Informatik I: Einführung in die Programmierung

5. Bedingungen, bedingte Ausführung

Albert-Ludwigs-Universität Freiburg



UNI FREIBURG

Prof. Dr. Peter Thiemann

03. November 2026



UNI FREIBURG

Bedingungen
Typ bool
Vergleichsoperationen
Logische Operatoren
Bedingte Anweisungen
Anwendung
Zusammenfassung

Bedingungen und der Typ `bool`

Der Typ `bool`



UNI FREIBURG

- 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`: `False` $\mapsto 0$, `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

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*?

Bedingungen
Typ bool
Vergleichsoperationen
Logische Operatoren
Bedingte Anweisungen
Anwendung
Zusammenfassung

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*?

bool_ex.py

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

Bedingungen
Typ bool
Vergleichsoperationen
Logische Operatoren
Bedingte Anweisungen
Anwendung
Zusammenfassung

Stringvergleich



UNI FREIBURG

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

Bedingungen
Typ `bool`
Vergleichsoperationen
Logische Operatoren
Bedingte Anweisungen
Anwendung
Zusammenfassung

Stringvergleich



UNI FREIBURG

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

Python-Interpreter

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


Bedingungen
Typ bool
Vergleichsoperationen
Logische Operatoren
Bedingte Anweisungen
Anwendung
Zusammenfassung

Stringvergleich



UNI FREIBURG

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

Python-Interpreter

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


Bedingungen
Typ bool
Vergleichsoperationen
Logische Operatoren
Bedingte Anweisungen
Anwendung
Zusammenfassung

Stringvergleich



UNI FREIBURG

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

Python-Interpreter

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


Bedingungen
Typ bool
Vergleichsoperationen
Logische Operatoren
Bedingte Anweisungen
Anwendung
Zusammenfassung

Stringvergleich



UNI FREIBURG

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

Python-Interpreter

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


Bedingungen
Typ bool
Vergleichsoperationen
Logische Operatoren
Bedingte Anweisungen
Anwendung
Zusammenfassung

Stringvergleich



UNI FREIBURG

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'
```


Bedingungen
Typ bool
Vergleichsoperationen
Logische Operatoren
Bedingte Anweisungen
Anwendung
Zusammenfassung

Stringvergleich



UNI FREIBURG

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
>>>
```


Bedingungen
Typ bool
Vergleichsoperationen
Logische Operatoren
Bedingte Anweisungen
Anwendung
Zusammenfassung

Stringvergleich



UNI FREIBURG

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'
```


Bedingungen
Typ bool
Vergleichsoperationen
Logische Operatoren
Bedingte Anweisungen
Anwendung
Zusammenfassung

Stringvergleich



UNI FREIBURG

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

Logische Operatoren



UNI FREIBURG

- Logische Operatoren auf `bool`

Bedingungen
Typ `bool`
Vergleichsoperationen
Logische Operatoren
Bedingte Anweisungen
Anwendung
Zusammenfassung

Logische Operatoren



- Logische Operatoren auf `bool`
- **`or`**, **`and`**, **`not`** – in aufsteigender Operatorpräzedenz.

Bedingungen
Typ bool
Vergleichsoperationen
Logische Operatoren
Bedingte Anweisungen
Anwendung
Zusammenfassung

Logische Operatoren



UNI FREIBURG

- Logische Operatoren auf `bool`
- **`or`**, **`and`**, **`not`** – in aufsteigender Operatorpräzedenz.
- Wie die Bitoperationen mit (`False` $\leftrightarrow$ 0, `True` $\leftrightarrow$ 1), d.h.

Bedingungen
Typ `bool`
Vergleichsoperationen
Logische Operatoren
Bedingte Anweisungen
Anwendung
Zusammenfassung

Logische Operatoren



UNI FREIBURG

- Logische Operatoren auf `bool`
- **`or`**, **`and`**, **`not`** – in aufsteigender Operatorpräzedenz.
- Wie die Bitoperationen mit (`False` ↔ 0, `True` ↔ 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.

Bedingungen
Typ bool
Vergleichsoperationen
Logische Operatoren
Bedingte Anweisungen
Anwendung
Zusammenfassung

Logische Operatoren



UNI FREIBURG

- 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.

Bedingungen
Typ bool
Vergleichsoperationen
Logische Operatoren
Bedingte Anweisungen
Anwendung
Zusammenfassung

Logische Operatoren



UNI FREIBURG

- 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`.

Bedingungen
Typ bool
Vergleichsoperationen
Logische Operatoren
Bedingte Anweisungen
Anwendung
Zusammenfassung

Logische Operatoren



UNI FREIBURG

- 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.

Bedingungen
Typ bool
Vergleichsoperationen
Logische Operatoren
Bedingte Anweisungen
Anwendung
Zusammenfassung

Logische Operatoren



UNI FREIBURG

- Logische Operatoren auf `bool`
- **`or`**, **`and`**, **`not`** – in aufsteigender Operatorpräzedenz.
- Wie die Bitoperationen mit (`False` ↔ 0, `True` ↔ 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`!

Bedingungen
Typ bool
Vergleichsoperationen
Logische Operatoren
Bedingte Anweisungen
Anwendung
Zusammenfassung

Logische Operatoren



UNI FREIBURG

- Logische Operatoren auf `bool`
- **`or`**, **`and`**, **`not`** – in aufsteigender Operatorpräzedenz.
- Wie die Bitoperationen mit (`False` ↔ 0, `True` ↔ 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-circuit evaluation)

Bedingungen
Typ bool
Vergleichsoperationen
Logische Operatoren
Bedingte Anweisungen
Anwendung
Zusammenfassung

Logische Operatoren in Aktion



UNI FREIBURG

Python-Interpreter

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


Bedingungen
Typ bool
Vergleichsoperationen
Logische Operatoren
Bedingte Anweisungen
Anwendung
Zusammenfassung

Logische Operatoren in Aktion



UNI FREIBURG

Python-Interpreter

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


Bedingungen
Typ bool
Vergleichsoperationen
Logische Operatoren
Bedingte Anweisungen
Anwendung
Zusammenfassung

Logische Operatoren in Aktion



UNI FREIBURG

Bedingungen
Typ bool
Vergleichsoperationen
Logische Operatoren
Bedingte Anweisungen
Anwendung
Zusammenfassung

Python-Interpreter

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

Logische Operatoren in Aktion



UNI FREIBURG

Bedingungen
Typ bool
Vergleichsoperationen
Logische Operatoren
Bedingte Anweisungen
Anwendung
Zusammenfassung

Python-Interpreter

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

Logische Operatoren in Aktion



UNI FREIBURG

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
```

Logische Operatoren in Aktion



UNI FREIBURG

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'
>>>
```

Logische Operatoren in Aktion



UNI FREIBURG

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'
```

Logische Operatoren in Aktion



UNI FREIBURG

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'
>>>
```

Logische Operatoren in Aktion



UNI FREIBURG

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'
```

Logische Operatoren in Aktion



UNI FREIBURG

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'
>>>
```

Logische Operatoren in Aktion



UNI FREIBURG

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
```

Logische Operatoren in Aktion



UNI FREIBURG

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
>>>
```

Logische Operatoren in Aktion



UNI FREIBURG

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)
```

Logische Operatoren in Aktion



UNI FREIBURG

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
```



UNI FREIBURG

Bedingungen
Bedingte Anweisungen
if-Anweisung
if-else-Anweisung
elif-Anweisung
Anwendung
Zusammenfassung

Bedingte Anweisungen

Kontrollfluss—Bedingte Ausführung



UNI FREIBURG

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

Bedingungen

Bedingte Anweisungen

`if`-Anweisung

`if-else`-Anweisung

`elif`-Anweisung

Anwendung

Zusammenfassung

Kontrollfluss—Bedingte Ausführung



UNI FREIBURG

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.

```
def strictly_positive(x : int) -> str:
    if x > 0:
        return str(x) + ' is strictly positive'
    return str(x) + ' is not strictly positive'
```

Python-Interpreter

```
>>> strictly_positive(3)
```

Kontrollfluss—Bedingte Ausführung



UNI FREIBURG

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.

```
def strictly_positive(x : int) -> str:
    if x > 0:
        return str(x) + ' is strictly positive'
    return str(x) + ' is not strictly positive'
```

Python-Interpreter

```
>>> strictly_positive(3)
'3 is strictly positive'
>>>
```

Kontrollfluss—Bedingte Ausführung



UNI FREIBURG

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

```
def strictly_positive(x : int) -> str:
    if x > 0:
        return str(x) + ' is strictly positive'
    return str(x) + ' is not strictly positive'
```

Python-Interpreter

```
>>> strictly_positive(3)
'3 is strictly positive'
>>> strictly_positive(0)
```


Bedingungen
Bedingte Anweisungen
if-Anweisung
if-else-Anweisung
elif-Anweisung
Anwendung
Zusammenfassung

Kontrollfluss—Bedingte Ausführung



UNI FREIBURG

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

```
def strictly_positive(x : int) -> str:
    if x > 0:
        return str(x) + ' is strictly positive'
    return str(x) + ' is not strictly positive'
```

Python-Interpreter

```
>>> strictly_positive(3)
'3 is strictly positive'
>>> strictly_positive(0)
'0 is not strictly positive'
>>>
```


Bedingungen
Bedingte Anweisungen
if-Anweisung
if-else-Anweisung
elif-Anweisung
Anwendung
Zusammenfassung

If-else



UNI FREIBURG

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

Bedingungen
Bedingte Anweisungen
if-Anweisung
if-else-Anweisung
elif-Anweisung
Anwendung
Zusammenfassung

If-else



UNI FREIBURG

- 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.

Bedingungen
Bedingte Anweisungen
if-Anweisung
if-else-Anweisung
elif-Anweisung
Anwendung
Zusammenfassung

If-else



UNI FREIBURG

- 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.

Bedingungen
Bedingte Anweisungen
if-Anweisung
if-else-Anweisung
elif-Anweisung
Anwendung
Zusammenfassung

If-else



UNI FREIBURG

- 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: int) -> str:
    if x % 2 == 0:
        return str(x) + ' is even'
    else:
        return str(x) + ' is odd'
```

Python-Interpreter

```
>>> evenodd(3)
```


Bedingungen
Bedingte Anweisungen
if-Anweisung
if-else-Anweisung
elif-Anweisung
Anwendung
Zusammenfassung

If-else



UNI FREIBURG

- 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: int) -> str:
    if x % 2 == 0:
        return str(x) + ' is even'
    else:
        return str(x) + ' is odd'
```

Python-Interpreter

```
>>> evenodd(3)
'3 is odd
```


Bedingungen
Bedingte Anweisungen
if-Anweisung
if-else-Anweisung
elif-Anweisung
Anwendung
Zusammenfassung

Verkettete bedingte Anweisungen



```
def compare(x : int, y : int) -> str:
    if x < y:
        return str(x) + ' is less than ' + str(y)
    elif x > y:
        return str(x) + ' is greater than ' + str(y)
    else:
        return str(x) + ' and ' + str(y) + ' are equal'
```

- 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)
```


Bedingungen
Bedingte Anweisungen
if-Anweisung
if-else-Anweisung
elif-Anweisung
Anwendung
Zusammenfassung

Verkettete bedingte Anweisungen



```
def compare(x : int, y : int) -> str:
    if x < y:
        return str(x) + ' is less than ' + str(y)
    elif x > y:
        return str(x) + ' is greater than ' + str(y)
    else:
        return str(x) + ' and ' + str(y) + ' are equal'
```

- 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 is greater than 0'
```


Bedingungen
Bedingte Anweisungen
if-Anweisung
if-else-Anweisung
elif-Anweisung
Anwendung
Zusammenfassung

Geschachtelte Konditionale



UNI FREIBURG

Bedingungen

Bedingte Anweisungen

if-Anweisung

if-else-Anweisung

elif-Anweisung

Anwendung

Zusammenfassung


```
def salesman(x:int) -> str:
    if x > 0:
        if x > 10:
            return 'successful encyclopedia salesman'
        ####
    else:
        return 'unsuccessful encyclopedia salesman'
    return 'tramp'
```

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



Bedingungen
Bedingte Anweisungen
Anwendung
Auswerten eines Tests
Nachspiel: Verträge
Freizeitpark
Zusammenfassung

Anwendung



Bedingungen
Bedingte Anweisungen
Anwendung
Auswerten eines Tests
Nachspiel: Verträge
Freizeitpark
Zusammenfassung

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.

Bedingungen
Bedingte Anweisungen
Anwendung
Auswerten eines Tests
Nachspiel: Verträge
Freizeitpark
Zusammenfassung

Anwendung — Auswerten eines Tests



UNI FREIBURG

Bedingungen
Bedingte Anweisungen
Anwendung
Auswerten eines Tests
Nachspiel: Verträge
Freizeitpark
Zusammenfassung

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

Bedingungen
Bedingte Anweisungen
Anwendung
Auswerten eines Tests
Nachspiel: Verträge
Freizeitpark
Zusammenfassung

Aufgabe

Entwickle eine Funktion `evaluate_test`, 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



Bedingungen
Bedingte Anweisungen
Anwendung
Auswerten eines Tests
Nachspiel: Verträge
Freizeitpark
Zusammenfassung

```
def evaluate_test(
        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

Bedingungen
Bedingte Anweisungen
Anwendung
Auswerten eines Tests
Nachspiel: Verträge
Freizeitpark
Zusammenfassung

```
assert(evaluate_test(100, 50, 50) == 'pass')
assert(evaluate_test(100, 50, 30) == 'fail')
assert(evaluate_test(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.

Schritt 4: Funktionsrumpf ausfüllen



UNI FREIBURG

Bedingungen

Bedingte Anweisungen

Anwendung

Auswerten eines Tests

Nachspiel: Verträge

Freizeitpark

Zusammenfassung


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

Nachspiel: Verträge



UNI FREIBURG

- Fertig?

Bedingungen
Bedingte Anweisungen
Anwendung
Auswerten eines Tests
Nachspiel: Verträge
Freizeitpark
Zusammenfassung

Nachspiel: Verträge



UNI FREIBURG

- Fertig?
- Was ist, wenn

Bedingungen

Bedingte Anweisungen

Anwendung

Auswerten eines Tests

Nachspiel: Verträge

Freizeitpark

Zusammenfassung

Nachspiel: Verträge



UNI FREIBURG

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

Bedingungen
Bedingte Anweisungen
Anwendung
Auswerten eines Tests
Nachspiel: Verträge
Freizeitpark
Zusammenfassung

Nachspiel: Verträge



UNI FREIBURG

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

Bedingungen
Bedingte Anweisungen
Anwendung
Auswerten eines Tests
Nachspiel: Verträge
Freizeitpark
Zusammenfassung

Nachspiel: Verträge



UNI FREIBURG

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

Bedingungen
Bedingte Anweisungen
Anwendung
Auswerten eines Tests
Nachspiel: Verträge
Freizeitpark
Zusammenfassung

Nachspiel: Verträge



UNI FREIBURG

Bedingungen

Bedingte Anweisungen

Anwendung

Auswerten eines Tests

Nachspiel: Verträge

Freizeitpark

Zusammenfassung

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

Nachspiel: Verträge



UNI FREIBURG

Bedingungen

Bedingte Anweisungen

Anwendung

Auswerten eines Tests

Nachspiel: Verträge

Freizeitpark

Zusammenfassung

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

Nachspiel: Verträge



UNI FREIBURG

Bedingungen

Bedingte Anweisungen

Anwendung

Auswerten eines Tests

Nachspiel: Verträge

Freizeitpark

Zusammenfassung

- 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

Defensives Programmieren

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

Bedingungen

Bedingte Anweisungen

Anwendung

Auswerten eines Tests

Nachspiel: Verträge

Freizeitpark

Zusammenfassung

Zwei Alternativen



UNI FREIBURG

Defensives Programmieren

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

Design by Contract

Bedingungen

Bedingte Anweisungen

Anwendung

Auswerten eines Tests

Nachspiel: Verträge

Freizeitpark

Zusammenfassung

Zwei Alternativen



UNI FREIBURG

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).

Bedingungen
Bedingte Anweisungen
Anwendung
Auswerten eines Tests
Nachspiel: Verträge
Freizeitpark
Zusammenfassung

Zwei Alternativen



UNI FREIBURG

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`

Bedingungen
Bedingte Anweisungen
Anwendung
Auswerten eines Tests
Nachspiel: Verträge
Freizeitpark
Zusammenfassung

Zwei Alternativen



UNI FREIBURG

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`

Bedingungen
Bedingte Anweisungen
Anwendung
Auswerten eines Tests
Nachspiel: Verträge
Freizeitpark
Zusammenfassung

Zwei Alternativen



UNI FREIBURG

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 Anweisungen
Anwendung
Auswerten eines Tests
Nachspiel: Verträge
Freizeitpark
Zusammenfassung



UNI FREIBURG

Bedingungen
Bedingte Anweisungen
Anwendung
Auswerten eines Tests
Nachspiel: Verträge
Freizeitpark
Zusammenfassung

Anwendung — Freizeitpark

Anwendung — Freizeitpark



UNI FREIBURG

Mitfahren oder nicht?

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

Bedingungen
Bedingte Anweisungen
Anwendung
Auswerten eines Tests
Nachspiel: Verträge
Freizeitpark
Zusammenfassung

Anwendung — Freizeitpark



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, kleiner 1,95m	unter 10 Jahre
blue fire	7 Jahre und 1,30m	—
Eurosat Cancan Coaster	6 Jahre und 1,20m, kleiner 1,95m	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



UNI FREIBURG

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.

Schritt 1: Bezeichner und Datentypen



UNI FREIBURG

Bedingungen

Bedingte Anweisungen

Anwendung

Auswerten eines Tests

Nachspiel: Verträge

Freizeitpark

Zusammenfassung

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!

Schritt 2: Funktionsgerüst



UNI FREIBURG

Bedingungen

Bedingte Anweisungen

Anwendung

Auswerten eines Tests

Nachspiel: Verträge

Freizeitpark

Zusammenfassung


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

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, ''))
```

Schritt 4: Funktionsrumpf ausfüllen



UNI FREIBURG

Bedingungen

Bedingte Anweisungen

Anwendung

Auswerten eines Tests

Nachspiel: Verträge

Freizeitpark

Zusammenfassung


```
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
```

Weitere Möglichkeiten



UNI FREIBURG

Bedingungen
Bedingte Anweisungen
Anwendung
Auswerten eines Tests
Nachspiel: Verträge
Freizeitpark
Zusammenfassung

- 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.



UNI FREIBURG

Bedingungen
Bedingte Anweisungen
Anwendung
Zusammenfassung

Zusammenfassung

Zusammenfassung



UNI FREIBURG

Bedingungen
Bedingte Anweisungen
Anwendung
Zusammenfassung

- `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