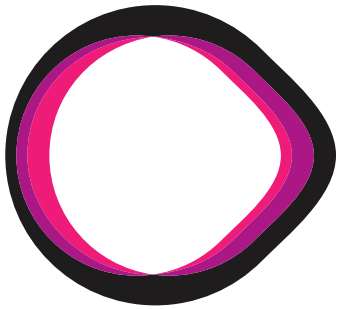
Python – Lektion 3
Exceptions und alles über Strings



OST
Ostschweizer
Fachhochschule

Disclaimer

Die Vertraulichkeitsklasse dieser Daten ist ”intern-erweitert”. Sie dürfen die Daten als OST-Angehörige nutzen, aber nicht an Dritte weitergeben oder veröffentlichen.

Rückblick

- Verzweigungen
  - `if`, `else`, `elif`

```
text = "Eins, Zwei, Drei"
if "Eins" in text:
    print("Wort gefunden!")
```

- Schleifen
  - `for`, `while`, `continue`, `break`

```
zahlen = [1, 2, 3, 4]
for zahl in zahlen:
        print(zahl)
```

- Funktionen definieren und benutzen
  - `def`, `return`, Standardwerte, `summe(20, b=4)`
  - lambda-Funktionen

`http://localhost:8888/notebooks/enum_zip.ipynb`

Heutige Themen

- Ausnahmebehandlungen
- Strings formatieren, manipulieren, testen

Ausnahmebehandlung

- Fehler[1] können auftreten, z.B.:

```
int("hallo")
```

```
Traceback (most recent call last):

  File "D:\exception_bsp.py", line 8, in <module>
    int('hallo')

ValueError: invalid literal for int() with base 10: 'hallo'
```

- und führen zu einem Abbruch des Programms.

```
int("bla")   => ValueError
5/0          => ZeroDivisionError
a[1000]      => IndexError
10 + "Fr."   => TypeError
```

[1] https://docs.python.org/3/tutorial/errors.html

Ausnahmebehandlung

- Fehler können abgefangen werden:

```
try:
    x = int(input("Zahl eingeben: "))
except ValueError:
    print("Falsche Eingabe!")
```

Ausnahmebehandlung

- Exception Handling für Program Flow Control
- oder: **LBYL** vs. **EAFP**

LBYL $\rightarrow$ *look before you leap*
Vor einer Operation wird auf alle möglichen Fälle geprüft, für welche die Operation failen könnte.
Nachteile:

- Die vielen Checks im Vorfeld erschweren die Lesbarkeit.
- Es gehen sehr wahrscheinlich Checks vergessen, die Operation failt trotzdem.
- Zwischen Check und Ausführung der Operation könnten sich die Bedingungen geändert haben (multi-threaded environment)

Ausnahmebehandlung

- Exception Handling für Program Flow Control
- oder: **LBYL** vs. **EAFP**

LBYL $\rightarrow$ *look before you leap*
Vor einer Operation wird auf alle möglichen Fälle geprüft, für welche die Operation failen könnte.
Beispiel:

```
eingabe = input("Zahl eingeben: ")
if(eingabe.lstrip('-+').isdecimal()):
    x = int(eingabe)
else:
    print("Falsche Eingabe!")
```

Ausnahmebehandlung

- Exception Handling für Program Flow Control
- oder: **LBYL** vs. **EAFP**

EAFP $\rightarrow$ *easier to ask for forgiveness than permission*
Es wird davon ausgegangen, dass die Operation mit den gegebenen Argumenten ausführbar ist. Sollte diese Annahme sich als falsch erweisen, werden mögliche Exceptions abgefangen.
Beispiel:

```
eingabe = input("Zahl eingeben: ")
try:
    x = int(eingabe)
except ValueError:
    print("Falsche Eingabe!")
```

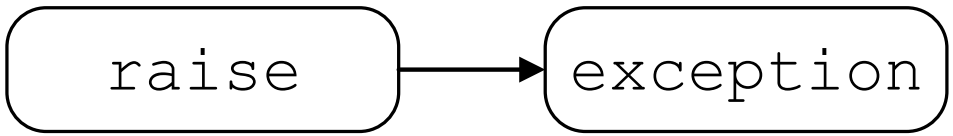
Performance Einbussen müssen in der Regel nicht befürchtet werden (wie es in C++ oder PHP der Fall ist).

Ausnahmebehandlung

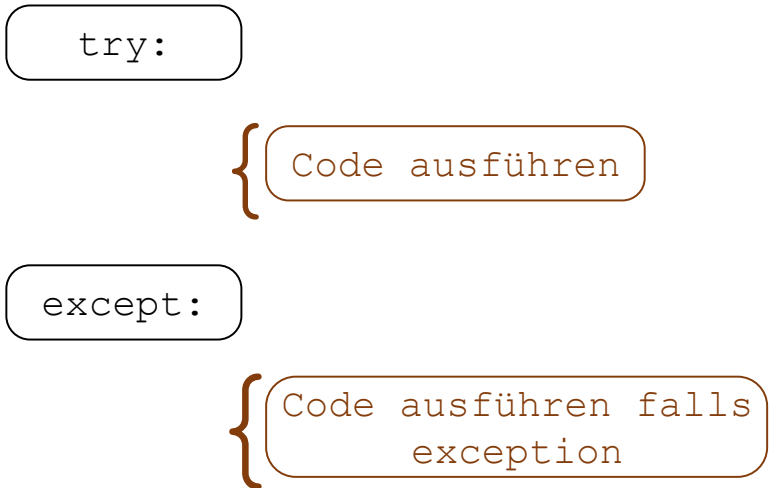
- `try`, `except`
- `else`
- `finally`
- `raise`

`http://localhost:8888/notebooks/ausnahmebehandlung.ipynb`

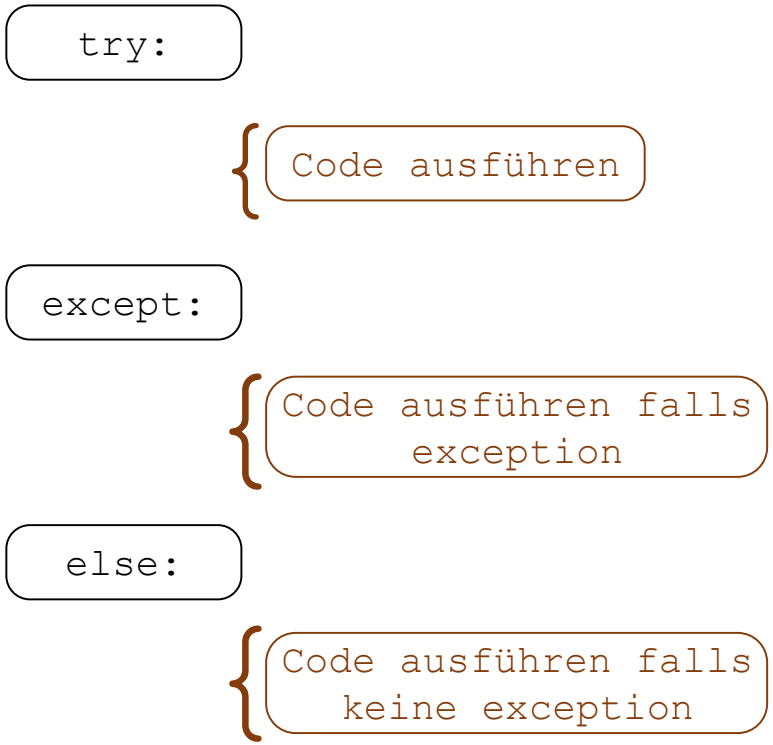
Ausnahmebehandlung - Raise



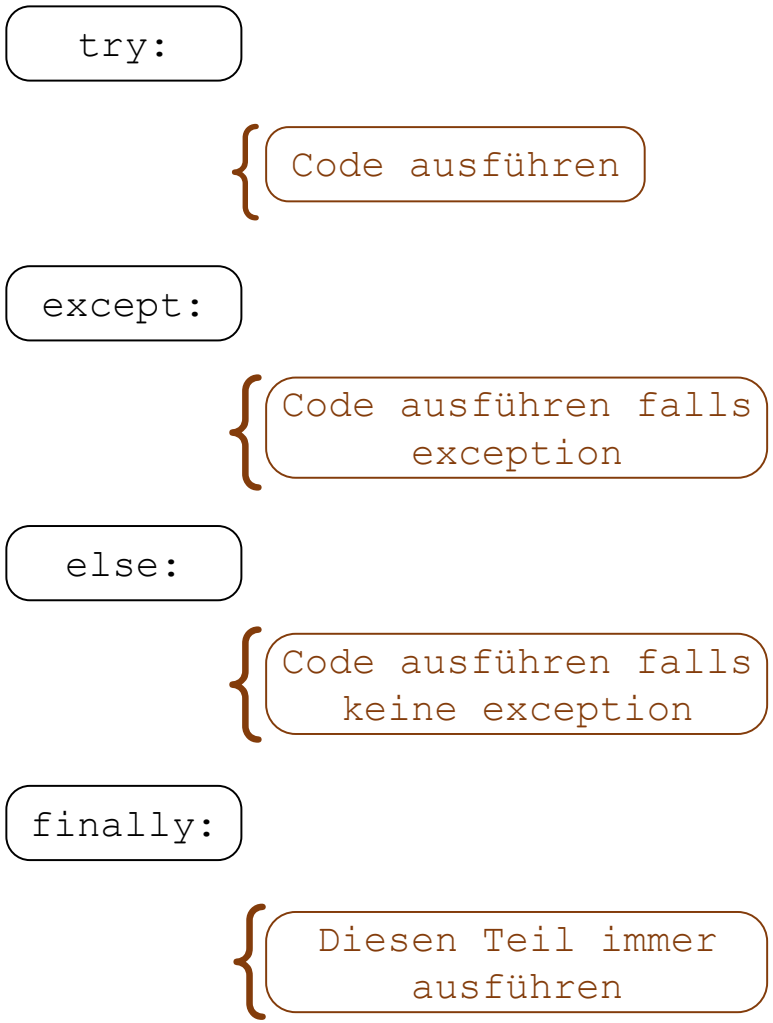
Ausnahmebehandlung - Try



Ausnahmebehandlung - Try



Ausnahmebehandlung - Try



Stringformatierung

- Stringformatierung benötigt man um Daten hübsch auszugeben

```
 Menge     Name        Wert
=======================
    3         R1       1.50k
    7         R2       0.10k
    2         R3      22.00k
    5         R4      47.00k
```

- oder systematisch abzuspeichern

```
Menge,Name,Wert
3,R1,1500
7,R2,100
2,R3,22000
5,R4,47000
```

Zwei Arten der Stringformatierung

Bis Python 3.6:

```
<template>.format(<pos_argument(s)>, <kw_argument(s)>)
```

Seit Python 3.6:

```
f<expression>
```

http://localhost:8888/notebooks/stringformatierung1.ipynb

Valide Syntax mit format():

```
"Der Preis für {} ist {} CHF.".format(artikel, preis)
```

Mit Positionsargumenten:

```
"Der Preis für {0} ist {1} CHF.".format(artikel, preis)
```

```
"Der Preis für {1} ist {0} CHF.".format(preis, artikel)
```

Mit Schlüsselwortargumenten:

```
"Der Preis für {artikel} ist {preis} CHF.".format(
    artikel=artikel,
    preis=preis,
)
```

Zwei Arten der Stringformatierung

Bis Python 3.6:

```
<template>.format(<pos_argument(s)>, <kw_argument(s)>)
```

Seit Python 3.6:

```
f<template>
```

http://localhost:8888/notebooks/stringformatierung1.ipynb

Valide Syntax mit f-strings:

```
f"Der Preis für {artikel} ist {preis} CHF."
```

```
"There should be one
- and preferably only one -
obvious way to do it."
```

Die Formatierung des Strings

Wo bleibt die Formatierung?

{[<name>][!<conversion>][:<format_spec>]}

Die Formatierung des Strings

`{[<name>][!<conversion>][:<format_spec>]}`

Spezifiziert die Konvertierungsmethode vor der eigentlichen Formatierung:

- `!s` → Konvertiere mit `str()`
- `!r` → Konvertiere mit `repr()`
- `!a` → Konvertiere mit `ascii()`[2]

`http://localhost:8888/notebooks/conversion.ipynb`

[2] https://en.wikipedia.org/wiki/ASCII

Die Formatierung des Strings

{[<name>][!<conversion>][:<format_spec>]}

Etwas detaillierter:

:[[<fill>]<align>][<sign>][#][0][<width>][<group>][.<precision>][<type>]

Die Formatierung des Strings

`:[[<fill>]<align>][<sign>][#][0][<width>][<group>][.<precision>][<type>]`

Angabe des Darstellungstyps:

[<type>]	**Beschreibung**
b	Binärer Integer
c	Charakter
d	Dezimalzahl
e oder E	Exponentialformat klein oder gross
f oder F	Fliesskommazahl klein oder gross
g oder G	Fliesskommazahl oder Exponentialformat abhängig von der Grösse des Wertes und der angegebenen Präzision, klein oder gross
o	Oktal Integer
s	String
x oder X	Hexadezimal Integer klein oder gross
%	Ausgabe eines Prozentzeichens

Die Formatierung des Strings

```
:[[<fill>]<align>][<sign>][#][0][<width>][<group>][.<precision>][<type>]
```

Minimale Grösse des Ausgabefeldes:

```
>>> f"{'ABC':4}"
"ABC "

>>> f"{123:4}"
" 123"
```

Die Formatierung des Strings

```
:[[<fill>]<align>][<sign>][#][0][<width>][<group>][.<precision>][<type>]
```

Anzahl Zeichen nach dem Komma bei floating point Zahlen:

```
>>> f"{1234.5678:8.2f}"
" 1234.57"
```

Bei Strings beschränkt es die Anzahl ausgegebener Zeichen:

```
>>> f"{'ABCDEFG':.4s}"
"ABCD"
```

Die Formatierung des Strings

`:[[<fill>]<align>][<sign>][#][0][<width>][<group>][.<precision>][<type>]`

Ausrichtung der Zeichen ([<width>] > Gesamtlänge):

[<align>]	**Beschreibung**
>	Rechtsbündig
<	Linksbündig
^	Mittig
=	Vorzeichen einer Zahl links aussen

```
>>> f"{123:>8}"
"     123"

>>> f"{123:<8}"
"123     "

>>> f"{123:^8}"
"  123   "

>>> f"{-123:=8}"
"-    123"
```

Die Formatierung des Strings

`:[[<fill>]<align>][<sign>][#][0][<width>][<group>][.<precision>][<type>]`

Angabe eines spezifischen Füllzeichen (immer mit `<align>`):

```
>>> f"{'ABC':->8}"
"-----ABC"

>>> f"{123:#<8}"
"123#####"

>>> f"{'foo':*^8}"
"**foo***"
```

Die Formatierung des Strings

```
:[[<fill>]<align>][<sign>][#][0][<width>][<group>][.<precision>][<type>]
```

Vorzeichen bei numerischen Werten:

```
>>> f"{123:+}"
"+123"

>>> f"{-123:+}"
"-123"
```

Die Formatierung des Strings

```
:[[<fill>]<align>][<sign>][#][0][<width>][<group>][.<precision>][<type>]
```

Alternative Darstellungsform einschalten (für integer, float, complex und dezimal):

```
>>> f"{16:b}, {16:#b}"
"10000, 0b10000"

>>> f"{16:x}, {16:#x}"
"10, 0x10"
```

Die Formatierung des Strings

:[[<fill>]<align>][<sign>][#][0][<width>][<group>][.<precision>][<type>]

Auffüllen mit Nullen (sofern [<fill>] und [<align>] nicht gesetzt)

```
>>> f"{123:05}"
"00123"

>>> f"{'ABC':>06}"
"000ABC"
```

Die Formatierung des Strings

```
:[[<fill>]<align>][<sign>][#][0][<width>][<group>][.<precision>][<type>]
```

Zahlen (int, float, hex, bin) in Gruppen (Tausender oder 4 Digits) darstellen, Trenncharakter entweder ein Komma oder ein Underscore:

```
>>> f"{1234567:,}"
"1,234,567"

>>> f"{1234567.89:_}"
"1_234_567.89"

>>> f"{0b111010100001:_b}"
"1110_1010_0001"
```

Alle Beispiele finden Sie hier:

`http://localhost:8888/notebooks/formatierung.ipynb`

Dokumentation:

`https://docs.python.org/3/library/string.html`

Alles über Strings

- Der `str`-Datentyp[3] besitzt viele nützliche Funktionen um Strings
  - zu bauen
  - zu manipulieren
  - zu untersuchen

Das Video zum Jupyter Notebook ”Alles über Strings” finden Sie auf Moodle.

`http://localhost:8888/notebooks/alles_ueber_strings.ipynb`

[3] https://docs.python.org/3/library/stdtypes.html#str

Alles über Strings

- Strings aufspalten

```
>>> "Guten Morgen   Welt.".split()
["Guten", "Morgen", "Welt."]
>>> "1;2;;;;3;4".split(';')
["1", "2", "", "", "", "3", "4"]
>>> "Guten\nMorgen\nWelt.".splitlines()
["Guten", "Morgen", "Welt."]
```

- Strings aufspalten

```
>>> "".join(["a", "b", "c"])
"abc"
```

- Strings bereinigen

```
>>> "  Hallo Welt!  \n".strip()
"Hallo Welt!"
>>> "  Hallo Welt!".rstrip("!")
"  Hallo Welt"
```

Alles über Strings

- Suchen und Ersetzen in Strings

```
>>> spruch = "Guten Morgen Welt"
>>> "Morgen" in spruch
True
>>> spruch.find("Morgen")
6
>>> spruch.count("Morgen")
1
>>> spruch.replace("Morgen", "Abend")
"Guten Abend Welt"
```

- Klein- und Grossschreibung

```
>>> "Passwort".lower()
"passwort"
>>> "Passwort".upper()
"PASSWORT"
```

Alles über Strings

- Strings testen

```
>>> '255'.isdigit()
True
>>> 'hallo'.isalpha()
True
>>> 'Gleis7'.isalnum()
True
>>> 'klein'.islower()
True
>>> 'GROSS'.isupper()
True
>>> 'Haus'.istitle()
True
```