*Empfohlene Unterrichtsinhalte als Vorbereitung für dieses Thema: 03-02/3 (Bedingungen, Schleifen), 05-01 (Funktionen)*

# Einführung in die computerlinguistische Programmierung mit Python 📖

## 11-01: Rekursion 1 ↘️

Rekursion ist ein mächtiges Werkzeug in der Programmierung, dass in sich jedoch auch Gefahren bergen kann, falls ein Algorithmus unsauber programmiert ist. Wir bezeichnen eine Funktion als rekursiv, falls sie sich selbst aufruft.

In [ ]:
```
def my_weird_recursive_function():
    return my_weird_recursive_function()

my_weird_recursive_function()
```

Die Funktion oben tut nichts anderes als einen Funktionsaufruf zurückzugeben. Dabei wird keine Alternative zu diesem Selbstaufruf bzw. eine andere Abbruchbedingung gegeben, sodass die Funktion nach einmaligem Aufruf theoretisch für immer so weiter laufen könnte. Glücklicherweise erkennt Python jedoch solche Probleme und gibt nach gewisser Zeit und einer bestimmten Anzahl an rekursiven Aufrufen eine Fehlermeldung aus:

```
RecursionError: maximum recursion depth exceeded .
```

Rekursive Algorithmen sollten deshalb immer eine Bedingung enthalten, die die Kette der Selbstaufruf unterbricht.

### Anwendungsbeispiele Rekursion

Im folgenden sollen zwei typische Beispiele für rekursive Algorithmen aus der Mathematik besprochen werden, um das Grundprinzip der Rekursion besser verständlich zu machen.

#### Berechnung der Fakultät *n!* einer Zahl *n*

Die Berechnung der Fakultät einer natürlichen Zahl *n!*, zum Beispiel *5!= 5 4 3 2 1*, lässt sich gut mit Hilfe eines rekursiven Algorithmus berechen. Dabei unterscheidet man zunächst zwei Fälle:

- Entweder Entweder ist *n=0*. Dann gilt: *n!=1*.
- Oder *n* ist eine positive ganze Zahl. Dann gilt: $n! = n \cdot (n-1)!$

Den ersten Fall bezeichnen wir als den **Rekursionsanfang**. Der zweite Fall beschreibt einen **Rekursionsschritt**. Mit Hilfe dieser beiden Fälle lässt sich die Fakultät einer Zahl wie folgt ermitteln:

Für eine beliebige natürlich Zahl, die nicht *0* ist, gilt: Wir starten mit einem Rekursionsschritt:

```
i.)   n! = n * (n-1)!              (Rekursionsschritt)
```

Nun muss geprüft werden welchen Wert *n-1* darstellt. Falls *n != 0* , wird wieder ein Rekursionsschritt gemacht:

```
ii.)  n! = n * (n-1)!
         = n * (n-1 * (n-2)!)
         = n * n-1 * (n-2)!
```

Die Fallüberprüfung wie in ii.) geschieht so, oft bis wir nach *n* Rekursionsschritten die Fakultät *(n-n)! = 0!* brechnen müssen. Diese ist im Rekursionsanfang als *1* definiert.

```
iii.) n! = n * n-1 * n-2 * ... * n-(n-1) * (n-n)!
         = n * n-1 * n-2 * ... * n-(n-1) * 0!
         = n * n-1 * n-2 * ... * n-(n-1) * 1
         = n * n-1 * n-2 * ... * n-(n-2)
```

Wenn wir den Rekursionsanfang erreicht haben, kennen wir alle Faktoren, die zur Berechnung der Fakultät notwendig sind.

Für den Wert *n = 5* berechnet sich die Fakultät wie folgt:

```
5! = 5 * (5-1)!
   = 5 * ((5-1) * (5-2)!)
   = 5 * (5-1) * ((5-2) * (5-3)!)
   = 5 * (5-1) * (5-2) * ((5-3) * (5-4)!)
   = 5 * (5-1) * (5-2) * (5-3) * ((5-4) * (5-5)!)
   = 5 *   4   *   3   *   2   *    1   *   0!
   = 5 *   4   *   3   *   2   *    1   *   1
   = 120
```

In [ ]:
```python
def fac(n):
    if n == 0:
        return 1
    else:
        return n * fac(n-1)

print(fac(5))
```

In [ ]:
```python

```

Die Funktion `fac()` gibt `1` zurück falls ihr Argument `n` den Wert `0` hat. Andernfalls gibt sie ein Produkt zurück, `n * fac(n)` . Der zweite Faktor dieses Produkts wird durch den selbst Aufruf mit dem Argument `n-1` dargestellt, `fac(n-1)` . Das `if` -Statement stellt also den Rekursionsanfang dar. Das `else` -Statement wiederum den Rekursionsschritt. Da `fac()` entweder ein Produkt oder eine einzelne Zahl zurückgibt, führen die ineinander verschachtelten Aufrufe also in jedem Fall zur Rückgabe eines Produkts mit `n + 1` Faktoren. Nach erreichen des Rekursionsanfangs alle `return` -Statements ausgewertet werden können. Man kann sich die Auswertung der `return` -Statements für `n = 5` wie folgt vorstellen:

```
(return 5 *                     # Rekursionsschritt
    (return 4 *                 # Rekursionsschritt
        (return 3 *             # Rekursionsschritt
            (return 2 *         # Rekursionsschritt
                (return 1 *     # Rekursionsschritt
                    (return 1)  # Rekursionsanfang
                 )
             )
         )
     )
 )
```

Jede neue Zeile symbolisiert einen Aufruf von `fac()`. Die Einrückung und Klammern stellen die Verschachtelung und die Auswertungsreihenfolge dar.

## Primfaktorzerlegung

Ein weiteres Beispiel für eine Rekursiven Algorithmus stellt das Verfahren zur Primzahlzerlegung einer natürlichen Zahl dar. Dabei wird eine Zahl in ein Produkt dessen Faktoren aus der Menge der Primzahlen stammen. Bei der Zerlegung einer Zahl *n* können zwei Fälle auftreten:

- Die Zahl *n* ist selbst eine Primzahl, dieser Fall stellt den Rekursionsanfang dar
- Die Zahl *n* lässt sich ganzzahlig, ohne Rest durch eine Primzahl *p* teilen. Falls dies der Fall ist, muss überprüft werden, ob sich wiederum der Quotient *n:p* in Primfaktoren zerlegen lässt. In jedem Fall ist *p* einer der Primfaktoren der Zahl *n*.

Mit Hilfe der o.g. Fallunterscheidung lassen sich alle Primfaktoren einer Zahl sammeln. Natürlich, kann diese Sammlung von Faktoren auch Wiederholungen enthalten (z.B $8 = 2^3$). *Natürlich setzt die Primfaktorzerlegung die Kenntnis der Primzahlen in dem jeweils relevanten Intervall [0, n]* voraus. Dies ist insbesondere für sehr große Zahlen kein triviales Problem.

Ein Algorithmus zur Primfaktorzerlegung kann wie folgt umgesetzt werden:

In [ ]:
```
prime_numbers = (2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37,
41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89,
97, 101, 103, 107, 109, 113, 127, 131, 137, 139, 149, 151,
157, 163, 167, 173, 179, 181, 191, 193, 197, 199, 211, 223,
227, 229, 233, 239, 241, 251, 257, 263, 269, 271, 277, 281,
283, 293, 307, 311, 313, 317, 331, 337, 347, 349, 353, 359,
367, 373, 379, 383, 389, 397, 401, 409, 419, 421, 431, 433,
439, 443, 449, 457, 461, 463, 467, 479, 487, 491, 499, 503,
509, 521, 523, 541, 547, 557, 563, 569, 571, 577, 587, 593,
599, 601, 607, 613, 617, 619, 631, 641, 643, 647, 653, 659,
661, 673, 677, 683, 691, 701, 709, 719, 727, 733, 739, 743,
751, 757, 761, 769, 773, 787, 797, 809, 811, 821, 823, 827,
829, 839, 853, 857, 859, 863, 877, 881, 883, 887, 907, 911,
919, 929, 937, 941, 947, 953, 967, 971, 977, 983, 991, 997)


def prime_factors(number):
    if number in prime_numbers:
        return (number, )
    for n in prime_numbers:
        if number % n == 0:
            return (n, ) + prime_factors(number//n)


print(prime_factors(936))
```

Mit den hier gelisteten Primzahlen lassen sich nur für die Zahlen im Intervall [0,1000] zuverlässig Primfaktoren finden, da die Primzahlen nur nachgeschlagen werden und kein anderer Primzahl-Test angewendet wird. Es wird nach und nach mit möglichst kleinen Primzahlen getestet, ob es möglich ist die Zahl `number` ganzzahlig, ohne Rest zu teilen. Falls dies der Fall ist, wird der Quotinet aus `number` und dem gefundenen Primfaktor an einen weiteren Aufruf von `prime_factors()` übergeben. Die gefundenen Primfaktoren werden im Anschluss als Tupel zurückgegeben und mit den Rückgaben der weiteren Aufrufe konkateniert.

## Rekursive Datenstrukturen

Einer der wichtigen Einsatzzwecke für Rekursion ist das Verarbeiten von rekursiven Datenstrukturen. Eine rekursive Datenstruktur ist eine Datenstruktur, die wiederum Datenstrukturen desselben Typs enthalten kann. Python-Listen können zum Beispiel rekursive Datenstrukturen darstellen, da sie selbst Python-Listen als Elemente enthalten können.

In [ ]:
```
my_recursive_list = ["I", ["saw", ["the", "man"],["with", ["the", "telescop

print(my_recursive_list)
```

Nehmen wir an wir möchten aus einer Struktur wie oben, auschließlich Strings ausgeben lassen.

In [1]:
```
def print_recursive_structures(some_list):
    for element in some_list:
        if type(element) is str:
            print(element)
        elif type(element) is list:
            print_recursive_structures(element)
        else:
            print("Sorry, I can't do that Dave")


my_recursive_list = ["I", ["saw", ["the", "man"],["with", ["the", "telescop
print_recursive_structures(my_recursive_list)
```

```
I
saw
the
man
with
the
telescope
```

Die Funktion oben löst die Aufgabe indem sie zunächst prüft, ob das aktuelle Listenelement den richtigen Typ hat und gibt entweder einen String aus oder startet einen Rekursionsschritt.

## Rekursion vs. Iteration

Wir benutzen rekursive Funktionen, wenn diese Lösungsmethode einen bekannten Algorithmus besser abbildet bzw. falls die Lösung eines Problems mit Hilfe von Rekursion intuitiver bzw. weniger aufwendig darzustellen ist als eine iterative Lösung. Generell sind rekursive Algorithmen für den Python-Interpreter aufwändiger als iterative, weil verschachtelte Funktionsaufrufe aufwändigere Speicherverwaltungsschritte notwendig machen. Das sollte uns aber nicht davon abhalten, einen Rekursion zu nutzen, wenn diese besser verständlicher und einfacher zu programmieren ist.

## Zusammenfassung

- Mit Hilfe rekursiver Funktionen können wir bekannte rekursive Algorithmen in Python abbilden.
- Rekursive Funktionen rufen sich im Rekursionschritt selbst auf.
- Mit jedem Rekursionsschritt nähert sich die Funktion der Rekursionsanfangsbedingung an.
- Rekursive Datenstrukturen können mit rekursiven Funktionen verarbeitet und erstellt werden.

## Weitere Themen dieser Woche

- Rekursion 2 (mit Debugger!) ↗️