6. Ausnahmebehandlung (*exception handling*)

Mit dem Begriff "Ausnahme" (*exception*) meint man häufig den Zustand eines Programms, der zu einem Fehler und damit zum Programmabstürz führt.

Die Ausnahmebehandlung (*exception handling*) ist ein Verfahren, um Fehlerzustände an andere Programmebenen weiterzuleiten.

Programmabstürze sind Katastrophen, die sogar zerstörerische Folgen haben können!

In [1]:
```
# TODO Beispiel 1
x = 1
y = 0
x/y
```

```
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
Cell In[1], line 4
      2 x = 1
      3 y = 0
----> 4 x/y

ZeroDivisionError: division by zero
```

In [2]:
```
# TODO Beispiel 2
eingabe = int(input("Geben Sie eine Zahl ein: "))
```

```
Geben Sie eine Zahl ein: asdf
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[2], line 2
      1 # TODO Beispiel 2
----> 2 eingabe = int(input("Geben Sie eine Zahl ein: "))

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

In [3]:
```
# TODO Beispiel 3
with open("michGibtsNicht.txt", "r") as datei:
    print(datei)
```

```
---------------------------------------------------------------------------
FileNotFoundError                         Traceback (most recent call last)
Cell In[3], line 2
      1 # TODO Beispiel 3
----> 2 with open("michGibtsNicht.txt", "r") as datei:
      3     print(datei)

File ~\anaconda3\Lib\site-packages\IPython\core\interactiveshell.py:310, in _modi
fied_open(file, *args, **kwargs)
    303 if file in {0, 1, 2}:
    304     raise ValueError(
    305         f"IPython won't let you open fd={file} by default "
    306         "as it is likely to crash IPython. If you know what you are doin
g, "
    307         "you can use builtins' open."
    308     )
--> 310 return io_open(file, *args, **kwargs)

FileNotFoundError: [Errno 2] No such file or directory: 'michGibtsNicht.txt'
```

Im Folgenden ist die Hierarchie der am häufigsten und allgemeinen Ausnahmen aufgeführt (s. auch in der Dokumentation):

- `Exception` (Generelle Fehlermeldung)
  - `StopIteration` (wird von *iterierbaren* Objekten (*Iterables*) ausgelöst, um z.B. `for` -Schleifen mitzuteilen, dass sie fertig durchlaufen sind)
  - `ArithmeticError` (Fehler bei Berechnungen)
    - `FloatingPointError` (Fehler spezifisch für Fließkommaberechnungen)
    - `OverflowError` (Ergebnis zu groß für den bereitgestellten Speicher)
    - `ZeroDivisionError` (Division durch Null)
  - `EOFError` (Lesezugriff nach Dateiende (*end of file*))
  - `ImportError` (Fehler bei Bearbeitung einer `import` -Zeile)
    - `ModuleNotFoundError` (Angegebenes Modul wurde nicht gefunden)
  - `LookupError` (Fehler bei Zugriff auf eine Datenstruktur oder -objekt)
    - `IndexError` (Ungültiger Index)
    - `KeyError` (Ungültiger Schlüssel bei `dictionary`)
  - `NameError` (Symbol mit diesem Namen existiert nicht)
  - `OSError` (Fehler bei Nutzung von Funktionen des Betriebsystems)
    - `FileExistsError` (Datei existiert bereits)
    - `FileNotFoundError` (Datei kann nicht gefunden werden)
  - `RuntimeError`
    - `NotImplementedError` (Methode/Variante einer Methode/Klasse wurde nicht implementiert)
    - `RecursionError`
  - `SyntaxError`
    - `IndentationError`
  - `TypeError` (Datentyp kann nicht verarbeitet werden)
  - `ValueError` (Unzulässiger Wert bei Funktionsaufruf)

(Auf die Hierachie achten!)

6.1 Abfangen von Ausnahmen mit `try`

6.1.1 `try-except` -Blöcke

Syntax in Python:

```
try:
    Code der eine Ausnahme auslösen könnte
except [ExceptionClass [as variable]]:
    Code zur Fehlerbehandlung
```

Mit `ExceptionClass` ist dabei der "Typ der Ausnahme" gemeint, also welche Fehlerklasse angezeigt wird bei Programmabbruch. Die Angabe ist optional genauso wie der Variablenname `variable`, mit dem man auf die Daten der Instanz `ExceptionClass` zugreifen kann.

In [5]:
```
# TODO Beispiel 1 try-except-Block
for x in range(-3, 3):
    try:
        print(1/x)
    except ZeroDivisionError as e:
        print("Exception was caught: ", e, type(e))
```

```
-0.3333333333333333
-0.5
-1.0
Exception was caught:  division by zero <class 'ZeroDivisionError'>
1.0
0.5
```

In [8]:
```
# TODO Beispiel 2 try-except-Block
try:
    for x in range(-3, 3):
        print(1/x)
except ArithmeticError as e:
    print("Exception was caught: ", e, type(e))
```

```
-0.3333333333333333
-0.5
-1.0
Exception was caught:  division by zero <class 'ZeroDivisionError'>
```

Ablauf der `try` -Anweisung:

- `try` -Anweisungsblock wird ausgeführt (oder zumindest versucht)
- Wenn **kein** Laufzeitfehler auftritt, wird die `except` -Klausel übersprungen
- falls ein Fehler auftritt, wird der `try` -Anweisungsblock sofort abgebrochen (und der Rest übersprungen), und stattdessen der `except` -Anweisungsblock ausgeführt

Eine `try` -Anweisung muss immer **mindestens** eine `except` (oder die `finally`)-Klausel besitzen. Wenn im Fehlerfall nichts passieren soll, schreibt man in den `except` -Block die Anweisung `pass` .

6.1.2 Mehrteilige `except`-Blöcke

Hinter `except` darf auch ein `tuple` von Fehlerklassen stehen, die dann jeweils mit dem gleichen Code behandelt werden. Der Syntax lautet z.B.:

```
except(ZeroDivisionError, ValueError) as e:
```

Wenn für verschiedene Fehlerklassen auch unterschiedlicher Code ausgeführt werden soll, können mehrere `except`-Blöcke hintereinander gesetzt werden. Dabei wir allerdings nur der **erste `except`-Block**, der zur ausgelösten Ausnahme passt, abgearbeitet.

In [12]:
```
# TODO mehrere except Blöcke


try:
    print(1/0)
except (ArithmeticError) as e:
    print(e, type(e))
```

```
division by zero <class 'ZeroDivisionError'>
```

6.1.3 optionale `else`-Klausel

Wollen Sie Code nur ausführen, wenn im `try`-Block **kein** Fehler aufgetreten ist, kann dies mit einem zusätzlichen `else`-Block realisiert werden. Dieser steht syntaktisch hinter den `except`-Blöcken und wird nur ausgeführt, nachdem die Behandlung des `try`-Blocks ohne Fehler abgeschlossen wurde.

In [14]:
```
# TODO else-Block für try-Anweisung
with open("neu.txt", "w") as neu:
    neu.write("eins\nzwei\ndrei")

filename = input("Dateiname: ")
try:
    f = open(filename, "r")
except IOError:
    print(filename, "lässt sich nicht öffnen")
else:
    print(f"{filename} hat {len(f.readlines())} Zeilen")
    f.close()
```

```
Dateiname: asdf
asdf lässt sich nicht öffnen
```

6.1.4 optionale `finally`-Klausel

Das Schlüsselwort `finally` leitet einen weiteren Anweisungblock ein, der *in jedem Fall* ausgeführt wird, auch wenn zuvor ein Laufzeitfehler aufgetreten ist. Man verwendet diese Kontrollstruktur beispielweise für Aufräumarbeiten bei Programmabbrüchen, wie Schließen und Speichern von Dateien, Trennen von Netzwerkverbindungen, usw.

Syntax:

```
try:
    anweisungsblock1
finally:
    anweisungsblock2
```

- `try`-Anweisungsblock wird ausgeführt (oder zumindest versucht)
- wenn ein Fehler auftritt, merkt sich das System die Ausnahme und führt zuerst noch die `finally`-Anweisung aus (diese wird auch ausgeführt, wenn *kein* Fehler auftritt)
- Im Fehlerfall folgen dann schließlich der Programmabbruch und die Meldung der Ausnahme

In [21]:
```
# TODO Beispiel
try:
    x = int(input("Eine Zahl: "))
# optional auch mit except
except:
    print("Fehler! ") # danach wird Programm weiter ausgeführt
else:
    print("Try war erfolgreich")
finally:
    print("wird immer ausgegeben")

print("wird nur ausgegeben, wenn das Programm nicht abbricht")
```

```
Eine Zahl: asd
Fehler!
wird immer ausgegeben
wird nur ausgegeben, wenn das Programm nicht abbricht
```

6.2 Ausnahmen generieren

6.2.1 Die `raise`-Klausel

Mit einer `raise`-Anweisung können sie gezielt Ausnahme-Ereignisse auslösen. Der Syntax lautet:

`raise ExceptionClass (assoziierter Wert)`
wobei `assoziierter Wert` ein String ist, der den Fehler näher erläutert und in der Fehlermeldung auftauchen soll.

In [22]:
```
# TODO raise-Anweisung
raise SyntaxError("Sorry, mein Fehler!")
```

```
Traceback (most recent call last):

  File ~\anaconda3\Lib\site-packages\IPython\core\interactiveshell.py:3553 in run
_code
    exec(code_obj, self.user_global_ns, self.user_ns)

  Cell In[22], line 2
    raise SyntaxError("Sorry, mein Fehler!")

  File <string>
SyntaxError: Sorry, mein Fehler!
```

Ausnahmen können nach der Verarbeitung "re-raised" werden. Wollen Sie beispielsweise eine detaillierte Fehlerbeschreibung ausgeben, aber dennoch ein Programmabbruch auslösen, können Sie dies realisieren, indem Sie im `except`-Block erneut `raise` (ohne weitere Argumente) ausführen:

In [28]:
```
# Detaillierte Fehlerbeschreibung
try:
    x = 7
    y = 0
    print(x/y)
except ZeroDivisionError as e:
    print("Division durch 0 aufgetreten")
    print("Fehlerbeschreibung: ", e)
    print("x = ", x)
    print("y = ", y)
    raise e

print("Wird nie ausgeführt")
```

```
Division durch 0 aufgetreten
Fehlerbeschreibung:  division by zero
x =  7
y =  0
```

```
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
Cell In[28], line 11
      9     print("x = ", x)
     10     print("y = ", y)
---> 11     raise e
     13 print("Wird nie ausgeführt")

Cell In[28], line 5
      3     x = 7
      4     y = 0
----> 5     print(x/y)
      6 except ZeroDivisionError as e:
      7     print("Division durch 0 aufgetreten")

ZeroDivisionError: division by zero
```

Eine solche `raise`-Anweisung verlässt den gesamten `try-except`-Block, d.h. auch nachgelagerte, passende `except`-Blöcke werden ignoriert. Ein äußerer `try`-Block dagegen kann die "re-raised"-Anweisung immernoch auffangen:

In [29]:
```
# Re-Raise auffangen
try:
    try:
        print(1/0)
    except ArithmeticError as ae:
        print("Arithmetic Unit Error")
        raise       # verlässt die gesamte innere Struktur
    except ZeroDivisionError as zde:
        print("Innerer Block wird übersprungen")
except ZeroDivisionError as zde:
    print("Äußerer Block wird behandelt")
```

```
Arithmetic Unit Error
Äußerer Block wird behandelt
```

6.2.2 Eigene Ausnahmen

Man kann auch eigene Exception-Klassen definieren. (Klassen werden jedoch erst später behandelt)

In [30]:
```python
# TODO Eigene Exception-Klasse
class LatinException(Exception):
    pass

raise LatinException("Errare humanum est")
```

```
---------------------------------------------------------------------------
LatinException                            Traceback (most recent call last)
Cell In[30], line 5
      2 class LatinException(Exception):
      3     pass
----> 5 raise LatinException("Errare humanum est")

LatinException: Errare humanum est
```

6.2.3 Testen von Vor- und Nachbedingungen

Eine weitere Technik, um das Fehlerrisiko bei Python-Programmen zu reduzieren, nennt man *Testen von Vor- und Nachbedingungen*. Eine Funktion arbeit in der Regel nur dann korrekt, wenn die übergebenen Argumente bestimmte Bedingungen erfüllen. Die Konjunktion (*und*-Verknüpfung) nennt man *Vorbedingung*.

Entsprechend gibt es auch *Nachbedingungen*. Sie definieren das, was die Funktion leisten soll. Ist die Nachbedingung bei erfüllter Vorbedingung ebenfalls erfüllt, arbeitet die Funktion korrekt.

In Python lassen sich Vor- und Nachbedingungen mithilfe der `assert` -Anweisung testen. Diese sichert zu, dass eine bestimmte Bedingung erfüllt ist.

```
assert bedingung [, fehlermeldung]
```
dies entspricht in etwa folgendem Syntax:

```
if not bedingung:
    raise AssertionError(fehlermeldung)
```
Damit ist die `assert` -Anweisung quasi eine bedingte `raise` -Anweisung. Sie sollte nicht zum Auffangen von Programmfehlern wie `x / 0` verwendet werden, weil Python diese selbst erkennt, sondern um benutzerdefinierte und semantische Einschränkungen aufzufangen.

In [35]:
```python
# TODO Testen auf Vor- und Nachbedinungen anhand fiblist()
# gibt eine Liste der erste n Fibonacci-Zahlen
def fiblist(n):
    # Prüfe Vorbedindung
    assert n > 1, "Zahl größer 1 erforderlich!"
    fib = [0, 1]
    for i in range(1, n-1):
```

```
        fib += [fib[-1] + fib[-2]]
    # Prüfe Nachbedinung
    assert len(fib) == n, f"es wurden {len(fib)} anstatt {n} Fibonaccizahlen erz
    return fib

try:
    print(fiblist(-3))
except Exception as e:
    print(type(e),e)
```

```
<class 'AssertionError'> Zahl größer 1 erforderlich!
```

6.3 Fehlerinformationen

Die genauen Fehlerinformationen kann man sich mit der Methode `exc_info()` des `sys`-Moduls anzeigen lassen:

In [39]:
```
# Fehlerinformationen anzeigen lassen
import sys

try:
    i = int("Hello")
except Exception:
    (type, value, traceback) = sys.exc_info()
    print("Unexpected Error:")
    print("Type: ", type)
    print("Value: ", value)
    print("Traceback: ", traceback)
```

```
Unexpected Error:
Type:  <class 'ValueError'>
Value:  invalid literal for int() with base 10: 'Hello'
Traceback:  <traceback object at 0x0000017002119700>
```

7. Modularisierung

Modulare Programmierung ist eine Software-Designtechnik. Unter modularem Design versteht man, ein komplexes System in kleinere selbständige Einheiten und Komponenten zu zerlegen. Diese Komponenten bezeichnet man als *Module*. Ein Modul kann unabhängig vom Gesamtsystem erzeugt und getestet werden, und in den meisten Fällen auch in anderen Systemen verwendet werden.

In Python unterscheidet man zwei Arten von Modulen:

- Bibliotheken (*Libraries*): Stellen Datentypen oder Funktionen für alle Python-Programme bereit, hierbei gibt es:
  - die umfangreiche Standardbibliothek
  - eigene Module
  - Module von Drittanbietern
- lokale Module: nur für ein Programm verfügbar

7.1 Modularten

Beim Einbinden eines Moduls sucht Python nach allen Dateien, die für Python importierbar sind (→ `sys.path`). Importiert werden können folgende Dateitypen:

- In Python geschriebene Module: `.py` (normaler Quellcode), `.pyc` (Bytecode), `.pyo` (optimierter Bytecode)
- dynamisch geladene C-Module: `.pyd`, `.dll` (DLLs unter Windows) und `.so` (dynamische Bibliotheken unter Linux/Unix-Systemen).

Weiterhin können *Packages*, die aus einem Ordner bestehen und importierbare Dateien enthalten, eingebunden werden. Diese Pakete kann man allgemein in der Form `import ordnername` ansprechen.

7.2 Einbinden von Modulen

Das Einbinden von Modulen spielt eine wichtige Rolle in Python. Man hat mehrere Möglichkeiten, Module zu importieren, wobei man ziemlich genau festlegen kann, was man genau importieren möchte.

- `import modul`: Das angegebene Modul wird vollständig in den aktuellen Namensraum aufgenommen und kann unter dem vollen Namen (*fully qualified*) „modul“ angesprochen werden.

In [41]:
```
# TODO Beispiel 1
import math
print(math.sin(1.234))
print(sin(1.234))
```

```
0.9438182093746337
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[41], line 4
      2 import math
      3 print(math.sin(1.234))
----> 4 print(sin(1.234))

NameError: name 'sin' is not defined
```

- `import modul as neuername`: Das angegebene Modul wird vollständig in den aktuellen Namensraum aufgenommen. Das Modul ist aber nicht mehr unter dem Namen „modul“ ansprechbar, sondern als „neuername“. Damit wird also das Modul für den internen Gebrauch umbenannt. Das eignet sich besonders, wenn man sich Schreibarbeit ersparen möchte.

In [42]:
```
# TODO Beispiel 2
import math as m
print(m.sin(1.234))
```

```
0.9438182093746337
```

- `from modul import name` : Mit diesem Aufruf wird aus dem Modul „modul“ der Teil „name“ importiert. Danach kann auch nur das Importierte als „name“ angesprochen werden.

In [44]:
```
# TODO Beispiel 3
from math import sin, pi
print(sin(pi))
print(cos(pi))
```

```
1.2246467991473532e-16
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[44], line 4
      2 from math import sin, pi
      3 print(sin(pi))
----> 4 print(cos(pi))

NameError: name 'cos' is not defined
```

- `from modul import name as neuername` : Mit diesem Aufruf wird aus dem Modul „modul“ der Teil „name“ unter „neuername“ importiert. Danach kann es mit „neuername“ angesprochen werden.

In [45]:
```
# TODO Beispiel 4
from math import cos as c
print(c(pi))
```

```
-1.0
```

- `from modul import *` : Die letzte Option ist ein Stern-Import. Der Stern dient als Platzhalter und signalisiert, dass das betreffende Modul vollständig importiert werden soll. Auf den ersten Blick ist das zwar bequem, weil man den Modulnamen nicht mehr vor die Funktionen usw. schreiben muss. Nachteilig ist dabei, dass bestehende Objekte (Funktionen, Klassen etc.) unbemerkt überschrieben werden können.

In [50]:
```
# TODO Beispiel 5
pi = 3
print(pi)
from math import *
print(pi)
pi = 4
print(pi)
print(tan(pi))
dir(math)
degrees(pi)
```

```
3
3.141592653589793
4
1.1578212823495777
```

Out[50]: 229.1831180523293

7.3 Inhalt eines Moduls

Mit der built-in Funktion `dir()` kann man sich die in einem Modul definierten Namen anzeigen lassen:

In [1]:
```
#from math import *
dir()
```

Out[1]:
```
['In',
 'Out',
 '_',
 '__',
 '___',
 '__builtin__',
 '__builtins__',
 '__doc__',
 '__loader__',
 '__name__',
 '__package__',
 '__spec__',
 '_dh',
 '_i',
 '_i1',
 '_ih',
 '_ii',
 '_iii',
 '_oh',
 'exit',
 'get_ipython',
 'open',
 'quit']
```

Ohne Argumente liefert `dir()` die Namen, die in den Namensraum geladen wurden. Je nachdem kann die Ausgabe dieser Methode variieren.

In [ ]:
```
# zuerst Kernel restarten
#import math
dir()
```

7.4 Eigene Module

In Python ist es extrem einfach, eigene Module zu schreiben. Viele tun es, ohne es zu wissen, denn jedes Python-Programm ist automatisch auch ein Modul.

In dem Python-File `fibonacci.py` befinden sich folgende Funktionen:

```
def fib(n):
    a, b = 0, 1
    for i in range(n):
        a, b = b, a + b
    return a

def fiblist(n):
    fib = [0, 1]
    for i in range(1, n):
```

```
            fib += [fib[-1] + fib[-2]]
        return fib
```

In [12]:
```
# TODO eigenes Modul fibonacci.py importieren
import fibonacci
print(fibonacci.fib(10))
fibonacci.fiblist(10)
```

```
55
```

Out[12]:
```
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55]
```

In [10]:
```
# Code um Module zu reloaden
import importlib as imp
imp.reload(fibonacci)
```

Out[10]:
```
<module 'fibonacci' from 'C:\\Users\\flori\\iCloudDrive\\Vorlesungen\\Python\\Workbooks\\06\\fibonacci.py'>
```

Dokumentation eigener Module

In [11]:
```
# Aufruf des pydoc-Moduls
help(fibonacci)
```

```
Help on module fibonacci:

NAME
    fibonacci - Modul mit wichtigen Funktionen zur Fibonacci-Folge

FUNCTIONS
    fib(n)
        Die Fibonacci-Zahl für die n+1-te Generation iterativ berechnet

    fiblist(n)
        Produziert eine Lister der Fibonacci-Zahlen

FILE
    c:\users\flori\iclouddrive\vorlesungen\python\workbooks\06\fibonacci.py
```

7.5 Pakete (*Packages*)

Um Programme, die aus mehreren Modulen bestehen, weiterhin nachvollziehbar zu gestalten, stellt Python das Paketkonzept bereit. Damit kann man beliebig viele Module zu einem Paket "schnüren". Der dazu erforderliche Mechanismus ist denkbar einfach gelöst:

- Zuerst erzeugt man einen Unterordner in einem Verzeichnis, in dem der Python-Interpreter auch Module erwartet bzw. danach sucht.
- Im angelegten Ordner muss nun eine Datei mit dem Namen `__init__.py` angelegt werden. Diese Datei kann leer sein oder Initialisierungscode enthalten, der beim Einbinden des Pakets einmalig ausgeführt wird.

In [13]:
```
# Neuen Unterordner erzeugen
import os
try:
    os.mkdir("simple_package")
except:
    print("Bereits vorhanden")
```

In [21]:
```
# Erzeugen der init.py Datei
try:
    with open("simple_package/__init__.py", "w") as d:
        # erst vor Ausführung der letzten Zelle entkommentieren:
        #d.write("from simple_package import a, b")
        pass

except:
    print("Konnte nicht erzeugt werden")
```

In [15]:
```
# Erzeugen zwei einfacher Module
try:
    with open("simple_package/a.py", "w") as a:
        a.write("def f1():\n\t")
        a.write("print('Hallo, hier ist f1 von Modul a ')")
except:
    print("Modul a.py konnte nicht angelegt werden")

try:
    with open("simple_package/b.py", "w") as b:
        b.write("def foo():\n\t")
        b.write("print('Hallo, hier ist foo von Modul b ')")
except:
    print("Modul b.py konnte nicht angelegt werden")
```

In [16]:
```
# TODO Was nicht funktioniert
import simple_package
simple_package.a.f1()
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
Cell In[16], line 3
      1 # TODO Was nicht funktioniert
      2 import simple_package
----> 3 simple_package.a.f1()

AttributeError: module 'simple_package' has no attribute 'a'
```

In [24]:
```
# TODO Module aus Paket aufrufen
from simple_package import *
a.f1()
b.foo()
dir()
```

```
Hallo, hier ist f1 von Modul a
Hallo, hier ist foo von Modul b
```

Out[24]:
```
['In',
 'NamespaceMagics',
 'Out',
 '_',
 '_1',
 '_10',
 '_12',
 '_2',
 '_22',
 '_3',
 '_4',
 '_5',
 '_8',
 '_Jupyter',
 '__',
 '___',
 '__builtin__',
 '__builtins__',
 '__doc__',
 '__loader__',
 '__name__',
 '__package__',
 '__spec__',
 '_dh',
 '_getcontentof',
 '_getshapeof',
 '_getsizeof',
 '_i',
 '_i1',
 '_i10',
 '_i11',
 '_i12',
 '_i13',
 '_i14',
 '_i15',
 '_i16',
 '_i17',
 '_i18',
 '_i19',
 '_i2',
 '_i20',
 '_i21',
 '_i22',
 '_i23',
 '_i24',
 '_i3',
 '_i4',
 '_i5',
 '_i6',
 '_i7',
 '_i8',
 '_i9',
 '_ih',
 '_ii',
 '_iii',
 '_nms',
 '_oh',
 'a',
 'acos',
 'acosh',
```

```
'asin',
'asinh',
'atan',
'atan2',
'atanh',
'b',
'cbrt',
'ceil',
'comb',
'copysign',
'cos',
'cosh',
'd',
'degrees',
'dist',
'e',
'erf',
'erfc',
'exit',
'exp',
'exp2',
'expm1',
'fabs',
'factorial',
'fibonacci',
'floor',
'fmod',
'frexp',
'fsum',
'gamma',
'gcd',
'get_ipython',
'getsizeof',
'hypot',
'imp',
'inf',
'isclose',
'isfinite',
'isinf',
'isnan',
'isqrt',
'json',
'lcm',
'ldexp',
'lgamma',
'log',
'log10',
'log1p',
'log2',
'math',
'modf',
'nan',
'nextafter',
'np',
'open',
'os',
'perm',
'pi',
'pow',
'prod',
```

```
 'quit',
 'radians',
 'remainder',
 'simple_package',
 'simple_package_v2',
 'sin',
 'sinh',
 'sqrt',
 'tan',
 'tanh',
 'tau',
 'trunc',
 'ulp',
 'var_dic_list']
```

In [22]:
```
# TODO nachdem die __init__.py angepasst wurde
imp.reload(simple_package)
import simple_package as simple_package_v2
simple_package_v2.a.f1()
```

Out[22]:
```
<module 'simple_package' from 'C:\\Users\\flori\\iCloudDrive\\Vorlesungen\\Pyth
on\\Workbooks\\06\\simple_package\\__init__.py'>
```