WORKSHOP-TERMIN 1

Drohnenprogrammierung und Automatisiertes Fliegen

12.04.2022

Organisatorisches I

- 12.04.: VM & Python Basics
- 10.05.: ROS & ROS Python
- 14.06.: Crazyswarm & Drohnenlabor
- 12.07.: Automatisiertes Fliegen

Organisatorisches II

- Patrik Golec
  - patrik.golec@fh-kufstein.ac.at
- wissenschaftlicher Mitarbeiter (seit Nov 2020)
  - Masterstudent WCIS
  - Webentwickler
- Zuständigkeit
  - Drohnenlabor (seit Nov 2021)
- Projekt DIH West
  - Schulungsreihe im Rahmen des Projekts

Agenda

- Kurze Vorstellungsrunde
- Überblick virtuelle Maschine
  - Probleme bei Einrichtung?
- Python-Einführung
  - bis 10:45
- PAUSE
  - 10:45 - 11:00
- Python-Einführung
  - bis 12:00
- Mögliche VM Probleme beheben
  - bis 13:00

Kurze Vorstellung

- Name, Firma, beruflicher Hintergrund
- bestehende Kenntnisse
- Erwartungen

Ziele

- VirtualBox
  - funktionierende VM für diesen und weitere Workshops

- Überblick Programmierkonzepte in Python
  - grundl. Verständnis Voraussetzung für ROS Python
  - Programmierbeispiele empfohlen!
    - https://github.com/patrik98/fhk-drohnenworkshop

Programmierparadigmen

Imperative Programmierung

Prozedurale Programmierung
- klassisch: C, Pascal, COBOL
- auch: Python, PHP, Perl, JavaScript

Objektorientierte Programmierung
- klassisch: JAVA, C++, Python, C#, PHP, JavaScript, Ruby, Swift
- auch: Lisp, Scala

Deklarative Programmierung

Funktionale Programmierung
- klassisch: Lisp, Haskell, Scala
- auch: Python, JavaScript

Logische Programmierung
- Prolog

Grundlagen Python I

- Python ist eine universelle, interpretierte höhere Programmiersprache

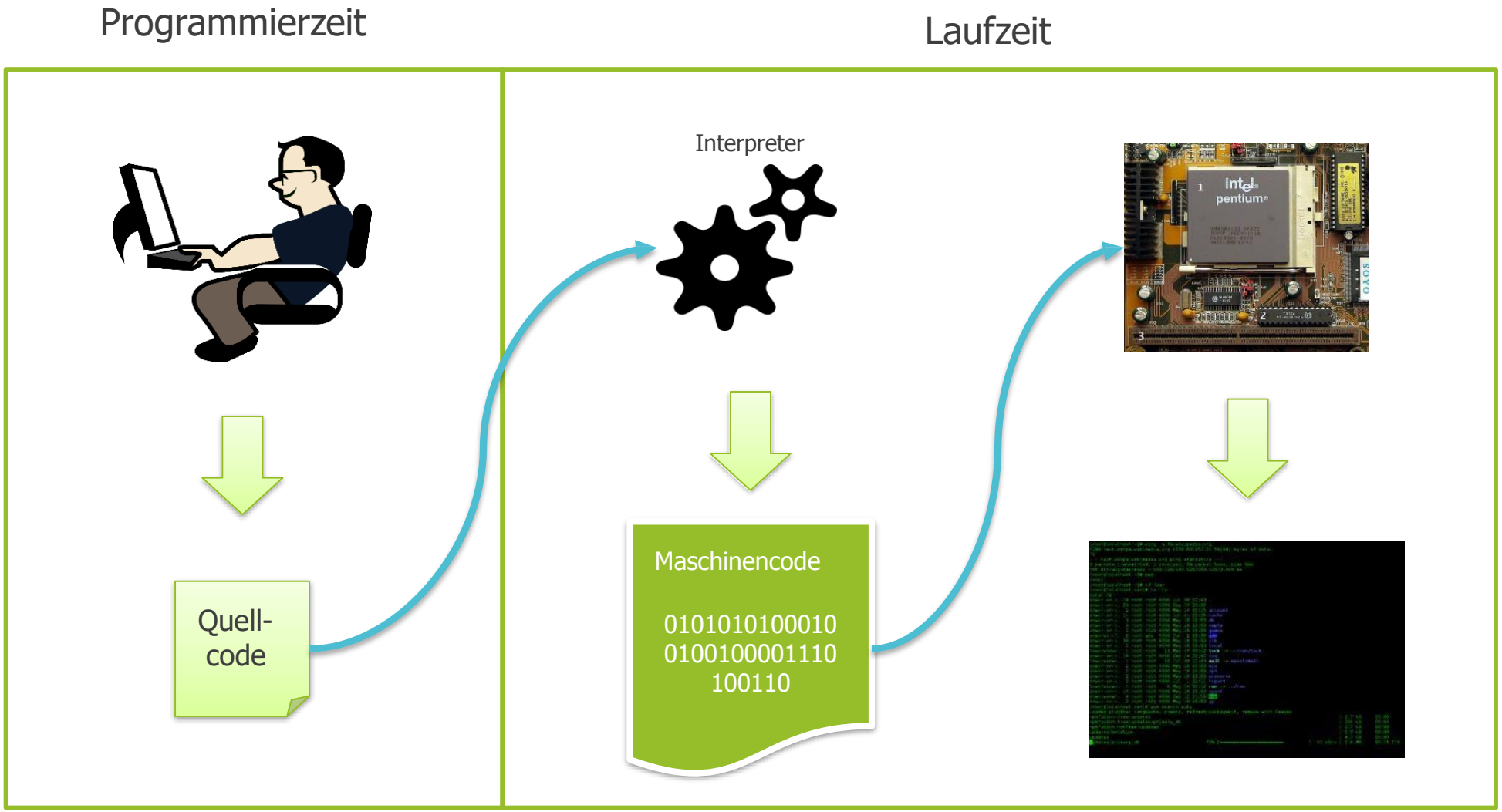
- Universell: Python kann für unterschiedliche Vorhaben eingesetzt werden
  - Data Science, TensorFlow
  - Web, Python Flask Webframework
  - SysAdmin, Automatisierungsskripts

Grundlagen Python II

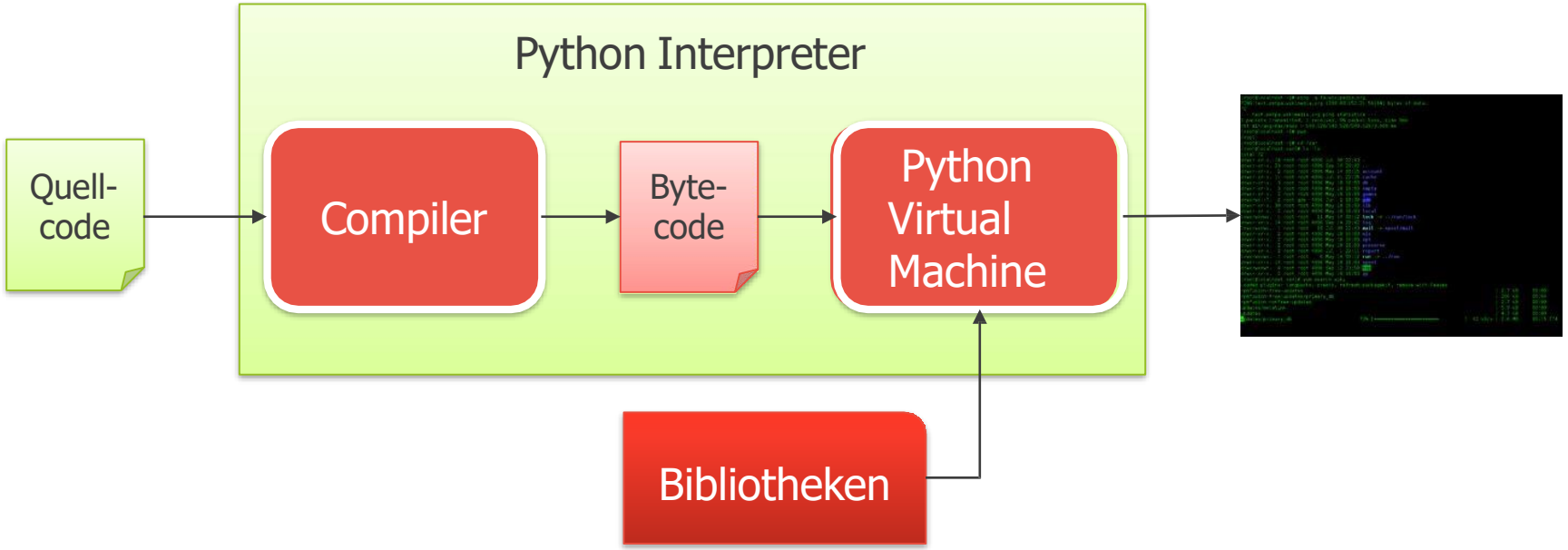
- Interpretiert: Python wird zur Laufzeit über einen Interpreter ausgeführt und nicht vorher in Maschinensprache kompiliert (üblicherweise)

- Höhere Programmiersprache: Python nutzt Abstraktionen von Konstrukten der Maschinensprache bzw. Assembler um es für Menschen einfacher programmierbar zu machen

Kompiliert vs Interpretiert II



Python Interpreter (CPython)



» Neben CPython gibt es viele andere Implementierungen, welche Python Code ausführbar machen!

Beispielprogamm „Hello World“ in Python

```
greeting = "Hello World"
print(greeting)
```

- Erklärung:
  - 1. Zeile: Es wird eine Variable greeting deklariert und der Variable wird der Wert Hello World zugewiesen. Der Wert Hello World ist ein String.
  - 2. Zeile: Es wird die Funktion print aufgerufen und als 1. Argument wird die Variable greeting übergeben.

Kommentare im Quellcode

- Raute (#) um einen einzeiligen Kommentar im Quellcode einzuführen
- Dreifaches Anführungszeichen an Anfang und Ende für docstring
  - Nicht zugeordnete Stringliterale werden von Interpreter ignoriert

```
# Python Kommentare:

# Kommentar für die ganze Zeile
a = 3 # Kommentar am Ende der Zeile

"""
This is a multi-line comment
with docstrings
"""
```

Variablen

- Variablen werden deklariert und mit einem Wert initialisiert (=)
  - Variablen haben einen Datentyp, dieser Datentyp ist dynamisch (kann zur Laufzeit des Programms verändert werden)
  - Der Datentyp wird implizit bestimmt
  - Die Built-in Funktion *type* kann genutzt werden um den Datentyp zu erfragen

```
my_number = 123
my_string = "a string"
print(type(my_number)) # <class 'int'>
print(type(my_string)) # <class 'str'>
```

Zulässige Variablennamen (in Python)

- Ein Variablenname darf mit einem alphabetischen Zeichen (a-z, A-Z) oder einem Unterstrich beginnen
- Der Variablenname darf ab der zweiten Stelle alphanumerische Zeichen (a-z, A-Z, 0-9) bzw. den Unterstrich enthalten
- Es gibt einige reservierte Wörter (zB if, for, while, ...), welche nicht als Variablennamen verwendet werden können, da sie eine andere Bedeutung haben

```
_var1 = "some text"

BlaBla9 = 123
Komma_Zahl = 32.32
Truthy = True
Nothing = None
```

Einfache Datentypen

- Ganzzahlen (int)
- Gleitkommazahlen (float)
- (Komplexe Zahlenwerte (complex))
- Boolsche Werte (bool)
- Zeichenketten (str)
- None (NoneType), (null oder nil in anderen Programmiersprachen)

```
print(type(123))      # <class 'int'>
print(type(1.34))     # <class 'float'>
print(type(3j))       # <class 'complex'>
print(type(True))     # <class 'bool'>
print(type("hello")) # <class 'str'>
print(type(None))     # <class 'NoneType'>
```

Statische vs. dynamische Typisierung

- Bei statisch typisierten Sprachen wird der Datentyp bereits während der Kompilierzeit festgelegt und kann während der Laufzeit nicht mehr verändert werden
- Unterschiede der statischen Typisierung:
  - Bestimmte Fehler können bereits während des Kompilierens gefunden werden
    - z.B.: ein Integer kann nicht von einem String subtrahiert werden
  - Programme werden effizienter, da der Aufwand zur Typfestlegung eliminiert wird
- Beispiele für Programmiersprachen:
  - statische Typisierung: Java, C#, Scala, Swift, …
  - dynamische Typisierung: Python, PHP, JavaScript, ...
- Dynamisch typisierte Sprachen werden auch als Skriptsprachen bezeichnet

Datentyp String (Zeichenkette) I

- Einzeilige Strings können mit einfachen bzw. doppelten Anführungszeichen definiert werden
- Mehrzeilige Strings können mit 3 einfachen bzw. doppelten Anführungszeichen definiert werden
- Ein String besteht aus einzelnen Zeichen (in anderen Programmiersprachen auch Chars genannt)
- Die Länge eines Strings ist als die Anzahl der Zeichen definiert
- Mit eckigen Klammern [] und einer Indexangabe kann auf einzelne Zeichen des String zugegriffen werden

Wichtig: Es ist egal, ob mit einfachen oder doppelten Anführungszeichen gearbeitet wird. Es müssen nur die selben Anführungszeichen am Anfang und am Ende verwendet werden.

Datentyp String (Zeichenkette) II

```
string1 = 'einzeilige Zeichenkette'

string2 = '''mehrzeilige
zeichen-
kette
'''

print(len(string1)) # 23
print(len(string2)) # 27
```

- Erklärung:
  - 1. Zeile: string1 wurde als einzeiliger String mit einfachen Anführungszeichen definiert
  - 3. Zeile: string2 wurde als mehrzeiliger String mit 3 einfachen Anführungszeichen definier

String Formatierung

- Innerhalb eines Strings können Platzhalter **{}** definiert werden, welche durch die Funktion format mit beliebigen Werten gefüllt werden können

Platzhalter 0

Platzhalter 1

```
template = „Name: {}, Alter: {}"
print(template.format("Peter", 42)) # Name: Peter, Alter: 42
```

Wert 0

Wert 1

- Neben dem reinen Ersetzen können auch komplexere Formatierungen durchgeführt werden, dazu hat Python eine „Mini- Sprache" integriert, Beispiel:

```
res = "Kosten: € {:> 8.2f}".format(159.9868)
print(res) # Kosten: €   159.99
```

Der Wert wird anhand der Formatvorgabe ausgegeben

String Formatierung: Minisprache

```
[[fill]align][sign][#][0][width][,][.precision][type]
```

- Innerhalb des Platzhalters **{}** im String kann mit einem führenden Doppelpunkt (**:**) eine Formatierungs-spezifikation angegeben werden
- Die wichtigsten Optionen:
  - type: **f** als Float bzw. **d** als Ganzahl
  - .precision: Ganzahl, welche die Anzahl der Nachkommastellen bei type f spezifiziert
  - ,: Tausenderpunkte anzeigen
  - width: Mindestbreite der gesamten Ausgabe
  - 0: mit 0 auffüllen bis zur Mindestbreite
  - #: Zahlenformat verändern (b für Binär, x für Hex, …)
  - sign: Vorzeichen Angabe (+, -, „ „)
  - align: Ausrichtung der Ausgabe in der Mindestbreite (< für linksbündig, > für rechtsbündig, ^ für zentriert, …)
  - fill: Zeichen zum Füllen links oder rechts neben der Ausgabe bis zu Mindestbreite

Alle Details: https://docs.python.org/3/library/string.html#format-string-syntax

Datentyp Integer (Ganzzahl)

- Das Dezimalsystem und das Binärsystem sind Stellenwertsysteme
- Die Anzahl der Stellen bzw. die entsprechende Ziffer an der Stelle bestimmen den Wert einer Zahl

Stellen	9	8	7	6	5	4	3	2	1	0
Repräsentations-möglichkeiten	2^9	2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
	512	256	128	64	32	16	8	4	2	1

Datentyp Float (Gleitkommazahl)

- Es gibt unterschiedliche Genauigkeiten:

Bezeichnung	Bit Gesamt	Bit Exponent	Bit Mantisse
single	32	8	23
double	64	11	52

- CPython float hat double precision

Operationen mit Variablen

- Eine Operation wird über einen Operator und 2 Operanden durchgeführt
- Es sind 7 Klassen von Operationen zu unterscheiden
  - Arithmetische Operationen (+, -, *, /, **, %)
  - Logische Operationen (and, or, not)
  - Vergleichsoperationen (==, <, >, <=, >=, !=, ...)
  - Zuweisungsoperationen (=, +=, -=, ...)
  - Bitweise Operationen (&, |, ^, <<, >>, ...)
  - Zugehörigkeitsoperationen (in, not in)
  - Identitätsoperationen (is, is not)

Wichtig: Der Datentyp bestimmt implizit die dafür definierten Operationen. ZB Die Subtraktion ist für Strings nicht definiert und führt zu einem Fehler.

Arithmetische Operationen

```python
# Addition +
print(5 + 3)   # 8
print(4 + -7)  # -3

# Subtraktion -
print(5 - 3) # 2

# Multiplikation *
print(5 * 5) # 25

# Division /
print(7 / 3) # 2.333333333
```

```python
# Modulo %
print(7 % 3) # 1

# Division mit Abrunden //
print( 7 // 3) # 2

# Potenzieren **
print(7 ** 3) # 343
```

Wichtig: Der Datentyp bestimmt implizit die dafür definierten Operationen. z. B. Die Subtraktion ist für Strings nicht definiert und führt zu einem Fehler.

Logische Operationen

- Logische Operationen basieren auf der Aussagenlogik und sind über Wahrheitstabellen definiert
- In Python sind nur die Konjunktion (and), Disjunktion (or) und Negation (not) definiert
- Logische Operationen können kombiniert werden

A	B	A and B	A or B
True	True	True	True
True	False	False	True
False	True	False	True
False	False	False	False

```
A   =  True
B   =  True
C = False

print(not A)          # False
print(A and B)        # True
print(A and C)        # False
print(A or B)         # True
print(A or C)         # True
print(True and B or not B) # True
```

Die Ausführungsreihenfolge von logischen Operationen ist: not, and, or

Vergleichsoperationen

- Eine Vergleichsoperation hat immer einen booleschen Wert als Ergebnis
- Vergleichsoperationen können verkettet werden, die Ausführung ist immer von links nach rechts je Operation und die einzelnen Operationen werden implizit mit **and** verknüpft

```
print(2    == 3)                    # False
print(3    > 5)                     # False
print(3    < 5)                     # True
print(2    >= 2)                    # True
print(5    <= 2)                    # False
print(1    != 2)                    # True
print(2    == 2 !=    4)            # True
print(3    < 5 < 12 > 11) #           True
```

Zuweisungsoperationen

- Variablen werden deklariert und initialisiert (Zuweisungsoperation)
- Es gibt einige Shortcuts (zB +=) um arithmetische Operationen mit Zuweisungen zu kombinieren

```
a  = 3          #  a  ==  3
a  = a +   5    #  a  ==  8
a  += 5         #  a  ==  13
a  -= 2         #  a  ==  11
```

Zugehörigkeitsoperation

» Die Zugehörigkeitsoperation prüft ob ein Element Teil einer Datenstruktur ist

```
print("c"  in  "abcdefg")     #   True
print("de" in  "abcdefg")     #   True
print("ed" in  "abcdefg")     #   False
```

Typkonvertierung

- Variablen können in einen anderen Datentyp konvertiert werden (Explizite Typkonvertierung)
- Es gibt unterschiedliche Funktionen dafür (ZB str(), int(), float(), ...)
  - Eine ungültiger Konvertierungsversuch führt zu einem Programmfehler

```
a  =  "10100"
b  =  "2.334343"
c  =  "no number"

 c1  = int(a)                          #  10100
 c2  = int(a, 2)                       #  20
 c3  = int(float(b))                   #  2
 c4  = int(c)                          #  ERROR
 c5  = 2 + " Versuche"                 #  ERROR
 c6  = str(2) + " Versuche"            #  2 Versuche
```

Interpretation von Wahr/Falsch in Python

- Bedingungen in Kontrollstrukturen müssen nicht zwingend einen booleschen Datentyp als Ergebnis haben
- Python interpretiert alles als True außer:
  - False
  - None
  - 0 oder 0.0 (Die Zahl 0)
  - „“ (Leerstring)
  - Leere Datenstrukturen (zB leere Liste)

```
if  42:      print("42")                    #  42
if  0.0:     print("0.0")
if  0.01:    print("0.01")                  #  0.01
if  "":      print("Leerstring")
if  "hi":    print("hi")                    #  hi
if  None:    print("None")
```

Kontrollstrukturen

- Um Algorithmen zu beschreiben, werden sog. Kontrollstrukturen benötigt
  - **Fallunterscheidungen (IF/ELSE)**
  - **Schleifen bzw. Wiederholungen (FOR, WHILE)**
- Anhand von Bedingungen (Boolescher Ausdruck) kann ein unterschiedlicher Pfad in der Programmausführung gewählt werden
  - **Steuerung des Programmablaufs**

Bedingte (if/else) Anweisung

- Die bedingte (if/else) Anweisung ist eine wesentliche Komponente der imperativen/prozeduralen Programmierung
- Anhand einer Bedingung, welche wahr oder falsch sein kann, wird der Programmablauf gesteuert
- Syntax im Pseudocode:
  IF *<Bedingung>* **THEN** *<Anweisung1>* **ELSE** *<Anweisung2>*
- Für die Formulierung der Bedingung können Vergleichsoperationen und/oder logische Operationen verwendet werden

if/else Beispiel

```
guess = int(input("Rate die Zahl: "))

if guess == 42:
    print("Super richtig geraten")
elif guess < 42:
    print("Leider zu klein")
else:
    print("Leider zu groß")
```

- Kurze Erklärung:
  - 1. Zeile: Der Benutzer wird aufgefordert eine Zahl einzugeben (**input**), welche in einen Integer konvertiert wird
  - 3. Zeile: Falls die Variable **guess** den Wert **42** hat, wird Zeile 4 ausgeführt (Anzeige: „Super richtig geraten")
  - 5. Zeile: Falls die Variable **guess** kleiner **42** ist, wird Zeile 6 ausgeführt (Anzeige: „Leider zu klein")
  - 7. Zeile: Alle anderen älle (hier **guess** ist größer **42**) führen zur Ausführung von Zeile 8 (Anzeige: „Leider zu groß")

Details zur Syntax (in Python)

Das Ende der Bedingung wird mit einem Doppelpunkt markiert

elif steht für else if, es können beliebig viele elif Teile eingebaut werden

```
if guess == 15:
    print("Super richtig geraten")
elif guess < 15:
    print("Leider zu klein")
else:
    print(„Leider zu groß")
```

Der „Block" an Anweisungen, welche zu einem Fall gehören, müssen mit einem Tab eingerückt werden

FOR-Schleife

- Die FOR-Schleife nutzt das sog. Iterator Design Pattern um zusammengesetzte Datenstrukturen sequentiell zu iterieren
- Der Iterator definiert eine Schnittstelle, welche von allen Iterables (iterierbaren Datenstrukturen) implementiert wird
- Die FOR-Schleife ist somit unabhängig von der entsprechenden Datenstruktur, solange der Iterator implementiert ist

```
for char in "abcd":
    print(char)

for item in ('a', 'b', 'c', 'd'):
    print(item)

# print 2x:
# a
# b
# c
# d
```

Egal welche Datenstruktur, solange sie iterierbar ist, kann die FOR-Schleife angewandt werden

Die range-Funktion

- Python hat eine built-in Funktion range(), welche gerade im Zusammenhang mit der FOR-Schleife sehr nützlich ist
- Mit range kann eine iterierbare Sequenz erzeugt werden, welche über die FOR-Schleife durchlaufen werden kann
- Folgende Funktionsaufrufe sind definiert:
  - range(end): von 0 bis end-1
  - range(start, end): von start bis end-1
  - range(start, end, step): von start bis end-1 in der Schrittweite step

inklusive 10 | exklusive 21 | Schrittweite 2

```
numbers = ""
for i in range(10,  21, 2):
    numbers += str(i) + " "
print(numbers) # 10 12 14 16 18 20
```

Kopfgesteuerte (WHILE) Schleife

- Schleifen sind eine weitere wesentliche Komponente der imperativen/prozeduralen Programmierung
- Anhand einer Bedingung, welche wahr oder falsch sein kann, wird ein Block von Anweisungen wiederholt
- Syntax im Pseudocode:
  WHILE *<Bedingung> <Anweisungen>*
- Man spricht bei der WHILE-Schleife auch vom Schleifenkopf als Bedingung und vom Schleifenrumpf als dem Block der Anweisungen

Hinweis: Fußgesteuerte Schleifen (DO-WHILE Schleifen) werden in Python nicht unterstützt.

WHILE Beispiel

```
secret = 42
guess = 0

while guess != secret:
    guess = int(input("Rate die Zahl: "))

print("Super die Zahl wurde erraten!")
```

- Kurze Erklärung:
  - 1. Zeile: Es wird eine Zahl als secret festgelegt
  - 2. Zeile: Die Variable guess wird mit 0 inititalisiert
  - 4. Zeile: Eine WHILE-Schleife wird angegeben, die Bedingung kann so formuliert werden: „Solange guess und secret nicht übereinstimmen, wiederhole den Schleifenrumpf“
  - 5. Zeile: Der Benutzer wird nach einer Zahl gefragt (input), welche in der Variable guess hinterlegt wird
  - 7: Zeile: Es wird „Super die Zahl wurde erraten!“ ausgegeben, dies passiert nur, wenn die Schleife zu einem Ende findet

Schleifensteuerung

- Es können Schleifen programmiert werden, welche ein Programm nicht mehr enden lassen (Endlosschleifen)
  - Programmierfehler
  - Eine Abbruchbedingung soll so gewählt werden, dass der Fall tatsächlich eintritt

- **continue** und **break** sind 2 Anweisungen, welche neben der Bedingung im Schleifenkopf zur Steuerung verwendet werden können
  - **continue: bricht den aktuellen Schleifendurchlauf ab**
  - **break: bricht die Schleife vollständig ab**

Wichtige built-in Funktionen

- Python hat einige built-in Standardfunktionen, welche für gängige Aufgaben genutzt werden können
- Folgend eine Übersicht:

Funktion	Beschreibung	Funktion	Beschreibung
len()	gibt die Länge eines Strings zurück	input()	Einlesen eines String vom Terminal (bis zum Enter Kommando)
print()	Ausgabe eines String auf dem Terminal	abs()	Absoluter Wert einer Zahl
min()	Gibt das Minimum der Zahlen zurück	max()	Gibt das Maximum der Zahlen zurück
round()	Runden einer Fließkommazahl	type()	Gibt des Datentyp einer Variable wieder

Literatur und Links

- Python 3 Keywords (reservierte Wörter)
  - https://docs.python.org/3/reference/lexical_analysis.html#keywords
- Python 3 Ausführungsreihenfolge und Operatoren Rangordnung
  - https://docs.python.org/3/reference/expressions.html#evaluation-order
- Python 3 Built-in Funktionen
  - https://docs.python.org/3/library/functions.html#built-in-functions

Funktionen bzw. Prozeduren

- Eine Funktion kapselt einen Programmteil als wiederverwendbare Einheit
- Eine Funktion wird definiert unter einem eindeutigen Namen
- Eine Funktion kann parametrisiert werden, über Funktionsargumente
- Eine Funktion kann ein Ergebnis als Rückgabewert haben
- Eine Funktion wird aufgerufen über den Namen und die entsprechenden Argumente

Einfaches Funktionsbeispiel

```
def say_hello(name="world"):
    print("hello " + name + "!")

say_hello()             # hello world!
say_hello("students") # hello
students!.
```

- Kurze Erklärung:
- 1. Zeile:
  - Mit dem Schlüsselwort **def** wird die **Funktionsdefinition** eingeleitet. Die erste Zeile der Funktionsdefinition wird auch **Signatur** oder **Funktionskopf** genannt.
  - **say_hello** wird als **Funktionsname** angegeben.
  - Mit den **Klammern ()** wird die **Parameterliste** definiert. Es gibt einen Parameter **name**, welcher einen **Defaultwert „world"** angegeben hat.
  - Der **Doppelpunkt** schließt die Funktionsdefinition ab.
- 2. Zeile:
  - Die Zeile ist mit einem **Tab** eingerückt (alle eingerückten Zeilen bilden den **Funktionsrumpf**)
  - Die Funktion **print** wird aufgerufen mit einem konkatenierten String als Argument. Das Argument **name** wird in der Konkatenation genutzt.
- 4-5 Zeile: Die Funktion wird genutzt. Im ersten Fall ohne Argumente (**name** wird mit Defaultargument genutzt) und im zweiten Fall wird **name** mit **„students"** belegt.

Lokale Variable, Funktionsaufruf und Rückgabewert

```
def create_hello(name="world"):
        hello = "hello " + name + "!"
        return hello

result = create_hello("students")
print(result) # hello students!
```

- Kurze Erklärung:
  - 1. Zeile: Die Funktion **create_hello** wird definiert
  - 2. Zeile: Die 1. Zeile im Funktionsrumpf deklariert eine **lokale Variable hello** mit dem konkatenierten String
  - 3. Zeile: Die 2. Zeile im Funktionsrumpf gibt die Variable **hello** als Rückgabewert über **return** als Ergebnis zurück
  - 5. Zeile: Die Funktion **create_hello** wird mit dem Argument **„students"** aufgerufen und das Ergebnis wird in die Variable **result** gelegt
  - 6. Zeile: Die Funktion **print** wird mit dem Argument **result** aufgerufen

Funktion: Schlüsselwort Aufruf

```
def say_hello(greeting="hello", name="world", end="!"):
    print(greeting + " " + name + end)

say_hello()                                    # hello world!
say_hello(end="!!!")                           # hello world!!!
say_hello(name="students",
          greeting="good evening")             # good evening students!
```

- Die Argumente können an die Funktion auch über die Angabe des Argumentnamen (Schlüsselwort) übergeben werden
- Normalerweise werden die Argumente anhand der Reihenfolge der Übergabe zugewiesen
- Beim Schlüsselwort Aufruf können die Parameter beliebig geordnet werden

Geltungsbereich (Scope, Kontext) einer Variable

- Variablen haben einen Geltungsbereich (lokal bzw. global)
- Variablen, welche innerhalb einer Funktion deklariert werden, sind innere oder lokale Variablen
  - lokale Variablen sind nur innerhalb der Funktion gültig
- Variablen, welche außerhalb einer Funktion deklariert werden, sind globale Variablen
- Mit dem Schlüsselwort global können Variablen im inneren Kontext (bzw. Geltungsbereich) als globale Variablen definiert werden

Beispiel: Geltungsbereich von Variablen

```
a = 10
def add():
    b = a + 10
    print(b)

add()     # print 20
print(a) # print 10
```

Lesender Zugriff auf eine globale Variable

- Kurze Erklärung:
  - 1. Zeile: Die globale Variable **a** wird mit dem Wert **10** initialisiert
  - 2. Zeile: Die Funktion **add** wird definiert
  - 3. Zeile: Die innere Variable **b** wird mit dem Wert **a + 10** initialisiert
  - 4. Zeile: Die Funktion **print** wird mit dem Argument **b** aufgerufen
  - 5. Zeile: Die Funktion **add** wird ausgeführt
  - 6. Zeile: Die Funktion **print** wird mit dem Argument **a** aufgerufen

Funktionen mit variablen Argumenten (*args, **kwargs)

- Für Funktionen können variable Parameter definiert werden, 2 Arten sind hier möglich:
  - unbenannte Parameter, welche als Tupel innerhalb der Funktion verfügbar sind (*args)
  - über Schlüsselwörter benannte Parameter welche als Dictionary innerhalb der Funktion verfügbar sind (**kwargs)

Unbenannte Argumente über *

```
def sum(*myparams):
    sum = 0
    for arg in myparams:
        sum += arg

    return sum

sum1 = sum(1, 2, 3, 4, 5)
sum2 = sum(10, 20)
sum3 = sum()

print(sum1, sum2, sum3)   # 15 30 0
```

Beispiel **kwargs

Keyword Argumente über **

```
def format_output(**names):
    for name, age in names.items():
        print("Name: " + name + ", Alter: " + str(age))

format_output(Peter=25, Hilde=31, Bruno=45)

# Name: Peter, Alter: 25
# Name: Hilde, Alter: 31
# Name: Bruno, Alter: 45
```

- Die Keyword Argumente sind ein Dictionary, als Funktionsargumente können beliebig viele Parameter angegeben werden

Zusammenfassung Funktionen

- Eine Funktion hat einen **Funktionskopf** (Signatur)
  - Schlüsselwort **def** als Einleitung
  - **Name der Funktion** (erlaubte Zeichen ähnlich Variablenname)
  - **Parameter** werden in Klammern angegeben
    - **Defaultparameter** mit = angeben
    - **VarArgs** können mit * oder ** angegeben werden
  - Abschluss mit Doppelpunkt
- Alle Zeilen des **Funktionskörpers** sind eingerückt über einen Tab
- Rückgabewerte werden über **return** definiert

Zusammengesetzte Datenstrukturen: list, tuple, set, dictionary

- Zur Verarbeitung von Datenmengen gibt es unterschiedliche Datenstrukturen, Python hat 4 built-in Datenstrukturen
- Datenstruktur **list**
  - Eine list ist eine sequentiell geordnete Liste von Elementen
  - Elemente werden über einen Index identifiziert
  - Die Länge der list ist dynamisch und es können jederzeit Elemente hinzugefügt oder entfernt werden (**mutable**)
- Datenstruktur **tuple**
  - Ein tupel ist eine sequentiell geordnete Liste von Elementen
  - Elemente werden über einen Index identifiziert
  - Die Elemente (Anzahl und Inhalt) des tupel ist nach der Initialisierung fixiert und unveränderlich (**immutable**)

Zusammengesetzte Datenstrukturen: list, tuple, set, dictionary

- Datenstruktur **set**
  - Ein set ist eine ungeordnete und eindeutige (keine Kopien) Menge von Elementen, vergleichbar mit Mengen aus der Mathematik
  - Elemente eines set besitzen keinen Index
  - Die Länge des set ist dynamisch und es können jederzeit Elemente hinzugefügt oder entfernt werden (**mutable**)
- Datenstruktur **dictionary**
  - Ein dictionary ist eine Datenstruktur, welche Schlüssel-Wert-Paare als Elemente hält
  - Elemente werden über einen selbstdefinierten **Schlüssel** identifiziert
  - Die Länge des dictionary ist dynamisch und es können jederzeit Elemente hinzugefügt oder entfernt werden (**mutable**)

Allgemeines zu Listen

- Listen werden über die Zeichen **[** und **]** initialisiert und können Elemente beliebigen Datentyps enthalten
- Listen können auch Listen als Elemente haben (mehrdimensionale Liste)

```
my_list = ['a', 'b', 3, ['sub', 'list']]
```

- Der Zugriff auf den Index (lesend/schreibend) wird ebenfalls über **[** und **]** durchgeführt
- Der Index beginnt mit **0** bis **Länge – 1**

```
print(my_list[1]) # 'b'
my_list[1] = 2
print(my_list[1]) # 2
```

- Mit der built-in Funktion **len** kann die Länge der Liste ermittelt werden

```
print(len(my_list)) # 4
```

Slicing Operationen mit Listen

» Ähnlich wie für Strings kann das „Slicing" auch für Listen verwendet werden, um Teillisten zu spezifizieren

» Die generelle Syntax ist **[start:end:step]**, folgendes ist zu beachten:

- start ist ein inklusiver Index
- end ist ein exklusiver Index
- für Indizes werden positive Werte von links nach rechts und negative Werte von rechts nach links gelesen
- mit step wird die Schrittweite angegeben

Eine Angabe welche keinen Sinn macht, oder außerhalb der Grenzen der Liste definiert ist, gibt keinen Fehler aber eine leere Liste

Operationen mit Listen: Einfügen, Löschen, …

Funktionen der Liste

```
data = ['a', 'b', 'c']

data.remove('c')      # ['a', 'b']
data.append('x')      # ['a', 'b', 'x']
data.reverse()        # ['x', 'b', 'a']

data.insert(1, 'y') # ['x', 'y', 'b', 'a']
data.pop(-1)          # ['x', 'y', 'b']

data.clear()          # []
```

Slice-basierte Syntax

```
data = ['a', 'b', 'c']

del data[2]
data[len(data):] = ['x']
data[::-1]



del data[:]
```

Weitere Operationen

```
data1 = ['a', 'b']
data2 = ['c', 'd']

print(data1 + data2)   # ['a', 'b', 'c', 'd']
print(data1 * 2)       # ['a', 'b', 'a', 'b']
```

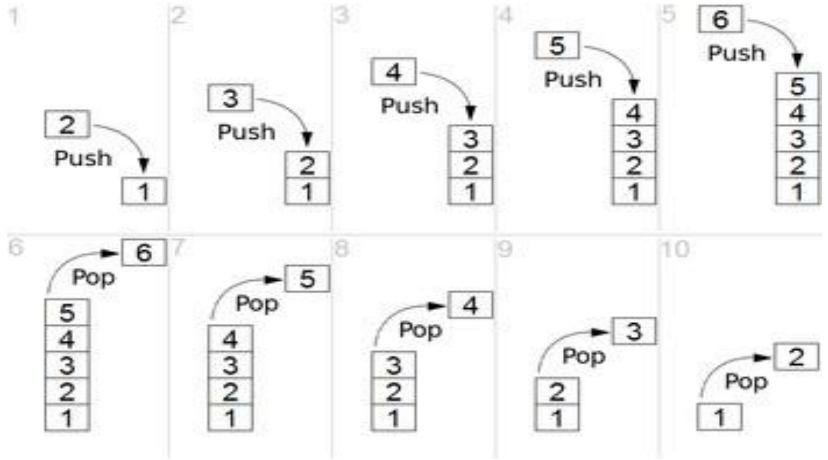
Grundlegende Datenstrukturen mit Listen: Stack (LIFO) und Queue (FIFO)

Stack (LIFO)

```
data = []
data.append('a')
data.append('b')

print(data)   # ['a',          'b']

data.pop()
print(data)   # ['a']
```

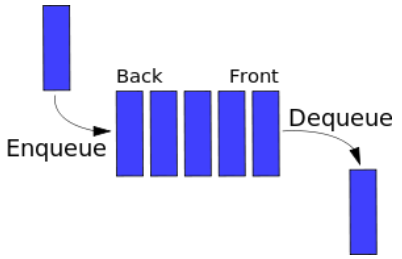


Queue (FIFO)

```
data = []
data.insert(0,        'a')
data.insert(0,        'b')

print(data)          # ['b', 'a']

data.pop()
print(data)          # ['b']
```



Hinweis: **deque** ist eine performantere FIFO Datenstruktur in Python (Teil der Standardbibliothek)

Iteration durch Listen

- Für die Iteration durch eine Liste eignet sich die FOR-Schleife
- Im Vergleich zur WHILE-Schleife benötigt die FOR- Schleife keine Bedingung, da die Anzahl der Schleifendurchläufe über die Länge der Liste definiert ist

```
for my_item in ['a', 'b', 'c']:
    print(my_item)
```

- Mit der Funktion enumerate kann auch der Index des aktuellen Elements abgefragt werden

```
for idx, item in enumerate(['a', 'b', 'c']):
    print(str(idx) + ": " + item)
```

Allgemeines zu Tuple

- Tuple werden über die Zeichen **(** und **)** initialisiert und können Elemente beliebigen Datentyps enthalten
  - Als Kurzschreibweise können die Klammern auch weggelassen werden
- Der Zugriff auf den Index (nur lesend) wird über **[** und **]** durchgeführt
- Die built-in Funktion **len** kann ebenfalls mit Tuple verwendet werden

```
my_tuple = ('a', 'b', 1)
same_tuple = 'a', 'b', 1
print(my_tuple[1], same_tuple[1])          # 'b' 'b'
print(len(my_tuple), len(same_tuple))      # 3 3
```

- Tuple können auch über FOR-Schleifen iteriert werden

```
for my_item in my_tuple:
        print(my_item)
```

Immutability vs. Mutability

- Im lesenden Zugriff unterscheiden sich Tuple und List nicht
- Tuple sind immutable (unveränderlich):
  - Kein Element kann gelöscht oder anders positioniert werden
  - Es können keine weiteren Elemente hinzugefügt werden
  - Tuple können als Schlüssel für Dictionaries verwendet werden

```
tuple = ('blub', ['b', 'a'])
tuple[1][0] = 'c'
```

Tuple können mutable Elemente (zB list, set, …) enthalten.
Vorsicht: Mutable Elemente eines Tuples können geändert werden. Falls ein Tupel mutable Elemente enthält, kann es nicht mehr als Schlüssel in einem Dictionary verwendet werden!

Allgemeines zu Sets

- Sets sind den mathematischen Mengen sehr ähnlich
- Sets werden über die Zeichen { und } initialisiert und können Elemente beliebigen Datentyps enthalten
  - Jedes Element kommt nur einmal vor (eindeutig)
  - Elemente unterliegen keiner Ordnung
- Definierte Mengenoperationen: Vereinigung, Durchschnitt, Differenz, symmetrische Differenz

```
A = {1, 2, 3, 4}
B = {3, 4, 5, 6}
print(A | B) # Vereinigung: {1, 2, 3, 4, 5, 6}
print(A & B) # Durchschnitt: {3, 4}
print(A - B) # Differenz: {1, 2}
print(B - A) # Differenz: {5, 6}
print(A ^ B) # Symmetrische Differenz: {1, 2, 5, 6}
```

Allgemeines zu Dictionaries

- Dictionaries werden über die Zeichen **{** und **}** initialisiert und müssen für jedes Element einen eindeutigen Schlüssel definieren
- Schlüssel-Wert-Paare werden über : getrennt
- Der Zugriff auf den Wert (lesend/schreibend) wird über **[** und **]** und dem Schlüssel durchgeführt

```
my_dict = { 'one' : 123 , 'two' : '2', 3 : ['a', 'b'] }
print(my_dict['one']) # 123
print(my_dict[3][0])  # 'a'
my_dict['one'] = 'changed'
print(my_dict['one']) # 'changed'
```

- Zuweisungen über einen Schlüssel der noch nicht existiert, führt zur Erweiterung des Dictionaries

```
print(len(my_dict)) # 3
my_dict['new'] = 'hello'
print(len(my_dict)) # 4
```

Iteration durch Dictionaries

- Die Elemente eines Dictionaries bestehen aus 2 Teilen, dem Schlüssel und dem Wert
- 2 mögliche Varianten zur Iteration:
  - Default wird durch die Schlüssel iteriert, der Wert kann über [key] abgefragt werden
  - Die Funktion items() gibt für jede Iteration ein Tupel zurück (Schlüssel, Wert)

```
names = {'Peter': 25, 'Hilde': 31, 'Bruno': 45}

for key in names:
    print("Name: " + key + ", Alter: " + str(names[key]))

for key, value in names.items():
    print("Name: " + key + ", Alter: " + str(value))
# print 2x
# Name: Peter, Alter: 25
# Name: Hilde, Alter: 31
# Name: Bruno, Alter: 45
```

Lambdas (Lambda Ausdrücke)

- Lambdas (in Python) sind anonyme Funktionen, welche nur einen Ausdruck enthalten dürfen, welcher auch der Rückgabewert ist
- Lambdas sind nur syntaktische Konstrukte um einfache Funktionsdefinitionen zu ersetzen
- Ein Lambda kann dort definiert werden wo es gebraucht wird
- Die folgenden 2 Beispiele sind semantisch vollkommen identisch:

```
sum = lambda a, b: a + b


print(sum(2, 5))   # 7
print(sum(4, 3))   # 7
```

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

print(sum(2, 5))   # 7
print(sum(4, 3))   # 7
```

Fragen?

- Gibt es Fragen?

- Python-Übungen zu finden unter:
  - https://github.com/patrik98/fhk-drohnenworkshop

Ausblick 10.05.

- ROS 1 (Noetic)
  - ROS-Architektur
  - Interprozesskommunikation
  - Werkzeuge (Visualisierung, Transformation, Simulation …)
  - anhand eines Beispiels – simulierter Roboter