

Grundkurs: Programmieren

Einführung in grundlegende Programmierkonzepte mit Python

Christoph Sonntag

WS 17/18

Universität Passau

Einführung in die Programmierung

- Erwartungen an den Kurs?
- Bereits Programmierkenntnisse aus Schule/Universität?
- Kursziele
 - grundlegendes Verständnis
 - “mit Informatikern reden können”
 - Angst nehmen

Die Programmiersprache Python

- Warum Python?
 - flache Lernkurve, sehenswerte Ergebnisse bereits nach dem ersten Tag
 - verankert in Forschung und Wirtschaft
 - der englischen Sprache sehr ähnlich

What's The Best Programming Language for First-Time Learners? (Poll Closed)

Java 17.63% (3,291 votes)



Ruby 8.39% (1,566 votes)



Python 34.16% (6,376 votes)



C/C++ 23.29% (4,347 votes)



JavaScript 16.53% (3,085 votes)



Total Votes: 18,665

Quelle: lifehacker.com

```
1 languages = ["C", "C++", "Java", "Python", "Fortran"]
2 modern_languages = \
3     list((x for x in languages if x is not "Fortran"))
```

Installieren von Python

- Python 3.6.3 unter <https://www.python.org/downloads/> herunterladen und ausführen
- Zum 'PATH' hinzufügen und '...for all users' deaktivieren



Hello World

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

- IDLE suchen und starten
- eintippen und mit Enter ausführen
- gibt den Text (String) 'Hello World!' auf der Konsole aus

Glückwunsch

Ihr habt gerade euer erstes Code-Fragment geschrieben und ausgeführt!

Aufgabe 1

Erweitere das Programm so, dass der String 'Hello World' 6-mal auf der Konsole ausgegeben wird.

Aufgabe 1

Erweitere das Programm so, dass der String 'Hello World' 6-mal auf der Konsole ausgegeben wird.

Lösung

```
1 print("Hello World")
2 print("Hello World")
3 print("Hello World")
4 print("Hello World")
5 print("Hello World")
6 print("Hello World")
```

Hello World

- `sleep(num)`
 - stoppt die Ausführung des Programms für `num` Sekunden
 - ist nicht Teil des Standard-Pythons, sondern muss importiert werden. `from time import sleep`

Hello World

- `sleep(num)`
 - stoppt die Ausführung des Programms für `num` Sekunden
 - ist nicht Teil des Standard-Pythons, sondern muss importiert werden. `from time import sleep`

Aufgabe 2

Lass das Programm einen realistischen Monolog ausführen.

Verwende hierzu `sleep`

Hello World

- `sleep(num)`
 - stoppt die Ausführung des Programms für `num` Sekunden
 - ist nicht Teil des Standard-Pythons, sondern muss importiert werden. `from time import sleep`

Aufgabe 2

Lass das Programm einen realistischen Monolog ausführen.

Verwende hierzu `sleep`

Lösung

```
1 from time import sleep
2
3 print("Hey, it's James!")
4 sleep(2)
5 print("I'm working on a ChatBot right now")
6 sleep(3)
7 ...
```

Entwicklungsumgebung einrichten

Achtung

Word, TextEdit, Notepad, oder Wordpad sind Textverarbeitungsprogramme, keine Quelltext-Editoren und schon gar keine Entwicklungsumgebungen

Achtung

Word, TextEdit, Notepad, oder Wordpad sind Textverarbeitungsprogramme, keine Quelltext-Editoren und schon gar keine Entwicklungsumgebungen

- Editoren wie Sublime Text, Atom oder IDLE sind für uns ausreichend
- große IDE's wie Eclipse, IntelliJ oder PyCharm bieten weitere Funktionen

- Pythonprogramme in IDLE schreiben und ausführen
 1. Datei > Neue Datei
 2. geeigneten Speicherort aussuchen, bspws.
Dokumente/GrundkursProgrammieren/helloworld.py
 3. Programm schreiben...
 4. Programm unter Run > Run Module ausführen oder F5 drücken

Wir haben ganz unbewusst bereits zwei Datentypen benutzt

- String, str:

```
1 print("Ich bin vom Typ String, eine Reihe von  
    Zeichen")
```

- Integer, int:

```
1 sleep(3)    # 3 ist ein Integer
```


Typen und Variablen

- Wahrheitswert (Boolean, bool):

```
1 wahr = True  
2 falsch = False
```

- Gleitkommazahlen (Float, float):

```
1 pi = 3.1415
```

- In Python gibt es sogar komplexe Zahlen complex:

```
1 a = complex('3+5j')
```

Typen und Variablen

- Zuweisung von Variablen mit dem Zuweisungsoperator =

```
1 a = 5
2 b = 3.14
3 c = "Hallo Grundkurs:Programmieren"
```

- der Variable kann auch das Ergebnis einer Operation zugewiesen werden

```
1 a = 1000
2 b = 200
3 percent = b / a * 100
```

- mehrfache Zuweisung

```
1 a, b, c = 5, 3.14, "Hallo Grundkurs:Programmieren"
2 x = y = z = 42
```

- streng-getypte Sprachen: **Java**, C/C++, ...

```
1  int x = 42
2  float pi = 3.14
3  String greeting = "Hallo Grundkurs:Programmieren"
4  boolean truth = true
```

- dynamisch (schwach) getypte Sprachen: **Python**, Javascript

```
1  x = 42
2  pi = 3.14
3  greeting = "Hey there"
4  truth = True
```

Aufgabe

Gib folgende Ausdrücke in den Python Interpreter ein:

```
1 >>> 3 + 3.14
2 >>> "Mein Alter: " + 5
3 >>> True + 1
```

Aufgabe

Gib folgende Ausdrücke in den Python Interpreter ein:

```
1 >>> 3 + 3.14
2 >>> "Mein Alter: " + 5
3 >>> True + 1
```

True + 1 == 2?

Intern werden die Keywords `True` und `False` auf die Werte 1 und 0 vom Typ `int` abgebildet.

Die `input()` Funktion nimmt Benutzereingaben auf der Kommandozeile entgegen und gibt sie zurück. Du brauchst kein zusätzliches Modul importieren.

Aufgabe

Lasse dich von deinem Programm begrüßen, indem du mit `input` deine Eingabe in einer Variable speicherst.

Typen und Variablen

Die `input()` Funktion nimmt Benutzereingaben auf der Kommandozeile entgegen und gibt sie zurück. Du brauchst kein zusätzliches Modul importieren.

Aufgabe

Lasse dich von deinem Programm begrüßen, indem du mit `input` deine Eingabe in einer Variable speicherst.

Lösung

```
1 >>> name = input()
2 'Christoph'
3 >>> print("Hallo " + name)
```

Achtung

Die `input()` Funktion interpretiert jede Benutzereingabe als `String`. Wenn man Zahlen aufnehmen will, muss der Typ 'gecastet' werden, d.h. 'umgewandelt'.

- `int()`: Castet zu `int`.
- `str()`: Castet zu `String`.

Was passiert bei `int("HalloWelt")`?

Achtung

Die `input()` Funktion interpretiert jede Benutzereingabe als `String`. Wenn man Zahlen aufnehmen will, muss der Typ 'gecastet' werden, d.h. 'umgewandelt'.

- `int()`: Castet zu `int`.
- `str()`: Castet zu `String`.

Was passiert bei `int("HalloWelt")`? Ein `ValueError` wird geworfen. (Exception handling nicht in diesem Kurs)

- Unterscheidungsmerkmale
 - Programmierparadigma: imperativ, funktional oder objektorientiert
 - Typsicherheit
 - kompiliert vs. interpretiert
 - allgemein vs. domänenspezifisch
 - hardwarenah vs. höhere Programmiersprachen

- Imperative Programmiersprachen: C/C++, C#, Java ...
- Funktionale Programmiersprachen: SQL, Haskell, Erlang, (Scala) ...
- Objektorientierte Programmiersprachen: C++, C#, Java, Javascript, PHP, Python ...

Imperative Sprachen (C/C++, C#, Python, Java, ...)

- ältestes Programmierparadigma
- große Verbreitung in der Industrie
- besteht aus Befehlen (lat. imperare = befehlen)
- Abarbeiten der Befehle 'Schritt für Schritt'
- sagt einem Computer, 'wie' er etwas tun soll

```
1 print("Hey, whats' up?")
2 sleep(3)
3 print("Learning Python right now")
4 sleep(2)
```

- Verwendung
 - 'Standard-Software', hardwarenahe Entwicklung

Funktionale Sprachen (Haskell, Erlang, SQL, Lisp, ...)

- vergleichsweise modern
- sagt einem Computer, 'was' das Ergebnis sein soll
- `SELECT name FROM students WHERE major='law' AND semester='1';`
- Verwendung
 - akademische Zwecke
 - sicherheitskritische und ...
 - hoch performante Anwendungen

```
1 square :: [Int] -> [Int]
2 square a = [2*x | x <- a]
```

$$x = x + 1$$

Objektorientierte Sprachen (Java, Python, C++, C#, ...)

- starke Verbreitung
- Abbilden der realen Welt der Dinge auf Objekte
- Klasse: Bauplan eines Objekts bestehend aus Eigenschaften (Attributen) und Methoden
- Vererbung möglich
- Verwendung
 - Standard-Software
 - Modellierung realer Projekte (Unternehmen, Mitarbeiter, Kunden, Waren, ...)
 - große Projekte (→ Planung durch Klassendiagramme)

Objektorientierung: Beispiel

```
1 class Konto:
2     def __init__(self, name, nr):
3         self.inhaber = name
4         self.kontonummer = nr
5         self.kontostand = 0
6     def einzahlen(self, betrag):
7         self.kontostand = kontostand + betrag
8     def auszahlen(self, betrag):
9         self.kontostand = kontostand - betrag
10    def ueberweisen(self, ziel, betrag):
11        ziel.einzahlen(self.betrag)
12        self.auszahlen(betrag)
13    def kontostand(self):
14        return self.kontostand
15
16 class Unternehmenskonto(Konto):
17     def erhalteBonus(self, bonus):
18         self.kontostand = kontostand + bonus
```


Kompilierte und Interpretierte Sprachen

- kompilierte Sprachen (Java, C/C++, C#, ...):
 - Übersetzung des (kompletten) Programmcodes in Maschinencode
 - dann Ausführung des Maschinencodes
- interpretierte Sprachen (Python, Lisp, PHP, JavaScript, ...):
 - Übersetzung einer einzelnen Programmanweisung
 - Ausführung dieser Anweisung
 - Übersetzung der nächsten Anweisung

Hardwarenahe und höhere Sprachen

- hardwarenah: abhängig von der Bauweise des Prozessors
- höhere Sprachen: von der Bauweise abstrahiert (print(), sleep())

```
1 .START ST
2   ST: MOV R1,#2
3       MOV R2,#1
4   M1: CMP R2,#20
5       BGT M2
6       MUL R1,R2
7       INI R2
8       JMP M1
9   M2: JSR PRINT
10      .END
```

```
1
2 a = 2;
3 i = 1;
4 # compare i ==
   20
5 # if True, jump
   to M2
6 a = a*i;
7 i++;
8 # jump to M1
9 print(a)
```

```
1 a = 2;
2 for i in range
   (1, 20) {
3
4     a = a*i;
5
6 }
7
8 print(a);
9
```

Populäre Programmiersprachen

- C++
 - imperativ, objektorientiert, typsicher, kompiliert, allgemein, höhere Sprache (dennoch hardwarenah)
 - große Verwendung in hocheffizienten Systemen (Betriebssysteme, Grafikberechnungen, Computerspiele, . . .)
 - Erweiterung von C mit Objektorientierung
- Java
 - imperativ, objektorientiert, typsicher, kompiliert, allgemein
 - im bayrischen Lehrplan und an vielen Universitäten 'erste' Sprache
 - ebenfalls große Verbreitung
- Python
 - (imperativ), (funktional), objektorientiert, dynamisch getypt, interpretiert, allgemein
 - große Verbreitung auch gerade im akademischen Umfeld, Web,

Machine Learning und Data Science

Exkurs: Kommentierung, Lesbarkeit und Wartbarkeit

```
1  """
2  Das ist ein Kommentar, der sich ueber mehrere Zeilen
3  erstreckt und in der Regel in ganzen Saetzen spricht.
4  Das Ziel ist etwas ausfuehrlich zu erklaren.
5  """
6  print("Wozu Kommentare?") # stellt eine Frage
```

- Zeilen, die mit '# ' beginnen, sind einzeilige Kommentare
- mehrzeilige Kommentare häufig bei Klassen (→ automatische Erzeugung von Dokumentationen)
- werden nicht ausgeführt
- zur Erläuterung von Programmcode
- Aufwendig, aber sehr wichtig! (Lesbarkeit, Wartbarkeit)

Exkurs: Kommentierung, Lesbarkeit und Wartbarkeit

```
1 # set the value of the age to an integer with the  
   value 32  
2 age = 32
```

```
1 # set the value of the age to an integer with the  
   value 32  
2 age = 32
```

Schlechtes Beispiel

Unnötige Erklärung einer offensichtlichen Sache

Exkurs: Kommentierung, Lesbarkeit und Wartbarkeit

```
1 def addSetEntry(set, value) {  
2     """  
3     Don't return 'set.add' because it's not chainable  
4         in Internet Explorer 11.  
5     """  
6     set.add(value)  
7     return set  
8 }
```

Exkurs: Kommentierung, Lesbarkeit und Wartbarkeit

```
1 def addSetEntry(set, value) {  
2     """  
3     Don't return 'set.add' because it's not chainable  
4         in Internet Explorer 11.  
5     """  
6     set.add(value)  
7     return set  
8 }
```

```
1 """  
2 This code sucks, you know it and I know it.  
3 Move on and call me an idiot later.  
4 """
```


Exkurs: Kommentierung, Lesbarkeit und Wartbarkeit

```
1 def addSetEntry(set, value) {  
2     """  
3     Don't return 'set.add' because it's not chainable  
4         in Internet Explorer 11.  
5     """  
6     set.add(value)  
7     return set  
}
```

```
1 """  
2 This code sucks, you know it and I know it.  
3 Move on and call me an idiot later.  
4 """
```

```
1 # Class used to workaround Richard being a f***ing  
    idiot
```

Datenstrukturen: Listen

Listen sind praktische Datenstrukturen, um eine Folge von Werten zu speichern oder zu erzeugen. Oft reichen Integer, Float und String Datentypen nicht aus. Meist wissen wir nämlich im Voraus nicht, wie viele Datensätze gespeichert werden sollen.

Datenstrukturen: Listen

Listen sind praktische Datenstrukturen, um eine Folge von Werten zu speichern oder zu erzeugen. Oft reichen Integer, Float und String Datentypen nicht aus. Meist wissen wir nämlich im Voraus nicht, wie viele Datensätze gespeichert werden sollen.

```
1 zahlen = [1, 2, 3, 4, 5, 6]
2 texte = ["Hallo", "Welt", ".", ["Grundkurs", "
    Programmieren"]]
```

```
1 >>> zahlen[3]
2 4
3 >>> texte[0]
4 "Hallo"
5 >>> texte[3]
6 ["Grundkurs", "Programmieren"]
```

Datenstrukturen: Listen

Die Liste bietet eine große Anzahl an Methoden (Funktionen), die auf ihnen ausgeführt werden können.

Aufgabe

```
1 >>> liste = ["Grundkurs", "Programmieren", 42, "Pie",  
2           3.14]  
3 >>> liste[2] = 99  
4 >>> len(liste)  
5 >>> liste.append("Passau")  
6 >>> liste.extend([4, 5, 3.14])  
7 >>> liste.insert(2, "Falke")  
8 >>> liste.count(3.14)  
9 >>> liste.index(3.14)  
10 >>> liste.remove(3.14)  
11 >>> liste.pop()  
12 >>> liste.reverse()  
13 >>> sum([1,3,5])  
14 >>> max([1,3,5])
```

Aufgabe

Versuche zu erraten, was die Ausgabe dieses Programms ist.

```
1 liste_a = ['Hallo', 'schoenes', 'Wetter']
2 liste_b = liste_a
3
4 liste_b[1] = 'schlechtes'
5
6 print(liste_a[0], liste_a[1], liste_a[2])
```

Aufgabe

Versuche zu erraten, was die Ausgabe dieses Programms ist.

```
1 liste_a = ['Hallo', 'schoenes', 'Wetter']
2 liste_b = liste_a
3
4 liste_b[1] = 'schlechtes'
5
6 print(liste_a[0], liste_a[1], liste_a[2])
```

Lösung

Hallo schlechtes Wetter

Aufgabe: Notendurchschnitt

Schreibe ein Programm, dass beliebig oft Prüfungs-Noten einliest, in einer Liste speichert und dir nach jeder Eingabe den Durchschnitt errechnet. *Hinweis:* Verwende eine while True: Schleife, um beliebig oft Noten einzulesen.

Aufgabe: Notendurchschnitt

Schreibe ein Programm, dass beliebig oft Prüfungs-Noten einliest, in einer Liste speichert und dir nach jeder Eingabe den Durchschnitt errechnet. *Hinweis:* Verwende eine while True: Schleife, um beliebig oft Noten einzulesen.

Lösung
siehe Beamer

- der Code wird durch Einrückungen (Tabs) strukturiert (vgl. for-Schleife)
- runde Klammern () sind meist für Parameter
(`print("Hier der Text")`)
- eckige Klammern [] sind meist für 'listenartige' Datenstrukturen (Arrays, Listen in Python)
- Leerzeichen und Zeilenumbrüche sind meist optional, verbessern aber die Lesbarkeit des Programms
- Groß- und Kleinschreibung muss meist beachtet werden

Konventionen für lesbareren Code

Damit Code einheitlich gut lesbar ist, gibt es für Programmiersprachen 'Coding Conventions', die zwar nicht erfüllt werden *müssen*, aber einen guten Eindruck hinterlassen und zur besseren Lesbarkeit beitragen.

Auszug (*PEP8*):

- Variablen- und Funktionsnamen klein und wenn nötig mit `_` schreiben
- Eine Einrückungsebene in Python entspricht genau 4 Leerzeichen (keine Tabulatorzeichen)
- Am Anfang jeder Python-Datei steht ein Doc-String (Kommentar), der kurz den Inhalt der Datei beschreibt
- ... bei Funktionen auch

Aufgabe

Schreibe ein Programm, dass von 1 bis 100 zählt.

```
1 print("Ich kann uebrigens bis 100 zaehlen ... ;-)")  
2 ...
```

Die for-Schleife

Aufgabe

Schreibe ein Programm, dass von 1 bis 100 zählt.

```
1 print("Ich kann uebrigens bis 100 zaehlen ... ;-)")  
2 ...
```

Lösungsvariante 1

```
1 print(1)  
2 print(2)  
3 print(3)  
4 print(4)  
5 print(5)  
6 print(6)  
7 print(7)  
8 ...
```

Vereinfachte weniger schreibintensive Variante:

Lösungsvariante 2

```
1 for i in range(1, 100):  
2     print(i)
```

- gibt die Werte von 1 bis 99 aus

Die for-Schleife

Vereinfachte weniger schreibintensive Variante:

Lösungsvariante 2

```
1 for i in range(1, 100):  
2     print(i)
```

- gibt die Werte von 1 bis 99 aus
- gib in den Interpreter `help()` ein, um genaueres über eine Funktion zu erfahren.

Die for-Schleife

Vereinfachte weniger schreibintensive Variante:

Lösungsvariante 2

```
1 for i in range(1, 100):  
2     print(i)
```

- gibt die Werte von 1 bis 99 aus
- gib in den Interpreter `help()` ein, um genaueres über eine Funktion zu erfahren.
- aus der Dokumentation: `range(i, j)` produces `i`, `i+1`, `i+2`, ..., `j-1`.

Die for-Schleife

- Andere Sprachen erzeugen kein 'extra Objekt', sondern zählen eine Variable hoch
- genauer: setzt $i = 1$, addiert in jedem Schritt 1 auf, solange $i < 100$ ist.

```
1  for(int i=1, i < 100, i++) {  
2      // gib i aus  
3  }
```


Aufgabe

Ändere das Programm von gerade eben so, dass wirklich bis 100 gezählt wird.

Aufgabe

Ändere das Programm von gerade eben so, dass wirklich bis 100 gezählt wird.

Lösung

```
1 for i in range(1, 101):  
2     print(i)
```

Aufgabe

Bildet 2-3er Gruppen und schreibt ein Programm, dass von 10 bis 0 herunterzählt. *Hinweis:* Verwendet zum pausieren zwischen den Zahlen wieder `sleep()`. `from time import sleep` nicht vergessen.

Aufgabe

Bildet 2-3er Gruppen und schreibt ein Programm, dass von 10 bis 0 herunterzählt. *Hinweis:* Verwendet zum pausieren zwischen den Zahlen wieder `sleep()`. `from time import sleep` nicht vergessen.

Lösung

siehe Beamer

Gültigkeit von Variablen - der Scope

Aufgabe

Schreibe ein Programm, dass sich dem Benutzer vorstellt und anschließend nach Namen und Alter fragt. Verwende hierzu die Funktionen `input()` und `print()`. (Wenn du willst auch `sleep()`).

Anfang

```
1 name = "James"
2 print("Hey, i'm " + name + " and what's you're name?")
3 ...
```

Gültigkeit von Variablen - der Scope

Aufgabe

Schreibe ein Programm, dass sich dem Benutzer vorstellt und anschließend nach Namen und Alter fragt. Verwende hierzu die Funktionen `input()` und `print()`. (Wenn du willst auch `sleep()`).

Anfang

```
1 name = "James"
2 print("Hey, i'm " + name + " and what's you're name?")
3 ...
```

mögliche Lösung

```
1 ...
2 name = input() # ueberschreibt die vorige Variable
3 sleep(2)
4 print("Ahh, so you're " + name)
5 ...
```

Den Zuweisungsoperator = haben wir unbewusst bereits kennengelernt. Er weist einem Wert bspws. einen Namen zu (`x = 42`). Die Operatoren `+`, `-`, `/` oder `*` verwenden wir intuitiv. Zum besseren Vergleichen brauchen wir weitere Operatoren:

- `==`: prüft zwei Werte auf Gleichheit
- `!=`: prüft zwei Werte auf Ungleichheit
- `>`: größer
- `<`: kleiner (vgl. for-Schleife)
- kleiner-gleich?

Den Zuweisungsoperator = haben wir unbewusst bereits kennengelernt. Er weist einem Wert bspws. einen Namen zu (`x = 42`). Die Operatoren `+`, `-`, `/` oder `*` verwenden wir intuitiv. Zum besseren Vergleichen brauchen wir weitere Operatoren:

- `==`: prüft zwei Werte auf Gleichheit
- `!=`: prüft zwei Werte auf Ungleichheit
- `>`: größer
- `<`: kleiner (vgl. for-Schleife)
- kleiner-gleich?
- `<=`, `>=`, kleiner-gleich, größer-gleich
- `and`: logisches 'Und'
- `or`: logisches 'Oder'
- `not`: verneint einen Ausdruck

logische Operatoren

Was ergeben folgende Ausdrücke? Überprüfe sie mit dem Python Interpreter.

1 `3 > 4`

1 `6 != 7`

1 `"Hallo" < "Hallo Welt!"`

1 `"Hallo" == "Hallo Welt"`

1 `"Hallo Welt" == "Hallo" and 3 > 4`

1 `not not 5 == 5`

Aufgabe

Schreibe ein Programm, welches vom Benutzer 6 Zahlen einliest, diese in einer Liste speichert und anschließend das Maximum ausgibt.

Lösung

```
1 numbers = []
2 for i in range(0,7):
3     input_num = int(input())
4     numbers.append(input_num)
5
6 maximum = max(numbers)
7 print("Maximum: " + str(maximum))
8 # print("Maximum: \"%d\" \"% maximum")
```

Bedingte Ausführung

Hilft uns beim strukturieren des Programms in verschiedene Richtungen. Struktur:

```
1  if Bedingung:
2      # Programmcode
3  else:
4      # alternativer Programmcode, wenn Bedingung nicht
        zutrifft
```

Bedingte Ausführung

Hilft uns beim strukturieren des Programms in verschiedene Richtungen. Struktur:

```
1  if Bedingung:
2      # Programmcode
3  else:
4      # alternativer Programmcode, wenn Bedingung nicht
      zutrifft
```

```
1  zahl = input()
2
3  if zahl > 10:
4      print("Die Zahl ist > 10.")
5  else:
6      print("Die Zahl ist < 10.")
```

Aufgabe

Schreibe ein Programm, das zwei Zahlen vom Benutzer einliest, testet, ob die erste durch die zweite Zahl teilbar ist und das Ergebnis ausgibt.

Aufgabe

Schreibe ein Programm, das zwei Zahlen vom Benutzer einliest, testet, ob die erste durch die zweite Zahl teilbar ist und das Ergebnis ausgibt.

```
1 zahl1 = int(input("Erste Zahl: "))
2 zahl2 = int(input("Zweite Zahl: "))
3
4 if zahl1 % zahl2 == 0:
5     print("Zahl 1 ist durch die Zweite teilbar!")
6 else:
7     print("Leider kann man die Zahlen nicht ohne Rest
      teilen")
```

Mehrere Bedingungen?

Mehrere Bedingungen?

```
1 zahl = input()
2
3 if zahl > 10:
4     print("Die Zahl ist > 10.")
5 elif zahl > 100:
6     print("Die Zahl ist > 100.")
7 else:
8     print("Die Zahl ist < 10.")
```

Aufgabe

Schreibe ein Programm, dass ein eingegebenes Hundealter in Menschenjahre umrechnet.

- 1 Hundejahr = 14 Jahre
- 2 Hundejahre = 22 Jahre
- Über 2 Jahren = $22 + (\text{Alter} - 2) * 5$

Lösung

```
1 age = int(input("Alter des Hundes: "))
2 if age < 0:
3     print("Das stimmt wohl kaum!")
4 elif age == 1:
5     print("entspricht ca. 14 Jahren")
6 elif age == 2:
7     print("entspricht ca. 22 Jahren")
8 elif age > 2:
9     human = 22 + (age - 2)*5
10    print("entspricht ca. " + str(human) + "Jahren")
```

Aufgabe

Schreibe ein Programm, das errechnet, ob ein eingegebenes Jahr ein Schaltjahr ist.

- durch 400 teilbar: ist ein Schaltjahr
- durch 100 teilbar: ist kein Schaltjahr
- falls durch 4 teilbar: ist ein Schaltjahr

Aufgabe

Schreibe ein Programm, das errechnet, ob ein eingegebenes Jahr ein Schaltjahr ist.

- durch 400 teilbar: ist ein Schaltjahr
- durch 100 teilbar: ist kein Schaltjahr
- falls durch 4 teilbar: ist ein Schaltjahr

gemeinsame Lösung

siehe Beamer

Die while-Schleife

Ähnlich zur for-Schleife mit dem Unterschied, dass nicht 'durch eine Liste gelaufen' wird, sondern die Schleife immer wiederholt wird, solange die Bedingung wahr ist.

```
1 while Bedingung:
2     # Programmcode
3
4 # ausserhalb der Schleife
```

Die while-Schleife

Aufgabe

Schreibe ein Programm, welches alle Zahlen von 1 bis 100 auf den Bildschirm schreibt, ohne dafür eine for Schleife zu verwenden.

Aufgabe

Die Folge aus Fibonacci-Zahlen wird wie folgt gebildet:

- Das erste und das zweite Element sind 0 und 1.
- Jedes folgende Element wird gebildet, in dem die letzten zwei Elemente addiert werden.

Das heisst, die Folge sieht so aus: 0,1,1,2,3,5,8,13,21,34,...

Schreibe ein Programm, welches die Fibonacci-Zahlen bis zu einer vom Benutzer gewählten Zahl ausgibt.

Lösung 1

```
1 num = 10
2 # num = int(input("Bis zu welcher Zahl? "))
3 fib1 = 0
4 fib2 = 1
5 count = 0
6
7 if num <= 0:
8     print("Bitte eine positive Zahl eingeben")
9 elif num == 1:
10    print("Fibonacci bis zur " + num + "ten Zahl:")
11    print(fib1)
12 else:
13    print("Fibonacci bis zur " + num + "ten Zahl:")
14    while count < num:
15        print(fib1)
16        fibn = fib1 + fib2
17        fib1 = fib2
18        fib2 = fibn
19        count = count + 1
20    # count += 1
```


Lösung 2

```
1 def fibonacci(n):  
2     a = 0  
3     b = 1  
4     for i in range(0, n):  
5         temp = a  
6         a = b  
7         b = temp + b  
8     return a
```

Aufgabe: Das kleine 1×1

Schreibe ein Programm, das den Benutzer 'zufällige' 1×1 Rechenaufgaben abfragt und die Lösung überprüft.

(`from random import randint`). Nach jeder Rechenaufgabe soll der Benutzer gefragt werden, ob noch eine Aufgabe gestellt werden soll.

Die while-Schleife

Lösung

```
1 from random import randint
2
3 print("Herzlich willkommen zum kleinen EinmalEins
  Rechner!")
4 weiter = "Ja"
5
6 while weiter == "Ja":
7     zahl1 = randint(1, 9)
8     zahl2 = randint(1, 9)
9     print(str(zahl1) + " x " + str(zahl2))
10
11     loesung = int(input("= "))
12     if loesung == zahl1 * zahl2:
13         print("Korrekt! Glueckwunsch")
14     else:
15         print("Leider nicht ganz korrekt. Richtig
          waere: " + str(zahl1 * zahl2))
16     weiter = input("Weiter? (Ja/Nein): ")
17
18 print("Vielen Dank!")
```

- Programme in der Regel wesentlich länger als 100 Zeilen

- Programme in der Regel wesentlich länger als 100 Zeilen
- Metrik zur Aufwandsberechnung z.B. Lines of Code. (LOC, SLOC)
 - Windows XP: ca. 40 Mio. SLOC
 - Mac OS X Tiger: ca. 86 Mio. SLOC

- Programme in der Regel wesentlich länger als 100 Zeilen
- Metrik zur Aufwandsberechnung z.B. Lines of Code. (LOC, SLOC)
 - Windows XP: ca. 40 Mio. SLOC
 - Mac OS X Tiger: ca. 86 Mio. SLOC
- nicht zwangsläufig Rückschlüsse auf die Komplexität oder Funktionalität

Je größer ein Projekt wird, desto wichtiger ist es den Überblick zu behalten und evtl. Programmblöcke, die etwas ähnliches machen zusammen zu fassen.

```
1 def greet():  
2     print("Hey!")  
3     print("How are you?")
```

Funktionen mit Parametern

Wie die uns bereits bekannten Funktionen `print()` und `sleep()` können auch eigene Funktionen Parameter aufnehmen und zurückgeben.

```
1 def sum(a, b):  
2     return a + b
```

Aufgabe

Schreibe eine Funktion, die eine Liste als Parameter nimmt und das Maximum zurückgibt.

- Universität Passau: 'Programmierung I' (5102) an der FIM
- Automate the Boring Stuff with Python: Practical Programming for Total Beginner (Sweigart, 2015)
- 'How to think like a Computer Scientist' (Wentworth, Peter and Elkner, Jeffrey and Downey, Allen B and Meyers, Chris, 2011)

- Danke für die Teilnahme! Informationen zu weiteren Kursen im jeweiligen Semester beim ZKK
- `www.evaluation.uni-passau.de` (Unter Umständen muss noch das Zertifikat heruntergeladen werden)
- Wintersemester 2017/18 > ZKK IT-Kurse > Token eingeben