

# Informatik I: Einführung in die Programmierung

## 5. Bedingungen, bedingte Ausführung

Albert-Ludwigs-Universität Freiburg



Prof. Dr. Peter Thiemann

28. October 2025

# 1 Bedingungen und der Typ bool



- Typ bool
- Vergleichsoperationen
- Logische Operatoren

## Bedingungen

Typ bool  
Vergleichsoperationen  
Logische Operatoren

## Bedingte Anweisungen

## Anwendung

## Zusammenfassung

- Neben arithmetischen Ausdrücken gibt es auch **Boolesche Ausdrücke**.
- Sie haben den Typ **bool**, dessen Werte entweder **True** oder **False** sind.
- Boolesche Ausdrücke: die Literale **True** und **False**, Vergleiche  $a == b$  usw.
- Arithmetische Operationen konvertieren Boolesche Werte nach **int**:  
 $\text{False} \mapsto 0, \text{True} \mapsto 1$

bool\_ex.py

```
assert 42 == 42
assert not ('egg' == 'spam')
assert type('egg' == 'spam') == bool
```

Bedingungen

Typ bool

Vergleichsoperationen

Logische Operatoren

Bedingte Anweisungen

Anwendung

Zusammenfassung

Es gibt die folgenden Vergleichsoperatoren:

Syntax	Bedeutung
<code>x == y</code>	Ist x gleich y?
<code>x != y</code>	Ist x ungleich y?
<code>x &gt; y</code>	Ist x echt größer als y?
<code>x &lt; y</code>	Ist x echt kleiner als y?
<code>x &gt;= y</code>	Ist x größer oder gleich y?
<code>x &lt;= y</code>	Ist x kleiner oder gleich y?

`bool_ex.py`

```
assert 2.1 - 2.0 > 0.1
assert not(2 - 1 < 1)
assert False < True
```

Bedingungen

Typ `bool`

Vergleichsoperatio-  
nen

Logische  
Operatoren

Bedingte An-  
weisungen

Anwendung

Zusammen-  
fassung

Strings werden anhand der **lexikographischen Ordnung** verglichen. Für Einzelzeichen wird das Ergebnis der `ord`-Funktion benutzt.

## Python-Interpreter

```
>>> 'anton' < 'antonia'
True
>>> 'anton' < 'berta'
True
>>> 'anton' < 'ulf'
True
>>> 'antonia' < 'antonella'
False
```

Bedingungen

Typ `bool`

Vergleichsoperationen

Logische  
Operatoren

Bedingte Anweisungen

Anwendung

Zusammenfassung

- Werte unvergleichbarer Typen sind ungleich: *ifb||s*.
- Bei den Anordnungsrelationen gibt es einen Fehler, wenn die Typen nicht zusammenpassen! *ifb||s*

## Python-Interpreter

```
>>> 42 == 'zweiundvierzig'
```

```
False
```

```
>>> 41 < '42'
```

```
Traceback (most recent call last): ...
```

```
TypeError: unorderable types: int() < str()
```

Bedingungen

Typ bool

Vergleichsoperationen

Logische Operatoren

Bedingte Anweisungen

Anwendung

Zusammenfassung

- **Logische Operatoren** auf `bool`
- `or`, `and`, `not` – in aufsteigender Operatorpräzedenz.
- Wie die Bitoperationen mit (`False`  $\leftrightarrow$  0, `True`  $\leftrightarrow$  1), d.h.
  - `x < 10 or y > 100`  
hat den Wert `True`, wenn `x` kleiner als 10 ist oder wenn `y` größer als 100 ist.
  - `1 <= x and x <= 10`  
hat den Wert `True`, wenn `x` zwischen 1 und 10 (inklusive) liegt.
  - Alternative Schreibweise dafür: `1 <= x <= 10`.
  - `not(x < y)` ist `True` wenn `x` nicht kleiner als `y` ist.
- **Nullwerte**: `None`, `0`, `0.0` und `' '` werden wie `False` behandelt, alle anderen Werte wie `True`!
- **Kurzschlussauswertung**: Die Auswertung der logischen Operatoren wird beendet, wenn das Ergebnis klar ist. (short-cut evaluation)

Bedingungen

Typ `bool`

Vergleichsoperationen

Logische Operatoren

Bedingte Anweisungen

Anwendung

Zusammenfassung

## Python-Interpreter

```
>>> 1 < 5 < 10
True
>>> 5 < 1 or 'spam' < 'egg'
False
>>> 'spam' or True
'spam'
>>> '' or 'default'
'default'
>>> 'good night' and 'ding ding ding'
'ding ding ding'
>>> 0 and 10 < 100
0
>>> not 'spam' and (None or 0.0 or 10 < 100)
False
```

Bedingungen

Typ bool

Vergleichsoperationen

Logische  
Operatoren

Bedingte Anweisungen

Anwendung

Zusammenfassung



## 2 Bedingte Anweisungen



- if-Anweisung
- if-else-Anweisung
- elif-Anweisung

Bedingungen

Bedingte Anweisungen

if-Anweisung  
if-else-Anweisung  
elif-Anweisung

Anwendung

Zusammenfassung

- Die **bedingte Anweisung** (Konditional, if-Anweisung) ermöglicht es, Anweisungen nur unter bestimmten Bedingungen auszuführen.

strictly\_positive.py

```
def strictly_positive(x):  
    if x > 0:  
        print(x, 'ist strikt positiv!')
```

Bedingungen

Bedingte Anweisungen

if-Anweisung

if-else-Anweisung

elif-Anweisung

Anwendung

Zusammenfassung

## Python-Interpreter

```
>>> strictly_positive(3)  
3 ist strikt positiv  
>>> strictly_positive(0)  
>>>
```

- Die **if-else-Anweisung** ermöglicht es, durch eine Bedingung zwischen zwei Blöcken von Anweisungen auszuwählen.
- Der **if-Block** wird ausgeführt, wenn die Bedingung erfüllt ist.
- Der **else-Block** wird ausgeführt, wenn die Bedingung **nicht** erfüllt ist.

```
def evenodd(x):  
    if x % 2 == 0:  
        print(x, 'ist gerade')  
    else:  
        print(x, 'ist ungerade')  
  
def evenodd2(x):  
    if x % 2:  
        print(x, 'ist ungerade')  
    else:  
        print(x, 'ist gerade')
```

Bedingungen

Bedingte Anweisungen

if-Anweisung  
if-else-Anweisung  
elif-Anweisung

Anwendung

Zusammenfassung

# Verkettete bedingte Anweisungen



```
def compare(x, y):  
    if x < y:  
        print(x, 'ist_kleiner_als', y)  
    elif x > y:  
        print(x, 'ist_größer_als', y)  
    else:  
        print(x, 'und', y, 'sind_gleich')
```

Bedingungen

Bedingte Anweisungen

if-Anweisung

if-else-Anweisung

elif-Anweisung

Anwendung

Zusammenfassung

- Eine **verkettete bedingte Anweisung** kann mehr als zwei Fälle behandeln.
- Die Bedingungen werden der Reihe nach ausgewertet. Der erste Block, dessen Bedingung erfüllt ist, wird ausgeführt.

## Python-Interpreter

```
>>> compare(3, 0)  
3 ist größer als 0
```

successful.py

```
def salesman(x):  
    if x > 0:  
        if x > 10:  
            print('successful_encyclopedia_salesman')  
            ####  
        else:  
            print('unsuccessful_encyclopedia_salesman')
```

Bedingungen

Bedingte Anweisungen

if-Anweisung

if-else-Anweisung

elif-Anweisung

Anwendung

Zusammenfassung

- Bedingte Anweisungen können geschachtelt werden.
- Durch die Einrückung ist immer klar, welcher Block durch welche Bedingung gesteuert wird!

# 3 Anwendung



- Auswerten eines Tests
- Nachspiel: Verträge
- Freizeitpark

Bedingungen

Bedingte An-  
weisungen

**Anwendung**

Auswerten eines  
Tests

Nachspiel:  
Verträge

Freizeitpark

Zusammen-  
fassung

# Anwendung — Auswerten eines Tests

Bedingungen

Bedingte An-  
weisungen

Anwendung

Auswerten eines  
Tests

Nachspiel:  
Verträge

Freizeitpark

Zusammen-  
fassung

## Bestanden oder nicht?

In einem Test kann eine maximale Punktzahl erreicht werden. Ein gewisser Prozentsatz an Punkten ist notwendig um den Test zu bestehen.

## Aufgabe

Entwickle eine Funktion, die die Eingaben

- maximale Punktzahl,
- Prozentsatz zum Bestehen und
- tatsächlich erreichte Punktzahl

nimmt und als Ergebnis entweder 'pass' oder 'fail' liefert.

Bedingungen

Bedingte An-  
weisungen

Anwendung

Auswerten eines  
Tests

Nachspiel:  
Verträge

Freizeitpark

Zusammen-  
fassung



## Aufgabe

Entwickle eine Funktion `test_result`, die die Eingaben

- `max_points: int` maximale Punktzahl,
- `percentage: int` Prozentsatz zum Bestehen und
- `points: int` tatsächlich erreichte Punktzahl

nimmt und als Ergebnis entweder `'pass'` oder `'fail'` (vom Typ `str`) liefert.

- Bezeichner für Funktion und Parameter festlegen
- Typen der Parameter angeben
- Typ des Rückgabewertes angeben

Bedingungen

Bedingte Anweisungen

Anwendung

Auswerten eines Tests

Nachspiel:  
Verträge

Freizeitpark

Zusammenfassung

## Schritt 2: Funktionsgerüst



```
def test_result(  
    max_points: int,  
    percentage: int,  
    points: int) -> str:  
    # fill in  
    return
```

Bedingungen

Bedingte An-  
weisungen

Anwendung

Auswerten eines  
Tests

Nachspiel:  
Verträge

Freizeitpark

Zusammen-  
fassung

- Funktionsgerüst aufschreiben.
- Wenn klar ist, dass eine Zeile fortgesetzt werden muss (hier: innerhalb einer Parameterliste), wird das durch zusätzliche Einrückung gekennzeichnet.
- Typen werden durch **Typannotationen** “: int” für Parameter bzw. “-> str” für das Ergebnis angegeben (ab Python 3.6).

## Schritt 3: Beispiele



```
assert(test_result(100, 50, 50) == 'pass')  
assert(test_result(100, 50, 30) == 'fail')  
assert(test_result(100, 50, 70) == 'pass')
```

- Sinnvolle Beispiele erarbeiten
  - Eingaben so wählen, dass alle mögliche Ergebnisse erreicht werden.
  - Randfälle bedenken (z.B. `points == max_points`, `points == 0`, `percentage == 0`, `percentage == 100`, ...)
- Ergebnisse der Beispiele von Hand ausrechnen!
- Die Funktion `assert` prüft, ob das Ergebnis wahr ist und gibt sonst eine Fehlermeldung. Kann unter der Definition im Programm bleiben.
- Die Beispiele dienen später als **Tests**, die belegen, dass der Code zumindest für die Beispiele funktioniert.

Bedingungen

Bedingte Anweisungen

Anwendung

Auswerten eines Tests

Nachspiel:  
Verträge

Freizeitpark

Zusammenfassung

## Schritt 4: Funktionsrumpf ausfüllen

```
def test_result(  
    max_points: int,  
    percentage: int,  
    points: int) -> str:  
    passed = (points >= max_points * percentage / 100)  
    if passed:  
        return 'pass'  
    else:  
        return 'fail'
```

Bedingungen

Bedingte An-  
weisungen

Anwendung

Auswerten eines  
Tests

Nachspiel:  
Verträge

Freizeitpark

Zusammen-  
fassung

- Fertig?
- Was ist, wenn
  - `max_points < 0?`
  - `percentage < 0?`
  - `percentage > 100?`
  - `points < 0?`
  - `points > max_points?`
- Wollen wir diese Fälle zulassen?

Bedingungen

Bedingte Anweisungen

Anwendung

Auswerten eines Tests

Nachspiel:  
Verträge

Freizeitpark

Zusammenfassung

## Defensives Programmieren

Fange alle unerwünschten Fälle im Code ab und erzeuge eine Fehlermeldung.

## Design by Contract

- **Spezifiziere** die Funktion durch einen **Vertrag** und programmiere unter der Annahme, dass nur die zulässigen Fälle auftreten (wie im Codebeispiel).
- `max_points >= 0`
- `0 <= percentage <= 100`
- `0 <= points <= max_points`

Bedingungen

Bedingte An-  
weisungen

Anwendung

Auswerten eines  
Tests

Nachspiel:  
Verträge

Freizeitpark

Zusammen-  
fassung

# Anwendung — Freizeitpark

Bedingungen

Bedingte An-  
weisungen

Anwendung

Auswerten eines  
Tests

Nachspiel:  
Verträge

**Freizeitpark**

Zusammen-  
fassung

## Mitfahren oder nicht?

In einem Freizeitpark gibt es verschiedene Attraktionen, die mit Alters- und Größenbeschränkungen belegt sind.

## Beispiel

Attraktion	Beschränkung	Begleitung
Silver-Star	11 Jahre und 1,40m	—
Euro-Mir	8 Jahre und 1,30m	unter 10 Jahre
blue fire	7 Jahre und 1,30m	—
Eurosat	6 Jahre und 1,20m	unter 10 Jahre
Matterhorn-Blitz	6 Jahre und 1,20m	unter 8 Jahre
Tiroler Wildwasserbahn	4 Jahre und 1,00m	unter 9 Jahre

Bedingungen

Bedingte Anweisungen

Anwendung

Auswerten eines Tests

Nachspiel:  
Verträge

Freizeitpark

Zusammenfassung



## Aufgabe

Entwickle eine Funktion zur Einlasskontrolle bei Euro-Mir, die als Eingaben

- das Alter,
- die Größe und
- ob ein erwachsener Begleiter dabei ist

nimmt und als Ergebnis entweder `True` oder `False` liefert.

Bedingungen

Bedingte An-  
weisungen

Anwendung

Auswerten eines  
Tests

Nachspiel:  
Verträge

Freizeitpark

Zusammen-  
fassung

## Aufgabe

Entwickle eine Funktion `enter_euro_mir` zur Einlasskontrolle bei Euro-Mir, die als Eingaben

- `age: int` das Alter (in Jahren),
  - `height: int` die Größe (in cm) und
  - `accompanied: bool` ob ein erwachsener Begleiter dabei ist
- nimmt und als Ergebnis entweder `True` oder `False` -> `bool` liefert.
- Festlegen von **Einheiten** für die Eingaben!

Bedingungen

Bedingte Anweisungen

Anwendung

Auswerten eines Tests

Nachspiel:  
Verträge

Freizeitpark

Zusammenfassung

## Schritt 2: Funktionsgerüst

```
def enter_euro_mir(  
    age: int,  
    height: int,  
    accompanied: bool  
    ) -> bool:  
    # fill in  
    return
```

Bedingungen

Bedingte An-  
weisungen

Anwendung

Auswerten eines  
Tests

Nachspiel:  
Verträge

Freizeitpark

Zusammen-  
fassung

## Schritt 3: Beispiele

```
assert(enter_euro_mir(4, 101, 'Mama') == False)
assert(enter_euro_mir(8, 125, 'Papa') == False)
assert(enter_euro_mir(7, 130, 'Oma') == False)
assert(enter_euro_mir(9, 135, 'Opa'))
assert(enter_euro_mir(10, 135, ''))
```

Bedingungen

Bedingte Anweisungen

Anwendung

Auswerten eines Tests

Nachspiel:  
Verträge

Freizeitpark

Zusammenfassung

## Schritt 4: Funktionsrumpf ausfüllen



```
def enter_euro_mir(  
    age: int,  
    height: int,  
    accompanied: bool  
    ) -> bool:  
    age_ok = age >= 8  
    height_ok = height >= 130  
    admitted = (age_ok  
        and height_ok  
        and (age >= 10 or accompanied))  
    return admitted
```

Bedingungen

Bedingte An-  
weisungen

Anwendung

Auswerten eines  
Tests

Nachspiel:  
Verträge

Freizeitpark

Zusammen-  
fassung

- Entwickle eine `enter` Funktion, die die Bedingungen aus den globalen Variablen `min_age`, `min_height` und `min_age_alone` berechnet.
- Ändere die Funktion, so dass sie bei einer Zurückweisung den Grund angibt. Zum Beispiel `'Du bist zu klein.'`, `'Du bist zu jung.'` usw.

# 4 Zusammenfassung



Bedingungen

Bedingte An-  
weisungen

Anwendung

Zusammen-  
fassung

- `bool` ist ein Typ, dessen einzige Werte `True` und `False` sind.
- Vergleichsoperationen, wie `==` oder `<`, liefern **Boolesche Werte**.
- Boolesche Werte werden bei Bedarf nach `int` konvertiert, wobei `True`  $\mapsto$  1 und `False`  $\mapsto$  0 gilt.
- Logische Operationen interpretieren **Nullwerte** als `False`, alle anderen Werte als `True`.
- **Bedingte Anweisungen** (`if-(elif)-else`) erlauben die Auswahl zwischen alternativen Blöcken von Anweisungen.
- **Defensives Programmieren** vs **Programmierung mit Verträgen**
- **Checkliste** zum Entwurf von Funktionen: Bezeichner und Datentypen, Funktionsgerüst, Beispiele, Funktionsrumpf

Bedingungen

Bedingte Anweisungen

Anwendung

Zusammenfassung