Python für Linguisten

Dozentin: Wiebke Petersen & Co-Dozentin: Esther Seyffarth

4. Foliensatz
bedingte Anweisungen und boolsche Werte,
Schleifen

Vergleichsoperatoren und Boolsche Werte

- numerische Vergleichsoperatoren: `1 < 2, 2 > 1, 1 == 1, 1 != 2`
  - kleiner als, Beispiel: `1 < 2`
  - größer als, Beispiel: `2 > 1`
  - gleich, Beispiel: `1 == 1`
  - ungleich, Beispiel: `1 != 2`
  - kleiner-gleich, Beispiel `1 <= 2`
  - größer-gleich, Beispiel `1 >= 2`
- Strings vergleichen:
  ```
  'this_string' == 't_string'
  'this_string' != 't_string'
  ```
- Typen prüfen:
  ```
  type(this_var) == str
  ```
  besser: `isinstance(this_var, str)`
- Vergleichsoperationen haben als Ergebnis immer einen boolschen Wert vom Datentype `bool`.
- `True` bedeutet logisch wahr. Beispiel: 2+3==5
- `False` bedeutet logisch falsch. Beispiel: 2<1

bedingte Anweisung: if

- Das Keyword `if` prüft, ob eine Bedingung zum boolschen Wert `True` evaluiert wird.
- `if` hat folgende Syntax:

```
if bedingung:
    # Beginn des Anweisungsblocks
    # beinhaltet entweder beliebig viele (>0) Anweisungen
    # oder die "Platzhalter"-Anweisung pass
    pass
```

- Der Code im `if`-Block wird nur ausgeführt, wenn die Bedingung zu `True` evaluiert.

```
if number < 2:
    print(number, "kleiner 2")
if number == 2:
    print(number, "gleich 2")
if number != 2:
    print(number, "ungleich 2")
```

if-Blöcke und else-Blöcke

- `if` wird meistens zusammen mit `else` verwendet.
- `else` leitet einen alternativen Anweisungsblock ein, der immer dann ausgeführt wird, wenn die `if`-Bedingung nicht zutrifft.
- Daher erlaubt `else` keine weiteren Bedingungen: Es deckt alle denkbaren Fälle ab, die nicht in den `if`-Block führen, und muss nicht näher beschrieben werden.

```
if number > 2:
    print(number, "groesser 2")
else:
    print(number, "kleiner gleich 2")
```

Syntactic Sugar: elif

- Wenn Sie mehr als eine Bedingung explizit prüfen wollen, gibt es zwei Möglichkeiten:
  - Sie können mehrere `if`/`else`-Konstruktionen verschachteln (**linkes Beispiel**).
  - Sie können das Keyword `elif` verwenden (**rechtes Beispiel**). Die Bedingung, mit der der `elif`-Block eingeleitet wird, wird nur geprüft, falls die Bedingung der entsprechenden `if`-Zeile nicht zutrifft.
- `elif` kann auch mehrfach verwendet werden (um mehrere Bedingungen, die sich gegenseitig ausschließen, abzufragen).

```
if number > 2:
    print(number,"groesser 2")
else:
    if number < 2:
        print(number,"kleiner 2")
    else:
        print(number,"gleich 2")
```

```
if number > 2:
    print(number,"groesser 2")
elif number < 2:
    print(number,"kleiner 2")
else:
    print(number,"gleich 2")
```

Unterschied zwischen if und elif

- Was ist der Unterschied zwischen:

```
if number == 2:
    print(number,"==2")
if number >= 2:
    print(number,">=2")
```

```
if number == 2:
    print(number,"==2")
elif number >= 2:
    print(number,">=2")
```

while-Schleifen

- Das Keyword `while` leitet Anweisungsblöcke ein, die immer wieder ausgeführt werden sollen. Die Bedingung nach dem `while` muss erfüllt sein, damit die Schleife erneut ausgeführt wird.
- Hat die Bedingung zu Beginn der Schleife den Wert `False`, verlässt der Python-Interpreter die Schleife.
- Syntax:

```
while boole:
    statement
```

Beispiel:

```
super_secret_password = "xyz"
my_guess = input("Please enter password: ")
while my_guess != super_secret_password:
    print("Incorrect password, please try again.")
    my_guess = input("Please enter password: ")

print("Access granted.")
```

Übungsaufgabe: Zahlen raten

- Schreiben Sie nun ein Programm, das eine Zufallszahl erzeugt, die vom Nutzer geraten werden soll.
- Um Zufallszahlen zu erzeugen, benötigen Sie die folgenden Befehle:

```
import random # imports stehen stets ganz oben in der Code-Datei
zahl = random.randint(kleinste, groesste)
# "kleinste" und "groesste" sind die Schranken, zwischen
# denen sich die Zahl befindet
```

- Solange die Nutzereingabe inkorrekt ist, soll das Programm Sie informieren, ob die zuletzt eingegebene Zahl zu klein oder zu groß ist.
- Bei Eingabe der richtigen Zahl haben Sie gewonnen und das Programm wird beendet.

Boolesche Operatoren bei if und while

- Sie können mit `if` und `while` auch komplexere Bedingungen abfragen, indem Sie mehrere boolsche Werte mit den Operatoren `and`, `or` und `not` verknüpfen.
- Die Operatoren verhalten sich erwartungsgemäß. Bei vielen Bedingungen empfiehlt es sich, Klammern zu setzen.
- Beispiel:

```
if zahl < lower or zahl > upper:
    print("Bitte eine Zahl zwischen", str(lower), "und",
            str(upper), "eingeben!")
```

```
finished = False
while not finished: # entspricht "while finished == False:"
    cont = input("Continue? y/n ")
    if cont == "n":
        finished = True
```

Boolsche Verwendung nicht-boolscher Datentypen

- Auch die bereits behandelten Datentypen Strings, Listen und Integers können in Wahrheitswerte umgewandelt werden.
- Geben Sie dazu `bool(element)` ein.
- Finden Sie einen String, eine Liste und einen Integer, die den Wert `False` ergeben?

Hausaufgabe: a- und o-Deklination im Lateinischen

Ziel ist die Entwicklung eines Wortformengenerators für das Lateinische (Aufgabe 3). Die ersten beiden Aufgaben sollen Ihnen bei der Programmierung helfen.

1. Definieren Sie eine Funktion, die zwei Argumente nimmt: 1. den Wortstamm eines lateinischen Nomens, 2. dessen Deklinationsklasse. Die Funktion soll die deklinierten Wortformen auf dem Bildschirm ausgeben.
2. Passen Sie Ihre Funktionsdefinition so an, dass die Stelligkeit der Funktion erhalten bleibt und die Funktion als 1. Argument die Wortform im Nominativ Singular nimmt. Das 2. Argument bleibt unverändert.
3. Passen Sie Ihre Funktionsdefinition so an, dass die Funktion lediglich ein Argument hat, nämlich die Wortform im Nominativ Singular, und die Deklinationsklasse selbst ermittelt.

Können Sie Ihre Definition so erweitern, dass bei falschem Funktionsaufruf eine entsprechende Fehlermeldung ausgegeben wird?

Hausaufgabe: a- und o-Deklination im Lateinischen, Beispiel In- und Output

1. `print_wordforms("domin", "a")`
2. `print_wordforms("domina", "a")`
3. `print_wordforms("domina")`

```
>>> print_wordforms("domina")
```

Singular:

Nom	domina
Gen	dominae
Dat	dominae
Acc	dominam
Abl	domina

Plural:

Nom	dominae
Gen	dominarum
Dat	dominis
Acc	dominas
Abl	dominis

```
>>> print_wordforms("dominus")
```

Singular:

Nom	dominus
Gen	domini
Dat	domino
Acc	dominum
Abl	domino

Plural:

Nom	domini
Gen	dominorum
Dat	dominis
Acc	dominos
Abl	dominis

Erweiterung der Hausaufgabe

- Wenn Sie Teil 3 der Aufgabe geschafft haben, versuchen Sie, die Ausgabe in eine Datei umzuleiten!

for-Schleifen

- Syntax:

```
for element in iterable:
    print(element)
```

- Mit dem Keyword `for ... in ...` wird der Schleifenkörper für alle Elemente in `iterable` ausgeführt.
- Das jeweilige Element ist als Variable `element` im Schleifenkörper verfügbar.
- Veränderungen (z.B. `element = element + 1`) betreffen aber nur das "lokale" Element, das Element in `iterable` bleibt unverändert!
- Mögliche Datentypen für `iterable` sind Listen, Strings und andere Datentypen, die wir noch behandeln werden.
- Durch die Zeilen einer zum Lesen geöffneten Datei kann man mit `for line in infile: ...` iterieren.
- Um durch Zahlen in bestimmten Bereichen zu iterieren, verwenden wir die Funktion `range`.

range

- Syntax:

```
range([von], bis, [schrittweite])
```

- Die Parameter in eckigen Klammern sind optional.
- Testen Sie die Funktion `range`. Was sind die Argumente, was das Ergebnis?
- `range` wird insbesondere in `for`-Schleifen eingesetzt:

```
print("Die 13er-Reihe ist:")
for number in range(0,131,13):
    print(number)
```

- Schreiben Sie eine Funktion, die für beliebige Integers die entsprechende Multiplikationsreihe ausgibt. Beispiel:

```
>>> reihe(4)
0  mal 4 ist 0
1  mal 4 ist 4
2  mal 4 ist 8
...
10 mal 4 ist 40
```

for-Schleifen: break und continue

- Syntax:

```
iterable = ['kiwi', 'apple', 'salad', 'cherry', 'pineapple']
for element in iterable:
    if element == 'salad':
        continue
    print(element)
```

- `continue` wird benutzt, um die aktuelle Iteration zu beenden und direkt in die nächste überzugehen.
- `break` wird benutzt, um die Iteration und die ganze Schleife zu beenden.

```
for element in iterable:
    if element == 'apple':
        continue
    elif element == 'salad':
        break
    print(element)
```

- Häufig brauchen Sie diese Befehle nicht, wenn sie die Schleifenart und die Bedingung sinnvoll wählen.

Aufgabe: Diskutieren

- Welche Art von Schleife ist in den folgenden Situationen sinnvoll, `for` oder `while`? Und mit welcher Bedingung?
  - Sie wollen eine Funktion auf jedes Element einer bereits bekannten Liste der Länge 100 anwenden.
  - Sie wollen den Benutzer 10 Strings eingeben lassen.
  - Der Benutzer soll einen String eingeben, der aber abgelehnt wird, solange man etwas anderes als "ENDE" eingibt.

Aufgabe: Programmieren (einfacher)

- Laden Sie die Datei punkte.txt von der Kurswebseite herunter.
- Jede Zeile der Datei hat das Format `TN-Name : Punktzahl`.
- Schreiben Sie ein Programm, das die Datei liest und die durchschnittliche Punktzahl aller Teilnehmer/innen errechnet und ausgibt.

Aufgabe: Programmieren (schwieriger)

- Bigramme sind Zeichenketten der Länge 2. Jede Zeichenkette, die aus 2 oder mehr Zeichen besteht, lässt sich in Bigramme zerlegen.
- Erzeugen Sie aus dem String "Python fuer Linguisten" eine Liste von Bigrammen!

```
>>> print(make_bigrams("Python fuer Linguisten"))
['Py', 'yt', 'th', 'ho', 'on', 'n ', ' f', 'fu', 'ue', 'er',
'r ', ' L', 'Li', 'in', 'ng', 'gu', 'ui', 'is', 'st', 'te', 'en']
```

- Tipp: Welche Beziehung herrscht zwischen der Stringlänge und der Anzahl der enthaltenen Bigramme?
- Tipp: Erstellen sie zunächst eine leere Liste und fügen Sie die Bigramme schrittweise ein. Pro neues Element können Sie mit dem Befehl `bigrams.append(element)` das Element ans Ende der Liste `bigrams` anhängen.
- Tipp: Ein Bigramm können Sie durch Slicen des Strings erhalten (siehe Foliensatz 2, Seiten 7 und 8).