindet man Funktionsbibliotheken ein

Paketimport

- Häufig möchte man Funktionalitäten nicht mehrfach erfinden müssen
- Deshalb bindet man Funktionsbibliotheken ein
- In Python (und Java) importiert man diese aus anderen Paketen

Paketimport

- Häufig möchte man Funktionalitäten nicht mehrfach erfinden müssen
- Deshalb bindet man Funktionsbibliotheken ein
- In Python (und Java) importiert man diese aus anderen Paketen
- Die Syntax ist dafür: `import <paket-name>`

Paketimport

Um zum Beispiel mathematische Funktionen nutzen können, gibt es im Python-Standard das `math`-Paket:

Import

```
>>> import math
>>> math.sin(math.pi * 0.5)
1.0
```

Dateioperationen

Dateioperationen

- Wir haben bereits `print()` und `input()` kennengelernt, um mit der Konsole zu interagieren

Dateioperationen

- Wir haben bereits `print()` und `input()` kennengelernt, um mit der Konsole zu interagieren
- Oft ist es aber nötig, Daten in Dateien zu speichern oder aus Dateien zu lesen, um Daten permanent zu speichern

Dateioperationen

- Wir haben bereits `print()` und `input()` kennengelernt, um mit der Konsole zu interagieren
- Oft ist es aber nötig, Daten in Dateien zu speichern oder aus Dateien zu lesen, um Daten permanent zu speichern

⇒ Python bietet dafür eine einfache Schnittstelle

Datei öffnen

```
1 f = open('<path>', '<mode>')
2 # Beispiel
3 f = open('main.py', 'r')
```

Datei öffnen

```
1 f = open('<path>', '<mode>')
2 # Beispiel
3 f = open('main.py', 'r')
```

- Der Pfad kann relativ oder absolut angegeben werden
- Beachtet dabei, in welchem Ordner euer Programm ausgeführt wird!

Datei öffnen

Für den Parameter ’mode’ gibt es folgende Möglichkeiten:

Kürzel	Bedeutung
r	Lesen
w	Schreiben
a	Anhängen
x	Dateierstellung
+	Lesen + Schreiben

mit Zusatz

Kürzel	Bedeutung
t	Text
b	Binärdaten

Datei öffnen

Für den Parameter ’mode’ gibt es folgende Möglichkeiten:

Kürzel	Bedeutung
r	Lesen
w	Schreiben
a	Anhängen
x	Dateierstellung
+	Lesen + Schreiben

mit Zusatz

Kürzel	Bedeutung
t	Text
b	Binärdaten

- Es ist immer nur ein Modus möglich, dieser kann nicht gewechselt werden, ohne die Datei neu zu öffnen
- Bis auf ’x’ überprüft die Operation nicht, ob eine Datei bereits existiert und überschreibt diese im Zweifelsfall

Lesen

- Der gesamte Inhalt einer Datei kann in einem großen String oder Buffer gelesen werden: `content = f.read()`

Lesen

- Der gesamte Inhalt einer Datei kann in einem großen String oder Buffer gelesen werden: `content = f.read()`
- `section = f.read(5)` liest die nächsten 5 Zeichen ein

Lesen

- Der gesamte Inhalt einer Datei kann in einem großen String oder Buffer gelesen werden: `content = f.read()`
- `section = f.read(5)` liest die nächsten 5 Zeichen ein
- Mit `line = f.readline()` kann man eine einzelne Zeile einlesen

Lesen

- Der gesamte Inhalt einer Datei kann in einem großen String oder Buffer gelesen werden: `content = f.read()`
- `section = f.read(5)` liest die nächsten 5 Zeichen ein
- Mit `line = f.readline()` kann man eine einzelne Zeile einlesen
- `lines = f.readlines()` liest eine ganze Datei als Liste von Zeilen einlesen

Lesen

- Der gesamte Inhalt einer Datei kann in einem großen String oder Buffer gelesen werden: `content = f.read()`
- `section = f.read(5)` liest die nächsten 5 Zeichen ein
- Mit `line = f.readline()` kann man eine einzelne Zeile einlesen
- `lines = f.readlines()` liest eine ganze Datei als Liste von Zeilen einlesen
  - Diese Liste kann wie gewohnt mit einer `for`-Schleife durchlaufen werden:
    ```
    for line in lines:
    ```

Schreiben

- Schreiben verhält sich ähnlich zum Lesen

Schreiben

- Schreiben verhält sich ähnlich zum Lesen
- Zu schreibende Daten (String oder Bytes) werden an die Funktion `f.write(<content>)` übergeben

Schreiben

- Schreiben verhält sich ähnlich zum Lesen
- Zu schreibende Daten (String oder Bytes) werden an die Funktion `f.write(<content>)` übergeben
- Beispiel: `f.write('Lorem ipsum')`

Schreiben

- Schreiben verhält sich ähnlich zum Lesen
- Zu schreibende Daten (String oder Bytes) werden an die Funktion `f.write(<content>)` übergeben
- Beispiel: `f.write('Lorem ipsum')`
- Der Daten werden an die aktuelle Schreibposition angefügt

Datei schließen

- Offene Dateien ”vermüllen” das System und sind deshalb unerwünscht
- Schreiboperationen sind erst garantiert abgeschlossen, wenn eine Datei geschlossen wurde

  Sonst ist ein Verlust an Daten möglich
- Einfache Funktion in Python: `f.close()`

Nutzung

- Offizielle Empfehlung von Python:

```
1 with open(path) as f:
2     content = f.read()
```

- `f.close()` wird automatisch ausgeführt

Funktionen

Funktionen

- Grundsätzlich kennt ihr bereits Funktionen und habt diese verwendet:

Funktionen

- Grundsätzlich kennt ihr bereits Funktionen und habt diese verwendet:

  $\Rightarrow$ `print()`, `input()`, etc.

Funktionen

- Grundsätzlich kennt ihr bereits Funktionen und habt diese verwendet:

  $\Rightarrow$ `print()`, `input()`, etc.
- Aber warum gibt es sie?

Warum Funktionen?

```
1 counter = 1
2
3 name = input('Name: ')
4 print(f'Hallo {name}, du bist der {counter}. Besucher')
5 counter += 1
6
7 name = input('Name: ')
8 print(f'Hallo {name}, du bist der {counter}. Besucher')
9 counter += 1
```

Das geht schöner!

Warum Funktionen?

```
counter = 1

def greet():
    global counter
    name = input('Name: ')
    print(f'Hallo {name}, du bist der {counter}. Besucher')
    counter += 1

greet()
greet()
```

Warum Funktionen?

- Funktionen reduzieren Redundanz

Warum Funktionen?

- Funktionen reduzieren Redundanz

  $\Rightarrow$ Erspart Programmierzeit

Warum Funktionen?

- Funktionen reduzieren Redundanz

  $\Rightarrow$ Erspart Programmierzeit
- Code wird dadurch häufig lesbarer (solange sinnvolle Namen gewählt werden)

Warum Funktionen?

- Funktionen reduzieren Redundanz

  $\Rightarrow$ Erspart Programmierzeit
- Code wird dadurch häufig lesbarer (solange sinnvolle Namen gewählt werden)
- Lokale Variablen und Logik können abgekapselt werden

Warum Funktionen?

- Funktionen reduzieren Redundanz

  $\Rightarrow$ Erspart Programmierzeit
- Code wird dadurch häufig lesbarer (solange sinnvolle Namen gewählt werden)
- Lokale Variablen und Logik können abgekapselt werden
- Unnötige Details werden (nach außen) versteckt

Definition & Syntax

Beispiele

Keine Parameter, kein Rückgabewert:

```
1 def fun1():
2     print('Keine Parameter')
```

Definition & Syntax

Beispiele

Keine Parameter, kein Rückgabewert:

```
1 def fun1():
2     print('Keine Parameter')
```

Ein Parameter, kein Rückgabewert:

```
1 def fun2(param1):
2     print(f'Nur ein Parameter: {param1}')
```

Definition & Syntax

Beispiele

Mit Rückgabewert:

```
1 def fun3(param1, param2):
2     print(f'Zwei Parameter: {param1} und {param2}')
3     return param1 + param2
```

Lokale Variablen

- Außerhalb der Funktionen sind lokale Variablen nicht sichtbar bzw. benutzbar

```
1 def greet():
2     name = input()
3     print(f'Hallo, {name}')
4
5 greet()
6 print(name) # => NameError: name 'name' is not defined
```

Lokale Variablen

- Deklaration von Variablen in Funktionen (einfache Zuweisung) ”verdecken” globale Variablen

```
1 counter = 1
2 def greet():
3     counter = 1
4     counter += 1
5 greet()
6 print(counter)
```

Lokale Variablen

- Möchte man globale Variablen dennoch nutzen, benötigt man eine Deklaration der Variable innerhalb der Funktion mit `global`

```
1 counter = 1
2 def greet():
3     global counter
4     counter += 1
5 greet()
6 print(counter)
```

Nutzung

- Die Verwendung von Funktionen kennen wir bereits von `print()`, `str()`

Beispiel:

```
1 def add1(a, b):
2     return a + b + 1
3
4 c = add1(5, 3)
```

Nutzung

- Die Verwendung von Funktionen kennen wir bereits von `print()`, `str()`
  $\Rightarrow$ Syntax: `rückgabewert = funktionsname(parameter)`

Beispiel:

```
1 def add1(a, b):
2     return a + b + 1
3
4 c = add1(5, 3)
```

Nutzung

- Die Verwendung von Funktionen kennen wir bereits von `print()`, `str()`
  $\Rightarrow$ Syntax: `rückgabewert = funktionsname(parameter)`
- Je nach Definition werden keine Parameter oder Rückgabewert gebraucht

Beispiel:

```
1 def add1(a, b):
2     return a + b + 1
3
4 c = add1(5, 3)
```

Nutzung

- Die Verwendung von Funktionen kennen wir bereits von `print()`, `str()`
  $\Rightarrow$ Syntax: `rückgabewert = funktionsname(parameter)`
- Je nach Definition werden keine Parameter oder Rückgabewert gebraucht
- Ein Rückgabewert muss nicht genutzt werden, man möchte dies aber üblicherweise tun

Beispiel:

```
1 def add1(a, b):
2     return a + b + 1
3
4 c = add1(5, 3)
```

Nutzung

- Da dies häufig für Verwirrung sorgt:

Nutzung

- Da dies häufig für Verwirrung sorgt:
  - Der Name von eingegebenen Variablen und Funktionsparametern muss nicht identisch sein

Nutzung

- Da dies häufig für Verwirrung sorgt:
  - Der Name von eingegebenen Variablen und Funktionsparametern muss nicht identisch sein
  - Beispiel: `c = add1(x, 5)` bei Funktionssignatur `def add1(a, b):`

Nutzung

- Da dies häufig für Verwirrung sorgt:
  - Der Name von eingegebenen Variablen und Funktionsparametern muss nicht identisch sein
  - Beispiel: `c = add1(x, 5)` bei Funktionssignatur `def add1(a, b):`
- Wichtig ist, dass sich eine Funktion nicht unendlich selbst aufruft

Nutzung

- Da dies häufig für Verwirrung sorgt:
  - Der Name von eingegebenen Variablen und Funktionsparametern muss nicht identisch sein
  - Beispiel: `c = add1(x, 5)` bei Funktionssignatur `def add1(a, b):`
- Wichtig ist, dass sich eine Funktion nicht unendlich selbst aufruft

  $\Rightarrow$ Mehr dazu morgen!

Quiz

Quiz

- Welches Ergebnis hat `s[3:0:-2]` für `s = [1,3,6,8,1,2]`?

Quiz

- Welches Ergebnis hat `s[3:0:-2]` für `s = [1,3,6,8,1,2]`?
- Welche Möglichkeiten gibt es, Werte (sofern vorhanden) aus einem Dictionary zu bekommen, ohne dass es dabei zu Fehlern kommt?

Quiz

- Welches Ergebnis hat `s[3:0:-2]` für `s = [1,3,6,8,1,2]`?
- Welche Möglichkeiten gibt es, Werte (sofern vorhanden) aus einem Dictionary zu bekommen, ohne dass es dabei zu Fehlern kommt?
- Welche Indizes werden in der Schleife `for i in range(1, 8, 3):` durchlaufen?

Ausblick

Ausblick

In der morgigen Vorlesung erwarten uns:

- Rekursion / Selbstreferenz von Funktionen

Ausblick

In der morgigen Vorlesung erwarten uns:

- Rekursion / Selbstreferenz von Funktionen
- Vergleich mit Java