

Python Crashkurs

Inhalt



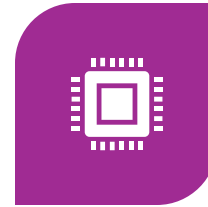
WAS SIND
PROGRAMMIERSPRACHE
N?



KLASSIFIKATION VON
PROGRAMMIERSPRACHE
N



COMPILER &
INTERPRETER



LIVE: VS-CODE &
TERMINAL



LIVE: PYTHON
EINFÜHRUNG



GRUNDLAGEN DER
BOOLESCHEN ALGEBRA



LIVE:
KONTROLLSTRUKTUREN:
IF & SCHLEIFEN



LIVE: FUNKTIONEN

Inhalt



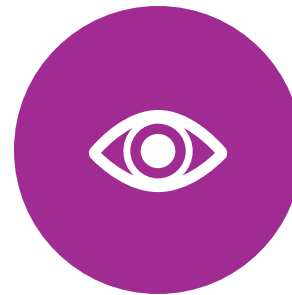
Live: Bibliotheken &
Packages



Live: Git



Grundlagen
Objektorientierung

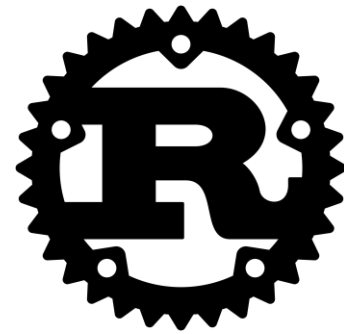


Ausblick



Programmiersprachen

⇒ GO



Was sind Programmiersprachen

Eine **Programmiersprache** ist eine formale Sprache, die aus einer Menge von Regeln (Syntax und Semantik) besteht und verwendet wird, um **Computerprogramme** zu schreiben, die von Computern interpretiert oder ausgeführt werden können.



Was sind Programmiersprachen

- **Syntax:** Legt fest, wie Programme geschrieben werden müssen (z. B. Schlüsselwörter, Klammern, Einrückung).
- **Semantik:** Beschreibt die Bedeutung der geschriebenen Befehle.
- **Formalisierung:** Im Gegensatz zur natürlichen Sprache ist sie eindeutig und maschinenlesbar.
- **Abstraktion:** Erlaubt die Modellierung komplexer Probleme auf unterschiedlichen Abstraktionsebenen.
- Beispiele: Python, Java, C, C++ , C#, JavaScript



Klassifikation von Programmiersprachen

- Programmiersprachen sind:
 - Turing-complete (können jede berechenbare Funktion ausdrücken)
 - Ermöglichen vollständige Programmlogik und Ablaufsteuerung
 - Kompilierte oder interpretierte Ausführung
 - Für System- und Anwendungsentwicklung geeignet

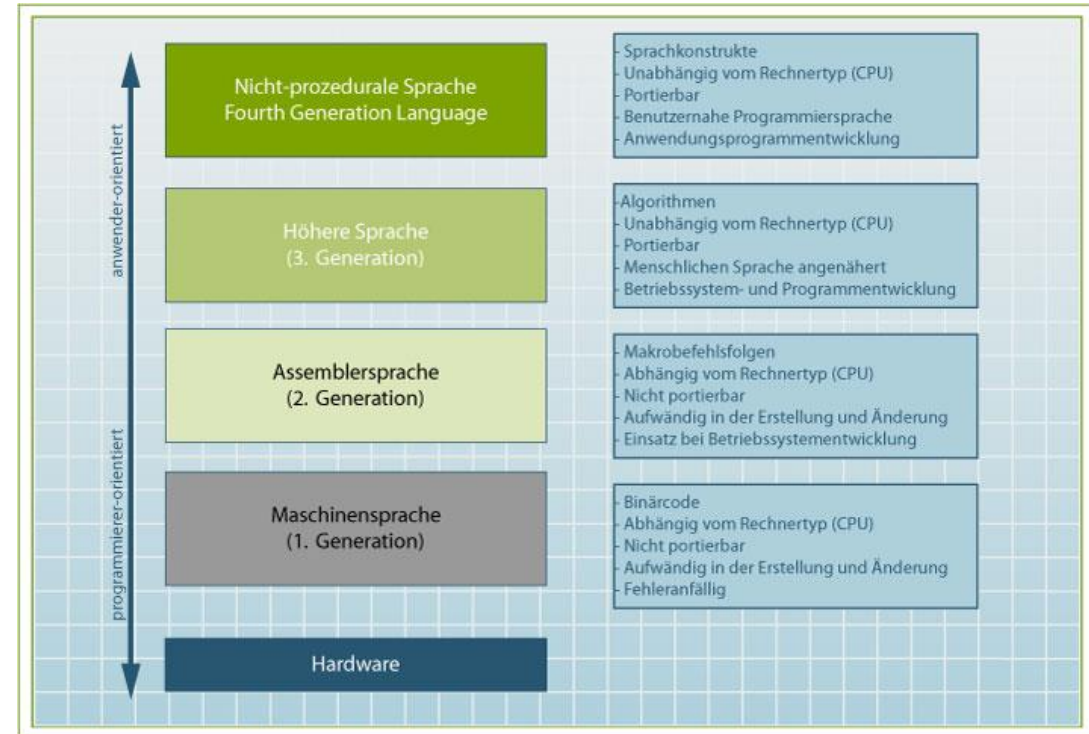


Klassifikation von Programmiersprachen

- Skripting-Sprachen als Unterkategorie
 - Häufig Interpretiert
 - Meist leichtgewichtig und dynamisch typisiert
 - Gut geeignet zur Automatisierung von abläufen in bestehenden Systemen
 - Beispiele: JavaScript (Web), Python, Lua

Klassifikation von Programmiersprachen

- Einteilung nach Paradigmen:
 - Paradigma = Art und Weise wie etwas programmiert wird
- Einteilung nach Abstraktionsniveau (Generation):
 - Generation 1-2: Niederen Programmiersprachen (Hardware nahe)
 - Generation 3-5: Höhere Programmiersprachen (problemorientiert)



Klassifikation von Programmiersprachen

Maschinencode (1. Gen)

8B 45 F8

8B 55 FC

01 D0

89 45 F4

Assembler (2. Gen)

```
mov eax, DWORD PTR [rbp-8]
```

```
mov edx, DWORD PTR [rbp-4]
```

```
add eax, edx
```

```
mov DWORD PTR [rbp-12], eax
```

Klassifikation von Programmiersprachen

- 3. Generation höheres Abstraktionsniveau
 - Verwendung von Variablen
 - Komplexe Datentypen
 - Kontrollstrukturen
 - Code-Recycling durch Funktionen

```
int c = a + b;
```



```
8B 45 F8  
8B 55 FC  
01 D0  
89 45 F4
```



Klassifikation von Programmiersprachen

- Programmierparadigma ist der fundamentale Programmierstil
 - Beschreibt das Modell, welches der Entwickler beim Programmieren im Kopf hat
 - Imperativ
 - Deklarativ
 - Funktional
 - Objektorientiert

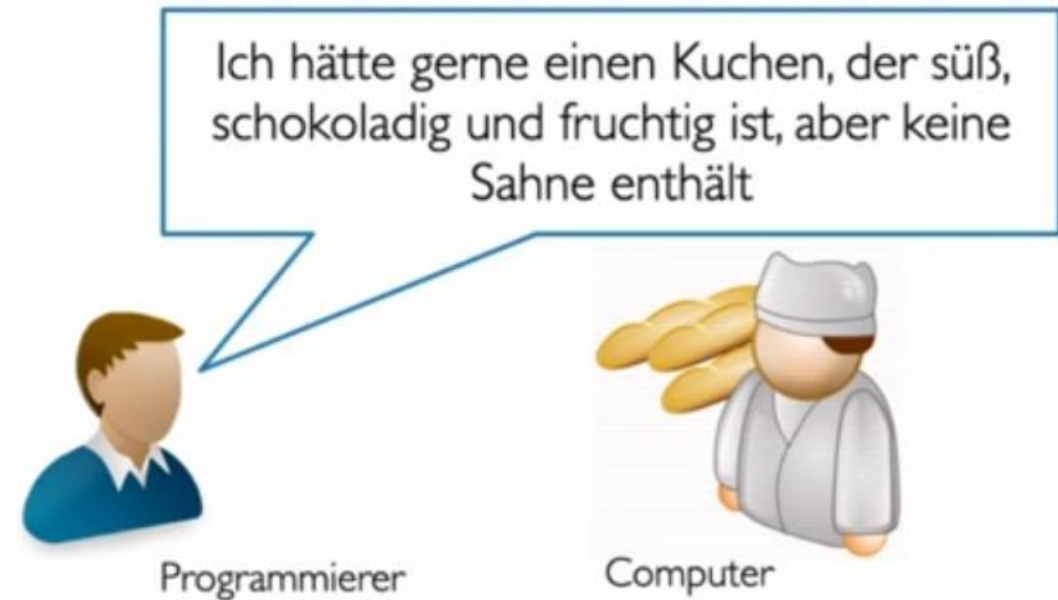
Klassifikation von Programmiersprachen

- Imperativ: Besteht aus Anweisungen, wie das Programm Ergebnisse erzeugt → Von unten nach oben
- Beispiel: C oder Python



Klassifikation von Programmiersprachen

- Deklarativ: Programm besteht aus Bedingungen, welche die Ausgabe des Programmes entsprechen muss (das Was)
- Bsp: SQL



Compiler & Interpreter

- Entwickler schreiben ein Programm als reinen Text, dem Quelltext
- Die CPU versteht nur Maschinencode
- Dafür gibt es zwei Möglichkeiten
 - Quelltext interpretieren
 - Quelltext kompilieren (übersetzen)



Interpreter

- Ein eigenes Programm, welches den Quelltext schrittweise interpretiert und ausführt
- Es wird nicht in Maschinencode übersetzt
- Somit braucht es den Interpreter, um das Programm auszuführen
- Beispiele: Python, PHP, JavaScript

Interpreter

- + Plattform-unabhängig
- + Programmiersprachen weniger strikt und sehr flexibel
- Programme sind langsamer
- Kein direkter Hardwarezugriff
- Keine Code-Optimierung

Compiler

```
int c = a + b;
```

Beispielcode: C



```
8B 45 F8  
8B 55 FC  
01 D0  
89 45 F4
```

Maschinencode

- Compiler übersetzen den Quelltext in Maschinensprache
- Nach der Kompilierung kann die Datei direkt gestartet werden
- Beispiel: C, C++

Compiler

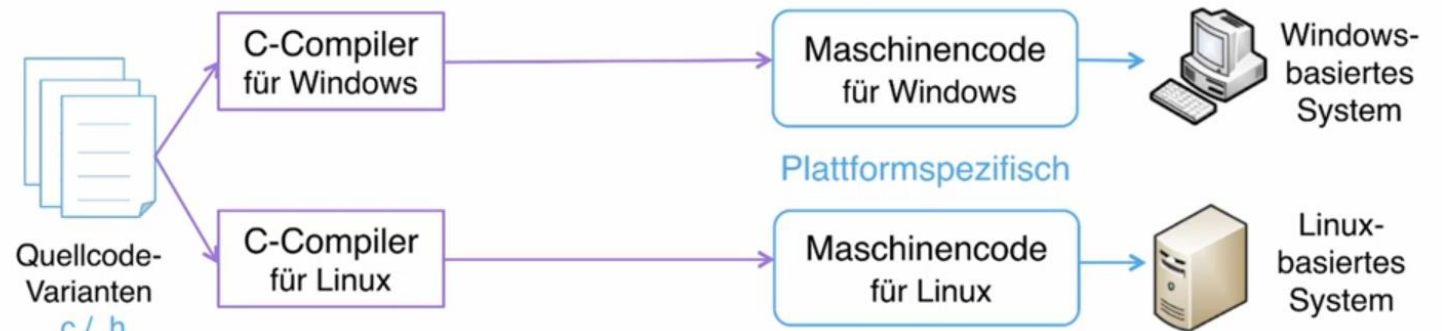
- + Effizienter, direktausführbarer Code
- + Nutzung von Hardware-Eigenheiten möglich
- + Code-Optimierung
- Programm nur auf Zielplattform lauffähig

Java

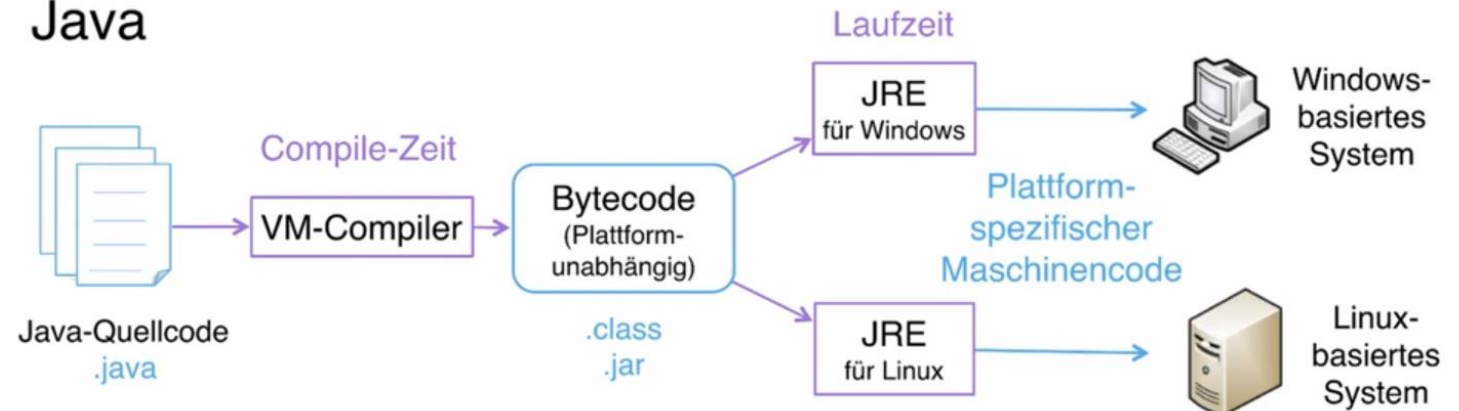
- Java wird kompiliert ist aber dennoch auf jeder Plattform ausführbar
- Java wird nicht direkt in Maschinencode übersetzt, sondern in Bytecode
- Der Java Bytecode in Plattformunabhängig
- Bytecode läuft auf der Java Runtime Environment (JRE)
- JRE muss für jede Plattform installiert werden

Compiler vs Java

C



Java



Java

- + Plattformunabhängigkeit
- + Bytecode kann vor Ausführung optimiert werden
- Benötigt JRE
- Zusätzlicher Interpretationsschritt kostet Laufzeit
- Reverse Engineering ist leichter



Python

Konsolenausgabe

- In Python reicht ein einzelner `print()` Befehl
- In den Klammern befindet sich die entsprechende Ausgabe

```
1 print("Hello World")
```


Variablen

- Speichern Werte ab
- Lassen sich über Operationen verändern
- Verschiedene Arten von Variablen: Zahlen (`int`, `float`), Texte (`String`), Wahrheitswerte (`Boolean`), Listen (`Arrays`)
- None ist ein nicht definierter Wert
- Bezeichner sollte beschreibend sein

```
1  x = 1
2  y = "Hello world"
3  z = True
4  l = []
5  |
```

Deklaration von Variablen

Variablen

- Zugriff auf listen mittels index operator `[]`
- Numerische Wert des index operators ergibt das entsprechende Element der Liste
- Informatiker fangen bei `0` an zu zählen
 - Erstes Element => `0`

```
1 list = ["Hallo", "Welt"]
2 print(list[0]) # Ausgabe: "Hallo"
3 |
```

Konsoleneingabe

- Mithilfe des `input()` Befehls kann eine Konsoleneingabe erfolgen

```
1  eingabe = input("Bitte gib etwas ein: ")
2  print("Du hast eingegeben: " + eingabe)
3
```

Boolesche Algebra

- Elementaraussagen können genau einem Wert zugeordnet sein
- wahr = $w = 1$
- falsch = $f = 0$


- " $2+2 = 5$ " falsch
- " $7 > 5$ " wahr



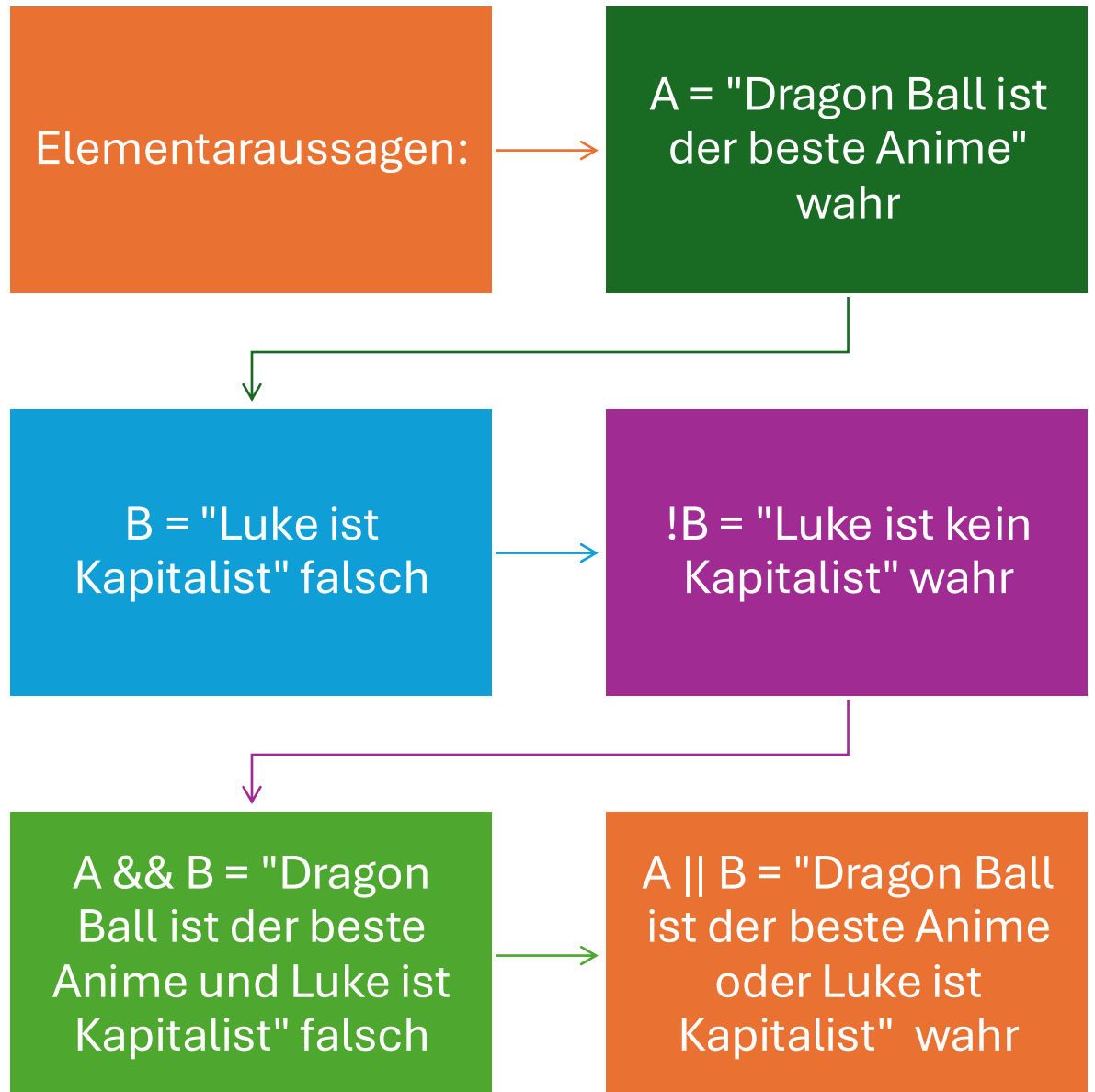


Zusammengesetzte Aussagen

- Aussagen lassen sich mit logischen Operatoren darstellen
 - && (und)
 - || (oder)
 - ! (nicht)

 - A && B
 - A || B
 - !A
- 

Zusammengesetzte Aussagen



Wahrheitstabellen

Und (Konjunktion)

A	B	$A \wedge B$
f	f	f
f	w	f
w	f	f
w	w	w

Oder (Disjunktion)

A	B	$A \vee B$
f	f	f
f	w	w
w	f	w
w	w	w

Nicht (Negation)

A	$\neg A$	$\neg \neg A$
w	f	w
f	w	f

Wahrheitstabellen

Äquivalenz


A	B	$A \leftrightarrow B$
f	f	w
f	w	f
w	f	f
w	w	w

Exklusives Oder (XOR)

A	B	$A \oplus B$
f	f	f
f	w	w
w	f	w
w	w	f

Vergleichsoperatoren

Operator	Bedeutung	Beispiel (a = 5 , b = 3)	Ergebnis
<code>==</code>	Gleichheit	<code>a == b</code>	<code>False</code>
<code>!=</code>	Ungleichheit	<code>a != b</code>	<code>True</code>
<code>></code>	Größer als	<code>a > b</code>	<code>True</code>
<code><</code>	Kleiner als	<code>a < b</code>	<code>False</code>
<code>>=</code>	Größer oder gleich	<code>a >= b</code>	<code>True</code>
<code><=</code>	Kleiner oder gleich	<code>a <= b</code>	<code>False</code>
<code>is</code>	Identität (gleiche Referenz)	<code>a is b</code>	<code>False</code>
<code>is not</code>	Keine Identität	<code>a is not b</code>	<code>True</code>
<code>in</code>	Mitgliedschaft in Sequenz	<code>'a' in 'abc'</code>	<code>True</code>
<code>not in</code>	Keine Mitgliedschaft	<code>'x' not in 'abc'</code>	<code>True</code>



Vergleichsoperationen

- Einfaches `=` ist Wertezuweisung
- Doppeltes `==` ist die Überprüfung auf Äquivalenz
- Listen können ebenfalls auf Gleichheit geprüft werden (Reihenfolge wird beachtet)

```
11  a = [1, 2]
12  b = [1, 2]
13  a == b      # True
```

Kontrollstruktur: `if-else`

- Immer dann wenn Code nur unter bestimmten vorraussetzungen ausgeführt werden soll, wird ein `if-else` verwendet
- Ergibt die Aussage im `if` statement `true` wird der entsprechend eingerückte Code ausgeführt

```
6   x = 1
7   if(x == 1):
8       print("Hello World")
9
```

Kontrollstruktur: `If-else`

- Ergibt die Aussage im `if` statement `false` wird der entsprechend eingerückte Code nicht ausgeführt

```
x = 2
if(x == 1):
    print("Hello World")
```

Kontrollstruktur: `If-else`

- Ergibt ein `if` Statement false und es befindet sich ein `else` dahinter wird der entsprechend eingerückte Code ausgeführt

```
6   x = 2
7   if(x == 1):
8       print("Hello World")
9   else:
10      print("Ich werde ausgeführt")
11
```

Kontrollstruktur: `If-else`

- Möchte man im `else` ebenfalls eine Überprüfung haben benutzt man `elif`

```
8   x = 2
9   if(x == 1):
10      print("Hello World")
11  elif(x == 2):
12      print("Ich werde ausgeführt")
13
```

Kontrollstruktur: `switch-case`

- Ähnlich wie bei `if-else` wird hier code bedingt ausgeführt
- `switch-case` prüft einen Wert und führt je nach Übereinstimmung mit einem Fall (case) einen bestimmten Codeblock aus
- `default(_)` wird ausgeführt, wenn kein case übereinstimmt

```
1  farbe = "blau"
2
3  match farbe:
4      case "rot":
5          print("Stop!")
6      case "gelb":
7          print("Achtung!")
8      case "grün":
9          print("Los!")
10     case _:
11         print("Unbekannte Farbe") # Default-Zweig
12
```

Kontrollstruktur: Schleifen

- Möchte man denselben Code wiederholen nutzt man dafür eine der verschiedenen **Wiederholschleifen**
- **while ()** bedingte Wiederholung
- **for** eine bestimmte Anzahl
- **for each** wiederholung für jedes Element (Listen)

Kontrollstruktur: `While`-Schleife

- Wird so lange ausgeführt, wie der logische Ausdruck `true` ergibt

```
1 word = input("Gebe ein Wort ein: ")
2 character = input("Gebe ein Buchstaben an: ")
3 index = 0
4
5 while(word[index] != character):
6     print(word[index])
7     index += 1
```

Kontrollstruktur: `While`-Schleife

- Möchte man frühzeitig abbrechen benutzt man ein `break`

```
1 word = input("Gebe ein Wort ein: ")
2 character = input("Gebe ein Buchstaben an: ")
3 index = 0
4
5 while(word[index] != character):
6     print(word[index])
7     index += 1
8     if(index >= len(word)):
9         print("Der Buchstabe " + character + " wurde nicht in " + word + " gefunden")
10        break
```

Kontrollstruktur: For-Schleife

- Zählschleife mit `iterator`
- Fängt ebenfalls bei 0 an

```
5  for i in range(10):  
6      print(i)
```

Ausgabe: 0 bis 9

```
1  word = input("Gebe ein Wort ein: ")  
2  character = input("Gebe ein Buchstaben an: ")  
3  index = 0  
4  
5  for i in range(len(word)):  
6      if word[i] == character:  
7          break  
8          print(word[i])  
9  
10 print("Der Buchstabe " + character + " konnte nicht in " + word + " gefunden werden")
```

Kontrollstruktur: For each-Schleife

- Wird vor allem bei Listen genutzt

```
1  names = ["Goku", "Guts", "Jotaro", "Luffy", "Naruto"]
2
3  for protagonist in names:
4      print(protagonist + " >", end=" ")
```

Dictionaries

- Schlüssel - Wert Paare
- Schlüssel bildet auf einen Wert ab
- Zugriff über `[]` operator

```
1  animeArotagonisten = {  
2      "Dragon Ball": "Son Goku",  
3      "One Piece": "Ruffy",  
4      "Berserk": "Guts"  
5  }  
6  
7  print(animeArotagonisten["Berserk"]) # Ausgabe Guts  
8
```

Dictionaries

Hinzufügen

```
1  animeArotagonisten = {
2      "Dragon Ball": "Son Goku",
3      "One Piece": "Ruffy",
4      "Berserk": "Guts"
5  }
6
7  animeArotagonisten["Bleach"] = "Ichigo"
8  print(animeArotagonisten["Bleach"]) # Ausgabe Ichigo
9  |
```

Iterieren

```
1  animeArotagonisten = {
2      "Dragon Ball": "Son Goku",
3      "One Piece": "Ruffy",
4      "Berserk": "Guts"
5  }
6
7  animeArotagonisten["Bleach"] = "Ichigo"
8
9  for key, value in animeArotagonisten.items():
10     print("Anime: " + key + " Protagonist: " + value)
11
```

Funktionen

- Vordefinierte Codeausschnitte
- Können Übergabeparameter erhalten
- Keyword: `def`

```
1  def square(number):  
2      squareNumber = number * number  
3      return squareNumber  
4  
5  numberInput = input("Quadratreehner: ")  
6  numberInput = int(numberInput)  
7  print(square(numberInput))  
8
```

Funktionen

- Funktionen müssen nichts zurückgeben

```
1  def ausgabe(ausgabe):  
2      print("Ich bin eine Zahlenausgabe: " + (ausgabe))  
3  
4  ausgabe("10")  
5
```


Bibliotheken und Packages

- Python kommt mit eingebauten Bibliotheken
- Mittels `import` kann man diese nutzen
- `math` – Mathematische Operationen und konstanten
- `random` – Zufallereignisse
- `datetime` – Uhrzeiten und Datum
- `os` – Betriebssystemfunktionen
- `sys` – Zugriff auf Systemfunktionen
- `json` – JSON Datenverarbeitung
- `re` – Reguläre Ausdrücke

Bibliothek: `math`

```
1  import math
2
3  print(math.pi) #3.141592653589793
4  print(math.sin(math.pi)) # fast 0
5  print(math.sqrt(16)) # 4
6  |
```

Bibliothek: `random`

```
1  import random
2
3  print(random.randint(1, 6))          # Zufallszahl zwischen 1 und 6
4  print(random.choice(["rot", "blau"])) # Zufälliges Element aus Liste
5
```

Bibliotheken: `datetime`, `os`, `sys`

```
1  import os, sys
2  from datetime import datetime
3
4  jetzt = datetime.now()
5  print(jetzt.strftime("%d.%m.%Y %H:%M")) # Formatiertes Datum
6
7  print(os.name)                        # 'posix' (Linux/Mac) oder 'nt' (Windows)
8  print(os.getcwd())                   # Aktuelles Arbeitsverzeichnis
9
10 print(sys.version)                    # Python-Version
11 sys.exit()                            # Skript beenden
12
```

Packages

- Packages sind Bibliotheken, welche erst installiert werden müssen
- Packagemanager: pip
- Ein **Package Manager** ist ein Programm, das dir hilft, **Software-Bibliotheken (Pakete)** zu finden, zu installieren, zu aktualisieren und zu verwalten.

```
> pip install numpy
```

Packages: `numpy`

- Einer der am häufigsten verwendeten Mathematik packages

```
1  import numpy as np
2
3  a = np.array([1, 2, 3])
4  b = np.array([4, 5, 6])
5
6  print(a + b)  # [5 7 9]
7  print(a * b)  # [4 10 18]
8  print(np.sqrt(a))  # [1.          1.41421356  1.73205081]
9  |
```

Packages: `Requirements.txt`

- In der `Requirements.txt` sind alle Abhängigkeiten mit Versionsnummer hinterlegt
- Mit `pip` ist es möglich direkt alle Abhängigkeiten aus der Datei zu installieren

```
requirements.txt  
1  numpy==1.26.4  
2  pandas>=2.1.0  
3  matplotlib  
4  requests<3.0
```

```
> pip install requirements.txt
```

Git

- Versionsverwaltung
- Erlaubt das gemeinsame Arbeiten am Code
- Cloudspeicher für Code
- Regeln für Zusammenarbeit können festgelegt werden
- Viele Provider bieten ein Wiki an
- Provider: GitHub, GitLab, GitBucket



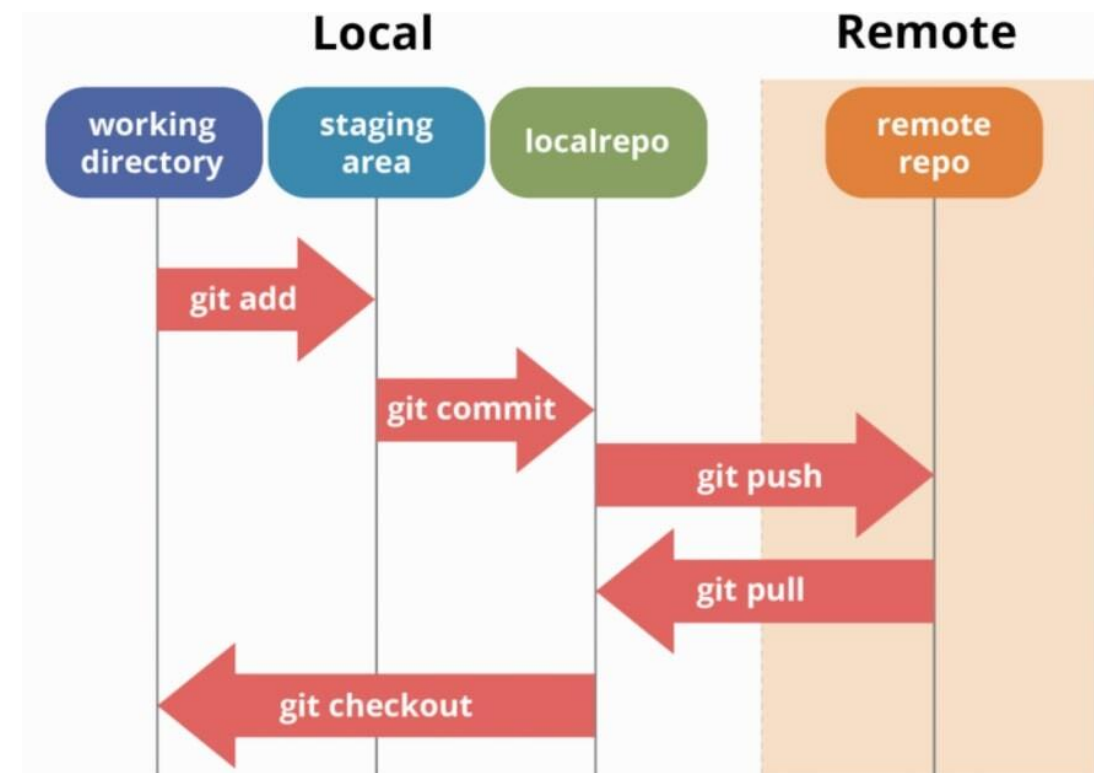
GitLab



GitBucket



- Steuerung über GUI oder CLI
- Durch **clone** holt man sich das Repo vom Server
- Durch **add** bringt man die lokalen Dateien in die Staging area
- Durch den **commit** wird es dann in das lokale Repository gepusht
- Durch **push** wird es auf dem git Server hochgeladen





```
1  git init                # Lokales Repo erstellen
2  git clone <url>         # Repo klonen
3
4  git status              # Zeigt den Status
5  git add <datei>         # Datei zur Staging-Area hinzufügen
6  git add .               # Alle Änderungen hinzufügen
7  git reset <datei>       # Aus Staging entfernen
8
9  git commit -m "Nachricht" # Änderungen committen
10 git commit -am "Nachricht" # Add + Commit (nur für bereits getrackte Dateien)
11
12 git remote -v           # Zeigt verbundene Remotes
13 git remote add origin <url> # Remote verbinden
14 git push -u origin main  # Hochladen (mit Tracking)
15 git push                # Änderungen hochladen
16 git pull                # Änderungen vom Remote holen
17 git fetch               # Änderungen abrufen (ohne Mergen)
18
```

Objektorientierung

- In der Objektorientierten Programmierung werden Klassen erstellt
- Klassen sind Baupläne für Objekte
- Klassen haben Attribute (Werte) und Methoden (Funktionen)
- Der Konstruktor der Klasse erzeugt das Objekt
- Konzepte wie Informationskapselung, Vererbung, Polymorphie, Separation of Concern und Dynamic Binding fördern sauberes, wartbares und wiederverwendbares Code-Design.
- Beispiele: Java, C#, C++, TypeScript, Python

Ausblick