CloudCommand

Cyber Security

CloudCommand GmbH chr.schumacher@gmx.tm

CloudCommand

Grundlagen der Programmierung

CloudCommand GmbH chr.schumacher@gmx.tm

Bedingte Verzweigungen

Vergleichsoperatoren

Sie definieren zum Beispiel Bedingungen in If-Anweisungen.

- **Gleicheit:** ==
- **Ungleicheit:** !=
- **Größer:** >, >=
- **Kleiner:** <, <=

Vergleichsoperatoren

```
x == 5
#ergibt True, wenn x den Wert 5 enthält

s == "Mein Text"
#Vergleichsoperatoren funktionieren nicht nur mit Zahlen
#ergibt True, wenn s genau den String "Mein Text" enthält

15 < 25
#ergibt True

15 > 25
#ergibt False
```

Logische Operatoren

Bedingungen können mit logischen Operatoren verknüpft werden. Die primären logischen Operatoren sind:

- **and:** Ergibt True, wenn beide Operanden True sind. Andernfalls False!
- **or:** Ergibt True, wenn mindestens einer der Operanden True ist. Andernfalls False!
- **not:** Negiert den Wert des folgenden Operanden.

Logische Operatoren

```
(True and True)
#True

(True and False)
#False

(True or False)
#True

not True
#False
```

Logische Operatoren

Zusätzliche Anmerkungen:

- Kombinationen tragen zur Erstellung komplexerer logischer Ausdrücke bei.
- Auswertung erfolgt von links nach rechts.
- Klammern steuern die Reihenfolge der Auswertung.

Der Mitgliedschaftsoperator “in”

Der in Operator in Python wird vorwiegend dazu verwendet, um die Mitgliedschaft eines Elements in einem Container-Objekt wie Listen, Tupeln oder Strings zu überprüfen. Er gibt ein Boolesches Ergebnis zurück: True, wenn das Element enthalten ist, und False andernfalls.

- **Überprüfung in Listen:** Ist der Wert in der Liste vorhanden?
- **Überprüfung in Strings:** Ist das Zeichen oder die Zeichenfolge Teil des Strings?
- **Überprüfung in Dictionaries:** Ist der Schlüssel im Dictionary vorhanden?

Der Mitgliedschaftsoperator “in”

```
3 in [1, 2, 3, 4]
#True
'a' in 'apple'
#True
'key' in {'key': 'value'}
#True
```

Bitweise Operatoren

Bitweisen Operatoren arbeiten auf einer „tieferen“ und komplexeren Ebene. Sie werden heute nicht behandelt, sollen aber der Vollständigkeit halber hier kurz erwähnt werden. Es handelt sich um folgende Operatoren:

- **&** Bitweises UND
- **|** Bitweises ODER
- **^** Bitweises XOR
- **~** Bitweises NICHT
- **<<** Bitshift links - verschiebt die Bits nach links
- **>>** Bitshift rechts - verschiebt die Bits nach rechts

Bedingte Abfragen: if

Was sind bedingte Abfragen?

- Anweisungen, die die Ausführung bestimmter Codeblöcke in Abhängigkeit von einer/mehreren Bedingungen ermöglichen.
- unerlässlich für die Steuerung des Programmflusses, die Datenvalidierung und die Fehlerbehandlung.

Bedingte Abfragen: if

```
if bedingung:
    # Code, der ausgeführt wird, wenn die Bedingung wahr ist

alter = 25

if alter >= 18:
    print("Du darfst wählen.")
else:
    print("Du darfst noch nicht wählen.")
```

Bedingte Abfragen: if

- Vergleichsoperatoren bestimmen, ob eine Bedingung wahr oder falsch ist.
- logischen Operatoren verknüpfen mehrere Bedingungen

```
if x == y:
    print("x und y sind gleich")

if x > 10:
    print("x ist größer als 10")

if x > 10 and x < 20:
    print("x liegt zwischen 10 und 20")
```

Verzweigung

- befindet sich BMW in x so wird die Ausgabe “Die Automarke ist gültig” ausgegeben.
- ansonsten erfolgt die Ausgabe “Die Automarke ist ungültig” ausgegeben

```
x = ["VW", "Skoda", "Seat"]

if "BMW" in x:
    print("Die Automarke ist gültig")
else:
    print("Die Automarke ist ungültig")
```

Merke: Wenn die Bedingung der if-Anweisung false ergibt, wird die Anweisung innerhalb des eingerückten else-Blocks ausgeführt!

Verzweigung

- ist x größer als 10, so wird “Größer als 10 ausgegeben”
- falls x nicht größer als 10 sondern gleich 10 ist, wird “Gleich 10”, ausgegeben
- in allen anderen Fällen (hier: x ist kleiner als 10) wird “Kleiner als 10” ausgegeben.

```
if x > 10:
    print("Größer als 10")
elif x == 10:
    print("Gleich 10")
else:
    print("Kleiner als 10")
```

Merke: Wenn die Bedingungen der if-Anweisung und der elif-Anweisung false ergeben, wird die Anweisung innerhalb des eingerückten else-Blocks ausgeführt!

Verschachtelung

- wenn die erste If-Bedingung, x kleiner als 10, richtig ist…
- wird die verschachtelte if-Bedingung geprüft…
- wenn y größer als 10 ist wird “Beide Werte sind größer als 10” ausgegeben

```
if x > 10:
    if y > 10:
        print("Beide Werte sind größer als 10")
```

Merke: Wenn die Bedingung der if-Anweisung true ergibt, wird die verschachtelte if-Anweisung überprüft!

if – wichtige Regeln

- Lesbarkeit beibehalten
- ordnungsgemäße Verschachtelung
- Verwendung kurzer, beschreibender Variablennamen

häufige Anwendungsfälle von If-Else

- Validieren von Eingabedaten
- Durchführen von Berechnungen auf der Grundlage unterschiedlicher Bedingungen
- Steuerung des Programmablaufs in Abhängigkeit von der Auswahl des Benutzers
- Vergleichen von Werten aus verschiedenen Quellen
- Bestimmung des Endergebnisses eines Spiels oder Wettbewerbs

Pattern matching

- Erweiterung der Möglichkeiten von if-Anweisungen
- Lesbarkeit teilweise stark verbessert

Hier eine Gegenüberstellung:

```python
if x == 1:
    print("x ist 1")
elif x == 2:
    print("x ist 2")
elif x == 3:
    print("x ist 3")
elif x == 4:
    print("x ist 4")
else:
    print("x ist nicht 1, 2, 3 oder 4")
```

```python
match x:
    case 1:
        print("x ist 1")
    case 2:
        print("x ist 2")
    case 3:
        print("x ist 3")
    case 4:
        print("x ist 4")
    case _:
        print("x ist nicht 1, 2, 3 oder 4")
```

Pattern matching

Regeln:

- Abarbeitung der Fälle, von oben nach unten
- Abarbeitung der Fälle bis es ein “match” gibt
- keine Abarbeitung der darunterliegenden Fälle
- “case _” wird als wildcard bezeichnet und wird ausgeführt, wenn es keine anderen Treffer gab

Pattern matching

Komplexere Bedingungen können ebenfalls durch Pattern Matching dargestellt werden:

```
match x:
    case 1:
        print("x ist 1")
    case 2 | 3 | 4:
        print("x ist 2, 3 oder 4")
    case x if x > 4:
        print("x ist größer als 4")
    case _:
        print("x ist weder eine ganze Zahl von 1 bis 4 noch eine Zahl größer als 4")
```

- **|** kann als **oder** verwendet werden, nicht mit bitweisen ODER verwechseln
- Bedingungen wie größer/kleiner geprüft werden.

Anwendungsbeispiel 1: Stringlänge

- prüfen einer Zeichenkette auf eine Mindestlänge von 6 Zeichen
- validiert die Benutzereingabe eines Namens

```
if len(username) >= 6:
    print("Gültiger Benutzername.")
else:
    print("Der Benutzername ist zu kurz.")
```

Anwendungsbeispiel 2: Zahl in einem Bereich

Ein etwas komplexeres Beispiel prüft, ob eine Zahl in einem bestimmten Bereich liegt. Auch hier wird eine Benutzereingabe validiert.

- Die Klammern sind optional, hier nur zur visuellen Abgrenzung.

```
if (1 <= choice) and (choice <= 5):
    print("Gültige Auswahl")
else:
    print("Ungültige Auswahl")
```

- eine andere Schreibweise des obigen Beispiels:

```
if 1 <= choice <= 5:
    print("Gültige Auswahl")
else:
    print("Ungültige Auswahl")
```

Anwendungsbeispiel 3.1: if-elif-else in einer Funktion

Einsatzmöglichkeit von if-elif-else, wie sie in der Webentwicklung zum Einsatz kommen könnte, zum HTTP-Statuscodes für den Benutzer darzustellen.

```
def getStatus(code):
    if code == 200:
        return "OK"
    elif code == 404:
        return "Not Found"
    else:
        return "Unknown"
```

- sich wiederholenden Code in eine eigene Funktion auslager

Anwendungsbeispiel 3.2: match-case in einer Funktion

Auch das letzte Beispiel kann, statt eine if-elif-else-Kette zu benutzen, mittels match-case abgebildet werden

```
def getStatus(code):
    match code:
        case 200:
            return "OK"
        case 404:
            return "Not Found"
        case _:
            return "Unknown"
```

Anwendungsbeispiel 4.1: best practice mit mehreren if

Addition drei positiver Zahlen:

- Verschachtelte if-Abfrage:Erfüllung mehrerer Bedingungen, um Codeblock auszuführen

```
def calculate(a, b, c):
    if a >= 0:
        if b >= 0:
            if c >= 0:
                return a + b + c
            else:
                return "c ist nicht positiv"
        else:
            return "b ist nicht positiv"
    else:
        return "a ist nicht positiv"
```

Anwendungsbeispiel 4.2: best practice mit mehreren if

- mehreren „guard clauses“ können wir die Verschachtelung auflösen:

```
def calculate(a, b, c):
    if a < 0:
        return "a ist nicht positiv"
    if b < 0:
        return "b ist nicht positiv"
    if c < 0:
        return "c ist nicht positiv"

    return a + b + c
```

Grundlagen der Fehlerbehandlung

Fehlerbehandlung ist ein integraler Bestandteil jeder Anwendung. Mit try und except können Fehler elegant abgefangen werden.

```
try:
    # riskanter Code
except SomeError:
    # Fehlerbehandlung
```

try-except

- try-Block umschließt Code der vermutlich Fehler enthält
- except-Block fängt möglichen Fehler auf
- mehrere except-Blöcke möglich (behandelt verschiedene Fehlerarten)
- bei Eintritt eines Fehlers springt die Ausführung zum except-Block
- tritt kein Fehler auf werden except-Blöcke übersprungen

```
try:
    print(x)
except:
    print('Konnte x nicht in der Konsole anzeigen')
# Konnte x nicht in der Konsole anzeigen
```

try-except-else

- der else-Block wird ausgeführt, wenn keine Fehler vorhanden sind
- die Blöcke try und except werden übersprungen

```
try:
    print("Es sind")
except:
    print("Fehler aufgetreten")
else:
    print("keine Fehler aufgetreten")
# Es sind:
# keine Fehler aufgetreten
```

try-except-finally

- der finally-Block wird ausgeführt, unabhängig davon ob ein Fehler vorliegt
- zunächst wird try und except in Abhängigkeit des Fehlers ausgeführt
- darauf folgt in jedem Fall der finally-Block

```
try:
    print(x)
except:
    print('Konnte x nicht in der Konsole anzeigen')
finally:
    print("Fertig.")
# Konnte x nicht in der Konsole anzeigen
# Fertig.
```

Erweiterte Fehlerbehandlung

Fehlerbehandlung kann optional auch **else** und/oder **finally** beinhalten:

```
x = 0

try:
    result = 10 / x
except ZeroDivisionError:
    print("Division durch Null ist nicht erlaubt.")
else:
    print(result)
finally:
    print("Dieser Block wird immer ausgeführt.")
```

Exception Handling

loggen der Fehler

- zukunftsorientierte Fehlersuche

nicht zu viele try-except-Blöcke

- Beibehaltung der Lesbarkeit

else-Block

- Erhöhung der Klarheit

contextlib.suppress()

- ignoriert aufgetretene Fehler
- nützlich wenn unkritischer Fehler erwartet wird

```
from contextlib import suppress

with suppress(ZeroDivisionError):
    # Dieser Code verursacht normalerweise einen ZeroDivisionError
    ergebnis = 10 / 0

print('Programm fortgesetzt.')
```

Fehlerausgabe

- Möglichkeit über den Fehler mehr Informationen auszugeben
- statement as, um den Fehler einer lokalen Variable zuzuweisen
- Variable nur in diesem except-Block gültig ist
- nach Ausführung des Blocks wird Variable gelöscht

```
try:
    # riskanter Code
except ValueError as ve:
    print(f"Fehlerdetails: {ve}")
```

Fehlerarten

SyntaxError:

- Python versteht Code nicht

IndexError:

- Zugriff auf nicht existentes Index eines Listenelements

KeyError:

- Schlüssel wird in Wörterbuch nicht gefunden

ValueError:

- Parameter in einer Funktion oder Operation, ist zwar gültiger Typ, hat aber unangemessenen Wert

NameError:

- Variable oder Funktion ist im Kontext des Programms nicht definiert

Fehlerarten

```
# Beispiel für einen SyntaxError
print('Hallo Welt'
# Beispiel für einen IndexError
liste = [1, 2, 3]
print(liste[3])
# Beispiel für einen KeyError
woerterbuch = {'name': 'Max'}
print(woerterbuch['alter'])
# Beispiel für einen ValueError
int('zehn')
```

Relevanz Fehlerarten

Warum ist es wichtig unterschiedliche Fehlerarten zu identifizieren?

- Verständnis für Fehler
- Entwicklung effektiver Lösungen
- wartungsfreundlichere und lesbare Gestaltung des Codes
- Vermeidung zukünftiger Fehler
- Verbesserung der Qualität von Programmen

Dateien sicher öffnen

Das angemessene öffnen von Dateien kann potenzielle Fehler vermeiden:

```
with open('datei.txt', 'r') as datei:
    inhalt = datei.read()
```

- with-Anweisung
- Datei wird richtig geschlossen, auch wenn ein Fehler auftritt
- gibt Ressourcen frei und vermeidet Fehler

FileNotFoundError

Zu öffnende Datei wird nicht gefunden.

```
try:
    with open('nicht_existierende_datei.txt', 'r') as datei:
        inhalt = datei.read()
except FileNotFoundError:
    print('Datei wurde nicht gefunden.')
```

PermissionError

- erforderlichen Berechtigungen fehlen, um eine Datei zu lesen, zu schreiben oder zu öffne

Lösung:

- Berechtigungen überprüfen
- sicherstellen, dass Python-Prozess Zugriff auf die Datei hat

CloudCommand

CloudCommand GmbH chr.schumacher@gmx.tm