# Informatik I: Einführung in die Programmierung

## 5. Bedingungen, bedingte Ausführung

Albert-Ludwigs-Universität Freiburg

UNI FREIBURG

Peter Thiemann

6. November 20189



# Bedingungen und der Typ `bool`

# Der Typ `bool`

UNI FREIBURG

- Neben *arithmetischen Ausdrücken* gibt es noch Boolesche Ausdrücke mit `True` oder `False` als Werten.
- Die einfachsten Booleschen Ausdrücke sind Vergleiche mit dem Gleichheitsoperator ==.

## Python-Interpreter

```
>>> 42 == 42
```

# Der Typ `bool`

UNI FREIBURG

- Neben *arithmetischen Ausdrücken* gibt es noch Boolesche Ausdrücke mit `True` oder `False` als Werten.
- Die einfachsten Booleschen Ausdrücke sind Vergleiche mit dem Gleichheitsoperator ==.

## Python-Interpreter

```
>>> 42 == 42
True
>>>
```

# Der Typ `bool`

UNI FREIBURG

- Neben *arithmetischen Ausdrücken* gibt es noch Boolesche Ausdrücke mit `True` oder `False` als Werten.
- Die einfachsten Booleschen Ausdrücke sind Vergleiche mit dem Gleichheitsoperator ==.

## Python-Interpreter

```
>>> 42 == 42
True
>>> 'egg' == 'spam'
```

# Der Typ `bool`

UNI FREIBURG

- Neben *arithmetischen Ausdrücken* gibt es noch Boolesche Ausdrücke mit `True` oder `False` als Werten.
- Die einfachsten Booleschen Ausdrücke sind Vergleiche mit dem Gleichheitsoperator ==.

## Python-Interpreter

```
>>> 42 == 42
True
>>> 'egg' == 'spam'
False
>>>
```

# Der Typ `bool`

UNI FREIBURG

- Neben *arithmetischen Ausdrücken* gibt es noch Boolesche Ausdrücke mit `True` oder `False` als Werten.
- Die einfachsten Booleschen Ausdrücke sind Vergleiche mit dem Gleichheitsoperator `==`.
- Die Werte `True` und `False` gehören zum Typ `bool`.

## Python-Interpreter

```
>>> 42 == 42
True
>>> 'egg' == 'spam'
False
>>> type('egg' == 'spam')
```

# Der Typ `bool`

UNI FREIBURG

- Neben *arithmetischen Ausdrücken* gibt es noch Boolesche Ausdrücke mit `True` oder `False` als Werten.
- Die einfachsten Booleschen Ausdrücke sind Vergleiche mit dem Gleichheitsoperator `==`.
- Die Werte `True` und `False` gehören zum Typ `bool`.

## Python-Interpreter

```
>>> 42 == 42
True
>>> 'egg' == 'spam'
False
>>> type('egg' == 'spam')
<class 'bool'>
>>>
```

# Der Typ `bool`

UNI FREIBURG



- Neben *arithmetischen Ausdrücken* gibt es noch Boolesche Ausdrücke mit `True` oder `False` als Werten.
- Die einfachsten Booleschen Ausdrücke sind Vergleiche mit dem Gleichheitsoperator ==.
- Die Werte `True` und `False` gehören zum Typ `bool`.
- Arithmetische Operationen konvertieren Boolesche Werte automatisch nach `int` (`False` $\mapsto$ 0, `True` $\mapsto$ 1):

## Python-Interpreter

```
>>> 42 == 42
True
>>> 'egg' == 'spam'
False
>>> type('egg' == 'spam')
<class 'bool'>
>>> True + True
2
```

# Vergleichsoperatoren

UNI FREIBURG

Es gibt die folgenden Vergleichsoperatoren:

| Syntax | Bedeutung |
|---|---|
| `x == y` | Ist *x* gleich *y*? |
| `x != y` | Ist *x* ungleich *y*? |
| `x > y` | Ist *x* echt größer als *y*? |
| `x < y` | Ist *x* echt kleiner als *y*? |
| `x >= y` | Ist *x* größer oder gleich *y*? |
| `x <= y` | Ist *x* kleiner oder gleich *y*? |

# Vergleichsoperatoren

Es gibt die folgenden Vergleichsoperatoren:

| Syntax | Bedeutung |
|---|---|
| `x == y` | Ist *x* gleich *y*? |
| `x != y` | Ist *x* ungleich *y*? |
| `x > y` | Ist *x* echt größer als *y*? |
| `x < y` | Ist *x* echt kleiner als *y*? |
| `x >= y` | Ist *x* größer oder gleich *y*? |
| `x <= y` | Ist *x* kleiner oder gleich *y*? |

## Python-Interpreter

```
>>>
```

# Vergleichsoperatoren

UNI FREIBURG

Es gibt die folgenden Vergleichsoperatoren:

| Syntax | Bedeutung |
|---|---|
| `x == y` | Ist *x* gleich *y*? |
| `x != y` | Ist *x* ungleich *y*? |
| `x > y` | Ist *x* echt größer als *y*? |
| `x < y` | Ist *x* echt kleiner als *y*? |
| `x >= y` | Ist *x* größer oder gleich *y*? |
| `x <= y` | Ist *x* kleiner oder gleich *y*? |

## Python-Interpreter

```
>>> 2.1 - 2.0 > 0.1
```

# Vergleichsoperatoren

Es gibt die folgenden Vergleichsoperatoren:

| Syntax | Bedeutung |
|---|---|
| `x == y` | Ist *x* gleich *y*? |
| `x != y` | Ist *x* ungleich *y*? |
| `x > y` | Ist *x* echt größer als *y*? |
| `x < y` | Ist *x* echt kleiner als *y*? |
| `x >= y` | Ist *x* größer oder gleich *y*? |
| `x <= y` | Ist *x* kleiner oder gleich *y*? |

## Python-Interpreter

```
>>> 2.1 - 2.0 > 0.1
True
>>>
```

# Vergleichsoperatoren

Es gibt die folgenden Vergleichsoperatoren:

| Syntax | Bedeutung |
|---|---|
| `x == y` | Ist *x* gleich *y*? |
| `x != y` | Ist *x* ungleich *y*? |
| `x > y` | Ist *x* echt größer als *y*? |
| `x < y` | Ist *x* echt kleiner als *y*? |
| `x >= y` | Ist *x* größer oder gleich *y*? |
| `x <= y` | Ist *x* kleiner oder gleich *y*? |

## Python-Interpreter

```
>>> 2.1 - 2.0 > 0.1
True
>>> 2 - 1 < 1
```

# Vergleichsoperatoren

UNI FREIBURG

Es gibt die folgenden Vergleichsoperatoren:

| Syntax | Bedeutung |
|---|---|
| `x == y` | Ist *x* gleich *y*? |
| `x != y` | Ist *x* ungleich *y*? |
| `x > y` | Ist *x* echt größer als *y*? |
| `x < y` | Ist *x* echt kleiner als *y*? |
| `x >= y` | Ist *x* größer oder gleich *y*? |
| `x <= y` | Ist *x* kleiner oder gleich *y*? |

## Python-Interpreter

```
>>> 2.1 - 2.0 > 0.1
True
>>> 2 - 1 < 1
False
>>>
```

# Vergleichsoperatoren

UNI FREIBURG

Es gibt die folgenden Vergleichsoperatoren:

| Syntax | Bedeutung |
|---|---|
| `x == y` | Ist *x* gleich *y*? |
| `x != y` | Ist *x* ungleich *y*? |
| `x > y` | Ist *x* echt größer als *y*? |
| `x < y` | Ist *x* echt kleiner als *y*? |
| `x >= y` | Ist *x* größer oder gleich *y*? |
| `x <= y` | Ist *x* kleiner oder gleich *y*? |

## Python-Interpreter

```
>>> 2.1 - 2.0 > 0.1
True
>>> 2 - 1 < 1
False
>>> False < True
```

# Vergleichsoperatoren

UNI FREIBURG

Es gibt die folgenden Vergleichsoperatoren:

| Syntax | Bedeutung |
|---|---|
| `x == y` | Ist *x* gleich *y*? |
| `x != y` | Ist *x* ungleich *y*? |
| `x > y` | Ist *x* echt größer als *y*? |
| `x < y` | Ist *x* echt kleiner als *y*? |
| `x >= y` | Ist *x* größer oder gleich *y*? |
| `x <= y` | Ist *x* kleiner oder gleich *y*? |

## Python-Interpreter

```
>>> 2.1 - 2.0 > 0.1
True
>>> 2 - 1 < 1
False
>>> False < True
True
```

# Stringvergleich

UNI FREIBURG

Strings werden anhand der lexikographischen Ordnung verglichen, wobei für Einzelzeichen der Unicode-Wert (Ergebnis der `ord`-Funktion) benutzt wird.

# Stringvergleich

UNI FREIBURG

Strings werden anhand der lexikographischen Ordnung verglichen, wobei für Einzelzeichen der Unicode-Wert (Ergebnis der `ord`-Funktion) benutzt wird.

## Python-Interpreter

```
>>> 'anton' < 'antonia'
```

# Stringvergleich

UNI FREIBURG



Strings werden anhand der lexikographischen Ordnung verglichen, wobei für Einzelzeichen der Unicode-Wert (Ergebnis der `ord`-Funktion) benutzt wird.

## Python-Interpreter

```
>>> 'anton' < 'antonia'
True
>>>
```

# Stringvergleich

UNI FREIBURG



Strings werden anhand der lexikographischen Ordnung verglichen, wobei für Einzelzeichen der Unicode-Wert (Ergebnis der `ord`-Funktion) benutzt wird.

## Python-Interpreter

```
>>> 'anton' < 'antonia'
True
>>> 'anton' < 'berta'
```

# Stringvergleich

UNI FREIBURG

Strings werden anhand der lexikographischen Ordnung verglichen, wobei für Einzelzeichen der Unicode-Wert (Ergebnis der `ord`-Funktion) benutzt wird.

## Python-Interpreter

```
>>> 'anton' < 'antonia'
True
>>> 'anton' < 'berta'
True
>>>
```

# Stringvergleich

UNI FREIBURG



Strings werden anhand der lexikographischen Ordnung verglichen, wobei für Einzelzeichen der Unicode-Wert (Ergebnis der `ord`-Funktion) benutzt wird.

## Python-Interpreter

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

# Stringvergleich

UNI FREIBURG



Strings werden anhand der lexikographischen Ordnung verglichen, wobei für Einzelzeichen der Unicode-Wert (Ergebnis der `ord`-Funktion) benutzt wird.

## Python-Interpreter

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

# Stringvergleich

UNI FREIBURG

Strings werden anhand der lexikographischen Ordnung verglichen, wobei für Einzelzeichen der Unicode-Wert (Ergebnis der `ord`-Funktion) benutzt wird.

## Python-Interpreter

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

# Stringvergleich

UNI FREIBURG

Strings werden anhand der lexikographischen Ordnung verglichen, wobei für Einzelzeichen der Unicode-Wert (Ergebnis der `ord`-Funktion) benutzt wird.

## Python-Interpreter

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



## Algorithmus: Lexikogaphische Ordnung (grob)

- Eingabe: zwei Strings $a$ und $b$
- Ausgabe: ist $a \leq b$ in der lexikographischen Ordnung?

1. Ist $a$ ein Präfix von $b$?
   Kann $b = a + b'$ geschrieben werden, wobei $b'$ ein beliebiger String ist?
2. Falls ja, ist das Ergebnis `True`.
3. Andernfalls suche das längste gemeinsame Präfix $c$ von $a$ und $b$.
   Für den String $c$ gilt, dass $a = c + a'$ und $b = c + b'$ und $a'$ und $b'$ fangen mit unterschiedlichen Zeichen $a_{k+1}$ und $b_{k+1}$ an.
4. Falls $a_{k+1} < b_{k+1}$, so ist das Ergebnis `True` sonst `False`.

# Stringvergleich

UNI FREIBURG

## Algorithmus: Lexikogaphische Ordnung (fein)

- Eingabe: zwei Strings
  $\vec{a}$ = "$a_1 a_2 \ldots a_m$" und $\vec{b}$ = "$b_1 b_2 \ldots b_n$" der Längen $m, n \geq 0$.
- Ausgabe: ist $a \leq b$ in der lexikographischen Ordnung?

Suche das längste gemeinsame Präfix

1. Setze $k = 0$
2. Falls $k \geq m$, ist das Ergebnis `True` ($\vec{a}$ ist Präfix von $\vec{b}$)
3. Falls $k \geq n$, ist das Ergebnis `False` ($\vec{b}$ ist Präfix von $\vec{a}$)
4. Falls $a_{k+1} == b_{k+1}$, setze $k = k + 1$ und weiter bei 2

   Längstes gemeinsames Präfix ist "$a_1 a_2 \ldots a_k$"
5. Falls $a_{k+1} < b_{k+1}$, ist das Ergebnis `True` sonst `False`

# MathematischeDefinition: Lexikographische Ordnung

UNI FREIBURG



## Gegeben

Zwei Strings der Längen $m, n \geq 0$:

$$\vec{a} = "a_1 a_2 \ldots a_m" \qquad \vec{b} = "b_1 b_2 \ldots b_n"$$

# MathematischeDefinition: Lexikographische Ordnung

UNI FREIBURG



## Gegeben

Zwei Strings der Längen $m, n \geq 0$:

$$\vec{a} = "a_1 a_2 \ldots a_m" \qquad \vec{b} = "b_1 b_2 \ldots b_n"$$

## $\vec{a} \leq \vec{b}$ in der lexikographischen Ordnung, falls

Es gibt $0 \leq k \leq \min(m, n)$, so dass

- $a_1 = b_1, \ldots, a_k = b_k$ und

  $$\vec{a} = "a_1 a_2 \ldots a_k a_{k+1} \ldots a_m" \qquad \vec{b} = "a_1 a_2 \ldots a_k b_{k+1} \ldots b_n"$$

  - entweder $k = m$

    $$\vec{a} = "a_1 a_2 \ldots a_m" \qquad \vec{b} = "a_1 a_2 \ldots a_m b_{m+1} \ldots b_n"$$

  - oder $k < m$ und $a_{k+1} < b_{k+1}$.

# Vergleich unterschiedlicher Typen

UNI FREIBURG



- Werte unvergleichbarer Typen sind ungleich.

## Python-Interpreter

```
>>>
```

# Vergleich unterschiedlicher Typen

UNI FREIBURG



- Werte unvergleichbarer Typen sind ungleich.

## Python-Interpreter

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

# Vergleich unterschiedlicher Typen

UNI FREIBURG



- Werte unvergleichbarer Typen sind ungleich.

## Python-Interpreter

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

# Vergleich unterschiedlicher Typen

UNI FREIBURG



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

## Python-Interpreter

```
>>> 42 == 'zweiundvierzig'
False
>>> 41 < '42'
```

# Vergleich unterschiedlicher Typen

UNI FREIBURG



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

## Python-Interpreter

```
>>> 42 == 'zweiundvierzig'
False
>>> 41 < '42'
Traceback (most recent call last): ...
TypeError: unorderable types: int() < str()
```

# Logische Operatoren

- Es gibt die folgenden logischen Operatoren: `or`, `and`, `not` – mit aufsteigender Operatorpräzedenz.

# Logische Operatoren

- Es gibt die folgenden logischen Operatoren: `or`, `and`, `not` – mit aufsteigender Operatorpräzedenz.
- Wie die Bitoperationen mit (`False` $\mapsto$ 0, `True` $\mapsto$ 1), d.h.

# Logische Operatoren

UNI FREIBURG

- Es gibt die folgenden logischen Operatoren: `or`, `and`, `not` – mit aufsteigender Operatorpräzedenz.
- Wie die Bitoperationen mit (`False` $\mapsto$ 0, `True` $\mapsto$ 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.

# Logische Operatoren

UNI FREIBURG

- Es gibt die folgenden logischen Operatoren: `or`, `and`, `not` – mit aufsteigender Operatorpräzedenz.
- Wie die Bitoperationen mit (`False` $\mapsto$ 0, `True` $\mapsto$ 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 $1 \leq x$ und $x \leq 10$, d.h. wenn $x$ zwischen 1 und 10 (inklusive) liegt.

# Logische Operatoren

UNI FREIBURG

- Es gibt die folgenden logischen Operatoren: `or`, `and`, `not` – mit aufsteigender Operatorpräzedenz.
- Wie die Bitoperationen mit (`False` $\mapsto$ 0, `True` $\mapsto$ 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 $1 \leq x$ und $x \leq 10$, d.h. wenn $x$ zwischen 1 und 10 (inklusive) liegt.
  - Alternative Schreibweise dafür: `1 <= x <= 10`.

# Logische Operatoren

UNI FREIBURG

- Es gibt die folgenden logischen Operatoren: `or`, `and`, `not` – mit aufsteigender Operatorpräzedenz.
- Wie die Bitoperationen mit (`False` $\mapsto$ 0, `True` $\mapsto$ 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 $1 \leq x$ und $x \leq 10$, d.h. 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.

# Logische Operatoren

UNI FREIBURG

- Es gibt die folgenden logischen Operatoren: `or`, `and`, `not` – mit aufsteigender Operatorpräzedenz.
- Wie die Bitoperationen mit (`False` $\mapsto$ 0, `True` $\mapsto$ 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 $1 \leq x$ und $x \leq 10$, d.h. 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 sind `None`, `0`, `0.0`, `(0 + 0j)` und `''`. Sie werden wie `False` behandelt, alle anderen Werte wie `True`!

# Logische Operatoren

UNI FREIBURG

- Es gibt die folgenden logischen Operatoren: `or`, `and`, `not` – mit aufsteigender Operatorpräzedenz.
- Wie die Bitoperationen mit (`False` $\mapsto$ 0, `True` $\mapsto$ 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 $1 \leq x$ und $x \leq 10$, d.h. 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 sind `None`, `0`, `0.0`, `(0 + 0j)` und `''`. Sie werden wie `False` behandelt, alle anderen Werte wie `True`!
- Die Auswertung der logischen Operatoren endet, sobald das Ergebnis klar ist.

# Logische Operatoren in Aktion

UNI FREIBURG

## Python-Interpreter

```
>>> 1 < 5 < 10
```

# Logische Operatoren in Aktion

UNI FREIBURG



## Python-Interpreter

```
>>> 1 < 5 < 10
True
>>>
```

# Logische Operatoren in Aktion

UNI FREIBURG



## Python-Interpreter

```
>>> 1 < 5 < 10
True
>>> 5 < 1 or 'spam' < 'egg'
```

# Logische Operatoren in Aktion

UNI FREIBURG



## Python-Interpreter

```
>>> 1 < 5 < 10
True
>>> 5 < 1 or 'spam' < 'egg'
False
>>>
```

# Logische Operatoren in Aktion

UNI FREIBURG



## Python-Interpreter

```
>>> 1 < 5 < 10
True
>>> 5 < 1 or 'spam' < 'egg'
False
>>> 'spam' or True
```

# Logische Operatoren in Aktion

UNI FREIBURG



## Python-Interpreter

```
>>> 1 < 5 < 10
True
>>> 5 < 1 or 'spam' < 'egg'
False
>>> 'spam' or True
'spam'
>>>
```

# Logische Operatoren in Aktion

UNI FREIBURG



## Python-Interpreter

```
>>> 1 < 5 < 10
True
>>> 5 < 1 or 'spam' < 'egg'
False
>>> 'spam' or True
'spam'
>>> '' or 'default'
```

# Logische Operatoren in Aktion

UNI FREIBURG



## Python-Interpreter

```
>>> 1 < 5 < 10
True
>>> 5 < 1 or 'spam' < 'egg'
False
>>> 'spam' or True
'spam'
>>> '' or 'default'
'default'
>>>
```

# Logische Operatoren in Aktion

UNI FREIBURG



## 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'
```

# Logische Operatoren in Aktion

UNI FREIBURG



## 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'
>>>
```

# Logische Operatoren in Aktion

UNI FREIBURG



## 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
```

# Logische Operatoren in Aktion

UNI FREIBURG



## 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
>>>
```

# Logische Operatoren in Aktion

UNI FREIBURG



## 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)
```

# Logische Operatoren in Aktion

UNI FREIBURG



## 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
```



# Bedingte Anweisungen

# Bedingte Ausführung

UNI FREIBURG

- Bisher wurde jede eingegebene Anweisung ausgeführt.
- Die bedingte Anweisung (Konditional, `if`-Anweisung) ermöglicht es, Anweisungen nur unter bestimmten Bedingungen auszuführen.

# Bedingte Ausführung

UNI FREIBURG

- Bisher wurde jede eingegebene Anweisung ausgeführt.
- Die bedingte Anweisung (Konditional, `if`-Anweisung) ermöglicht es, Anweisungen nur unter bestimmten Bedingungen auszuführen.

```
def check(x):
    if x > 0:
        print ("x␣ist␣strikt␣positiv")
```

# Bedingte Ausführung

UNI FREIBURG

- Bisher wurde jede eingegebene Anweisung ausgeführt.
- Die bedingte Anweisung (Konditional, `if`-Anweisung) ermöglicht es, Anweisungen nur unter bestimmten Bedingungen auszuführen.

```
def check(x):
    if x > 0:
        print ("x␣ist␣strikt␣positiv")
```

## Python-Interpreter

```
>>> from checker import check
>>> x = 3
>>> check(x)
```

# Bedingte Ausführung



- Bisher wurde jede eingegebene Anweisung ausgeführt.
- Die bedingte Anweisung (Konditional, `if`-Anweisung) ermöglicht es, Anweisungen nur unter bestimmten Bedingungen auszuführen.

```
def check(x):
    if x > 0:
        print ("x␣ist␣strikt␣positiv")
```

## Python-Interpreter

```
>>> from checker import check
>>> x = 3
>>> check(x)
x ist strikt positiv
>>>
```

# Bedingte Ausführung

UNI FREIBURG

- Bisher wurde jede eingegebene Anweisung ausgeführt.
- Die bedingte Anweisung (Konditional, `if`-Anweisung) ermöglicht es, Anweisungen nur unter bestimmten Bedingungen auszuführen.

```
def check(x):
    if x > 0:
        print ("x␣ist␣strikt␣positiv")
```

## Python-Interpreter

```
>>> from checker import check
>>> x = 3
>>> check(x)
x ist strikt positiv
>>> x = 0
>>> check(x)
```

# Bedingte Ausführung

UNI FREIBURG

- Bisher wurde jede eingegebene Anweisung ausgeführt.
- Die bedingte Anweisung (Konditional, `if`-Anweisung) ermöglicht es, Anweisungen nur unter bestimmten Bedingungen auszuführen.

```
def check(x):
    if x > 0:
        print ("x␣ist␣strikt␣positiv")
```

## Python-Interpreter

```
>>> from checker import check
>>> x = 3
>>> check(x)
x ist strikt positiv
>>> x = 0
>>> check(x)
>>>
```

# If-else

UNI FREIBURG

- Die `if-else`-Anweisung ermöglicht es, durch eine Bedingung zwischen zwei Anweisungen auszuwählen.

# If-else

UNI FREIBURG

- Die `if-else`-Anweisung ermöglicht es, durch eine Bedingung zwischen zwei Anweisungen auszuwählen.
- Der `else`-Zweig wird ausgewertet, wenn die Bedingung nicht erfüllt ist.

# If-else

UNI FREIBURG

- Die `if-else`-Anweisung ermöglicht es, durch eine Bedingung zwischen zwei Anweisungen auszuwählen.
- Der `else`-Zweig wird ausgewertet, wenn die Bedingung nicht erfüllt ist.

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

# If-else

UNI FREIBURG

- Die `if-else`-Anweisung ermöglicht es, durch eine Bedingung zwischen zwei Anweisungen auszuwählen.
- Der `else`-Zweig wird ausgewertet, wenn die Bedingung nicht erfüllt ist.

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

**Python-Interpreter**

```
>>> from checker import even_odd
>>> x = 3
>>> even_odd(x)
```

# If-else

UNI FREIBURG

- Die `if-else`-Anweisung ermöglicht es, durch eine Bedingung zwischen zwei Anweisungen auszuwählen.
- Der `else`-Zweig wird ausgewertet, wenn die Bedingung nicht erfüllt ist.

```
def even_odd(x):
    if x % 2 == 0:
        print('x␣ist␣gerade')
    else:
        print('x␣ist␣ungerade')
```

**Python-Interpreter**

```
>>> from checker import even_odd
>>> x = 3
>>> even_odd(x)
x ist ungerade
```

# Verkettete bedingte Anweisungen

UNI FREIBURG

- Eine verkettete bedingte Anweisung kann mehr als zwei Fälle behandeln.

# Verkettete bedingte Anweisungen

UNI FREIBURG

- Eine verkettete bedingte Anweisung kann mehr als zwei Fälle behandeln.

```
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')
```

# Verkettete bedingte Anweisungen

UNI FREIBURG

- Eine verkettete bedingte Anweisung kann mehr als zwei Fälle behandeln.

```
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')
```

**Python-Interpreter**

```
>>> compare(x = 3, y = 0)
```

# Verkettete bedingte Anweisungen

UNI FREIBURG

- Eine verkettete bedingte Anweisung kann mehr als zwei Fälle behandeln.

```
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')
```

**Python-Interpreter**

```
>>> compare(x = 3, y = 0)
x ist größer als y
```

# Verkettete bedingte Anweisungen

UNI FREIBURG

- Eine verkettete bedingte Anweisung kann mehr als zwei Fälle behandeln.

```
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')
```

**Python-Interpreter**

```
>>> compare(x = 3, y = 0)
x ist größer als y
```

- Die Bedingungen werden der Reihe nach ausgewertet. Der erste Block, dessen Bedingung erfüllt ist, wird ausgeführt.

# Geschachtelte Konditionale

UNI FREIBURG

- Bedingte Anweisungen können geschachtelt werden.
- Durch die Einrückung ist immer klar, wozu die bedingte Anweisung gehört!

```
def nested(x):
    if x > 0:
        if x > 10:
            print('successful encyclopedia salesman')
    else:
        print('unsuccessful encyclopedia salesman')
```

## Python-Interpreter

```
>>> nested(x)
```

# Geschachtelte Konditionale

UNI FREIBURG

- Bedingte Anweisungen können geschachtelt werden.
- Durch die Einrückung ist immer klar, wozu die bedingte Anweisung gehört!

```
def nested(x):
    if x > 0:
        if x > 10:
            print('successful encyclopedia salesman')
    else:
        print('unsuccessful encyclopedia salesman')
```

## Python-Interpreter

```
>>> nested(x)
>>>
```

# Sollte die Missachtung der Regeln für die Einrückung ein Fehler sein?

UNI FREIBURG

# Sollte die Missachtung der Regeln für die Einrückung ein Fehler sein?

UNI FREIBURG



**Ja** **Nein**





# Anwendung



# Anwendung — Auswerten eines Tests

# Anwendung — Auswerten eines Tests

UNI FREIBURG



## Bestanden oder nicht?

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

# Anwendung — Auswerten eines Tests

UNI FREIBURG



## 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.

# Schritt 1: Bezeichner und Datentypen

UNI FREIBURG



## Aufgabe

Entwickle eine Funktion `result_for`, 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

# Schritt 2: Funktionsgerüst

UNI FREIBURG



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

- 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

UNI FREIBURG



```
result_for(100, 50, 50) == 'pass'
result_for(100, 50, 30) == 'fail'
result_for(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 Beispiele dienen später als Tests, dass der Code zumindest für die Beispiele funktioniert.

# Schritt 4: Funktionsrumpf ausfüllen

UNI FREIBURG



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

- Die Zuweisung an `passed` erstreckt sich über zwei Zeilen.
- Dafür muss der Ausdruck rechts geklammert sein.
- Zeilenumbruch vor dem Operator >=.

# Nachspiel



- Fertig?

# Nachspiel



- Fertig?
- Was ist, wenn

# Nachspiel



- Fertig?
- Was ist, wenn
  - `max_points < 0`?

# Nachspiel



- Fertig?
- Was ist, wenn
  - `max_points < 0`?
  - `percentage < 0`?

# Nachspiel



- Fertig?
- Was ist, wenn
  - `max_points < 0`?
  - `percentage < 0`?
  - `percentage > 100`?

# Nachspiel



- Fertig?
- Was ist, wenn
  - `max_points < 0`?
  - `percentage < 0`?
  - `percentage > 100`?
  - `points < 0`?

# Nachspiel



- Fertig?
- Was ist, wenn
  - `max_points < 0`?
  - `percentage < 0`?
  - `percentage > 100`?
  - `points < 0`?
  - `points > max_points`?

# Nachspiel

UNI FREIBURG



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

# Zwei Alternativen

UNI FREIBURG

## 1. Defensives Programmieren

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

# Zwei Alternativen

UNI FREIBURG

## 1. Defensives Programmieren

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

## 2. Design by Contract

Spezifiziere die Funktion und programmiere unter der Annahme, dass nur die zulässigen Fälle auftreten.

# Zwei Alternativen

UNI FREIBURG



## 1. Defensives Programmieren

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

## 2. Design by Contract

Spezifiziere die Funktion und programmiere unter der Annahme, dass nur die zulässigen Fälle auftreten.

## Im Codebeispiel: Design by Contract

Annahmen (verschärfen den Typ)

- `max_points >= 0`
- `0 <= percentage <= 100`
- `0 <= points <= max_points`





# Anwendung — Freizeitpark

# Anwendung — Freizeitpark

UNI FREIBURG



## Mitfahren oder nicht?

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

# Anwendung — Freizeitpark

UNI FREIBURG



## 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-Bltz | 6 Jahre und 1,20m | unter 8 Jahre |
| Tiroler Wildwasserbahn | 4 Jahre und 1,00m | unter 9 Jahre |

# Anwendung — Freizeitpark



## 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 `'Du␣darfst␣mitfahren!'` oder `'Du␣musst␣leider␣draussenbleiben.'` liefert.

# Schritt 1: Bezeichner und Datentypen

UNI FREIBURG



## Aufgabe

Entwickle eine Funktion `admit_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` ('Du darfst mitfahren!') oder `False` ('Du musst draussenbleiben.') `-> bool` liefert.

- Festlegen von Einheiten für die Eingaben!

# Schritt 2: Funktionsgerüst

UNI FREIBURG



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

# Schritt 3: Beispiele



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

# Schritt 4: Funktionsrumpf ausfüllen

UNI FREIBURG



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

# Weitere Möglichkeiten



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

# Zusammenfassung

# Zusammenfassung



- `bool` ist ein weiterer Datentyp, dessen einzige Werte `True` und `False` sind.
- Vergleiche, wie z.B. == oder <, liefern Boolesche Werte.
- Boolesche Werte werden bei Bedarf nach `int` konvertiert, wobei `True` $\mapsto$ 1 und `False` $\mapsto$ 0 gilt.
- Nullwerte werden als `False` interpretiert, alle anderen Werte als `True`.
- Bedingte Anweisungen (`if`-(`elif`)-`else`) erlauben die Auswahl zwischen alternativen Anweisungen.
- Checkliste zum Entwurf von Funktionen: Bezeichner und Datentypen, Funktionsgerüst, Beispiele, Funktionsrumpf