合肥大學
HEFEI UNIVERSITY

Programming with Python

26. Funktionen definieren und aufrufen

Thomas Weise (汤卫思)
tweise@hfuu.edu.cn

Institute of Applied Optimization (IAO) | 应用优化研究所
School of Artificial Intelligence and Big Data | 人工智能与大数据学院
Hefei University | 合肥大学
Hefei, Anhui, China | 中国安徽省合肥市

Version: 2026-01-29

Programming with Python

Dies ist ein Kurs über das Programmieren mit der Programmiersprache Python an der Universität Hefei (合肥大学).

Die Webseite mit dem Lehrmaterial dieses Kurses ist https://thomasweise.github.io/programmingWithPython (siehe auch den QR-Kode unten rechts). Dort können Sie das Kursbuch (in Englisch) und diese Slides finden. Das Repository mit den Beispielprogrammen in Python finden Sie unter https://github.com/thomasWeise/programmingWithPythonCode.

Outline

Einleitung

Einleitung

- Funktionen sind Blöcke von Kode, die von anderen Orten in Programmen aus aufgerufen werden können.

Einleitung

- Funktionen sind Blöcke von Kode, die von anderen Orten in Programmen aus aufgerufen werden können.
- Sie haben bereits mehrere Funktionen kennengelernt, von der `print` bis `sqrt`.

Einleitung

- Funktionen sind Blöcke von Kode, die von anderen Orten in Programmen aus aufgerufen werden können.
- Sie haben bereits mehrere Funktionen kennengelernt, von der `print` bis `sqrt`.
- Wir unterscheiden die Definition und den Aufruf einer Funktion.

Einleitung

- Funktionen sind Blöcke von Kode, die von anderen Orten in Programmen aus aufgerufen werden können.
- Sie haben bereits mehrere Funktionen kennengelernt, von der `print` bis `sqrt`.
- Wir unterscheiden die Definition und den Aufruf einer Funktion.
- In der Funktionsdefinition spezifizieren wir den Name, die Parameter, den Rückgabewert, und den Körper (also den eigentlichen Kode) der Funktion.

Einleitung

- Funktionen sind Blöcke von Kode, die von anderen Orten in Programmen aus aufgerufen werden können.
- Sie haben bereits mehrere Funktionen kennengelernt, von der `print` bis `sqrt`.
- Wir unterscheiden die Definition und den Aufruf einer Funktion.
- In der Funktionsdefinition spezifizieren wir den Name, die Parameter, den Rückgabewert, und den Körper (also den eigentlichen Kode) der Funktion.
- Eine Funktion kann dann überall in unserem Kode über ihren Namen aufgerufen werden, wobei dann Werte für die Parameter übergeben und gegebenenfalls der Rückgabewert z. B. in einer Variable gespeichert wird.

Einleitung

- Funktionen sind Blöcke von Kode, die von anderen Orten in Programmen aus aufgerufen werden können.
- Sie haben bereits mehrere Funktionen kennengelernt, von der `print` bis `sqrt`.
- Wir unterscheiden die Definition und den Aufruf einer Funktion.
- In der Funktionsdefinition spezifizieren wir den Name, die Parameter, den Rückgabewert, und den Körper (also den eigentlichen Kode) der Funktion.
- Eine Funktion kann dann überall in unserem Kode über ihren Namen aufgerufen werden, wobei dann Werte für die Parameter übergeben und gegebenenfalls der Rückgabewert z. B. in einer Variable gespeichert wird.
- Jetzt wollen wir unsere eigenen Funktionen definieren[22].

Funktionen Definieren

Funktionen Definieren

- Die Definition einer Funktion beginnt mit dem Schlüsselwort `def`, gefolgt von dem Funktionsnamen, eventuell Parametern in Klammern, einem Rückgabewert-Type-Hint, dem Doppelpunkt `:`, und dann dem mit vier Leerzeichen eingerückten Funktionskörper.

Funktionen Definieren

- Die Definition einer Funktion beginnt mit dem Schlüsselwort `def`, gefolgt von dem Funktionsnamen, eventuell Parametern in Klammern, einem Rückgabewert-Type-Hint, dem Doppelpunkt `:`, und dann dem mit vier Leerzeichen eingerückten Funktionskörper.

```
"""The syntax of function definitions and function calls."""

def my_function(param_1: type_hint, param_2: type_hint) -> result_type:
    body of function 1
    body of function 2
    return result  # if result_type is not None we return something


normal statement 1  # some random code outside of the function
normal statement 2
my_function(argument_1, argument_2)  # We call the function like this.
```

Funktionen Definieren

- Wir haben also folgende Syntax:

Gute Praxis

Die Namen von Funktionen werden mit Kleinbuchstaben geschrieben, wobei Worte mit Unterstrichen getrennt werden[32].

```
"""The syntax of function definitions and function calls."""

def my_function(param_1: type_hint, param_2: type_hint) -> result_type:
    body of function 1
    body of function 2
    return result  # if result_type is not None we return something


normal statement 1  # some random code outside of the function
normal statement 2
my_function(argument_1, argument_2)  # We call the function like this.
```

Funktionen Definieren

- Wir haben also folgende Syntax:
- Funktionsnamen werden mit Kleinbuchstaben geschrieben, Worte mit Unterstrichen getrennt.

```python
"""The syntax of function definitions and function calls."""

def my_function(param_1: type_hint, param_2: type_hint) -> result_type:
    body of function 1
    body of function 2
    return result  # if result_type is not None we return something


normal statement 1  # some random code outside of the function
normal statement 2
my_function(argument_1, argument_2)  # We call the function like this.
```

Funktionen Definieren

- Wir haben also folgende Syntax:
- Funktionsnamen werden mit Kleinbuchstaben geschrieben, Worte mit Unterstrichen getrennt.
- Nach den Funktionsnamen folgen eine öffnende und eine schließende runde Klammer.

```
"""The syntax of function definitions and function calls."""

def my_function(param_1: type_hint, param_2: type_hint) -> result_type:
    body of function 1
    body of function 2
    return result  # if result_type is not None we return something


normal statement 1  # some random code outside of the function
normal statement 2
my_function(argument_1, argument_2)  # We call the function like this.
```

Funktionen Definieren

- Wir haben also folgende Syntax:
- Funktionsnamen werden mit Kleinbuchstaben geschrieben, Worte mit Unterstrichen getrennt.
- Nach den Funktionsnamen folgen eine öffnende und eine schließende runde Klammer.
- Eine Funktion kann Parameter haben, also Werte, die ihr bei der Ausführung übergeben werden.

```python
"""The syntax of function definitions and function calls."""

def my_function(param_1: type_hint, param_2: type_hint) -> result_type:
    body of function 1
    body of function 2
    return result  # if result_type is not None we return something


normal statement 1  # some random code outside of the function
normal statement 2
my_function(argument_1, argument_2)  # We call the function like this.
```

Funktionen Definieren

- Funktionsnamen werden mit Kleinbuchstaben geschrieben, Worte mit Unterstrichen getrennt.
- Nach den Funktionsnamen folgen eine öffnende und eine schließende runde Klammer.
- Eine Funktion kann Parameter haben, also Werte, die ihr bei der Ausführung übergeben werden.
- Im Falle der Funktion `print` war das der String, der ausgegeben werden soll.

```python
"""The syntax of function definitions and function calls."""

def my_function(param_1: type_hint, param_2: type_hint) -> result_type:
    body of function 1
    body of function 2
    return result  # if result_type is not None we return something


normal statement 1  # some random code outside of the function
normal statement 2
my_function(argument_1, argument_2)  # We call the function like this.
```

Funktionen Definieren

- Nach den Funktionsnamen folgen eine öffnende und eine schließende runde Klammer.
- Eine Funktion kann Parameter haben, also Werte, die ihr bei der Ausführung übergeben werden.
- Im Falle der Funktion `print` war das der String, der ausgegeben werden soll.
- Im Falle der Funktion `sqrt` war das die Zahl, deren Wurzel berechnet werden sollte.

```python
"""The syntax of function definitions and function calls."""

def my_function(param_1: type_hint, param_2: type_hint) -> result_type:
    body of function 1
    body of function 2
    return result  # if result_type is not None we return something


normal statement 1  # some random code outside of the function
normal statement 2
my_function(argument_1, argument_2)  # We call the function like this.
```

Funktionen Definieren

- Eine Funktion kann Parameter haben, also Werte, die ihr bei der Ausführung übergeben werden.
- Im Falle der Funktion `print` war das der String, der ausgegeben werden soll.
- Im Falle der Funktion `sqrt` war das die Zahl, deren Wurzel berechnet werden sollte.
- In der Funktion werden die Parameter wie Variablen verwendet.

```
"""The syntax of function definitions and function calls."""

def my_function(param_1: type_hint, param_2: type_hint) -> result_type:
    body of function 1
    body of function 2
    return result  # if result_type is not None we return something


normal statement 1  # some random code outside of the function
normal statement 2
my_function(argument_1, argument_2)  # We call the function like this.
```

Funktionen Definieren

- Im Falle der Funktion `print` war das der String, der ausgegeben werden soll.
- Im Falle der Funktion `sqrt` war das die Zahl, deren Wurzel berechnet werden sollte.
- In der Funktion werden die Parameter wie Variablen verwendet.
- Die Werte dieser Variablen werden allerdings der Funktion bei ihrer Ausführung übergeben.

```python
"""The syntax of function definitions and function calls."""

def my_function(param_1: type_hint, param_2: type_hint) -> result_type:
    body of function 1
    body of function 2
    return result  # if result_type is not None we return something


normal statement 1  # some random code outside of the function
normal statement 2
my_function(argument_1, argument_2)  # We call the function like this.
```

Funktionen Definieren

- Im Falle der Funktion `sqrt` war das die Zahl, deren Wurzel berechnet werden sollte.
- In der Funktion werden die Parameter wie Variablen verwendet.
- Die Werte dieser Variablen werden allerdings der Funktion bei ihrer Ausführung übergeben.
- Sie stammen also vom Kode, der die Funktion aufruft.

```
"""The syntax of function definitions and function calls."""

def my_function(param_1: type_hint, param_2: type_hint) -> result_type:
    body of function 1
    body of function 2
    return result  # if result_type is not None we return something


normal statement 1  # some random code outside of the function
normal statement 2
my_function(argument_1, argument_2)  # We call the function like this.
```

Funktionen Definieren

- Die Werte dieser Variablen werden allerdings der Funktion bei ihrer Ausführung übergeben.
- Sie stammen also vom Kode, der die Funktion aufruft.

Definition: Parameter

Ein Funktionsparameter ist eine Variable, die in einer Funktion definiert ist, ihren Wert aber vom aufrufenden Kode erhält.

```
"""The syntax of function definitions and function calls."""

def my_function(param_1: type_hint, param_2: type_hint) -> result_type:
    body of function 1
    body of function 2
    return result  # if result_type is not None we return something


normal statement 1  # some random code outside of the function
normal statement 2
my_function(argument_1, argument_2)  # We call the function like this.
```

Funktionen Definieren

- Die Werte dieser Variablen werden allerdings der Funktion bei ihrer Ausführung übergeben.
- Sie stammen also vom Kode, der die Funktion aufruft.
- Die Parameter werden zwischen der öffnenden und der schließenden Klammer im Funktionskopf definiert.

```python
"""The syntax of function definitions and function calls."""

def my_function(param_1: type_hint, param_2: type_hint) -> result_type:
    body of function 1
    body of function 2
    return result  # if result_type is not None we return something


normal statement 1  # some random code outside of the function
normal statement 2
my_function(argument_1, argument_2)  # We call the function like this.
```

Funktionen Definieren

- Die Werte dieser Variablen werden allerdings der Funktion bei ihrer Ausführung übergeben.
- Sie stammen also vom Kode, der die Funktion aufruft.
- Die Parameter werden zwischen der öffnenden und der schließenden Klammer im Funktionskopf definiert.
- Jeder Parameter hat einen Namen, unter dem wir dann in der Funktion auf seinen Wert zugreifen können.

```python
"""The syntax of function definitions and function calls."""

def my_function(param_1: type_hint, param_2: type_hint) -> result_type:
    body of function 1
    body of function 2
    return result  # if result_type is not None we return something


normal statement 1  # some random code outside of the function
normal statement 2
my_function(argument_1, argument_2)  # We call the function like this.
```

Funktionen Definieren

- Die Werte dieser Variablen werden allerdings der Funktion bei ihrer Ausführung übergeben.
- Sie stammen also vom Kode, der die Funktion aufruft.
- Die Parameter werden zwischen der öffnenden und der schließenden Klammer im Funktionskopf definiert.
- Jeder Parameter hat einen Namen, unter dem wir dann in der Funktion auf seinen Wert zugreifen können.
- Die Parameter sind durch Kommas getrennt.

```python
"""The syntax of function definitions and function calls."""

def my_function(param_1: type_hint, param_2: type_hint) -> result_type:
    body of function 1
    body of function 2
    return result  # if result_type is not None we return something


normal statement 1  # some random code outside of the function
normal statement 2
my_function(argument_1, argument_2)  # We call the function like this.
```

Funktionen Definieren

- Sie stammen also vom Kode, der die Funktion aufruft.
- Die Parameter werden zwischen der öffnenden und der schließenden Klammer im Funktionskopf definiert.
- Jeder Parameter hat einen Namen, unter dem wir dann in der Funktion auf seinen Wert zugreifen können.
- Die Parameter sind durch Kommas getrennt.
- Wir können keine oder beliebig viele Parameter definieren.

```
"""The syntax of function definitions and function calls."""

def my_function(param_1: type_hint, param_2: type_hint) -> result_type:
    body of function 1
    body of function 2
    return result  # if result_type is not None we return something


normal statement 1  # some random code outside of the function
normal statement 2
my_function(argument_1, argument_2)  # We call the function like this.
```

Funktionen Definieren

- Die Parameter werden zwischen der öffnenden und der schließenden Klammer im Funktionskopf definiert.
- Jeder Parameter hat einen Namen, unter dem wir dann in der Funktion auf seinen Wert zugreifen können.
- Die Parameter sind durch Kommas getrennt.
- Wir können keine oder beliebig viele Parameter definieren.
- Genau wie Variablen sollten Parameter mit Type Hints annotiert werden[34].

```python
"""The syntax of function definitions and function calls."""

def my_function(param_1: type_hint, param_2: type_hint) -> result_type:
    body of function 1
    body of function 2
    return result  # if result_type is not None we return something


normal statement 1  # some random code outside of the function
normal statement 2
my_function(argument_1, argument_2)  # We call the function like this.
```

Funktionen Definieren

- Wir können keine oder beliebig viele Parameter definieren.
- Genau wie Variablen sollten Parameter mit Type Hints annotiert werden[34].
- Wir wollen ja schließlich, dass die Benutzer unserer Funktion genau verstehen, ob sie Ganzzahlen, Strings, oder Fließkommazahlen an unsere Funktion übergeben können oder nicht.

```python
"""The syntax of function definitions and function calls."""

def my_function(param_1: type_hint, param_2: type_hint) -> result_type:
    body of function 1
    body of function 2
    return result  # if result_type is not None we return something


normal statement 1  # some random code outside of the function
normal statement 2
my_function(argument_1, argument_2)  # We call the function like this.
```

Funktionen Definieren

- Genau wie Variablen sollten Parameter mit Type Hints annotiert werden[34].
- Wir wollen ja schließlich, dass die Benutzer unserer Funktion genau verstehen, ob sie Ganzzahlen, Strings, oder Fließkommazahlen an unsere Funktion übergeben können oder nicht.
- Wenn wir eine Funktion `add(value_1, value_2)` definieren, dann ist erstmal gar nicht klar, was für Datentypen für `value_1` und `value_2` in Frage kommen.

```python
"""The syntax of function definitions and function calls."""

def my_function(param_1: type_hint, param_2: type_hint) -> result_type:
    body of function 1
    body of function 2
    return result  # if result_type is not None we return something


normal statement 1  # some random code outside of the function
normal statement 2
my_function(argument_1, argument_2)  # We call the function like this.
```

Funktionen Definieren

- Wenn wir eine Funktion `add(value_1, value_2)` definieren, dann ist erstmal gar nicht klar, was für Datentypen für `value_1` und `value_2` in Frage kommen.
- Definieren wir die Funktion dagegen mit Type Hints, z. B. `add(value_1: int, value_2: int)`, dann ist es offensichtlich.

```python
"""The syntax of function definitions and function calls."""

def my_function(param_1: type_hint, param_2: type_hint) -> result_type:
    body of function 1
    body of function 2
    return result  # if result_type is not None we return something


normal statement 1  # some random code outside of the function
normal statement 2
my_function(argument_1, argument_2)  # We call the function like this.
```

Funktionen Definieren

- Wenn wir eine Funktion `add(value_1, value_2)` definieren, dann ist erstmal gar nicht klar, was für Datentypen für `value_1` und `value_2` in Frage kommen.
- Definieren wir die Funktion dagegen mit Type Hints, z. B. `add(value_1: int, value_2: int)`, dann ist es offensichtlich.
- Funktionen können Ergebnisse zurückliefern.

```python
"""The syntax of function definitions and function calls."""

def my_function(param_1: type_hint, param_2: type_hint) -> result_type:
    body of function 1
    body of function 2
    return result  # if result_type is not None we return something


normal statement 1  # some random code outside of the function
normal statement 2
my_function(argument_1, argument_2)  # We call the function like this.
```

Funktionen Definieren

- Wenn wir eine Funktion `add(value_1, value_2)` definieren, dann ist erstmal gar nicht klar, was für Datentypen für `value_1` und `value_2` in Frage kommen.
- Definieren wir die Funktion dagegen mit Type Hints, z. B. `add(value_1: int, value_2: int)`, dann ist es offensichtlich.
- Funktionen können Ergebnisse zurückliefern.
- Die `sqrt`-Funktion, z. B. liefert die Quadratwurzel zurück.

```python
"""The syntax of function definitions and function calls."""

def my_function(param_1: type_hint, param_2: type_hint) -> result_type:
    body of function 1
    body of function 2
    return result  # if result_type is not None we return something


normal statement 1  # some random code outside of the function
normal statement 2
my_function(argument_1, argument_2)  # We call the function like this.
```

Funktionen Definieren

- Definieren wir die Funktion dagegen mit Type Hints, z. B. `add(value_1: int, value_2: int)`, dann ist es offensichtlich.
- Funktionen können Ergebnisse zurückliefern.
- Die `sqrt`-Funktion, z. B. liefert die Quadratwurzel zurück.
- Die Funktion `print` liefert nichts zurück.

```python
"""The syntax of function definitions and function calls."""

def my_function(param_1: type_hint, param_2: type_hint) -> result_type:
    body of function 1
    body of function 2
    return result  # if result_type is not None we return something


normal statement 1  # some random code outside of the function
normal statement 2
my_function(argument_1, argument_2)  # We call the function like this.
```

Funktionen Definieren

- Funktionen können Ergebnisse zurückliefern.
- Die `sqrt`-Funktion, z. B. liefert die Quadratwurzel zurück.
- Die Funktion `print` liefert nichts zurück.
- Wenn Funktionen ein Ergebnis zurückliefern, dann sollte der Typ dieses Rückgabewertes mit einem Type Hint spezifiziert werden.

```python
"""The syntax of function definitions and function calls."""

def my_function(param_1: type_hint, param_2: type_hint) -> result_type:
    body of function 1
    body of function 2
    return result  # if result_type is not None we return something


normal statement 1  # some random code outside of the function
normal statement 2
my_function(argument_1, argument_2)  # We call the function like this.
```

Funktionen Definieren

- Funktionen können Ergebnisse zurückliefern.
- Die `sqrt`-Funktion, z. B. liefert die Quadratwurzel zurück.
- Die Funktion `print` liefert nichts zurück.
- Wenn Funktionen ein Ergebnis zurückliefern, dann sollte der Typ dieses Rückgabewertes mit einem Type Hint spezifiziert werden.
- Nach der schließenden Klammer im Funktionskopf folgt dieser Datentyp dann als

```python
"""The syntax of function definitions and function calls."""

def my_function(param_1: type_hint, param_2: type_hint) -> result_type:
    body of function 1
    body of function 2
    return result  # if result_type is not None we return something


normal statement 1  # some random code outside of the function
normal statement 2
my_function(argument_1, argument_2)  # We call the function like this.
```

Funktionen Definieren

- Die `sqrt`-Funktion, z. B. liefert die Quadratwurzel zurück.
- Die Funktion `print` liefert nichts zurück.
- Wenn Funktionen ein Ergebnis zurückliefern, dann sollte der Typ dieses Rückgabewertes mit einem Type Hint spezifiziert werden.
- Nach der schließenden Klammer im Funktionskopf folgt dieser Datentyp dann als `-> result_type`.

Definition: Signatur

Die Abfolge der Parameter-Datentypen und der Datentyp des Rückgabewertes einer Funktion zusammen werden als die *Signatur* der Funktion bezeichnet.

Funktionen Definieren

- Die Funktion `print` liefert nichts zurück.
- Wenn Funktionen ein Ergebnis zurückliefern, dann sollte der Typ dieses Rückgabewertes mit einem Type Hint spezifiziert werden.
- Nach der schließenden Klammer im Funktionskopf folgt dieser Datentyp dann als `-> result_type`.
- Der Funktionskopf endet mit einem Doppelpunkt (`:`).

```python
"""The syntax of function definitions and function calls."""

def my_function(param_1: type_hint, param_2: type_hint) -> result_type:
    body of function 1
    body of function 2
    return result  # if result_type is not None we return something


normal statement 1  # some random code outside of the function
normal statement 2
my_function(argument_1, argument_2)  # We call the function like this.
```

Funktionen Definieren

- Die Funktion `print` liefert nichts zurück.
- Wenn Funktionen ein Ergebnis zurückliefern, dann sollte der Typ dieses Rückgabewertes mit einem Type Hint spezifiziert werden.
- Nach der schließenden Klammer im Funktionskopf folgt dieser Datentyp dann als `-> result_type`.
- Der Funktionskopf endet mit einem Doppelpunkt (`:`).

Gute Praxis

Alle Parameter und der Rückgabewert einer Funktion sollten mit Type Hints[34] annotiert werden. *Eine Funktion ohne Type Hints ist falsch.*

Funktionen Definieren

- Wenn Funktionen ein Ergebnis zurückliefern, dann sollte der Typ dieses Rückgabewertes mit einem Type Hint spezifiziert werden.
- Nach der schließenden Klammer im Funktionskopf folgt dieser Datentyp dann als `-> result_type`.
- Der Funktionskopf endet mit einem Doppelpunkt (`:`).
- Nach dem Funktionskopf folgt, eingerückt mit vier Leerzeichen, der Körper der Funktion.

```
"""The syntax of function definitions and function calls."""

def my_function(param_1: type_hint, param_2: type_hint) -> result_type:
    body of function 1
    body of function 2
    return result  # if result_type is not None we return something


normal statement 1  # some random code outside of the function
normal statement 2
my_function(argument_1, argument_2)  # We call the function like this.
```

Funktionen Definieren

- Nach der schließenden Klammer im Funktionskopf folgt dieser Datentyp dann als `-> result_type`.
- Der Funktionskopf endet mit einem Doppelpunkt (`:`).
- Nach dem Funktionskopf folgt, eingerückt mit vier Leerzeichen, der Körper der Funktion.

Gute Praxis

Der Körper einer Funktion ist mit vier Leerzeichen eingerückt.

Funktionen Definieren

- Der Funktionskopf endet mit einem Doppelpunkt (`:`).
- Nach dem Funktionskopf folgt, eingerückt mit vier Leerzeichen, der Körper der Funktion.
- Der Körper einer Funktion ist ein beliebiger Block von Kode.

```python
"""The syntax of function definitions and function calls."""

def my_function(param_1: type_hint, param_2: type_hint) -> result_type:
    body of function 1
    body of function 2
    return result  # if result_type is not None we return something


normal statement 1  # some random code outside of the function
normal statement 2
my_function(argument_1, argument_2)  # We call the function like this.
```

Funktionen Definieren

- Der Funktionskopf endet mit einem Doppelpunkt (`:`).
- Nach dem Funktionskopf folgt, eingerückt mit vier Leerzeichen, der Körper der Funktion.
- Der Körper einer Funktion ist ein beliebiger Block von Kode.
- Er kann alles beinhalten, was wir bisher gelernt haben.

```python
"""The syntax of function definitions and function calls."""

def my_function(param_1: type_hint, param_2: type_hint) -> result_type:
    body of function 1
    body of function 2
    return result  # if result_type is not None we return something


normal statement 1  # some random code outside of the function
normal statement 2
my_function(argument_1, argument_2)  # We call the function like this.
```

Funktionen Definieren

- Nach dem Funktionskopf folgt, eingerückt mit vier Leerzeichen, der Körper der Funktion.
- Der Körper einer Funktion ist ein beliebiger Block von Kode.
- Er kann alles beinhalten, was wir bisher gelernt haben.
- Von `if...else` über `for`- und `while`-Schleifen zu Variablenzuweisungen und dem Aufrufen anderer Funktionen – alles ist OK.

```python
"""The syntax of function definitions and function calls."""

def my_function(param_1: type_hint, param_2: type_hint) -> result_type:
    body of function 1
    body of function 2
    return result  # if result_type is not None we return something


normal statement 1  # some random code outside of the function
normal statement 2
my_function(argument_1, argument_2)  # We call the function like this.
```

Funktionen Definieren

- Nach dem Funktionskopf folgt, eingerückt mit vier Leerzeichen, der Körper der Funktion.
- Der Körper einer Funktion ist ein beliebiger Block von Kode.
- Er kann alles beinhalten, was wir bisher gelernt haben.
- Von `if...else` über `for`- und `while`-Schleifen zu Variablenzuweisungen und dem Aufrufen anderer Funktionen – alles ist OK.
- Wenn eine Funktion aufgerufen wird, dann wird dieser Kodeblock ausgeführt.

```
"""The syntax of function definitions and function calls."""

def my_function(param_1: type_hint, param_2: type_hint) -> result_type:
    body of function 1
    body of function 2
    return result  # if result_type is not None we return something


normal statement 1  # some random code outside of the function
normal statement 2
my_function(argument_1, argument_2)  # We call the function like this.
```

Funktionen Definieren

- Er kann alles beinhalten, was wir bisher gelernt haben.
- Von `if...else` über `for`- und `while`-Schleifen zu Variablenzuweisungen und dem Aufrufen anderer Funktionen – alles ist OK.
- Wenn eine Funktion aufgerufen wird, dann wird dieser Kodeblock ausgeführt.
- Wenn das Ende des Blocks erreicht wird, dann endet auch die Funktion (ohne einen Wert zurückzugeben).

```python
"""The syntax of function definitions and function calls."""

def my_function(param_1: type_hint, param_2: type_hint) -> result_type:
    body of function 1
    body of function 2
    return result  # if result_type is not None we return something


normal statement 1  # some random code outside of the function
normal statement 2
my_function(argument_1, argument_2)  # We call the function like this.
```

Funktionen Definieren

- Von `if...else` über `for`- und `while`-Schleifen zu Variablenzuweisungen und dem Aufrufen anderer Funktionen – alles ist OK.
- Wenn eine Funktion aufgerufen wird, dann wird dieser Kodeblock ausgeführt.
- Wenn das Ende des Blocks erreicht wird, dann endet auch die Funktion (ohne einen Wert zurückzugeben).
- Dann geht der Kontrollfluss in dem Block weiter, der die Funktion aufgerufen hat, und zwar genau nach dem Funktionsaufruf.

```python
"""The syntax of function definitions and function calls."""

def my_function(param_1: type_hint, param_2: type_hint) -> result_type:
    body of function 1
    body of function 2
    return result  # if result_type is not None we return something


normal statement 1  # some random code outside of the function
normal statement 2
my_function(argument_1, argument_2)  # We call the function like this.
```

Funktionen Definieren

- Wenn eine Funktion aufgerufen wird, dann wird dieser Kodeblock ausgeführt.
- Wenn das Ende des Blocks erreicht wird, dann endet auch die Funktion (ohne einen Wert zurückzugeben).
- Dann geht der Kontrollfluss in dem Block weiter, der die Funktion aufgerufen hat, und zwar genau nach dem Funktionsaufruf.
- Die Funktion kann auch an jedem Punkt mit Hilfe des `return`-Statement verlassen werden.

```python
"""The syntax of function definitions and function calls."""

def my_function(param_1: type_hint, param_2: type_hint) -> result_type:
    body of function 1
    body of function 2
    return result  # if result_type is not None we return something


normal statement 1  # some random code outside of the function
normal statement 2
my_function(argument_1, argument_2)  # We call the function like this.
```

Funktionen Definieren

- Wenn das Ende des Blocks erreicht wird, dann endet auch die Funktion (ohne einen Wert zurückzugeben).
- Dann geht der Kontrollfluss in dem Block weiter, der die Funktion aufgerufen hat, und zwar genau nach dem Funktionsaufruf.
- Die Funktion kann auch an jedem Punkt mit Hilfe des `return`-Statement verlassen werden.
- Soll die Funktion einen Wert `result` zurückliefern, dann geht das über `return result`.

```python
"""The syntax of function definitions and function calls."""

def my_function(param_1: type_hint, param_2: type_hint) -> result_type:
    body of function 1
    body of function 2
    return result  # if result_type is not None we return something


normal statement 1  # some random code outside of the function
normal statement 2
my_function(argument_1, argument_2)  # We call the function like this.
```

Funktionen Definieren

- Dann geht der Kontrollfluss in dem Block weiter, der die Funktion aufgerufen hat, und zwar genau nach dem Funktionsaufruf.
- Die Funktion kann auch an jedem Punkt mit Hilfe des `return`-Statement verlassen werden.
- Soll die Funktion einen Wert `result` zurückliefern, dann geht das über `return result`.
- Genau wie das `break`-Statement für Schleifen können wir das `return`-Statement in Funktionen an beliebiger Stelle verwenden.

```python
"""The syntax of function definitions and function calls."""

def my_function(param_1: type_hint, param_2: type_hint) -> result_type:
    body of function 1
    body of function 2
    return result  # if result_type is not None we return something


normal statement 1  # some random code outside of the function
normal statement 2
my_function(argument_1, argument_2)  # We call the function like this.
```

Funktionen Definieren

- Die Funktion kann auch an jedem Punkt mit Hilfe des `return`-Statement verlassen werden.
- Soll die Funktion einen Wert `result` zurückliefern, dann geht das über `return result`.
- Genau wie das `break`-Statement für Schleifen können wir das `return`-Statement in Funktionen an beliebiger Stelle verwenden.
- Wir können mehrere `return`-Statements an verschiedenen Stellen in der Funktion verwenden.

```python
"""The syntax of function definitions and function calls."""

def my_function(param_1: type_hint, param_2: type_hint) -> result_type:
    body of function 1
    body of function 2
    return result  # if result_type is not None we return something


normal statement 1  # some random code outside of the function
normal statement 2
my_function(argument_1, argument_2)  # We call the function like this.
```

Funktionen Definieren

- Genau wie das `break`-Statement für Schleifen können wir das `return`-Statement in Funktionen an beliebiger Stelle verwenden.
- Wir können mehrere `return`-Statements an verschiedenen Stellen in der Funktion verwenden.
- Die Funktion `my_function` can dann in beliebigem Kode aufgerufen werden, in dem wir `my_function(value_1, value_2)` schreiben.

```python
"""The syntax of function definitions and function calls."""

def my_function(param_1: type_hint, param_2: type_hint) -> result_type:
    body of function 1
    body of function 2
    return result  # if result_type is not None we return something


normal statement 1  # some random code outside of the function
normal statement 2
my_function(argument_1, argument_2)  # We call the function like this.
```

Funktionen Definieren

- Wir können mehrere `return`-Statements an verschiedenen Stellen in der Funktion verwenden.
- Die Funktion `my_function` can dann in beliebigem Kode aufgerufen werden, in dem wir `my_function(value_1, value_2)` schreiben.
- Hier wird `value_1` als Wert für `param_1` und `value_2` als Wert für `param_2` angegeben.

```
"""The syntax of function definitions and function calls."""

def my_function(param_1: type_hint, param_2: type_hint) -> result_type:
    body of function 1
    body of function 2
    return result  # if result_type is not None we return something


normal statement 1  # some random code outside of the function
normal statement 2
my_function(argument_1, argument_2)  # We call the function like this.
```

Funktionen Definieren

- Die Funktion `my_function` can dann in beliebigem Kode aufgerufen werden, in dem wir `my_function(value_1, value_2)` schreiben.
- Hier wird `value_1` als Wert für `param_1` und `value_2` als Wert für `param_2` angegeben.
- Das folgt also dem selben Schema für Funktionsaufrufe, das wir schon oft angewandt haben.

```
"""The syntax of function definitions and function calls."""

def my_function(param_1: type_hint, param_2: type_hint) -> result_type:
    body of function 1
    body of function 2
    return result  # if result_type is not None we return something


normal statement 1  # some random code outside of the function
normal statement 2
my_function(argument_1, argument_2)  # We call the function like this.
```

Funktionen Definieren

- Die Funktion `my_function` can dann in beliebigem Kode aufgerufen werden, in dem wir `my_function(value_1, value_2)` schreiben.
- Hier wird `value_1` als Wert für `param_1` und `value_2` als Wert für `param_2` angegeben.
- Das folgt also dem selben Schema für Funktionsaufrufe, das wir schon oft angewandt haben.

Definition: Argument

Ein *Argument* ist der Wert, der für einen Funktionsparameter beim Funktionsaufruf angegeben wird.

Funktionen Definieren

- Hier wird `value_1` als Wert für `param_1` und `value_2` als Wert für `param_2` angegeben.
- Das folgt also dem selben Schema für Funktionsaufrufe, das wir schon oft angewandt haben.
- Zwischen dem Kopf und dem Körper der Funktion müssen wir immer einen sogenannten Docstring platzieren in Form eines mehrzeiligen Strings.

```
"""The syntax of function definitions and function calls."""

def my_function(param_1: type_hint, param_2: type_hint) -> result_type:
    """
    Short sentence describing the function.

    The title of the so-called docstring is a short sentence stating
    what the function does. It can be followed by several paragraphs of
    text describing it in more detail. Then follows the list of
    parameters, return values, and raised exceptions (if any).

    :param param_1: the description of the first parameter (if any)
    :param param_2: the description of the second parameter (if any)
    :returns: the description of the return value (unless `-> None`).
    """
    body of function 1
    body of function 2
    return result  # if result_type is not None we return something
```

Funktionen Definieren

- Hier wird `value_1` als Wert für `param_1` und `value_2` als Wert für `param_2` angegeben.
- Das folgt also dem selben Schema für Funktionsaufrufe, das wir schon oft angewandt haben.
- Zwischen dem Kopf und dem Körper der Funktion müssen wir immer einen sogenannten Docstring platzieren in Form eines mehrzeiligen Strings.
- Dieser String beginnt mit einer Titelzeile, die kurz beschreibt, was die Funktion tut.

```python
"""The syntax of function definitions and function calls."""

def my_function(param_1: type_hint, param_2: type_hint) -> result_type:
    """
    Short sentence describing the function.

    The title of the so-called docstring is a short sentence stating
    what the function does. It can be followed by several paragraphs of
    text describing it in more detail. Then follows the list of
    parameters, return values, and raised exceptions (if any).

    :param param_1: the description of the first parameter (if any)
    :param param_2: the description of the second parameter (if any)
    :returns: the description of the return value (unless `-> None`).
    """
    body of function 1
    body of function 2
    return result  # if result_type is not None we return something
```

Funktionen Definieren

- Das folgt also dem selben Schema für Funktionsaufrufe, das wir schon oft angewandt haben.
- Zwischen dem Kopf und dem Körper der Funktion müssen wir immer einen sogenannten Docstring platzieren in Form eines mehrzeiligen Strings.
- Dieser String beginnt mit einer Titelzeile, die kurz beschreibt, was die Funktion tut.
- Dann folgt eine Leerzeile.

```python
"""The syntax of function definitions and function calls."""

def my_function(param_1: type_hint, param_2: type_hint) -> result_type:
    """
    Short sentence describing the function.

    The title of the so-called docstring is a short sentence stating
    what the function does. It can be followed by several paragraphs of
    text describing it in more detail. Then follows the list of
    parameters, return values, and raised exceptions (if any).

    :param param_1: the description of the first parameter (if any)
    :param param_2: the description of the second parameter (if any)
    :returns: the description of the return value (unless `-> None`).
    """
    body of function 1
    body of function 2
    return result  # if result type is not None we return something
```

Funktionen Definieren

- Zwischen dem Kopf und dem Körper der Funktion müssen wir immer einen sogenannten Docstring platzieren in Form eines mehrzeiligen Strings.
- Dieser String beginnt mit einer Titelzeile, die kurz beschreibt, was die Funktion tut.
- Dann folgt eine Leerzeile.
- Danach können wir Textabschnitte mit einer genaueren Beschreibung der Funktion einfügen.

```
"""The syntax of function definitions and function calls."""

def my_function(param_1: type_hint, param_2: type_hint) -> result_type:
    """
    Short sentence describing the function.

    The title of the so-called docstring is a short sentence stating
    what the function does. It can be followed by several paragraphs of
    text describing it in more detail. Then follows the list of
    parameters, return values, and raised exceptions (if any).

    :param param_1: the description of the first parameter (if any)
    :param param_2: the description of the second parameter (if any)
    :returns: the description of the return value (unless `-> None`).
    """
    body of function 1
    body of function 2
    return result  # if result type is not None we return something
```

Funktionen Definieren

- Dieser String beginnt mit einer Titelzeile, die kurz beschreibt, was die Funktion tut.
- Dann folgt eine Leerzeile.
- Danach können wir Textabschnitte mit einer genaueren Beschreibung der Funktion einfügen.
- Dann folgt eine Liste mit den Beschreibungen der Parameter `parameter_name`, wobei für jeden Parameter ein Eintrag der Form `:param parameter_name: Beschreibung` angegeben wird.

```python
"""The syntax of function definitions and function calls."""

def my_function(param_1: type_hint, param_2: type_hint) -> result_type:
    """
    Short sentence describing the function.

    The title of the so-called docstring is a short sentence stating
    what the function does. It can be followed by several paragraphs of
    text describing it in more detail. Then follows the list of
    parameters, return values, and raised exceptions (if any).

    :param param_1: the description of the first parameter (if any)
    :param param_2: the description of the second parameter (if any)
    :returns: the description of the return value (unless `-> None`).
    """
    body of function 1
    body of function 2
    return result  # if result type is not None we return something
```

Funktionen Definieren

- Danach können wir Textabschnitte mit einer genaueren Beschreibung der Funktion einfügen.
- Dann folgt eine Liste mit den Beschreibungen der Parameter `parameter_name`, wobei für jeden Parameter ein Eintrag der Form `:param parameter_name: Beschreibung` angegeben wird.
- Hat die Funktion einen Rückgabewert, dann folgt eine Zeile der Form `:returns: Beschreibung`, die den Rückgabewert erklärt.

```python
"""The syntax of function definitions and function calls."""

def my_function(param_1: type_hint, param_2: type_hint) -> result_type:
    """
    Short sentence describing the function.

    The title of the so-called docstring is a short sentence stating
    what the function does. It can be followed by several paragraphs of
    text describing it in more detail. Then follows the list of
    parameters, return values, and raised exceptions (if any).

    :param param_1: the description of the first parameter (if any)
    :param param_2: the description of the second parameter (if any)
    :returns: the description of the return value (unless `-> None`).
    """
    body of function 1
    body of function 2
    return result  # if result type is not None we return something
```

Funktionen Definieren

- Danach können wir Textabschnitte mit einer genaueren Beschreibung der Funktion einfügen.
- Dann folgt eine Liste mit den Beschreibungen der Parameter `parameter_name`, wobei für jeden Parameter ein Eintrag der Form `:param parameter_name: Beschreibung` angegeben wird.
- Hat die Funktion einen Rückgabewert, dann folgt eine Zeile der Form `:returns: Beschreibung`, die den Rückgabewert erklärt.

Gute Praxis

Jede Funktion sollte mit einem Docstring dokumentiert werden. Wenn Sie in einem Team arbeiten oder Ihren Kode mit anderen Teilen wollen, dann erhört ein Docstring die Wahrscheinlichkeit, dass Ihre Funktion richtig aufgerufen wird, ganz enorm. *Eine Funktion ohne Docstring ist falsch.*

Funktionen Definieren

- Dann folgt eine Liste mit den Beschreibungen der Parameter `parameter_name`, wobei für jeden Parameter ein Eintrag der Form `:param parameter_name: Beschreibung` angegeben wird.
- Hat die Funktion einen Rückgabewert, dann folgt eine Zeile der Form `:returns: Beschreibung`, die den Rückgabewert erklärt.

Gute Praxis

Nach dem Ende des Funktionskörper werden zwei Leerzeilen eingefügt, bevor weiterer Programmtext folgt[32].

Funktionen Definieren

- Hat die Funktion einen Rückgabewert, dann folgt eine Zeile der Form `:returns: Beschreibung`, die den Rückgabewert erklärt.
- Damit kennen wir nun das generelle Schema, nach dem wir Funktionen definieren können.

```python
"""The syntax of function definitions and function calls."""

def my_function(param_1: type_hint, param_2: type_hint) -> result_type:
    """
    Short sentence describing the function.

    The title of the so-called docstring is a short sentence stating
    what the function does. It can be followed by several paragraphs of
    text describing it in more detail. Then follows the list of
    parameters, return values, and raised exceptions (if any).

    :param param_1: the description of the first parameter (if any)
    :param param_2: the description of the second parameter (if any)
    :returns: the description of the return value (unless `-> None`).
    """
    body of function 1
    body of function 2
    return result  # if result_type is not None we return something


normal statement 1  # some random code outside of the function
normal statement 2
my_function(argument_1, argument_2)  # We call the function like this.
```

Beispiele

Fakultät

- Jetzt kennen wir die generelle Struktur, nach der Funktionen definiert werden können.

Fakultät

- Jetzt kennen wir die generelle Struktur, nach der Funktionen definiert werden können.
- Nach dieser sehr langen Einleitung wollen wir loslegen.

```python
"""Implementing the factorial as a function."""


def factorial(a: int) -> int:  # 1 `int` parameter and `int` result
    """
    Compute the factorial of a positive integer `a`.

    :param a: the number to compute the factorial of
    :return: the factorial of `a`, i.e., `a!`.
    """
    product: int = 1  # Initialize `product` as `1`.
    for i in range(2, a + 1):  # `i` goes from `2` to `a`.
        product *= i  # Multiply `i` to the product.
    return product  # Return the product, which now is the factorial.


for j in range(10):  # Test the `factorial` function for `i` in 0..9`.
    print(f"The factorial of {j} is {factorial(j)}.")
```

↓ `python3 def_factorial.py` ↓

```
The factorial of 0 is 1.
The factorial of 1 is 1.
The factorial of 2 is 2.
The factorial of 3 is 6.
The factorial of 4 is 24.
The factorial of 5 is 120.
The factorial of 6 is 720.
The factorial of 7 is 5040.
The factorial of 8 is 40320.
The factorial of 9 is 362880.
```

Fakultät

- Jetzt kennen wir die generelle Struktur, nach der Funktionen definiert werden können.
- Nach dieser sehr langen Einleitung wollen wir loslegen.
- Lassen Sie uns die Fakultäts-Funktion als, nunja, Funktion implementieren.

```python
"""Implementing the factorial as a function."""


def factorial(a: int) -> int:  # 1 `int` parameter and `int` result
    """
    Compute the factorial of a positive integer `a`.

    :param a: the number to compute the factorial of
    :return: the factorial of `a`, i.e., `a!`.
    """
    product: int = 1  # Initialize `product` as `1`.
    for i in range(2, a + 1):  # `i` goes from `2` to `a`.
        product *= i  # Multiply `i` to the product.
    return product  # Return the product, which now is the factorial.


for j in range(10):  # Test the `factorial` function for `i` in 0..9`.
    print(f"The factorial of {j} is {factorial(j)}.")
```

↓ `python3 def_factorial.py` ↓

```
The factorial of 0 is 1.
The factorial of 1 is 1.
The factorial of 2 is 2.
The factorial of 3 is 6.
The factorial of 4 is 24.
The factorial of 5 is 120.
The factorial of 6 is 720.
The factorial of 7 is 5040.
The factorial of 8 is 40320.
The factorial of 9 is 362880.
```

Fakultät

- Jetzt kennen wir die generelle Struktur, nach der Funktionen definiert werden können.
- Nach dieser sehr langen Einleitung wollen wir loslegen.
- Lassen Sie uns die Fakultäts-Funktion als, nunja, Funktion implementieren.
- Die Fakultät ist wie folgt definiert[4,7]:

$$a! = \begin{cases} 1 & \text{if } a = 0 \\ \prod_{i=1}^{a} i & \text{otherwise, i.e., if } a > 0 \end{cases} \tag{1}$$

wobei $\prod_{i=1}^{a} i$ für das Produkt $1 * 2 * 3 * \cdots * (a-1) * a$ steht.

```python
"""Implementing the factorial as a function."""


def factorial(a: int) -> int:  # 1 `int` parameter and `int` result
    """
    Compute the factorial of a positive integer `a`.

    :param a: the number to compute the factorial of
    :return: the factorial of `a`, i.e., `a!`.
    """
    product: int = 1  # Initialize `product` as `1`.
    for i in range(2, a + 1):  # `i` goes from `2` to `a`.
        product *= i  # Multiply `i` to the product.
    return product  # Return the product, which now is the factorial.


for j in range(10):  # Test the `factorial` function for `i` in 0..9`.
    print(f"The factorial of {j} is {factorial(j)}.")
```

↓ `python3 def_factorial.py` ↓

```
The factorial of 0 is 1.
The factorial of 1 is 1.
The factorial of 2 is 2.
The factorial of 3 is 6.
The factorial of 4 is 24.
The factorial of 5 is 120.
The factorial of 6 is 720.
The factorial of 7 is 5040.
The factorial of 8 is 40320.
The factorial of 9 is 362880.
```

Fakultät

- Jetzt kennen wir die generelle Struktur, nach der Funktionen definiert werden können.
- Nach dieser sehr langen Einleitung wollen wir loslegen.
- Lassen Sie uns die Fakultäts-Funktion als, nunja, Funktion implementieren.
- Die Fakultät ist wie folgt definiert[4,7]:

$$a! = \begin{cases} 1 & \text{if } a = 0 \\ \prod_{i=1}^{a} i & \text{otherwise, i.e., if } a > 0 \end{cases} \tag{1}$$

  wobei $\prod_{i=1}^{a} i$ für das Produkt $1 * 2 * 3 * \cdots * (a-1) * a$ steht.

- Wir implementieren dese Funktion in Python und nennen Sie `factorial`.

```python
"""Implementing the factorial as a function."""


def factorial(a: int) -> int:  # 1 `int` parameter and `int` result
    """
    Compute the factorial of a positive integer `a`.

    :param a: the number to compute the factorial of
    :return: the factorial of `a`, i.e., `a!`.
    """
    product: int = 1  # Initialize `product` as `1`.
    for i in range(2, a + 1):  # `i` goes from `2` to `a`.
        product *= i  # Multiply `i` to the product.
    return product  # Return the product, which now is the factorial.


for j in range(10):  # Test the `factorial` function for `i` in 0..9`.
    print(f"The factorial of {j} is {factorial(j)}.")
```

↓ `python3 def_factorial.py` ↓

```
The factorial of 0 is 1.
The factorial of 1 is 1.
The factorial of 2 is 2.
The factorial of 3 is 6.
The factorial of 4 is 24.
The factorial of 5 is 120.
The factorial of 6 is 720.
The factorial of 7 is 5040.
The factorial of 8 is 40320.
The factorial of 9 is 362880.
```

Fakultät

- Nach dieser sehr langen Einleitung wollen wir loslegen.
- Lassen Sie uns die Fakultäts-Funktion als, nunja, Funktion implementieren.
- Die Fakultät ist wie folgt definiert[4,7]:

$$a! = \begin{cases} 1 & \text{if } a = 0 \\ \prod_{i=1}^{a} i & \text{otherwise, i.e., if } a > 0 \end{cases} \tag{1}$$

  wobei $\prod_{i=1}^{a} i$ für das Produkt $1 * 2 * 3 * \cdots * (a-1) * a$ steht.

- Wir implementieren dese Funktion in Python und nennen Sie `factorial`.
- Der Header unserer Funktion beginnt daher erstmal mit `def factorial(`.

```python
"""Implementing the factorial as a function."""


def factorial(a: int) -> int:  # 1 `int` parameter and `int` result
    """
    Compute the factorial of a positive integer `a`.

    :param a: the number to compute the factorial of
    :return: the factorial of `a`, i.e., `a!`.
    """
    product: int = 1  # Initialize `product` as `1`.
    for i in range(2, a + 1):  # `i` goes from `2` to `a`.
        product *= i  # Multiply `i` to the product.
    return product  # Return the product, which now is the factorial.


for j in range(10):  # Test the `factorial` function for `i` in 0..9`.
    print(f"The factorial of {j} is {factorial(j)}.")
```

↓ `python3 def_factorial.py` ↓

```
The factorial of 0 is 1.
The factorial of 1 is 1.
The factorial of 2 is 2.
The factorial of 3 is 6.
The factorial of 4 is 24.
The factorial of 5 is 120.
The factorial of 6 is 720.
The factorial of 7 is 5040.
The factorial of 8 is 40320.
The factorial of 9 is 362880.
```

Fakultät

- Lassen Sie uns die Fakultäts-Funktion als, nunja, Funktion implementieren.
- Die Fakultät ist wie folgt definiert[4,7]:

$$a! = \begin{cases} 1 & \text{if } a = 0 \\ \prod_{i=1}^{a} i & \text{otherwise, i.e., if } a > 0 \end{cases} \quad (1)$$

  wobei $\prod_{i=1}^{a} i$ für das Produkt $1 * 2 * 3 * \cdots * (a-1) * a$ steht.
- Wir implementieren dese Funktion in Python und nennen Sie `factorial`.
- Der Header unserer Funktion beginnt daher erstmal mit `def factorial(`.
- Die Funktion soll einen einzelnen Paramter `a` erhalten, was wir also in die Klammern schreiben.

```python
"""Implementing the factorial as a function."""


def factorial(a: int) -> int:  # 1 `int` parameter and `int` result
    """
    Compute the factorial of a positive integer `a`.

    :param a: the number to compute the factorial of
    :return: the factorial of `a`, i.e., `a!`.
    """
    product: int = 1  # Initialize `product` as `1`.
    for i in range(2, a + 1):  # `i` goes from `2` to `a`.
        product *= i  # Multiply `i` to the product.
    return product  # Return the product, which now is the factorial.


for j in range(10):  # Test the `factorial` function for `i` in 0..9`.
    print(f"The factorial of {j} is {factorial(j)}.")
```

↓ `python3 def_factorial.py` ↓

```
The factorial of 0 is 1.
The factorial of 1 is 1.
The factorial of 2 is 2.
The factorial of 3 is 6.
The factorial of 4 is 24.
The factorial of 5 is 120.
The factorial of 6 is 720.
The factorial of 7 is 5040.
The factorial of 8 is 40320.
The factorial of 9 is 362880.
```

Fakultät

- Wir implementieren dese Funktion in Python und nennen Sie `factorial`.
- Der Header unserer Funktion beginnt daher erstmal mit `def factorial(`.
- Die Funktion soll einen einzelnen Paramter `a` erhalten, was wir also in die Klammern schreiben.
- `a` wird über einen Type-Hint als Ganzzahl definiert, i.e., wird als `a: int` angegeben.

```python
"""Implementing the factorial as a function."""


def factorial(a: int) -> int:  # 1 `int` parameter and `int` result
    """
    Compute the factorial of a positive integer `a`.

    :param a: the number to compute the factorial of
    :return: the factorial of `a`, i.e., `a!`.
    """
    product: int = 1  # Initialize `product` as `1`.
    for i in range(2, a + 1):  # `i` goes from `2` to `a`.
        product *= i  # Multiply `i` to the product.
    return product  # Return the product, which now is the factorial.


for j in range(10):  # Test the `factorial` function for `i` in 0..9`.
    print(f"The factorial of {j} is {factorial(j)}.")
```

↓ `python3 def_factorial.py` ↓

```
The factorial of 0 is 1.
The factorial of 1 is 1.
The factorial of 2 is 2.
The factorial of 3 is 6.
The factorial of 4 is 24.
The factorial of 5 is 120.
The factorial of 6 is 720.
The factorial of 7 is 5040.
The factorial of 8 is 40320.
The factorial of 9 is 362880.
```

Fakultät

- Wir implementieren dese Funktion in Python und nennen Sie `factorial`.
- Der Header unserer Funktion beginnt daher erstmal mit `def factorial(`.
- Die Funktion soll einen einzelnen Paramter `a` erhalten, was wir also in die Klammern schreiben.
- `a` wird über einen Type-Hint als Ganzzahl definiert, i.e., wird als `a: int` angegeben.
- Das Ergebnis wird auch als Ganzzahl ge-type-hinted, wir fügen also `-> int` an.

```python
"""Implementing the factorial as a function."""


def factorial(a: int) -> int:  # 1 `int` parameter and `int` result
    """
    Compute the factorial of a positive integer `a`.

    :param a: the number to compute the factorial of
    :return: the factorial of `a`, i.e., `a!`.
    """
    product: int = 1  # Initialize `product` as `1`.
    for i in range(2, a + 1):  # `i` goes from `2` to `a`.
        product *= i  # Multiply `i` to the product.
    return product  # Return the product, which now is the factorial.


for j in range(10):  # Test the `factorial` function for `i` in 0..9`.
    print(f"The factorial of {j} is {factorial(j)}.")
```

↓ `python3 def_factorial.py` ↓

```
The factorial of 0 is 1.
The factorial of 1 is 1.
The factorial of 2 is 2.
The factorial of 3 is 6.
The factorial of 4 is 24.
The factorial of 5 is 120.
The factorial of 6 is 720.
The factorial of 7 is 5040.
The factorial of 8 is 40320.
The factorial of 9 is 362880.
```

Fakultät

- Der Header unserer Funktion beginnt daher erstmal mit `def factorial(`.
- Die Funktion soll einen einzelnen Paramter `a` erhalten, was wir also in die Klammern schreiben.
- `a` wird über einen Type-Hint als Ganzzahl definiert, i.e., wird als `a: int` angegeben.
- Das Ergebnis wird auch als Ganzzahl ge-type-hinted, wir fügen also `-> int` an.
- Insgesamt haben wir also den Header `def factorial(a: int)-> int:`.

```python
"""Implementing the factorial as a function."""


def factorial(a: int) -> int:  # 1 `int` parameter and `int` result
    """
    Compute the factorial of a positive integer `a`.

    :param a: the number to compute the factorial of
    :return: the factorial of `a`, i.e., `a!`.
    """
    product: int = 1  # Initialize `product` as `1`.
    for i in range(2, a + 1):  # `i` goes from `2` to `a`.
        product *= i  # Multiply `i` to the product.
    return product  # Return the product, which now is the factorial.


for j in range(10):  # Test the `factorial` function for `i` in 0..9`.
    print(f"The factorial of {j} is {factorial(j)}.")
```

↓ `python3 def_factorial.py` ↓

```
The factorial of 0 is 1.
The factorial of 1 is 1.
The factorial of 2 is 2.
The factorial of 3 is 6.
The factorial of 4 is 24.
The factorial of 5 is 120.
The factorial of 6 is 720.
The factorial of 7 is 5040.
The factorial of 8 is 40320.
The factorial of 9 is 362880.
```

Fakultät

- Die Funktion soll einen einzelnen Paramter `a` erhalten, was wir also in die Klammern schreiben.
- `a` wird über einen Type-Hint als Ganzzahl definiert, i.e., wird als `a: int` angegeben.
- Das Ergebnis wird auch als Ganzzahl ge-type-hinted, wir fügen also `-> int` an.
- Insgesamt haben wir also den Header `def factorial(a: int)-> int:`.
- Der Körper der Funktion ist einfach.

```python
"""Implementing the factorial as a function."""


def factorial(a: int) -> int:  # 1 `int` parameter and `int` result
    """
    Compute the factorial of a positive integer `a`.

    :param a: the number to compute the factorial of
    :return: the factorial of `a`, i.e., `a!`.
    """
    product: int = 1  # Initialize `product` as `1`.
    for i in range(2, a + 1):  # `i` goes from `2` to `a`.
        product *= i  # Multiply `i` to the product.
    return product  # Return the product, which now is the factorial.


for j in range(10):  # Test the `factorial` function for `i` in 0..9`.
    print(f"The factorial of {j} is {factorial(j)}.")
```

↓ `python3 def_factorial.py` ↓

```
The factorial of 0 is 1.
The factorial of 1 is 1.
The factorial of 2 is 2.
The factorial of 3 is 6.
The factorial of 4 is 24.
The factorial of 5 is 120.
The factorial of 6 is 720.
The factorial of 7 is 5040.
The factorial of 8 is 40320.
The factorial of 9 is 362880.
```

Fakultät

- `a` wird über einen Type-Hint als Ganzzahl definiert, i.e., wird als `a: int` angegeben.
- Das Ergebnis wird auch als Ganzzahl ge-type-hinted, wir fügen also `-> int` an.
- Insgesamt haben wir also den Header `def factorial(a: int)-> int:`.
- Der Körper der Funktion ist einfach.
- Zuerst initialisieren wir die Variable `product` mit dem Wert `1`.

```python
"""Implementing the factorial as a function."""


def factorial(a: int) -> int:  # 1 `int` parameter and `int` result
    """
    Compute the factorial of a positive integer `a`.

    :param a: the number to compute the factorial of
    :return: the factorial of `a`, i.e., `a!`.
    """
    product: int = 1  # Initialize `product` as `1`.
    for i in range(2, a + 1):  # `i` goes from `2` to `a`.
        product *= i  # Multiply `i` to the product.
    return product  # Return the product, which now is the factorial.


for j in range(10):  # Test the `factorial` function for `i` in 0..9`.
    print(f"The factorial of {j} is {factorial(j)}.")
```

↓ `python3 def_factorial.py` ↓

```
The factorial of 0 is 1.
The factorial of 1 is 1.
The factorial of 2 is 2.
The factorial of 3 is 6.
The factorial of 4 is 24.
The factorial of 5 is 120.
The factorial of 6 is 720.
The factorial of 7 is 5040.
The factorial of 8 is 40320.
The factorial of 9 is 362880.
```

Fakultät

- Das Ergebnis wird auch als Ganzzahl ge-type-hinted, wir fügen also `-> int` an.
- Insgesamt haben wir also den Header `def factorial(a: int)-> int:`.
- Der Körper der Funktion ist einfach.
- Zuerst initialisieren wir die Variable `product` mit dem Wert `1`.
- Dann brauchen wir eine Schleife, die über alle positiven Ganzzahlen kleiner/gleich `a` iteriert.

```python
"""Implementing the factorial as a function."""


def factorial(a: int) -> int:  # 1 `int` parameter and `int` result
    """
    Compute the factorial of a positive integer `a`.

    :param a: the number to compute the factorial of
    :return: the factorial of `a`, i.e., `a!`.
    """
    product: int = 1  # Initialize `product` as `1`.
    for i in range(2, a + 1):  # `i` goes from `2` to `a`.
        product *= i  # Multiply `i` to the product.
    return product  # Return the product, which now is the factorial.


for j in range(10):  # Test the `factorial` function for `i` in 0..9`.
    print(f"The factorial of {j} is {factorial(j)}.")
```

↓ `python3 def_factorial.py` ↓

```
The factorial of 0 is 1.
The factorial of 1 is 1.
The factorial of 2 is 2.
The factorial of 3 is 6.
The factorial of 4 is 24.
The factorial of 5 is 120.
The factorial of 6 is 720.
The factorial of 7 is 5040.
The factorial of 8 is 40320.
The factorial of 9 is 362880.
```

Fakultät

- Insgesamt haben wir also den Header `def factorial(a: int)-> int:`.
- Der Körper der Funktion ist einfach.
- Zuerst initialisieren wir die Variable `product` mit dem Wert `1`.
- Dann brauchen wir eine Schleife, die über alle positiven Ganzzahlen kleiner/gleich `a` iteriert.
- Wir müssen diese Werte an `product` heranmultiplizieren.

```python
"""Implementing the factorial as a function."""


def factorial(a: int) -> int:  # 1 `int` parameter and `int` result
    """
    Compute the factorial of a positive integer `a`.

    :param a: the number to compute the factorial of
    :return: the factorial of `a`, i.e., `a!`.
    """
    product: int = 1  # Initialize `product` as `1`.
    for i in range(2, a + 1):  # `i` goes from `2` to `a`.
        product *= i  # Multiply `i` to the product.
    return product  # Return the product, which now is the factorial.


for j in range(10):  # Test the `factorial` function for `i` in 0..9`.
    print(f"The factorial of {j} is {factorial(j)}.")
```

↓ `python3 def_factorial.py` ↓

```
The factorial of 0 is 1.
The factorial of 1 is 1.
The factorial of 2 is 2.
The factorial of 3 is 6.
The factorial of 4 is 24.
The factorial of 5 is 120.
The factorial of 6 is 720.
The factorial of 7 is 5040.
The factorial of 8 is 40320.
The factorial of 9 is 362880.
```

Fakultät

- Der Körper der Funktion ist einfach.
- Zuerst initialisieren wir die Variable `product` mit dem Wert `1`.
- Dann brauchen wir eine Schleife, die über alle positiven Ganzzahlen kleiner/gleich `a` iteriert.
- Wir müssen diese Werte an `product` heranmultiplizieren.
- Wir können dabei `i` $=1$ überspringen, weil es sinnlos wäre.

```python
"""Implementing the factorial as a function."""


def factorial(a: int) -> int:  # 1 `int` parameter and `int` result
    """
    Compute the factorial of a positive integer `a`.

    :param a: the number to compute the factorial of
    :return: the factorial of `a`, i.e., `a!`.
    """
    product: int = 1  # Initialize `product` as `1`.
    for i in range(2, a + 1):  # `i` goes from `2` to `a`.
        product *= i  # Multiply `i` to the product.
    return product  # Return the product, which now is the factorial.


for j in range(10):  # Test the `factorial` function for `i` in 0..9`.
    print(f"The factorial of {j} is {factorial(j)}.")
```

↓ `python3 def_factorial.py` ↓

```
The factorial of 0 is 1.
The factorial of 1 is 1.
The factorial of 2 is 2.
The factorial of 3 is 6.
The factorial of 4 is 24.
The factorial of 5 is 120.
The factorial of 6 is 720.
The factorial of 7 is 5040.
The factorial of 8 is 40320.
The factorial of 9 is 362880.
```

Fakultät

- Zuerst initialisieren wir die Variable `product` mit dem Wert `1`.
- Dann brauchen wir eine Schleife, die über alle positiven Ganzzahlen kleiner/gleich `a` iteriert.
- Wir müssen diese Werte an `product` heranmultiplizieren.
- Wir können dabei `i` $= 1$ überspringen, weil es sinnlos wäre.
- Wir definieren also eine `for`-Schleife, die `i` über `range(2, a + 1)` iteriert.

```python
"""Implementing the factorial as a function."""


def factorial(a: int) -> int:  # 1 `int` parameter and `int` result
    """
    Compute the factorial of a positive integer `a`.

    :param a: the number to compute the factorial of
    :return: the factorial of `a`, i.e., `a!`.
    """
    product: int = 1  # Initialize `product` as `1`.
    for i in range(2, a + 1):  # `i` goes from `2` to `a`.
        product *= i  # Multiply `i` to the product.
    return product  # Return the product, which now is the factorial.


for j in range(10):  # Test the `factorial` function for `i` in 0..9`.
    print(f"The factorial of {j} is {factorial(j)}.")
```

↓ `python3 def_factorial.py` ↓

```
The factorial of 0 is 1.
The factorial of 1 is 1.
The factorial of 2 is 2.
The factorial of 3 is 6.
The factorial of 4 is 24.
The factorial of 5 is 120.
The factorial of 6 is 720.
The factorial of 7 is 5040.
The factorial of 8 is 40320.
The factorial of 9 is 362880.
```

Fakultät

- Dann brauchen wir eine Schleife, die über alle positiven Ganzzahlen kleiner/gleich `a` iteriert.
- Wir müssen diese Werte an `product` heranmultiplizieren.
- Wir können dabei `i` $= 1$ überspringen, weil es sinnlos wäre.
- Wir definieren also eine `for`-Schleife, die `i` über `range(2, a + 1)` iteriert.
- Damit startet `i` mit `2`.

```python
"""Implementing the factorial as a function."""


def factorial(a: int) -> int:  # 1 `int` parameter and `int` result
    """
    Compute the factorial of a positive integer `a`.

    :param a: the number to compute the factorial of
    :return: the factorial of `a`, i.e., `a!`.
    """
    product: int = 1  # Initialize `product` as `1`.
    for i in range(2, a + 1):  # `i` goes from `2` to `a`.
        product *= i  # Multiply `i` to the product.
    return product  # Return the product, which now is the factorial.


for j in range(10):  # Test the `factorial` function for `i` in 0..9`.
    print(f"The factorial of {j} is {factorial(j)}.")
```

↓ `python3 def_factorial.py` ↓

```
The factorial of 0 is 1.
The factorial of 1 is 1.
The factorial of 2 is 2.
The factorial of 3 is 6.
The factorial of 4 is 24.
The factorial of 5 is 120.
The factorial of 6 is 720.
The factorial of 7 is 5040.
The factorial of 8 is 40320.
The factorial of 9 is 362880.
```

Fakultät

- Wir müssen diese Werte an `product` heranmultiplizieren.
- Wir können dabei `i` $= 1$ überspringen, weil es sinnlos wäre.
- Wir definieren also eine `for`-Schleife, die `i` über `range(2, a + 1)` iteriert.
- Damit startet `i` mit `2`.
- Die exklusive Obergrenze `a + 1` der `range`, so das die Vorschleife läuft, bis `i` irgendwann `a` erreicht.

```python
"""Implementing the factorial as a function."""


def factorial(a: int) -> int:  # 1 `int` parameter and `int` result
    """
    Compute the factorial of a positive integer `a`.

    :param a: the number to compute the factorial of
    :return: the factorial of `a`, i.e., `a!`.
    """
    product: int = 1  # Initialize `product` as `1`.
    for i in range(2, a + 1):  # `i` goes from `2` to `a`.
        product *= i  # Multiply `i` to the product.
    return product  # Return the product, which now is the factorial.


for j in range(10):  # Test the `factorial` function for `i` in 0..9`.
    print(f"The factorial of {j} is {factorial(j)}.")
```

↓ `python3 def_factorial.py` ↓

```
The factorial of 0 is 1.
The factorial of 1 is 1.
The factorial of 2 is 2.
The factorial of 3 is 6.
The factorial of 4 is 24.
The factorial of 5 is 120.
The factorial of 6 is 720.
The factorial of 7 is 5040.
The factorial of 8 is 40320.
The factorial of 9 is 362880.
```

Fakultät

- Wir können dabei `i` $= 1$ überspringen, weil es sinnlos wäre.
- Wir definieren also eine `for`-Schleife, die `i` über `range(2, a + 1)` iteriert.
- Damit startet `i` mit `2`.
- Die exklusive Obergrenze `a + 1` der `range`, so das die Vorschleife läuft, bis `i` irgendwann `a` erreicht.
- Sehen Sie, dass wir `a` hier wie eine ganz normale Variable verwenden?

```python
"""Implementing the factorial as a function."""


def factorial(a: int) -> int:  # 1 `int` parameter and `int` result
    """
    Compute the factorial of a positive integer `a`.

    :param a: the number to compute the factorial of
    :return: the factorial of `a`, i.e., `a!`.
    """
    product: int = 1  # Initialize `product` as `1`.
    for i in range(2, a + 1):  # `i` goes from `2` to `a`.
        product *= i  # Multiply `i` to the product.
    return product  # Return the product, which now is the factorial.


for j in range(10):  # Test the `factorial` function for `i` in 0..9`.
    print(f"The factorial of {j} is {factorial(j)}.")
```

↓ `python3 def_factorial.py` ↓

```
The factorial of 0 is 1.
The factorial of 1 is 1.
The factorial of 2 is 2.
The factorial of 3 is 6.
The factorial of 4 is 24.
The factorial of 5 is 120.
The factorial of 6 is 720.
The factorial of 7 is 5040.
The factorial of 8 is 40320.
The factorial of 9 is 362880.
```

Fakultät

- Wir definieren also eine `for`-Schleife, die `i` über `range(2, a + 1)` iteriert.
- Damit startet `i` mit `2`.
- Die exklusive Obergrenze `a + 1` der `range`, so das die Vorschleife läuft, bis `i` irgendwann `a` erreicht.
- Sehen Sie, dass wir `a` hier wie eine ganz normale Variable verwenden?
- In dem Körper der Schleife berechnen wir `product *= i`, was das selbe ist wie `product = product * i`.

```python
"""Implementing the factorial as a function."""


def factorial(a: int) -> int:  # 1 `int` parameter and `int` result
    """
    Compute the factorial of a positive integer `a`.

    :param a: the number to compute the factorial of
    :return: the factorial of `a`, i.e., `a!`.
    """
    product: int = 1  # Initialize `product` as `1`.
    for i in range(2, a + 1):  # `i` goes from `2` to `a`.
        product *= i  # Multiply `i` to the product.
    return product  # Return the product, which now is the factorial.


for j in range(10):  # Test the `factorial` function for `i` in 0..9`.
    print(f"The factorial of {j} is {factorial(j)}.")
```

↓ `python3 def_factorial.py` ↓

```
The factorial of 0 is 1.
The factorial of 1 is 1.
The factorial of 2 is 2.
The factorial of 3 is 6.
The factorial of 4 is 24.
The factorial of 5 is 120.
The factorial of 6 is 720.
The factorial of 7 is 5040.
The factorial of 8 is 40320.
The factorial of 9 is 362880.
```

Fakultät

- Damit startet `i` mit `2`.
- Die exklusive Obergrenze `a + 1` der `range`, so das die Vorschleife läuft, bis `i` irgendwann `a` erreicht.
- Sehen Sie, dass wir `a` hier wie eine ganz normale Variable verwenden?
- In dem Körper der Schleife berechnen wir `product *= i`, was das selbe ist wie `product = product * i`.
- Nach der Schleife steht `a`! in `product`.

```python
"""Implementing the factorial as a function."""


def factorial(a: int) -> int:  # 1 `int` parameter and `int` result
    """
    Compute the factorial of a positive integer `a`.

    :param a: the number to compute the factorial of
    :return: the factorial of `a`, i.e., `a!`.
    """
    product: int = 1  # Initialize `product` as `1`.
    for i in range(2, a + 1):  # `i` goes from `2` to `a`.
        product *= i  # Multiply `i` to the product.
    return product  # Return the product, which now is the factorial.


for j in range(10):  # Test the `factorial` function for `i` in 0..9`.
    print(f"The factorial of {j} is {factorial(j)}.")
```

↓ `python3 def_factorial.py` ↓

```
The factorial of 0 is 1.
The factorial of 1 is 1.
The factorial of 2 is 2.
The factorial of 3 is 6.
The factorial of 4 is 24.
The factorial of 5 is 120.
The factorial of 6 is 720.
The factorial of 7 is 5040.
The factorial of 8 is 40320.
The factorial of 9 is 362880.
```

Fakultät

- Die exklusive Obergrenze `a + 1` der `range`, so das die Vorschleife läuft, bis `i` irgendwann `a` erreicht.
- Sehen Sie, dass wir `a` hier wie eine ganz normale Variable verwenden?
- In dem Körper der Schleife berechnen wir `product *= i`, was das selbe ist wie `product = product * i`.
- Nach der Schleife steht `a`! in `product`.
- Wir können diesen Wert zurückliefern, in dem wir `return product` schreiben.

```python
"""Implementing the factorial as a function."""


def factorial(a: int) -> int:  # 1 `int` parameter and `int` result
    """
    Compute the factorial of a positive integer `a`.

    :param a: the number to compute the factorial of
    :return: the factorial of `a`, i.e., `a!`.
    """
    product: int = 1  # Initialize `product` as `1`.
    for i in range(2, a + 1):  # `i` goes from `2` to `a`.
        product *= i  # Multiply `i` to the product.
    return product  # Return the product, which now is the factorial.


for j in range(10):  # Test the `factorial` function for `i` in 0..9`.
    print(f"The factorial of {j} is {factorial(j)}.")
```

↓ `python3 def_factorial.py` ↓

```
The factorial of 0 is 1.
The factorial of 1 is 1.
The factorial of 2 is 2.
The factorial of 3 is 6.
The factorial of 4 is 24.
The factorial of 5 is 120.
The factorial of 6 is 720.
The factorial of 7 is 5040.
The factorial of 8 is 40320.
The factorial of 9 is 362880.
```

Fakultät

- Sehen Sie, dass wir `a` hier wie eine ganz normale Variable verwenden?
- In dem Körper der Schleife berechnen wir `product *= i`, was das selbe ist wie `product = product * i`.
- Nach der Schleife steht `a`! in `product`.
- Wir können diesen Wert zurückliefern, in dem wir `return product` schreiben.
- Wir können nun die Fakultät von jeder positiven Ganzzahl `x` berechnen, in dem wir `factorial(x)` aufrufen.

```python
"""Implementing the factorial as a function."""


def factorial(a: int) -> int:  # 1 `int` parameter and `int` result
    """
    Compute the factorial of a positive integer `a`.

    :param a: the number to compute the factorial of
    :return: the factorial of `a`, i.e., `a!`.
    """
    product: int = 1  # Initialize `product` as `1`.
    for i in range(2, a + 1):  # `i` goes from `2` to `a`.
        product *= i  # Multiply `i` to the product.
    return product  # Return the product, which now is the factorial.


for j in range(10):  # Test the `factorial` function for `i` in 0..9`.
    print(f"The factorial of {j} is {factorial(j)}.")
```

↓ `python3 def_factorial.py` ↓

```
The factorial of 0 is 1.
The factorial of 1 is 1.
The factorial of 2 is 2.
The factorial of 3 is 6.
The factorial of 4 is 24.
The factorial of 5 is 120.
The factorial of 6 is 720.
The factorial of 7 is 5040.
The factorial of 8 is 40320.
The factorial of 9 is 362880.
```

Fakultät

- In dem Körper der Schleife berechnen wir `product *= i`, was das selbe ist wie `product = product * i`.
- Nach der Schleife steht `a`! in `product`.
- Wir können diesen Wert zurückliefern, in dem wir `return product` schreiben.
- Wir können nun die Fakultät von jeder positiven Ganzzahl `x` berechnen, in dem wir `factorial(x)` aufrufen.
- Nach dem Körper der Funktion lassen wir zwei Leerzeilen.

```python
"""Implementing the factorial as a function."""


def factorial(a: int) -> int:  # 1 `int` parameter and `int` result
    """
    Compute the factorial of a positive integer `a`.

    :param a: the number to compute the factorial of
    :return: the factorial of `a`, i.e., `a!`.
    """
    product: int = 1  # Initialize `product` as `1`.
    for i in range(2, a + 1):  # `i` goes from `2` to `a`.
        product *= i  # Multiply `i` to the product.
    return product  # Return the product, which now is the factorial.


for j in range(10):  # Test the `factorial` function for `i` in 0..9`.
    print(f"The factorial of {j} is {factorial(j)}.")
```

↓ `python3 def_factorial.py` ↓

```
The factorial of 0 is 1.
The factorial of 1 is 1.
The factorial of 2 is 2.
The factorial of 3 is 6.
The factorial of 4 is 24.
The factorial of 5 is 120.
The factorial of 6 is 720.
The factorial of 7 is 5040.
The factorial of 8 is 40320.
The factorial of 9 is 362880.
```

Fakultät

- Nach der Schleife steht `a`! in `product`.
- Wir können diesen Wert zurückliefern, in dem wir `return product` schreiben.
- Wir können nun die Fakultät von jeder positiven Ganzzahl `x` berechnen, in dem wir `factorial(x)` aufrufen.
- Nach dem Körper der Funktion lassen wir zwei Leerzeilen.
- Dann berechnen wir die Fakultäten von 1 bis 9 in einer `for`-Schleife und drucken sie mit Hilfe von einem f-String aus.

```python
"""Implementing the factorial as a function."""


def factorial(a: int) -> int:  # 1 `int` parameter and `int` result
    """
    Compute the factorial of a positive integer `a`.

    :param a: the number to compute the factorial of
    :return: the factorial of `a`, i.e., `a!`.
    """
    product: int = 1  # Initialize `product` as `1`.
    for i in range(2, a + 1):  # `i` goes from `2` to `a`.
        product *= i  # Multiply `i` to the product.
    return product  # Return the product, which now is the factorial.


for j in range(10):  # Test the `factorial` function for `i` in 0..9`.
    print(f"The factorial of {j} is {factorial(j)}.")
```

↓ `python3 def_factorial.py` ↓

```
The factorial of 0 is 1.
The factorial of 1 is 1.
The factorial of 2 is 2.
The factorial of 3 is 6.
The factorial of 4 is 24.
The factorial of 5 is 120.
The factorial of 6 is 720.
The factorial of 7 is 5040.
The factorial of 8 is 40320.
The factorial of 9 is 362880.
```

Fakultät

- Wir können diesen Wert zurückliefern, in dem wir `return product` schreiben.
- Wir können nun die Fakultät von jeder positiven Ganzzahl `x` berechnen, in dem wir `factorial(x)` aufrufen.
- Nach dem Körper der Funktion lassen wir zwei Leerzeilen.
- Dann berechnen wir die Fakultäten von 1 bis 9 in einer `for`-Schleife und drucken sie mit Hilfe von einem f-String aus.
- In der Schleife und dem f-String benutzen wir `factorial` genau wie jede andere Funktion, genau wie `sqrt` oder `sin`.

```python
"""Implementing the factorial as a function."""


def factorial(a: int) -> int:  # 1 `int` parameter and `int` result
    """
    Compute the factorial of a positive integer `a`.

    :param a: the number to compute the factorial of
    :return: the factorial of `a`, i.e., `a!`.
    """
    product: int = 1  # Initialize `product` as `1`.
    for i in range(2, a + 1):  # `i` goes from `2` to `a`.
        product *= i  # Multiply `i` to the product.
    return product  # Return the product, which now is the factorial.


for j in range(10):  # Test the `factorial` function for `i` in 0..9`.
    print(f"The factorial of {j} is {factorial(j)}.")
```

↓ `python3 def_factorial.py` ↓

```
The factorial of 0 is 1.
The factorial of 1 is 1.
The factorial of 2 is 2.
The factorial of 3 is 6.
The factorial of 4 is 24.
The factorial of 5 is 120.
The factorial of 6 is 720.
The factorial of 7 is 5040.
The factorial of 8 is 40320.
The factorial of 9 is 362880.
```

Fakultät

- Nach dem Körper der Funktion lassen wir zwei Leerzeilen.
- Dann berechnen wir die Fakultäten von 1 bis 9 in einer `for`-Schleife und drucken sie mit Hilfe von einem f-String aus.
- In der Schleife und dem f-String benutzen wir `factorial` genau wie jede andere Funktion, genau wie `sqrt` oder `sin`.
- Als kleine Seiteninformation sei gesagt, dass es Möglichkeiten gibt, die Fakultät schneller zu berechnen als mit dieser Produktmethode[19].

```python
"""Implementing the factorial as a function."""


def factorial(a: int) -> int:  # 1 `int` parameter and `int` result
    """
    Compute the factorial of a positive integer `a`.

    :param a: the number to compute the factorial of
    :return: the factorial of `a`, i.e., `a!`.
    """
    product: int = 1  # Initialize `product` as `1`.
    for i in range(2, a + 1):  # `i` goes from `2` to `a`.
        product *= i  # Multiply `i` to the product.
    return product  # Return the product, which now is the factorial.


for j in range(10):  # Test the `factorial` function for `i` in 0..9`.
    print(f"The factorial of {j} is {factorial(j)}.")
```

↓ `python3 def_factorial.py` ↓

```
The factorial of 0 is 1.
The factorial of 1 is 1.
The factorial of 2 is 2.
The factorial of 3 is 6.
The factorial of 4 is 24.
The factorial of 5 is 120.
The factorial of 6 is 720.
The factorial of 7 is 5040.
The factorial of 8 is 40320.
The factorial of 9 is 362880.
```

Euklidischer Algorithmus für den Größten Gemeinsamen Teiler

- Funktionen können mehr als einen Parameter haben oder auch gar keinen Parameter haben.

Euklidischer Algorithmus für den Größten Gemeinsamen Teiler

- Funktionen können mehr als einen Parameter haben oder auch gar keinen Parameter haben.
- Funktionen können ein Ergebnis zurückliefern oder auch nichts

.

Euklidischer Algorithmus für den Größten Gemeinsamen Teiler

- Funktionen können mehr als einen Parameter haben oder auch gar keinen Parameter haben.
- Funktionen können ein Ergebnis zurückliefern oder auch nichts (wobei sie dann `None` zurückliefern, wie wir in Einheit 12 gelernt haben).

Euklidischer Algorithmus für den Größten Gemeinsamen Teiler

- Funktionen können mehr als einen Parameter haben oder auch gar keinen Parameter haben.
- Funktionen können ein Ergebnis zurückliefern oder auch nichts (wobei sie dann `None` zurückliefern, wie wir in Einheit 12 gelernt haben).
- Funktionen können auch aus anderen Funktionen heraus aufgerufen werden.

Euklidischer Algorithmus für den Größten Gemeinsamen Teiler

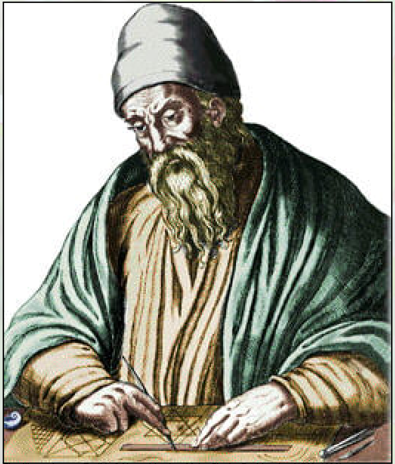
- Funktionen können mehr als einen Parameter haben oder auch gar keinen Parameter haben.
- Funktionen können ein Ergebnis zurückliefern oder auch nichts (wobei sie dann `None` zurückliefern, wie wir in Einheit 12 gelernt haben).
- Funktionen können auch aus anderen Funktionen heraus aufgerufen werden.
- Probieren wir das mal aus an einem weiteren spannenden Beispiel.

Euklidischer Algorithmus für den Größten Gemeinsamen Teiler

- Funktionen können mehr als einen Parameter haben oder auch gar keinen Parameter haben.
- Funktionen können ein Ergebnis zurückliefern oder auch nichts (wobei sie dann `None` zurückliefern, wie wir in Einheit 12 gelernt haben).
- Funktionen können auch aus anderen Funktionen heraus aufgerufen werden.
- Probieren wir das mal aus an einem weiteren spannenden Beispiel.
- Schauen wir uns die Berechnung des größten gemeinsamen Teilers (EN: *greatest common divisor*), kurz `gcd` an.

Euklidischer Algorithmus für den Größten Gemeinsamen Teiler

- Der GCD kann mit dem Euclidischen Algorithmus[1,8,10] von Euclid of Alexandria ($E\acute{\upsilon}\kappa\lambda\varepsilon\acute{\iota}\delta\eta\varsigma$) der 300 *before Common Era (BCE)* gelebt hat berechnet werden.



Eine Illustration von Euclid of Alexandria, dem Maler Charles Paul Landon (1760–1826) zugeschrieben. Quelle: Vikidia, wo es als *domaine public*, steht, also als in der Public Domain.

Euklidischer Algorithmus für den Größten Gemeinsamen Teiler

- Der GCD kann mit dem Euclidischen Algorithmus[1,8,10] von Euclid of Alexandria ($E\acute{\upsilon}\kappa\lambda\varepsilon\acute{\iota}\delta\eta\varsigma$) der 300 *before Common Era (BCE)* gelebt hat berechnet werden.
- Der größte gemeinsame Teiler zweier positiver natürlicher Zahlen $a \in \mathbb{N}_1$ und $b \in \mathbb{N}_1$ ist die größte Zahl $g \in \mathbb{N}_1 =$ `gcd`(a, b) für die gilt das $a \bmod g = 0$ und $b \bmod g = 0$, wobei mod die Modulo Division ist, also der Rest einer Division, was wiederum äquivalent zum Operator `%` von Python ist.

Euklidischer Algorithmus für den Größten Gemeinsamen Teiler

- Der GCD kann mit dem Euclidischen Algorithmus[1,8,10] von Euclid of Alexandria ($E\acute{\upsilon}\kappa\lambda\varepsilon\acute{\iota}\delta\eta\varsigma$) der 300 *before Common Era (BCE)* gelebt hat berechnet werden.
- Der größte gemeinsame Teiler zweier positiver natürlicher Zahlen $a \in \mathbb{N}_1$ und $b \in \mathbb{N}_1$ ist die größte Zahl $g \in \mathbb{N}_1 = \texttt{gcd}(a, b)$ für die gilt das $a \bmod g = 0$ und $b \bmod g = 0$, wobei mod die Modulo Division ist, also der Rest einer Division, was wiederum äquivalent zum Operator `%` von Python ist.
- Das bedeutet, dass g sowohl a als auch b ohne Rest teilt.

Euklidischer Algorithmus für den Größten Gemeinsamen Teiler

- Der GCD kann mit dem Euclidischen Algorithmus[1,8,10] von Euclid of Alexandria ($E\acute{\upsilon}\kappa\lambda\varepsilon\acute{\iota}\delta\eta\varsigma$) der 300 *before Common Era (BCE)* gelebt hat berechnet werden.
- Der größte gemeinsame Teiler zweier positiver natürlicher Zahlen $a \in \mathbb{N}_1$ und $b \in \mathbb{N}_1$ ist die größte Zahl $g \in \mathbb{N}_1 =$ `gcd`(a, b) für die gilt das $a \bmod g = 0$ und $b \bmod g = 0$, wobei mod die Modulo Division ist, also der Rest einer Division, was wiederum äquivalent zum Operator `%` von Python ist.
- Das bedeutet, dass g sowohl a als auch b ohne Rest teilt.
- Ist $a = b$, dann gilt offensichtlich `gcd`$(a, b) = a = b$.

Euklidischer Algorithmus für den Größten Gemeinsamen Teiler

- Der GCD kann mit dem Euclidischen Algorithmus[1,8,10] von Euclid of Alexandria ($E\acute{\upsilon}\kappa\lambda\varepsilon\acute{\iota}\delta\eta\varsigma$) der 300 *before Common Era (BCE)* gelebt hat berechnet werden.
- Der größte gemeinsame Teiler zweier positiver natürlicher Zahlen $a \in \mathbb{N}_1$ und $b \in \mathbb{N}_1$ ist die größte Zahl $g \in \mathbb{N}_1 = \texttt{gcd}(a, b)$ für die gilt das $a \bmod g = 0$ und $b \bmod g = 0$, wobei mod die Modulo Division ist, also der Rest einer Division, was wiederum äquivalent zum Operator `%` von Python ist.
- Das bedeutet, dass g sowohl a als auch b ohne Rest teilt.
- Ist $a = b$, dann gilt offensichtlich $\texttt{gcd}(a, b) = a = b$.
- Andernfalls wissen wir, dass $a = ig$ für irgendein $i \in \mathbb{N}_1$ und das $b = jg$ für irgendein $j \in \mathbb{N}_1$ gilt.

Euklidischer Algorithmus für den Größten Gemeinsamen Teiler

- Der GCD kann mit dem Euclidischen Algorithmus[1,8,10] von Euclid of Alexandria ($E\acute{\upsilon}\kappa\lambda\varepsilon\acute{\iota}\delta\eta\varsigma$) der 300 *before Common Era (BCE)* gelebt hat berechnet werden.
- Der größte gemeinsame Teiler zweier positiver natürlicher Zahlen $a \in \mathbb{N}_1$ und $b \in \mathbb{N}_1$ ist die größte Zahl $g \in \mathbb{N}_1 = \texttt{gcd}(a, b)$ für die gilt das $a \bmod g = 0$ und $b \bmod g = 0$, wobei mod die Modulo Division ist, also der Rest einer Division, was wiederum äquivalent zum Operator `%` von Python ist.
- Das bedeutet, dass g sowohl a als auch b ohne Rest teilt.
- Ist $a = b$, dann gilt offensichtlich $\texttt{gcd}(a, b) = a = b$.
- Andernfalls wissen wir, dass $a = ig$ für irgendein $i \in \mathbb{N}_1$ und das $b = jg$ für irgendein $j \in \mathbb{N}_1$ gilt.
- Ohne Beschränkung der Allgemeinheit nehmen wir an dass $a > b$.

Euklidischer Algorithmus für den Größten Gemeinsamen Teiler

- Der GCD kann mit dem Euclidischen Algorithmus[1,8,10] von Euclid of Alexandria ($E\acute{\upsilon}\kappa\lambda\varepsilon\acute{\iota}\delta\eta\varsigma$) der 300 *before Common Era (BCE)* gelebt hat berechnet werden.
- Der größte gemeinsame Teiler zweier positiver natürlicher Zahlen $a \in \mathbb{N}_1$ und $b \in \mathbb{N}_1$ ist die größte Zahl $g \in \mathbb{N}_1 = \texttt{gcd}(a, b)$ für die gilt das $a \bmod g = 0$ und $b \bmod g = 0$, wobei mod die Modulo Division ist, also der Rest einer Division, was wiederum äquivalent zum Operator `%` von Python ist.
- Das bedeutet, dass g sowohl a als auch b ohne Rest teilt.
- Ist $a = b$, dann gilt offensichtlich $\texttt{gcd}(a, b) = a = b$.
- Andernfalls wissen wir, dass $a = ig$ für irgendein $i \in \mathbb{N}_1$ und das $b = jg$ für irgendein $j \in \mathbb{N}_1$ gilt.
- Ohne Beschränkung der Allgemeinheit nehmen wir an dass $a > b$.
- Dann gilt das $c = a - b = (i - j)g$.

Euklidischer Algorithmus für den Größten Gemeinsamen Teiler

- Der GCD kann mit dem Euclidischen Algorithmus[1,8,10] von Euclid of Alexandria ($E\acute{\upsilon}\kappa\lambda\varepsilon\acute{\iota}\delta\eta\varsigma$) der 300 *before Common Era (BCE)* gelebt hat berechnet werden.
- Der größte gemeinsame Teiler zweier positiver natürlicher Zahlen $a \in \mathbb{N}_1$ und $b \in \mathbb{N}_1$ ist die größte Zahl $g \in \mathbb{N}_1 =$ `gcd`(a, b) für die gilt das $a \bmod g = 0$ und $b \bmod g = 0$, wobei mod die Modulo Division ist, also der Rest einer Division, was wiederum äquivalent zum Operator `%` von Python ist.
- Das bedeutet, dass g sowohl a als auch b ohne Rest teilt.
- Ist $a = b$, dann gilt offensichtlich `gcd`$(a, b) = a = b$.
- Andernfalls wissen wir, dass $a = ig$ für irgendein $i \in \mathbb{N}_1$ und das $b = jg$ für irgendein $j \in \mathbb{N}_1$ gilt.
- Ohne Beschränkung der Allgemeinheit nehmen wir an dass $a > b$.
- Dann gilt das $c = a - b = (i - j)g$.
- Es ist klar das $c \bmod g = (a - b) \bmod g = (i - j)g \bmod g = 0$ gilt.

Euklidischer Algorithmus für den Größten Gemeinsamen Teiler

- Der GCD kann mit dem Euclidischen Algorithmus[1,8,10] von Euclid of Alexandria ($E\acute{\upsilon}\kappa\lambda\varepsilon\acute{\iota}\delta\eta\varsigma$) der 300 *before Common Era (BCE)* gelebt hat berechnet werden.
- Der größte gemeinsame Teiler zweier positiver natürlicher Zahlen $a \in \mathbb{N}_1$ und $b \in \mathbb{N}_1$ ist die größte Zahl $g \in \mathbb{N}_1 =$ `gcd`(a, b) für die gilt das $a \bmod g = 0$ und $b \bmod g = 0$, wobei mod die Modulo Division ist, also der Rest einer Division, was wiederum äquivalent zum Operator `%` von Python ist.
- Das bedeutet, dass g sowohl a als auch b ohne Rest teilt.
- Ist $a = b$, dann gilt offensichtlich `gcd`$(a, b) = a = b$.
- Andernfalls wissen wir, dass $a = ig$ für irgendein $i \in \mathbb{N}_1$ und das $b = jg$ für irgendein $j \in \mathbb{N}_1$ gilt.
- Ohne Beschränkung der Allgemeinheit nehmen wir an dass $a > b$.
- Dann gilt das $c = a - b = (i - j)g$.
- Es ist klar das $c \bmod g = (a - b) \bmod g = (i - j)g \bmod g = 0$ gilt.
- Es ist auch klar, dass $a - b < a$.

Euklidischer Algorithmus für den Größten Gemeinsamen Teiler

- Der GCD kann mit dem Euclidischen Algorithmus[1,8,10] von Euclid of Alexandria ($E\acute{\upsilon}\kappa\lambda\varepsilon\acute{\iota}\delta\eta\varsigma$) der 300 *before Common Era (BCE)* gelebt hat berechnet werden.
- Der größte gemeinsame Teiler zweier positiver natürlicher Zahlen $a \in \mathbb{N}_1$ und $b \in \mathbb{N}_1$ ist die größte Zahl $g \in \mathbb{N}_1 = \texttt{gcd}(a, b)$ für die gilt das $a \bmod g = 0$ und $b \bmod g = 0$, wobei mod die Modulo Division ist, also der Rest einer Division, was wiederum äquivalent zum Operator `%` von Python ist.
- Das bedeutet, dass g sowohl a als auch b ohne Rest teilt.
- Ist $a = b$, dann gilt offensichtlich $\texttt{gcd}(a, b) = a = b$.
- Andernfalls wissen wir, dass $a = ig$ für irgendein $i \in \mathbb{N}_1$ und das $b = jg$ für irgendein $j \in \mathbb{N}_1$ gilt.
- Ohne Beschränkung der Allgemeinheit nehmen wir an dass $a > b$.
- Dann gilt das $c = a - b = (i - j)g$.
- Es ist klar das $c \bmod g = (a - b) \bmod g = (i - j)g \bmod g = 0$ gilt.
- Es ist auch klar, dass $a - b < a$.
- Anstelle der Differenz c von a und b können wir auch den Rest der Division a/b nehmen.

Euklidischer Algorithmus für den Größten Gemeinsamen Teiler

- Der größte gemeinsame Teiler zweier positiver natürlicher Zahlen $a \in \mathbb{N}_1$ und $b \in \mathbb{N}_1$ ist die größte Zahl $g \in \mathbb{N}_1 = \texttt{gcd}(a, b)$ für die gilt das $a \bmod g = 0$ und $b \bmod g = 0$, wobei $\bmod$ die Modulo Division ist, also der Rest einer Division, was wiederum äquivalent zum Operator `%` von Python ist.
- Das bedeutet, dass g sowohl a als auch b ohne Rest teilt.
- Ist $a = b$, dann gilt offensichtlich $\texttt{gcd}(a, b) = a = b$.
- Andernfalls wissen wir, dass $a = ig$ für irgendein $i \in \mathbb{N}_1$ und das $b = jg$ für irgendein $j \in \mathbb{N}_1$ gilt.
- Ohne Beschränkung der Allgemeinheit nehmen wir an dass $a > b$.
- Dann gilt das $c = a - b = (i - j)g$.
- Es ist klar das $c \bmod g = (a - b) \bmod g = (i - j)g \bmod g = 0$ gilt.
- Es ist auch klar, dass $a - b < a$.
- Anstelle der Differenz c von a und b können wir auch den Rest der Division a/b nehmen:
- $d = a \bmod b = ig \bmod (jg) = ig - \lfloor i/j \rfloor * jg = g(i - j\lfloor i/j \rfloor)$ ist natürlich immer noch teilbar durch g, also hat $d \bmod g = 0$.

Euklidischer Algorithmus für den Größten Gemeinsamen Teiler

- Das bedeutet, dass g sowohl a als auch b ohne Rest teilt.
- Ist $a = b$, dann gilt offensichtlich `gcd`$(a, b) = a = b$.
- Andernfalls wissen wir, dass $a = ig$ für irgendein $i \in \mathbb{N}_1$ und das $b = jg$ für irgendein $j \in \mathbb{N}_1$ gilt.
- Ohne Beschränkung der Allgemeinheit nehmen wir an dass $a > b$.
- Dann gilt das $c = a - b = (i - j)g$.
- Es ist klar das $c \bmod g = (a - b) \bmod g = (i - j)g \bmod g = 0$ gilt.
- Es ist auch klar, dass $a - b < a$.
- Anstelle der Differenz c von a und b können wir auch den Rest der Division a/b nehmen:
- $d = a \bmod b = ig \bmod (jg) = ig - \lfloor i/j \rfloor * jg = g(i - j\lfloor i/j \rfloor)$ ist natürlich immer noch teilbar durch g, also hat $d \bmod g = 0$.
- Und natürlich ist auch $a \bmod b < a$.

Euklidischer Algorithmus für den Größten Gemeinsamen Teiler

- Ist $a = b$, dann gilt offensichtlich $\texttt{gcd}(a, b) = a = b$.
- Andernfalls wissen wir, dass $a = ig$ für irgendein $i \in \mathbb{N}_1$ und das $b = jg$ für irgendein $j \in \mathbb{N}_1$ gilt.
- Ohne Beschränkung der Allgemeinheit nehmen wir an dass $a > b$.
- Dann gilt das $c = a - b = (i - j)g$.
- Es ist klar das $c \bmod g = (a - b) \bmod g = (i - j)g \bmod g = 0$ gilt.
- Es ist auch klar, dass $a - b < a$.
- Anstelle der Differenz c von a und b können wir auch den Rest der Division a/b nehmen:
- $d = a \bmod b = ig \bmod (jg) = ig - \lfloor i/j \rfloor * jg = g(i - j\lfloor i/j \rfloor)$ ist natürlich immer noch teilbar durch g, also hat $d \bmod g = 0$.
- Und natürlich ist auch $a \bmod b < a$.
- Weil sowohl d als auch c kleiner als a sind, aber immer noch durch g teilbar sind, können wir a mit einem von ihnen ersetzen.

Euklidischer Algorithmus für den Größten Gemeinsamen Teiler

- Andernfalls wissen wir, dass $a = ig$ für irgendein $i \in \mathbb{N}_1$ und das $b = jg$ für irgendein $j \in \mathbb{N}_1$ gilt.
- Ohne Beschränkung der Allgemeinheit nehmen wir an dass $a > b$.
- Dann gilt das $c = a - b = (i - j)g$.
- Es ist klar das $c \bmod g = (a - b) \bmod g = (i - j)g \bmod g = 0$ gilt.
- Es ist auch klar, dass $a - b < a$.
- Anstelle der Differenz c von a und b können wir auch den Rest der Division a/b nehmen:
- $d = a \bmod b = ig \bmod (jg) = ig - \lfloor i/j \rfloor * jg = g(i - j\lfloor i/j \rfloor)$ ist natürlich immer noch teilbar durch g, also hat $d \bmod g = 0$.
- Und natürlich ist auch $a \bmod b < a$.
- Weil sowohl d als auch c kleiner als a sind, aber immer noch durch g teilbar sind, können wir a mit einem von ihnen ersetzen.
- Das Tolle an d ist, dass $d < a$ und $d < b$.

Euklidischer Algorithmus für den Größten Gemeinsamen Teiler

- Ohne Beschränkung der Allgemeinheit nehmen wir an dass $a > b$.
- Dann gilt das $c = a - b = (i - j)g$.
- Es ist klar das $c \bmod g = (a - b) \bmod g = (i - j)g \bmod g = 0$ gilt.
- Es ist auch klar, dass $a - b < a$.
- Anstelle der Differenz c von a und b können wir auch den Rest der Division a/b nehmen:
- $d = a \bmod b = ig \bmod (jg) = ig - \lfloor i/j \rfloor * jg = g(i - j\lfloor i/j \rfloor)$ ist natürlich immer noch teilbar durch g, also hat $d \bmod g = 0$.
- Und natürlich ist auch $a \bmod b < a$.
- Weil sowohl d als auch c kleiner als a sind, aber immer noch durch g teilbar sind, können wir a mit einem von ihnen ersetzen.
- Das Tolle an d ist, dass $d < a$ und $d < b$.
- Wir könnten also das „alte“ a mit b ersetzen und d in der Variable b speichern.

Euklidischer Algorithmus für den Größten Gemeinsamen Teiler

- Dann gilt das $c = a - b = (i - j)g$.
- Es ist klar das $c \bmod g = (a - b) \bmod g = (i - j)g \bmod g = 0$ gilt.
- Es ist auch klar, dass $a - b < a$.
- Anstelle der Differenz c von a und b können wir auch den Rest der Division a/b nehmen:
- $d = a \bmod b = ig \bmod (jg) = ig - \lfloor i/j \rfloor * jg = g(i - j\lfloor i/j \rfloor)$ ist natürlich immer noch teilbar durch g, also hat $d \bmod g = 0$.
- Und natürlich ist auch $a \bmod b < a$.
- Weil sowohl d als auch c kleiner als a sind, aber immer noch durch g teilbar sind, können wir a mit einem von ihnen ersetzen.
- Das Tolle an d ist, dass $d < a$ und $d < b$.
- Wir könnten also das „alte“ a mit b ersetzen und d in der Variable b speichern.
- Wenn wir das immer wieder wiederholen, dann werden unsere Werte immer kleiner, bleiben aber durch g teilbar.

Euklidischer Algorithmus für den Größten Gemeinsamen Teiler

- Es ist klar das $c \bmod g = (a-b) \bmod g = (i-j)g \bmod g = 0$ gilt.
- Es ist auch klar, dass $a - b < a$.
- Anstelle der Differenz c von a und b können wir auch den Rest der Division a/b nehmen:
- $d = a \bmod b = ig \bmod (jg) = ig - \lfloor i/j \rfloor * jg = g(i - j\lfloor i/j \rfloor)$ ist natürlich immer noch teilbar durch g, also hat $d \bmod g = 0$.
- Und natürlich ist auch $a \bmod b < a$.
- Weil sowohl d als auch c kleiner als a sind, aber immer noch durch g teilbar sind, können wir a mit einem von ihnen ersetzen.
- Das Tolle an d ist, dass $d < a$ und $d < b$.
- Wir könnten also das „alte“ a mit b ersetzen und d in der Variable b speichern.
- Wenn wir das immer wieder wiederholen, dann werden unsere Werte immer kleiner, bleiben aber durch g teilbar.
- Irgendwann kommen wir bei $b = 0$ und $a = g$ an.

Euklidischer Algorithmus für den Größten Gemeinsamen Teiler

- Es ist auch klar, dass $a - b < a$.
- Anstelle der Differenz c von a und b können wir auch den Rest der Division a/b nehmen:
- $d = a \bmod b = ig \bmod (jg) = ig - \lfloor i/j \rfloor * jg = g(i - j\lfloor i/j \rfloor)$ ist natürlich immer noch teilbar durch g, also hat $d \bmod g = 0$.
- Und natürlich ist auch $a \bmod b < a$.
- Weil sowohl d als auch c kleiner als a sind, aber immer noch durch g teilbar sind, können wir a mit einem von ihnen ersetzen.
- Das Tolle an d ist, dass $d < a$ und $d < b$.
- Wir könnten also das „alte“ a mit b ersetzen und d in der Variable b speichern.
- Wenn wir das immer wieder wiederholen, dann werden unsere Werte immer kleiner, bleiben aber durch g teilbar.
- Irgendwann kommen wir bei $b = 0$ und $a = g$ an.
- Wir müssen noch nicht mal anehmen, dass $a > b$.

Euklidischer Algorithmus für den Größten Gemeinsamen Teiler

- Anstelle der Differenz c von a und b können wir auch den Rest der Division a/b nehmen:
- $d = a \bmod b = ig \bmod (jg) = ig - \lfloor i/j \rfloor * jg = g(i - j\lfloor i/j \rfloor)$ ist natürlich immer noch teilbar durch g, also hat $d \bmod g = 0$.
- Und natürlich ist auch $a \bmod b < a$.
- Weil sowohl d als auch c kleiner als a sind, aber immer noch durch g teilbar sind, können wir a mit einem von ihnen ersetzen.
- Das Tolle an d ist, dass $d < a$ und $d < b$.
- Wir könnten also das „alte" a mit b ersetzen und d in der Variable b speichern.
- Wenn wir das immer wieder wiederholen, dann werden unsere Werte immer kleiner, bleiben aber durch g teilbar.
- Irgendwann kommen wir bei $b = 0$ und $a = g$ an.
- Wir müssen noch nicht mal anehmen, dass $a > b$.
- Wenn $b > a$ gilt, dann wäre $a \bmod b = a$ und wir würden im ersten Rechenschritt a und b tauschen.

Euklidischer Algorithmus für den Größten Gemeinsamen Teiler

- $d = a \bmod b = ig \bmod (jg) = ig - \lfloor i/j \rfloor * jg = g(i - j\lfloor i/j \rfloor)$ ist natürlich immer noch teilbar durch g, also hat $d \bmod g = 0$.
- Und natürlich ist auch $a \bmod b < a$.
- Weil sowohl d als auch c kleiner als a sind, aber immer noch durch g teilbar sind, können wir a mit einem von ihnen ersetzen.
- Das Tolle an d ist, dass $d < a$ und $d < b$.
- Wir könnten also das „alte" a mit b ersetzen und d in der Variable b speichern.
- Wenn wir das immer wieder wiederholen, dann werden unsere Werte immer kleiner, bleiben aber durch g teilbar.
- Irgendwann kommen wir bei $b = 0$ und $a = g$ an.
- Wir müssen noch nicht mal anehmen, dass $a > b$.
- Wenn $b > a$ gilt, dann wäre $a \bmod b = a$ und wir würden im ersten Rechenschritt a und b tauschen.
- Wären $a = b$, dann ist $a \bmod b = 0$ und wir würden im ersten Rechenschritt aufhören und hätten a als größten gemeinsamen Teiler.

Euklidischer Algorithmus für den Größten Gemeinsamen Teiler

- Implementieren wir das also als Funktion.

Euklidischer Algorithmus für den Größten Gemeinsamen Teiler

- Implementieren wir das also als Funktion.
- Unsere neue Funktion `gcd` hat zwei ganzzahlige Parameters, `a` und `b`.

```python
"""Euclidian Algorithm for the Greatest Common Divisor as a function."""

from math import gcd as math_gcd  # Use math's gcd under name `math_gcd
  ↪ `.


def gcd(a: int, b: int) -> int:  # 2 `int` parameters and `int` result
    """
    Compute the greatest common divisor of two numbers `a` and `b`.

    :param a: the first number
    :param b: the second number
    :return: the greatest common divisor of `a` and `b`
    """
    while b != 0:  # Repeat in a loop until `b == 0`.
        a, b = b, a % b  # the same as `t = b`; `b = a % b`; `b = t`.
    return a  # If `b` becomes `0`, then the gcd is in `a`.


def print_gcd(a: int, b: int) -> None:  # `-> None` == returns nothing
    """
    Print the result of the gcd of `a` and `b`.

    :param a: the first number
    :param b: the second number
    """
    print(f"gcd({a}, {b})={gcd(a, b)}, math_gcd={math_gcd(a, b)}.")
    # Notice: no `return` statement. Because we return nothing.


print_gcd(1, 0)
print_gcd(0, 1)
print_gcd(765, 273)
print_gcd(24359573700, 35943207300)
```

↓ `python3 def_gcd.py` ↓

```
gcd(1, 0)=1, math_gcd=1.
gcd(0, 1)=1, math_gcd=1.
gcd(765, 273)=3, math_gcd=3.
gcd(24359573700, 35943207300)=2148300, math_gcd=2148300.
```

Euklidischer Algorithmus für den Größten Gemeinsamen Teiler

- Implementieren wir das also als Funktion.
- Unsere neue Funktion `gcd` hat zwei ganzzahlige Parameters, `a` und `b`.
- Sie liefert eine andere Ganzzahl zurück.

```python
"""Euclidian Algorithm for the Greatest Common Divisor as a function."""

from math import gcd as math_gcd  # Use math's gcd under name `math_gcd
    ↪ `.


def gcd(a: int, b: int) -> int:  # 2 `int` parameters and `int` result
    """
    Compute the greatest common divisor of two numbers `a` and `b`.

    :param a: the first number
    :param b: the second number
    :return: the greatest common divisor of `a` and `b`
    """
    while b != 0:  # Repeat in a loop until `b == 0`.
        a, b = b, a % b  # the same as `t = b`; `b = a % b`; `b = t`.
    return a  # If `b` becomes `0`, then the gcd is in `a`.


def print_gcd(a: int, b: int) -> None:  # `-> None` == returns nothing
    """
    Print the result of the gcd of `a` and `b`.

    :param a: the first number
    :param b: the second number
    """
    print(f"gcd({a}, {b})={gcd(a, b)}, math_gcd={math_gcd(a, b)}.")
    # Notice: no `return` statement. Because we return nothing.


print_gcd(1, 0)
print_gcd(0, 1)
print_gcd(765, 273)
print_gcd(24359573700, 35943207300)
```

↓ `python3 def_gcd.py` ↓

```
gcd(1, 0)=1, math_gcd=1.
gcd(0, 1)=1, math_gcd=1.
gcd(765, 273)=3, math_gcd=3.
gcd(24359573700, 35943207300)=2148300, math_gcd=2148300.
```

Euklidischer Algorithmus für den Größten Gemeinsamen Teiler

- Implementieren wir das also als Funktion.
- Unsere neue Funktion `gcd` hat zwei ganzzahlige Parameters, `a` und `b`.
- Sie liefert eine andere Ganzzahl zurück.
- Beachten Sie, dass wir alles mit Type Hints annotiert haben.

```python
"""Euclidian Algorithm for the Greatest Common Divisor as a function."""

from math import gcd as math_gcd  # Use math's gcd under name `math_gcd`.


def gcd(a: int, b: int) -> int:  # 2 `int` parameters and `int` result
    """
    Compute the greatest common divisor of two numbers `a` and `b`.

    :param a: the first number
    :param b: the second number
    :return: the greatest common divisor of `a` and `b`
    """
    while b != 0:  # Repeat in a loop until `b == 0`.
        a, b = b, a % b  # the same as `t = b`; `b = a % b`; `b = t`.
    return a  # If `b` becomes `0`, then the gcd is in `a`.


def print_gcd(a: int, b: int) -> None:  # `-> None` == returns nothing
    """
    Print the result of the gcd of `a` and `b`.

    :param a: the first number
    :param b: the second number
    """
    print(f"gcd({a}, {b})={gcd(a, b)}, math_gcd={math_gcd(a, b)}.")
    # Notice: no `return` statement. Because we return nothing.


print_gcd(1, 0)
print_gcd(0, 1)
print_gcd(765, 273)
print_gcd(24359573700, 35943207300)
```

↓ `python3 def_gcd.py` ↓

```
gcd(1, 0)=1, math_gcd=1.
gcd(0, 1)=1, math_gcd=1.
gcd(765, 273)=3, math_gcd=3.
gcd(24359573700, 35943207300)=2148300, math_gcd=2148300.
```

Euklidischer Algorithmus für den Größten Gemeinsamen Teiler

- Implementieren wir das also als Funktion.
- Unsere neue Funktion `gcd` hat zwei ganzzahlige Parameters, `a` und `b`.
- Sie liefert eine andere Ganzzahl zurück.
- Beachten Sie, dass wir alles mit Type Hints annotiert haben.
- Nach dem Kopf der Funktion folgt ein Docstring, in dem die Funktion, die Parameter, und der Rückgabewert ordentlich beschrieben sind.

```python
"""Euclidian Algorithm for the Greatest Common Divisor as a function."""

from math import gcd as math_gcd  # Use math's gcd under name `math_gcd`.


def gcd(a: int, b: int) -> int:  # 2 `int` parameters and `int` result
    """
    Compute the greatest common divisor of two numbers `a` and `b`.

    :param a: the first number
    :param b: the second number
    :return: the greatest common divisor of `a` and `b`
    """
    while b != 0:  # Repeat in a loop until `b == 0`.
        a, b = b, a % b  # the same as `t = b`; `b = a % b`; `b = t`.
    return a  # If `b` becomes `0`, then the gcd is in `a`.


def print_gcd(a: int, b: int) -> None:  # `-> None` == returns nothing
    """
    Print the result of the gcd of `a` and `b`.

    :param a: the first number
    :param b: the second number
    """
    print(f"gcd({a}, {b})={gcd(a, b)}, math_gcd={math_gcd(a, b)}.")
    # Notice: no `return` statement. Because we return nothing.


print_gcd(1, 0)
print_gcd(0, 1)
print_gcd(765, 273)
print_gcd(24359573700, 35943207300)
```

↓ `python3 def_gcd.py` ↓

```
gcd(1, 0)=1, math_gcd=1.
gcd(0, 1)=1, math_gcd=1.
gcd(765, 273)=3, math_gcd=3.
gcd(24359573700, 35943207300)=2148300, math_gcd=2148300.
```

Euklidischer Algorithmus für den Größten Gemeinsamen Teiler

- Unsere neue Funktion `gcd` hat zwei ganzzahlige Parameters, `a` und `b`.
- Sie liefert eine andere Ganzzahl zurück.
- Beachten Sie, dass wir alles mit Type Hints annotiert haben.
- Nach dem Kopf der Funktion folgt ein Docstring, in dem die Funktion, die Parameter, und der Rückgabewert ordentlich beschrieben sind.
- Der Körper der Funktion ist überraschend kurz.

```python
"""Euclidian Algorithm for the Greatest Common Divisor as a function."""

from math import gcd as math_gcd  # Use math's gcd under name `math_gcd
    ↪ `.


def gcd(a: int, b: int) -> int:  # 2 `int` parameters and `int` result
    """
    Compute the greatest common divisor of two numbers `a` and `b`.

    :param a: the first number
    :param b: the second number
    :return: the greatest common divisor of `a` and `b`
    """
    while b != 0:  # Repeat in a loop until `b == 0`.
        a, b = b, a % b  # the same as `t = b`; `b = a % b`; `b = t`.
    return a  # If `b` becomes `0`, then the gcd is in `a`.


def print_gcd(a: int, b: int) -> None:  # `-> None` == returns nothing
    """
    Print the result of the gcd of `a` and `b`.

    :param a: the first number
    :param b: the second number
    """
    print(f"gcd({a}, {b})={gcd(a, b)}, math_gcd={math_gcd(a, b)}.")
    # Notice: no `return` statement. Because we return nothing.


print_gcd(1, 0)
print_gcd(0, 1)
print_gcd(765, 273)
print_gcd(24359573700, 35943207300)
```

↓ `python3 def_gcd.py` ↓

```
gcd(1, 0)=1, math_gcd=1.
gcd(0, 1)=1, math_gcd=1.
gcd(765, 273)=3, math_gcd=3.
gcd(24359573700, 35943207300)=2148300, math_gcd=2148300.
```

Euklidischer Algorithmus für den Größten Gemeinsamen Teiler

- Sie liefert eine andere Ganzzahl zurück.
- Beachten Sie, dass wir alles mit Type Hints annotiert haben.
- Nach dem Kopf der Funktion folgt ein Docstring, in dem die Funktion, die Parameter, und der Rückgabewert ordentlich beschrieben sind.
- Der Körper der Funktion ist überraschend kurz.
- Wir benutzen eine `while`-Schleife, die so lange iteriert, wie `b > 0`.

```python
"""Euclidian Algorithm for the Greatest Common Divisor as a function."""

from math import gcd as math_gcd  # Use math's gcd under name `math_gcd`.


def gcd(a: int, b: int) -> int:  # 2 `int` parameters and `int` result
    """
    Compute the greatest common divisor of two numbers `a` and `b`.

    :param a: the first number
    :param b: the second number
    :return: the greatest common divisor of `a` and `b`
    """
    while b != 0:  # Repeat in a loop until `b == 0`.
        a, b = b, a % b  # the same as `t = b`; `b = a % b`; `b = t`.
    return a  # If `b` becomes `0`, then the gcd is in `a`.


def print_gcd(a: int, b: int) -> None:  # `-> None` == returns nothing
    """
    Print the result of the gcd of `a` and `b`.

    :param a: the first number
    :param b: the second number
    """
    print(f"gcd({a}, {b})={gcd(a, b)}, math_gcd={math_gcd(a, b)}.")
    # Notice: no `return` statement. Because we return nothing.


print_gcd(1, 0)
print_gcd(0, 1)
print_gcd(765, 273)
print_gcd(24359573700, 35943207300)
```

↓ `python3 def_gcd.py` ↓

```
gcd(1, 0)=1, math_gcd=1.
gcd(0, 1)=1, math_gcd=1.
gcd(765, 273)=3, math_gcd=3.
gcd(24359573700, 35943207300)=2148300, math_gcd=2148300.
```

Euklidischer Algorithmus für den Größten Gemeinsamen Teiler

- Beachten Sie, dass wir alles mit Type Hints annotiert haben.
- Nach dem Kopf der Funktion folgt ein Docstring, in dem die Funktion, die Parameter, und der Rückgabewert ordentlich beschrieben sind.
- Der Körper der Funktion ist überraschend kurz.
- Wir benutzen eine `while`-Schleife, die so lange iteriert, wie `b > 0`.
- Nach der Schleife geben wir `a` als Ergebnis zurück, in dem wir `return a` aufrufen.

```python
"""Euclidian Algorithm for the Greatest Common Divisor as a function."""

from math import gcd as math_gcd  # Use math's gcd under name `math_gcd`.


def gcd(a: int, b: int) -> int:  # 2 `int` parameters and `int` result
    """
    Compute the greatest common divisor of two numbers `a` and `b`.

    :param a: the first number
    :param b: the second number
    :return: the greatest common divisor of `a` and `b`
    """
    while b != 0:  # Repeat in a loop until `b == 0`.
        a, b = b, a % b  # the same as `t = b`; `b = a % b`; `b = t`.
    return a  # If `b` becomes `0`, then the gcd is in `a`.


def print_gcd(a: int, b: int) -> None:  # `-> None` == returns nothing
    """
    Print the result of the gcd of `a` and `b`.

    :param a: the first number
    :param b: the second number
    """
    print(f"gcd({a}, {b})={gcd(a, b)}, math_gcd={math_gcd(a, b)}.")
    # Notice: no `return` statement. Because we return nothing.


print_gcd(1, 0)
print_gcd(0, 1)
print_gcd(765, 273)
print_gcd(24359573700, 35943207300)
```

↓ `python3 def_gcd.py` ↓

```
gcd(1, 0)=1, math_gcd=1.
gcd(0, 1)=1, math_gcd=1.
gcd(765, 273)=3, math_gcd=3.
gcd(24359573700, 35943207300)=2148300, math_gcd=2148300.
```

Euklidischer Algorithmus für den Größten Gemeinsamen Teiler

- Nach dem Kopf der Funktion folgt ein Docstring, in dem die Funktion, die Parameter, und der Rückgabewert ordentlich beschrieben sind.
- Der Körper der Funktion ist überraschend kurz.
- Wir benutzen eine `while`-Schleife, die so lange iteriert, wie `b > 0`.
- Nach der Schleife geben wir `a` als Ergebnis zurück, in dem wir `return a` aufrufen.
- Sollte `b == 0` sein, dann wird die Schleife gar nicht erst ausgeführt und `a` direkt zurückgeliefert, was richtig ist: Es gilt $\texttt{gcd}(a, 0) = a$ für alle $a \in \mathbb{N}_1$.

```python
"""Euclidian Algorithm for the Greatest Common Divisor as a function."""

from math import gcd as math_gcd  # Use math's gcd under name `math_gcd`.


def gcd(a: int, b: int) -> int:  # 2 `int` parameters and `int` result
    """
    Compute the greatest common divisor of two numbers `a` and `b`.

    :param a: the first number
    :param b: the second number
    :return: the greatest common divisor of `a` and `b`
    """
    while b != 0:  # Repeat in a loop until `b == 0`.
        a, b = b, a % b  # the same as `t = b`; `b = a % b`; `b = t`.
    return a  # If `b` becomes `0`, then the gcd is in `a`.


def print_gcd(a: int, b: int) -> None:  # `-> None` == returns nothing
    """
    Print the result of the gcd of `a` and `b`.

    :param a: the first number
    :param b: the second number
    """
    print(f"gcd({a}, {b})={gcd(a, b)}, math_gcd={math_gcd(a, b)}.")
    # Notice: no `return` statement. Because we return nothing.


print_gcd(1, 0)
print_gcd(0, 1)
print_gcd(765, 273)
print_gcd(24359573700, 35943207300)
```

↓ `python3 def_gcd.py` ↓

```
gcd(1, 0)=1, math_gcd=1.
gcd(0, 1)=1, math_gcd=1.
gcd(765, 273)=3, math_gcd=3.
gcd(24359573700, 35943207300)=2148300, math_gcd=2148300.
```

Euklidischer Algorithmus für den Größten Gemeinsamen Teiler

- Der Körper der Funktion ist überraschend kurz.
- Wir benutzen eine `while`-Schleife, die so lange iteriert, wie `b > 0`.
- Nach der Schleife geben wir `a` als Ergebnis zurück, in dem wir `return a` aufrufen.
- Sollte `b == 0` sein, dann wird die Schleife gar nicht erst ausgeführt und `a` direkt zurückgeliefert, was richtig ist: Es gilt `gcd`$(a, 0) = a$ für alle $a \in \mathbb{N}_1$.
- Ist aber `b > 0`, dann wird der Körper der Schleife ausgeführt.

```python
"""Euclidian Algorithm for the Greatest Common Divisor as a function."""

from math import gcd as math_gcd  # Use math's gcd under name `math_gcd`.


def gcd(a: int, b: int) -> int:  # 2 `int` parameters and `int` result
    """
    Compute the greatest common divisor of two numbers `a` and `b`.

    :param a: the first number
    :param b: the second number
    :return: the greatest common divisor of `a` and `b`
    """
    while b != 0:  # Repeat in a loop until `b == 0`.
        a, b = b, a % b  # the same as `t = b`; `b = a % b`; `b = t`.
    return a  # If `b` becomes `0`, then the gcd is in `a`.


def print_gcd(a: int, b: int) -> None:  # `-> None` == returns nothing
    """
    Print the result of the gcd of `a` and `b`.

    :param a: the first number
    :param b: the second number
    """
    print(f"gcd({a}, {b})={gcd(a, b)}, math_gcd={math_gcd(a, b)}.")
    # Notice: no `return` statement. Because we return nothing.


print_gcd(1, 0)
print_gcd(0, 1)
print_gcd(765, 273)
print_gcd(24359573700, 35943207300)
```

↓ `python3 def_gcd.py` ↓

```
gcd(1, 0)=1, math_gcd=1.
gcd(0, 1)=1, math_gcd=1.
gcd(765, 273)=3, math_gcd=3.
gcd(24359573700, 35943207300)=2148300, math_gcd=2148300.
```

Euklidischer Algorithmus für den Größten Gemeinsamen Teiler

- Wir benutzen eine `while`-Schleife, die so lange iteriert, wie `b > 0`.
- Nach der Schleife geben wir `a` als Ergebnis zurück, in dem wir `return a` aufrufen.
- Sollte `b == 0` sein, dann wird die Schleife gar nicht erst ausgeführt und `a` direkt zurückgeliefert, was richtig ist: Es gilt $\texttt{gcd}(a, 0) = a$ für alle $a \in \mathbb{N}_1$.
- Ist aber `b > 0`, dann wird der Körper der Schleife ausgeführt.
- Der Körper ist nur eine einzige Zeile Kode.

```python
"""Euclidian Algorithm for the Greatest Common Divisor as a function."""

from math import gcd as math_gcd  # Use math's gcd under name `math_gcd`.


def gcd(a: int, b: int) -> int:  # 2 `int` parameters and `int` result
    """
    Compute the greatest common divisor of two numbers `a` and `b`.

    :param a: the first number
    :param b: the second number
    :return: the greatest common divisor of `a` and `b`
    """
    while b != 0:  # Repeat in a loop until `b == 0`.
        a, b = b, a % b  # the same as `t = b`; `b = a % b`; `b = t`.
    return a  # If `b` becomes `0`, then the gcd is in `a`.


def print_gcd(a: int, b: int) -> None:  # `-> None` == returns nothing
    """
    Print the result of the gcd of `a` and `b`.

    :param a: the first number
    :param b: the second number
    """
    print(f"gcd({a}, {b})={gcd(a, b)}, math_gcd={math_gcd(a, b)}.")
    # Notice: no `return` statement. Because we return nothing.


print_gcd(1, 0)
print_gcd(0, 1)
print_gcd(765, 273)
print_gcd(24359573700, 35943207300)
```

↓ `python3 def_gcd.py` ↓

```
gcd(1, 0)=1, math_gcd=1.
gcd(0, 1)=1, math_gcd=1.
gcd(765, 273)=3, math_gcd=3.
gcd(24359573700, 35943207300)=2148300, math_gcd=2148300.
```

Euklidischer Algorithmus für den Größten Gemeinsamen Teiler

- Nach der Schleife geben wir `a` als Ergebnis zurück, in dem wir `return a` aufrufen.
- Sollte `b == 0` sein, dann wird die Schleife gar nicht erst ausgeführt und `a` direkt zurückgeliefert, was richtig ist: Es gilt $\texttt{gcd}(a, 0) = a$ für alle $a \in \mathbb{N}_1$.
- Ist aber `b > 0`, dann wird der Körper der Schleife ausgeführt.
- Der Körper ist nur eine einzige Zeile Kode: der Mehrfachzuweisung `a, b = b, a %b`.

```python
"""Euclidian Algorithm for the Greatest Common Divisor as a function."""

from math import gcd as math_gcd  # Use math's gcd under name `math_gcd`.


def gcd(a: int, b: int) -> int:  # 2 `int` parameters and `int` result
    """
    Compute the greatest common divisor of two numbers `a` and `b`.

    :param a: the first number
    :param b: the second number
    :return: the greatest common divisor of `a` and `b`
    """
    while b != 0:  # Repeat in a loop until `b == 0`.
        a, b = b, a % b  # the same as `t = b`; `b = a % b`; `b = t`.
    return a  # If `b` becomes `0`, then the gcd is in `a`.


def print_gcd(a: int, b: int) -> None:  # `-> None` == returns nothing
    """
    Print the result of the gcd of `a` and `b`.

    :param a: the first number
    :param b: the second number
    """
    print(f"gcd({a}, {b})={gcd(a, b)}, math_gcd={math_gcd(a, b)}.")
    # Notice: no `return` statement. Because we return nothing.


print_gcd(1, 0)
print_gcd(0, 1)
print_gcd(765, 273)
print_gcd(24359573700, 35943207300)
```

↓ `python3 def_gcd.py` ↓

```
gcd(1, 0)=1, math_gcd=1.
gcd(0, 1)=1, math_gcd=1.
gcd(765, 273)=3, math_gcd=3.
gcd(24359573700, 35943207300)=2148300, math_gcd=2148300.
```

Euklidischer Algorithmus für den Größten Gemeinsamen Teiler

- Sollte `b == 0` sein, dann wird die Schleife gar nicht erst ausgeführt und `a` direkt zurückgeliefert, was richtig ist: Es gilt $\texttt{gcd}(a, 0) = a$ für alle $a \in \mathbb{N}_1$.
- Ist aber `b > 0`, dann wird der Körper der Schleife ausgeführt.
- Der Körper ist nur eine einzige Zeile Kode: der Mehrfachzuweisung `a, b = b, a %b`.
- Diese Zeile funktioniert in etwa so:

```python
"""Euclidian Algorithm for the Greatest Common Divisor as a function."""

from math import gcd as math_gcd  # Use math's gcd under name `math_gcd`.


def gcd(a: int, b: int) -> int:  # 2 `int` parameters and `int` result
    """
    Compute the greatest common divisor of two numbers `a` and `b`.

    :param a: the first number
    :param b: the second number
    :return: the greatest common divisor of `a` and `b`
    """
    while b != 0:  # Repeat in a loop until `b == 0`.
        a, b = b, a % b  # the same as `t = b`; `b = a % b`; `b = t`.
    return a  # If `b` becomes `0`, then the gcd is in `a`.


def print_gcd(a: int, b: int) -> None:  # `-> None` == returns nothing
    """
    Print the result of the gcd of `a` and `b`.

    :param a: the first number
    :param b: the second number
    """
    print(f"gcd({a}, {b})={gcd(a, b)}, math_gcd={math_gcd(a, b)}.")
    # Notice: no `return` statement. Because we return nothing.


print_gcd(1, 0)
print_gcd(0, 1)
print_gcd(765, 273)
print_gcd(24359573700, 35943207300)
```

↓ `python3 def_gcd.py` ↓

```
gcd(1, 0)=1, math_gcd=1.
gcd(0, 1)=1, math_gcd=1.
gcd(765, 273)=3, math_gcd=3.
gcd(24359573700, 35943207300)=2148300, math_gcd=2148300.
```

Euklidischer Algorithmus für den Größten Gemeinsamen Teiler

- Ist aber `b > 0`, dann wird der Körper der Schleife ausgeführt.
- Der Körper ist nur eine einzige Zeile Kode: der Mehrfachzuweisung `a, b = b, a %b`.
- Diese Zeile funktioniert in etwa so:
- Zuerst wird die rechte Seite ausgewertet.

```python
"""Euclidian Algorithm for the Greatest Common Divisor as a function."""

from math import gcd as math_gcd  # Use math's gcd under name `math_gcd`.


def gcd(a: int, b: int) -> int:  # 2 `int` parameters and `int` result
    """
    Compute the greatest common divisor of two numbers `a` and `b`.

    :param a: the first number
    :param b: the second number
    :return: the greatest common divisor of `a` and `b`
    """
    while b != 0:  # Repeat in a loop until `b == 0`.
        a, b = b, a % b  # the same as `t = b`; `b = a % b`; `b = t`.
    return a  # If `b` becomes `0`, then the gcd is in `a`.


def print_gcd(a: int, b: int) -> None:  # `-> None` == returns nothing
    """
    Print the result of the gcd of `a` and `b`.

    :param a: the first number
    :param b: the second number
    """
    print(f"gcd({a}, {b})={gcd(a, b)}, math_gcd={math_gcd(a, b)}.")
    # Notice: no `return` statement. Because we return nothing.


print_gcd(1, 0)
print_gcd(0, 1)
print_gcd(765, 273)
print_gcd(24359573700, 35943207300)
```

↓ `python3 def_gcd.py` ↓

```
gcd(1, 0)=1, math_gcd=1.
gcd(0, 1)=1, math_gcd=1.
gcd(765, 273)=3, math_gcd=3.
gcd(24359573700, 35943207300)=2148300, math_gcd=2148300.
```

Euklidischer Algorithmus für den Größten Gemeinsamen Teiler

- Der Körper ist nur eine einzige Zeile Kode: der Mehrfachzuweisung `a, b = b, a %b`.
- Diese Zeile funktioniert in etwa so:
- Zuerst wird die rechte Seite ausgewertet.
- Es entsteht ein Tupel dessen erster Wert `b` und dessen zweiter Wert `a % b` ist.

```python
"""Euclidian Algorithm for the Greatest Common Divisor as a function."""

from math import gcd as math_gcd  # Use math's gcd under name `math_gcd
    ↪ `.


def gcd(a: int, b: int) -> int:  # 2 `int` parameters and `int` result
    """
    Compute the greatest common divisor of two numbers `a` and `b`.

    :param a: the first number
    :param b: the second number
    :return: the greatest common divisor of `a` and `b`
    """
    while b != 0:  # Repeat in a loop until `b == 0`.
        a, b = b, a % b  # the same as `t = b`; `b = a % b`; `b = t`.
    return a  # If `b` becomes `0`, then the gcd is in `a`.


def print_gcd(a: int, b: int) -> None:  # `-> None` == returns nothing
    """
    Print the result of the gcd of `a` and `b`.

    :param a: the first number
    :param b: the second number
    """
    print(f"gcd({a}, {b})={gcd(a, b)}, math_gcd={math_gcd(a, b)}.")
    # Notice: no `return` statement. Because we return nothing.


print_gcd(1, 0)
print_gcd(0, 1)
print_gcd(765, 273)
print_gcd(24359573700, 35943207300)
```

↓ `python3 def_gcd.py` ↓

```
gcd(1, 0)=1, math_gcd=1.
gcd(0, 1)=1, math_gcd=1.
gcd(765, 273)=3, math_gcd=3.
gcd(24359573700, 35943207300)=2148300, math_gcd=2148300.
```

Euklidischer Algorithmus für den Größten Gemeinsamen Teiler

- Diese Zeile funktioniert in etwa so:
- Zuerst wird die rechte Seite ausgewertet.
- Es entsteht ein Tupel dessen erster Wert `b` und dessen zweiter Wert `a % b` ist.
- Dieses Tupel wird dann ausgepackt, und zwar in die Variablen `a` und `b`.

```python
"""Euclidian Algorithm for the Greatest Common Divisor as a function."""

from math import gcd as math_gcd  # Use math's gcd under name `math_gcd
    ↪ `.


def gcd(a: int, b: int) -> int:  # 2 `int` parameters and `int` result
    """
    Compute the greatest common divisor of two numbers `a` and `b`.

    :param a: the first number
    :param b: the second number
    :return: the greatest common divisor of `a` and `b`
    """
    while b != 0:  # Repeat in a loop until `b == 0`.
        a, b = b, a % b  # the same as `t = b`; `b = a % b`; `b = t`.
    return a  # If `b` becomes `0`, then the gcd is in `a`.


def print_gcd(a: int, b: int) -> None:  # `-> None` == returns nothing
    """
    Print the result of the gcd of `a` and `b`.

    :param a: the first number
    :param b: the second number
    """
    print(f"gcd({a}, {b})={gcd(a, b)}, math_gcd={math_gcd(a, b)}.")
    # Notice: no `return` statement. Because we return nothing.


print_gcd(1, 0)
print_gcd(0, 1)
print_gcd(765, 273)
print_gcd(24359573700, 35943207300)
```

↓ `python3 def_gcd.py` ↓

```
gcd(1, 0)=1, math_gcd=1.
gcd(0, 1)=1, math_gcd=1.
gcd(765, 273)=3, math_gcd=3.
gcd(24359573700, 35943207300)=2148300, math_gcd=2148300.
```

Euklidischer Algorithmus für den Größten Gemeinsamen Teiler

- Zuerst wird die rechte Seite ausgewertet.
- Es entsteht ein Tupel dessen erster Wert `b` und dessen zweiter Wert `a % b` ist.
- Dieses Tupel wird dann ausgepackt, und zwar in die Variablen `a` und `b`.
- `a` bekommt also den alten Wert von `b`.

```python
"""Euclidian Algorithm for the Greatest Common Divisor as a function."""

from math import gcd as math_gcd  # Use math's gcd under name `math_gcd
    ↪ `.


def gcd(a: int, b: int) -> int:  # 2 `int` parameters and `int` result
    """
    Compute the greatest common divisor of two numbers `a` and `b`.

    :param a: the first number
    :param b: the second number
    :return: the greatest common divisor of `a` and `b`
    """
    while b != 0:  # Repeat in a loop until `b == 0`.
        a, b = b, a % b  # the same as `t = b`; `b = a % b`; `b = t`.
    return a  # If `b` becomes `0`, then the gcd is in `a`.


def print_gcd(a: int, b: int) -> None:  # `-> None` == returns nothing
    """
    Print the result of the gcd of `a` and `b`.

    :param a: the first number
    :param b: the second number
    """
    print(f"gcd({a}, {b})={gcd(a, b)}, math_gcd={math_gcd(a, b)}.")
    # Notice: no `return` statement. Because we return nothing.


print_gcd(1, 0)
print_gcd(0, 1)
print_gcd(765, 273)
print_gcd(24359573700, 35943207300)
```

↓ `python3 def_gcd.py` ↓

```
gcd(1, 0)=1, math_gcd=1.
gcd(0, 1)=1, math_gcd=1.
gcd(765, 273)=3, math_gcd=3.
gcd(24359573700, 35943207300)=2148300, math_gcd=2148300.
```

Euklidischer Algorithmus für den Größten Gemeinsamen Teiler

- Es entsteht ein Tupel dessen erster Wert `b` und dessen zweiter Wert `a % b` ist.
- Dieses Tupel wird dann ausgepackt, und zwar in die Variablen `a` und `b`.
- `a` bekommt also den alten Wert von `b`.
- `b` bekommt den vorher ausgerechneten Wert von `a % b`.

```python
"""Euclidian Algorithm for the Greatest Common Divisor as a function."""

from math import gcd as math_gcd  # Use math's gcd under name `math_gcd`.


def gcd(a: int, b: int) -> int:  # 2 `int` parameters and `int` result
    """
    Compute the greatest common divisor of two numbers `a` and `b`.

    :param a: the first number
    :param b: the second number
    :return: the greatest common divisor of `a` and `b`
    """
    while b != 0:  # Repeat in a loop until `b == 0`.
        a, b = b, a % b  # the same as `t = b`; `b = a % b`; `b = t`.
    return a  # If `b` becomes `0`, then the gcd is in `a`.


def print_gcd(a: int, b: int) -> None:  # `-> None` == returns nothing
    """
    Print the result of the gcd of `a` and `b`.

    :param a: the first number
    :param b: the second number
    """
    print(f"gcd({a}, {b})={gcd(a, b)}, math_gcd={math_gcd(a, b)}.")
    # Notice: no `return` statement. Because we return nothing.


print_gcd(1, 0)
print_gcd(0, 1)
print_gcd(765, 273)
print_gcd(24359573700, 35943207300)
```

↓ `python3 def_gcd.py` ↓

```
gcd(1, 0)=1, math_gcd=1.
gcd(0, 1)=1, math_gcd=1.
gcd(765, 273)=3, math_gcd=3.
gcd(24359573700, 35943207300)=2148300, math_gcd=2148300.
```

Euklidischer Algorithmus für den Größten Gemeinsamen Teiler

- Dieses Tupel wird dann ausgepackt, und zwar in die Variablen `a` und `b`.
- `a` bekommt also den alten Wert von `b`.
- `b` bekommt den vorher ausgerechneten Wert von `a % b`.
- Mit anderen Worten, `b` wird in `a` gespeichert und der Rest der Division des alten `a` durch das alte `b` wird in `b` gespeichert.

```python
"""Euclidian Algorithm for the Greatest Common Divisor as a function."""

from math import gcd as math_gcd  # Use math's gcd under name `math_gcd`.


def gcd(a: int, b: int) -> int:  # 2 `int` parameters and `int` result
    """
    Compute the greatest common divisor of two numbers `a` and `b`.

    :param a: the first number
    :param b: the second number
    :return: the greatest common divisor of `a` and `b`
    """
    while b != 0:  # Repeat in a loop until `b == 0`.
        a, b = b, a % b  # the same as `t = b`; `b = a % b`; `b = t`.
    return a  # If `b` becomes `0`, then the gcd is in `a`.


def print_gcd(a: int, b: int) -> None:  # `-> None` == returns nothing
    """
    Print the result of the gcd of `a` and `b`.

    :param a: the first number
    :param b: the second number
    """
    print(f"gcd({a}, {b})={gcd(a, b)}, math_gcd={math_gcd(a, b)}.")
    # Notice: no `return` statement. Because we return nothing.


print_gcd(1, 0)
print_gcd(0, 1)
print_gcd(765, 273)
print_gcd(24359573700, 35943207300)
```

↓ `python3 def_gcd.py` ↓

```
gcd(1, 0)=1, math_gcd=1.
gcd(0, 1)=1, math_gcd=1.
gcd(765, 273)=3, math_gcd=3.
gcd(24359573700, 35943207300)=2148300, math_gcd=2148300.
```

Euklidischer Algorithmus für den Größten Gemeinsamen Teiler

- `a` bekommt also den alten Wert von `b`.
- `b` bekommt den vorher ausgerechneten Wert von `a % b`.
- Mit anderen Worten, `b` wird in `a` gespeichert und der Rest der Division des alten `a` durch das alte `b` wird in `b` gespeichert.
- Es ist klar, dass `b` in jeder Iteration kleiner wird.

```python
"""Euclidian Algorithm for the Greatest Common Divisor as a function."""

from math import gcd as math_gcd  # Use math's gcd under name `math_gcd`.


def gcd(a: int, b: int) -> int:  # 2 `int` parameters and `int` result
    """
    Compute the greatest common divisor of two numbers `a` and `b`.

    :param a: the first number
    :param b: the second number
    :return: the greatest common divisor of `a` and `b`
    """
    while b != 0:  # Repeat in a loop until `b == 0`.
        a, b = b, a % b  # the same as `t = b`; `b = a % b`; `b = t`.
    return a  # If `b` becomes `0`, then the gcd is in `a`.


def print_gcd(a: int, b: int) -> None:  # `-> None` == returns nothing
    """
    Print the result of the gcd of `a` and `b`.

    :param a: the first number
    :param b: the second number
    """
    print(f"gcd({a}, {b})={gcd(a, b)}, math_gcd={math_gcd(a, b)}.")
    # Notice: no `return` statement. Because we return nothing.


print_gcd(1, 0)
print_gcd(0, 1)
print_gcd(765, 273)
print_gcd(24359573700, 35943207300)
```

↓ `python3 def_gcd.py` ↓

```
gcd(1, 0)=1, math_gcd=1.
gcd(0, 1)=1, math_gcd=1.
gcd(765, 273)=3, math_gcd=3.
gcd(24359573700, 35943207300)=2148300, math_gcd=2148300.
```

Euklidischer Algorithmus für den Größten Gemeinsamen Teiler

- `b` bekommt den vorher ausgerechneten Wert von `a % b`.
- Mit anderen Worten, `b` wird in `a` gespeichert und der Rest der Division des alten `a` durch das alte `b` wird in `b` gespeichert.
- Es ist klar, dass `b` in jeder Iteration kleiner wird.
- Da es niemals negativ werden kann, wird es irgendwann `0`.

```python
"""Euclidian Algorithm for the Greatest Common Divisor as a function."""

from math import gcd as math_gcd  # Use math's gcd under name `math_gcd`.


def gcd(a: int, b: int) -> int:  # 2 `int` parameters and `int` result
    """
    Compute the greatest common divisor of two numbers `a` and `b`.

    :param a: the first number
    :param b: the second number
    :return: the greatest common divisor of `a` and `b`
    """
    while b != 0:  # Repeat in a loop until `b == 0`.
        a, b = b, a % b  # the same as `t = b`; `b = a % b`; `b = t`.
    return a  # If `b` becomes `0`, then the gcd is in `a`.


def print_gcd(a: int, b: int) -> None:  # `-> None` == returns nothing
    """
    Print the result of the gcd of `a` and `b`.

    :param a: the first number
    :param b: the second number
    """
    print(f"gcd({a}, {b})={gcd(a, b)}, math_gcd={math_gcd(a, b)}.")
    # Notice: no `return` statement. Because we return nothing.


print_gcd(1, 0)
print_gcd(0, 1)
print_gcd(765, 273)
print_gcd(24359573700, 35943207300)
```

↓ `python3 def_gcd.py` ↓

```
gcd(1, 0)=1, math_gcd=1.
gcd(0, 1)=1, math_gcd=1.
gcd(765, 273)=3, math_gcd=3.
gcd(24359573700, 35943207300)=2148300, math_gcd=2148300.
```

Euklidischer Algorithmus für den Größten Gemeinsamen Teiler

- Mit anderen Worten, `b` wird in `a` gespeichert und der Rest der Division des alten `a` durch das alte `b` wird in `b` gespeichert.
- Es ist klar, dass `b` in jeder Iteration kleiner wird.
- Da es niemals negativ werden kann, wird es irgendwann `0`.
- Dann hört die Schleife auf.

```python
"""Euclidian Algorithm for the Greatest Common Divisor as a function."""

from math import gcd as math_gcd  # Use math's gcd under name `math_gcd`.


def gcd(a: int, b: int) -> int:  # 2 `int` parameters and `int` result
    """
    Compute the greatest common divisor of two numbers `a` and `b`.

    :param a: the first number
    :param b: the second number
    :return: the greatest common divisor of `a` and `b`
    """
    while b != 0:  # Repeat in a loop until `b == 0`.
        a, b = b, a % b  # the same as `t = b`; `b = a % b`; `b = t`.
    return a  # If `b` becomes `0`, then the gcd is in `a`.


def print_gcd(a: int, b: int) -> None:  # `-> None` == returns nothing
    """
    Print the result of the gcd of `a` and `b`.

    :param a: the first number
    :param b: the second number
    """
    print(f"gcd({a}, {b})={gcd(a, b)}, math_gcd={math_gcd(a, b)}.")
    # Notice: no `return` statement. Because we return nothing.


print_gcd(1, 0)
print_gcd(0, 1)
print_gcd(765, 273)
print_gcd(24359573700, 35943207300)
```

↓ `python3 def_gcd.py` ↓

```
gcd(1, 0)=1, math_gcd=1.
gcd(0, 1)=1, math_gcd=1.
gcd(765, 273)=3, math_gcd=3.
gcd(24359573700, 35943207300)=2148300, math_gcd=2148300.
```

Euklidischer Algorithmus für den Größten Gemeinsamen Teiler

- Es ist klar, dass `b` in jeder Iteration kleiner wird.
- Da es niemals negativ werden kann, wird es irgendwann `0`.
- Dann hört die Schleife auf.
- Der größte gemeinsame Teiler geht in der Schleife auch nicht verloren.

```python
"""Euclidian Algorithm for the Greatest Common Divisor as a function."""

from math import gcd as math_gcd  # Use math's gcd under name `math_gcd
    ↪ `.


def gcd(a: int, b: int) -> int:  # 2 `int` parameters and `int` result
    """
    Compute the greatest common divisor of two numbers `a` and `b`.

    :param a: the first number
    :param b: the second number
    :return: the greatest common divisor of `a` and `b`
    """
    while b != 0:  # Repeat in a loop until `b == 0`.
        a, b = b, a % b  # the same as `t = b`; `b = a % b`; `b = t`.
    return a  # If `b` becomes `0`, then the gcd is in `a`.


def print_gcd(a: int, b: int) -> None:  # `-> None` == returns nothing
    """
    Print the result of the gcd of `a` and `b`.

    :param a: the first number
    :param b: the second number
    """
    print(f"gcd({a}, {b})={gcd(a, b)}, math_gcd={math_gcd(a, b)}.")
    # Notice: no `return` statement. Because we return nothing.


print_gcd(1, 0)
print_gcd(0, 1)
print_gcd(765, 273)
print_gcd(24359573700, 35943207300)
```

↓ `python3 def_gcd.py` ↓

```
gcd(1, 0)=1, math_gcd=1.
gcd(0, 1)=1, math_gcd=1.
gcd(765, 273)=3, math_gcd=3.
gcd(24359573700, 35943207300)=2148300, math_gcd=2148300.
```

Euklidischer Algorithmus für den Größten Gemeinsamen Teiler

- Da es niemals negativ werden kann, wird es irgendwann `0`.
- Dann hört die Schleife auf.
- Der größte gemeinsame Teiler geht in der Schleife auch nicht verloren.
- Er ist der Wert, den `a` am Ende annimmt.

```python
"""Euclidian Algorithm for the Greatest Common Divisor as a function."""

from math import gcd as math_gcd  # Use math's gcd under name `math_gcd`.


def gcd(a: int, b: int) -> int:  # 2 `int` parameters and `int` result
    """
    Compute the greatest common divisor of two numbers `a` and `b`.

    :param a: the first number
    :param b: the second number
    :return: the greatest common divisor of `a` and `b`
    """
    while b != 0:  # Repeat in a loop until `b == 0`.
        a, b = b, a % b  # the same as `t = b`; `b = a % b`; `b = t`.
    return a  # If `b` becomes `0`, then the gcd is in `a`.


def print_gcd(a: int, b: int) -> None:  # `-> None` == returns nothing
    """
    Print the result of the gcd of `a` and `b`.

    :param a: the first number
    :param b: the second number
    """
    print(f"gcd({a}, {b})={gcd(a, b)}, math_gcd={math_gcd(a, b)}.")
    # Notice: no `return` statement. Because we return nothing.


print_gcd(1, 0)
print_gcd(0, 1)
print_gcd(765, 273)
print_gcd(24359573700, 35943207300)
```

↓ `python3 def_gcd.py` ↓

```
gcd(1, 0)=1, math_gcd=1.
gcd(0, 1)=1, math_gcd=1.
gcd(765, 273)=3, math_gcd=3.
gcd(24359573700, 35943207300)=2148300, math_gcd=2148300.
```

Euklidischer Algorithmus für den Größten Gemeinsamen Teiler

- Dann hört die Schleife auf.
- Der größte gemeinsame Teiler geht in der Schleife auch nicht verloren.
- Er ist der Wert, den `a` am Ende annimmt.
- Und dieser Wert wird zurückgegeben.
- Mit `gcd` haben wir also eine Funktion mit zwei Parametern und einem Rückgabewert implementiert.

```python
"""Euclidian Algorithm for the Greatest Common Divisor as a function."""

from math import gcd as math_gcd  # Use math's gcd under name `math_gcd`.


def gcd(a: int, b: int) -> int:  # 2 `int` parameters and `int` result
    """
    Compute the greatest common divisor of two numbers `a` and `b`.

    :param a: the first number
    :param b: the second number
    :return: the greatest common divisor of `a` and `b`
    """
    while b != 0:  # Repeat in a loop until `b == 0`.
        a, b = b, a % b  # the same as `t = b`; `b = a % b`; `b = t`.
    return a  # If `b` becomes `0`, then the gcd is in `a`.


def print_gcd(a: int, b: int) -> None:  # `-> None` == returns nothing
    """
    Print the result of the gcd of `a` and `b`.

    :param a: the first number
    :param b: the second number
    """
    print(f"gcd({a}, {b})={gcd(a, b)}, math_gcd={math_gcd(a, b)}.")
    # Notice: no `return` statement. Because we return nothing.


print_gcd(1, 0)
print_gcd(0, 1)
print_gcd(765, 273)
print_gcd(24359573700, 35943207300)
```

↓ `python3 def_gcd.py` ↓

```
gcd(1, 0)=1, math_gcd=1.
gcd(0, 1)=1, math_gcd=1.
gcd(765, 273)=3, math_gcd=3.
gcd(24359573700, 35943207300)=2148300, math_gcd=2148300.
```

Euklidischer Algorithmus für den Größten Gemeinsamen Teiler

- Der größte gemeinsame Teiler geht in der Schleife auch nicht verloren.
- Er ist der Wert, den `a` am Ende annimmt.
- Und dieser Wert wird zurückgegeben.
- Mit `gcd` haben wir also eine Funktion mit zwei Parametern und einem Rückgabewert implementiert.
- Implementieren wir jetzt eine zweite Funktion, diesmal ohne Rückgabewert.

```python
"""Euclidian Algorithm for the Greatest Common Divisor as a function."""

from math import gcd as math_gcd  # Use math's gcd under name `math_gcd
    ↪ `.


def gcd(a: int, b: int) -> int:  # 2 `int` parameters and `int` result
    """
    Compute the greatest common divisor of two numbers `a` and `b`.

    :param a: the first number
    :param b: the second number
    :return: the greatest common divisor of `a` and `b`
    """
    while b != 0:  # Repeat in a loop until `b == 0`.
        a, b = b, a % b  # the same as `t = b`; `b = a % b`; `b = t`.
    return a  # If `b` becomes `0`, then the gcd is in `a`.


def print_gcd(a: int, b: int) -> None:  # `-> None` == returns nothing
    """
    Print the result of the gcd of `a` and `b`.

    :param a: the first number
    :param b: the second number
    """
    print(f"gcd({a}, {b})={gcd(a, b)}, math_gcd={math_gcd(a, b)}.")
    # Notice: no `return` statement. Because we return nothing.


print_gcd(1, 0)
print_gcd(0, 1)
print_gcd(765, 273)
print_gcd(24359573700, 35943207300)
```

↓ `python3 def_gcd.py` ↓

```
gcd(1, 0)=1, math_gcd=1.
gcd(0, 1)=1, math_gcd=1.
gcd(765, 273)=3, math_gcd=3.
gcd(24359573700, 35943207300)=2148300, math_gcd=2148300.
```

Euklidischer Algorithmus für den Größten Gemeinsamen Teiler

- Er ist der Wert, den `a` am Ende annimmt.
- Und dieser Wert wird zurückgegeben.
- Mit `gcd` haben wir also eine Funktion mit zwei Parametern und einem Rückgabewert implementiert.
- Implementieren wir jetzt eine zweite Funktion, diesmal ohne Rückgabewert.
- Unsere neue Funktion `print_gcd` akzeptiert ebenfalls zwei Parameter `a` und `b`, liefert aber diesmal nichts zurück.

```python
"""Euclidian Algorithm for the Greatest Common Divisor as a function."""

from math import gcd as math_gcd  # Use math's gcd under name `math_gcd
    ↪ `.


def gcd(a: int, b: int) -> int:  # 2 `int` parameters and `int` result
    """
    Compute the greatest common divisor of two numbers `a` and `b`.

    :param a: the first number
    :param b: the second number
    :return: the greatest common divisor of `a` and `b`
    """
    while b != 0:  # Repeat in a loop until `b == 0`.
        a, b = b, a % b  # the same as `t = b`; `b = a % b`; `b = t`.
    return a  # If `b` becomes `0`, then the gcd is in `a`.


def print_gcd(a: int, b: int) -> None:  # `-> None` == returns nothing
    """
    Print the result of the gcd of `a` and `b`.

    :param a: the first number
    :param b: the second number
    """
    print(f"gcd({a}, {b})={gcd(a, b)}, math_gcd={math_gcd(a, b)}.")
    # Notice: no `return` statement. Because we return nothing.


print_gcd(1, 0)
print_gcd(0, 1)
print_gcd(765, 273)
print_gcd(24359573700, 35943207300)
```

↓ `python3 def_gcd.py` ↓

```
gcd(1, 0)=1, math_gcd=1.
gcd(0, 1)=1, math_gcd=1.
gcd(765, 273)=3, math_gcd=3.
gcd(24359573700, 35943207300)=2148300, math_gcd=2148300.
```

Euklidischer Algorithmus für den Größten Gemeinsamen Teiler

- Und dieser Wert wird zurückgegeben.
- Mit `gcd` haben wir also eine Funktion mit zwei Parametern und einem Rückgabewert implementiert.
- Implementieren wir jetzt eine zweite Funktion, diesmal ohne Rückgabewert.
- Unsere neue Funktion `print_gcd` akzeptiert ebenfalls zwei Parameter `a` und `b`, liefert aber diesmal nichts zurück.
- Stattdessen druckt es den `gcd` schön mit `print` und einem f-String aus.

```python
"""Euclidian Algorithm for the Greatest Common Divisor as a function."""

from math import gcd as math_gcd  # Use math's gcd under name `math_gcd`.


def gcd(a: int, b: int) -> int:  # 2 `int` parameters and `int` result
    """
    Compute the greatest common divisor of two numbers `a` and `b`.

    :param a: the first number
    :param b: the second number
    :return: the greatest common divisor of `a` and `b`
    """
    while b != 0:  # Repeat in a loop until `b == 0`.
        a, b = b, a % b  # the same as `t = b`; `b = a % b`; `b = t`.
    return a  # If `b` becomes `0`, then the gcd is in `a`.


def print_gcd(a: int, b: int) -> None:  # `-> None` == returns nothing
    """
    Print the result of the gcd of `a` and `b`.

    :param a: the first number
    :param b: the second number
    """
    print(f"gcd({a}, {b})={gcd(a, b)}, math_gcd={math_gcd(a, b)}.")
    # Notice: no `return` statement. Because we return nothing.


print_gcd(1, 0)
print_gcd(0, 1)
print_gcd(765, 273)
print_gcd(24359573700, 35943207300)
```

↓ `python3 def_gcd.py` ↓

```
gcd(1, 0)=1, math_gcd=1.
gcd(0, 1)=1, math_gcd=1.
gcd(765, 273)=3, math_gcd=3.
gcd(24359573700, 35943207300)=2148300, math_gcd=2148300.
```

Euklidischer Algorithmus für den Größten Gemeinsamen Teiler

- Mit `gcd` haben wir also eine Funktion mit zwei Parametern und einem Rückgabewert implementiert.
- Implementieren wir jetzt eine zweite Funktion, diesmal ohne Rückgabewert.
- Unsere neue Funktion `print_gcd` akzeptiert ebenfalls zwei Parameter `a` und `b`, liefert aber diesmal nichts zurück.
- Stattdessen druckt es den `gcd` schön mit `print` und einem f-String aus.
- Beachten Sie, dass wir die Funktion wieder schön mit Type Hints und einem Docstring annotieren.

```python
"""Euclidian Algorithm for the Greatest Common Divisor as a function."""

from math import gcd as math_gcd  # Use math's gcd under name `math_gcd`.


def gcd(a: int, b: int) -> int:  # 2 `int` parameters and `int` result
    """
    Compute the greatest common divisor of two numbers `a` and `b`.

    :param a: the first number
    :param b: the second number
    :return: the greatest common divisor of `a` and `b`
    """
    while b != 0:  # Repeat in a loop until `b == 0`.
        a, b = b, a % b  # the same as `t = b`; `b = a % b`; `b = t`.
    return a  # If `b` becomes `0`, then the gcd is in `a`.


def print_gcd(a: int, b: int) -> None:  # `-> None` == returns nothing
    """
    Print the result of the gcd of `a` and `b`.

    :param a: the first number
    :param b: the second number
    """
    print(f"gcd({a}, {b})={gcd(a, b)}, math_gcd={math_gcd(a, b)}.")
    # Notice: no `return` statement. Because we return nothing.


print_gcd(1, 0)
print_gcd(0, 1)
print_gcd(765, 273)
print_gcd(24359573700, 35943207300)
```

↓ `python3 def_gcd.py` ↓

```
gcd(1, 0)=1, math_gcd=1.
gcd(0, 1)=1, math_gcd=1.
gcd(765, 273)=3, math_gcd=3.
gcd(24359573700, 35943207300)=2148300, math_gcd=2148300.
```

Euklidischer Algorithmus für den Größten Gemeinsamen Teiler

- Implementieren wir jetzt eine zweite Funktion, diesmal ohne Rückgabewert.
- Unsere neue Funktion `print_gcd` akzeptiert ebenfalls zwei Parameter `a` und `b`, liefert aber diesmal nichts zurück.
- Stattdessen druckt es den `gcd` schön mit `print` und einem f-String aus.
- Beachten Sie, dass wir die Funktion wieder schön mit Type Hints und einem Docstring annotieren.
- Das Modul `math` hat auch eine Funktion mit dem Namen `gcd`.

```python
"""Euclidian Algorithm for the Greatest Common Divisor as a function."""

from math import gcd as math_gcd  # Use math's gcd under name `math_gcd
    ↪ `.


def gcd(a: int, b: int) -> int:  # 2 `int` parameters and `int` result
    """
    Compute the greatest common divisor of two numbers `a` and `b`.

    :param a: the first number
    :param b: the second number
    :return: the greatest common divisor of `a` and `b`
    """
    while b != 0:  # Repeat in a loop until `b == 0`.
        a, b = b, a % b  # the same as `t = b`; `b = a % b`; `b = t`.
    return a  # If `b` becomes `0`, then the gcd is in `a`.


def print_gcd(a: int, b: int) -> None:  # `-> None` == returns nothing
    """
    Print the result of the gcd of `a` and `b`.

    :param a: the first number
    :param b: the second number
    """
    print(f"gcd({a}, {b})={gcd(a, b)}, math_gcd={math_gcd(a, b)}.")
    # Notice: no `return` statement. Because we return nothing.


print_gcd(1, 0)
print_gcd(0, 1)
print_gcd(765, 273)
print_gcd(24359573700, 35943207300)
```

↓ `python3 def_gcd.py` ↓

```
gcd(1, 0)=1, math_gcd=1.
gcd(0, 1)=1, math_gcd=1.
gcd(765, 273)=3, math_gcd=3.
gcd(24359573700, 35943207300)=2148300, math_gcd=2148300.
```

Euklidischer Algorithmus für den Größten Gemeinsamen Teiler

- Unsere neue Funktion `print_gcd` akzeptiert ebenfalls zwei Parameter `a` und `b`, liefert aber diesmal nichts zurück.
- Stattdessen druckt es den `gcd` schön mit `print` und einem f-String aus.
- Beachten Sie, dass wir die Funktion wieder schön mit Type Hints und einem Docstring annotieren.
- Das Modul `math` hat auch eine Funktion mit dem Namen `gcd`.
- Diese berechnet ebenfalls den größten gemeinsamen Teiler.

```python
"""Euclidian Algorithm for the Greatest Common Divisor as a function."""

from math import gcd as math_gcd  # Use math's gcd under name `math_gcd`.


def gcd(a: int, b: int) -> int:  # 2 `int` parameters and `int` result
    """
    Compute the greatest common divisor of two numbers `a` and `b`.

    :param a: the first number
    :param b: the second number
    :return: the greatest common divisor of `a` and `b`
    """
    while b != 0:  # Repeat in a loop until `b == 0`.
        a, b = b, a % b  # the same as `t = b`; `b = a % b`; `b = t`.
    return a  # If `b` becomes `0`, then the gcd is in `a`.


def print_gcd(a: int, b: int) -> None:  # `-> None` == returns nothing
    """
    Print the result of the gcd of `a` and `b`.

    :param a: the first number
    :param b: the second number
    """
    print(f"gcd({a}, {b})={gcd(a, b)}, math_gcd={math_gcd(a, b)}.")
    # Notice: no `return` statement. Because we return nothing.


print_gcd(1, 0)
print_gcd(0, 1)
print_gcd(765, 273)
print_gcd(24359573700, 35943207300)
```

↓ `python3 def_gcd.py` ↓

```
gcd(1, 0)=1, math_gcd=1.
gcd(0, 1)=1, math_gcd=1.
gcd(765, 273)=3, math_gcd=3.
gcd(24359573700, 35943207300)=2148300, math_gcd=2148300.
```

Euklidischer Algorithmus für den Größten Gemeinsamen Teiler

- Stattdessen druckt es den `gcd` schön mit `print` und einem f-String aus.
- Beachten Sie, dass wir die Funktion wieder schön mit Type Hints und einem Docstring annotieren.
- Das Modul `math` hat auch eine Funktion mit dem Namen `gcd`.
- Diese berechnet ebenfalls den größten gemeinsamen Teiler.
- Natürlich wollen wir das Ergebnis unserer Funktion mit ihr vergleichen.

```python
"""Euclidian Algorithm for the Greatest Common Divisor as a function."""

from math import gcd as math_gcd  # Use math's gcd under name `math_gcd`.


def gcd(a: int, b: int) -> int:  # 2 `int` parameters and `int` result
    """
    Compute the greatest common divisor of two numbers `a` and `b`.

    :param a: the first number
    :param b: the second number
    :return: the greatest common divisor of `a` and `b`
    """
    while b != 0:  # Repeat in a loop until `b == 0`.
        a, b = b, a % b  # the same as `t = b`; `b = a % b`; `b = t`.
    return a  # If `b` becomes `0`, then the gcd is in `a`.


def print_gcd(a: int, b: int) -> None:  # `-> None` == returns nothing
    """
    Print the result of the gcd of `a` and `b`.

    :param a: the first number
    :param b: the second number
    """
    print(f"gcd({a}, {b})={gcd(a, b)}, math_gcd={math_gcd(a, b)}.")
    # Notice: no `return` statement. Because we return nothing.


print_gcd(1, 0)
print_gcd(0, 1)
print_gcd(765, 273)
print_gcd(24359573700, 35943207300)
```

↓ `python3 def_gcd.py` ↓

```
gcd(1, 0)=1, math_gcd=1.
gcd(0, 1)=1, math_gcd=1.
gcd(765, 273)=3, math_gcd=3.
gcd(24359573700, 35943207300)=2148300, math_gcd=2148300.
```

Euklidischer Algorithmus für den Größten Gemeinsamen Teiler

- Beachten Sie, dass wir die Funktion wieder schön mit Type Hints und einem Docstring annotieren.
- Das Modul `math` hat auch eine Funktion mit dem Namen `gcd`.
- Diese berechnet ebenfalls den größten gemeinsamen Teiler.
- Natürlich wollen wir das Ergebnis unserer Funktion mit ihr vergleichen.
- Natürlich können wir nicht zwei Funktionen mit dem gleichen Namen (`gcd`) im selben Kontext haben.

```python
"""Euclidian Algorithm for the Greatest Common Divisor as a function."""

from math import gcd as math_gcd  # Use math's gcd under name `math_gcd`.


def gcd(a: int, b: int) -> int:  # 2 `int` parameters and `int` result
    """
    Compute the greatest common divisor of two numbers `a` and `b`.

    :param a: the first number
    :param b: the second number
    :return: the greatest common divisor of `a` and `b`
    """
    while b != 0:  # Repeat in a loop until `b == 0`.
        a, b = b, a % b  # the same as `t = b`; `b = a % b`; `b = t`.
    return a  # If `b` becomes `0`, then the gcd is in `a`.


def print_gcd(a: int, b: int) -> None:  # `-> None` == returns nothing
    """
    Print the result of the gcd of `a` and `b`.

    :param a: the first number
    :param b: the second number
    """
    print(f"gcd({a}, {b})={gcd(a, b)}, math_gcd={math_gcd(a, b)}.")
    # Notice: no `return` statement. Because we return nothing.


print_gcd(1, 0)
print_gcd(0, 1)
print_gcd(765, 273)
print_gcd(24359573700, 35943207300)
```

↓ `python3 def_gcd.py` ↓

```
gcd(1, 0)=1, math_gcd=1.
gcd(0, 1)=1, math_gcd=1.
gcd(765, 273)=3, math_gcd=3.
gcd(24359573700, 35943207300)=2148300, math_gcd=2148300.
```

Euklidischer Algorithmus für den Größten Gemeinsamen Teiler

- Das Modul `math` hat auch eine Funktion mit dem Namen `gcd`.
- Diese berechnet ebenfalls den größten gemeinsamen Teiler.
- Natürlich wollen wir das Ergebnis unserer Funktion mit ihr vergleichen.
- Natürlich können wir nicht zwei Funktionen mit dem gleichen Namen (`gcd`) im selben Kontext haben.
- Deshalb importieren wir die Funktion aus dem Modul `math` *unter einem anderen Namen*.

```python
"""Euclidian Algorithm for the Greatest Common Divisor as a function."""

from math import gcd as math_gcd  # Use math's gcd under name `math_gcd`.


def gcd(a: int, b: int) -> int:  # 2 `int` parameters and `int` result
    """
    Compute the greatest common divisor of two numbers `a` and `b`.

    :param a: the first number
    :param b: the second number
    :return: the greatest common divisor of `a` and `b`
    """
    while b != 0:  # Repeat in a loop until `b == 0`.
        a, b = b, a % b  # the same as `t = b`; `b = a % b`; `b = t`.
    return a  # If `b` becomes `0`, then the gcd is in `a`.


def print_gcd(a: int, b: int) -> None:  # `-> None` == returns nothing
    """
    Print the result of the gcd of `a` and `b`.

    :param a: the first number
    :param b: the second number
    """
    print(f"gcd({a}, {b})={gcd(a, b)}, math_gcd={math_gcd(a, b)}.")
    # Notice: no `return` statement. Because we return nothing.


print_gcd(1, 0)
print_gcd(0, 1)
print_gcd(765, 273)
print_gcd(24359573700, 35943207300)
```

↓ `python3 def_gcd.py` ↓

```
gcd(1, 0)=1, math_gcd=1.
gcd(0, 1)=1, math_gcd=1.
gcd(765, 273)=3, math_gcd=3.
gcd(24359573700, 35943207300)=2148300, math_gcd=2148300.
```

Euklidischer Algorithmus für den Größten Gemeinsamen Teiler

- Diese berechnet ebenfalls den größten gemeinsamen Teiler.
- Natürlich wollen wir das Ergebnis unserer Funktion mit ihr vergleichen.
- Natürlich können wir nicht zwei Funktionen mit dem gleichen Namen (`gcd`) im selben Kontext haben.
- Deshalb importieren wir die Funktion aus dem Modul `math` *unter einem anderen Namen*:
- `from math import gcd as math_gcd` importiert die Funktion `gcd` aus dem Module `math` und stellt sie unter dem Namen `math_gcd` zur Verfügung.

```python
"""Euclidian Algorithm for the Greatest Common Divisor as a function."""

from math import gcd as math_gcd  # Use math's gcd under name `math_gcd`.


def gcd(a: int, b: int) -> int:  # 2 `int` parameters and `int` result
    """
    Compute the greatest common divisor of two numbers `a` and `b`.

    :param a: the first number
    :param b: the second number
    :return: the greatest common divisor of `a` and `b`
    """
    while b != 0:  # Repeat in a loop until `b == 0`.
        a, b = b, a % b  # the same as `t = b`; `b = a % b`; `b = t`.
    return a  # If `b` becomes `0`, then the gcd is in `a`.


def print_gcd(a: int, b: int) -> None:  # `-> None` == returns nothing
    """
    Print the result of the gcd of `a` and `b`.

    :param a: the first number
    :param b: the second number
    """
    print(f"gcd({a}, {b})={gcd(a, b)}, math_gcd={math_gcd(a, b)}.")
    # Notice: no `return` statement. Because we return nothing.


print_gcd(1, 0)
print_gcd(0, 1)
print_gcd(765, 273)
print_gcd(24359573700, 35943207300)
```

↓ `python3 def_gcd.py` ↓

```
gcd(1, 0)=1, math_gcd=1.
gcd(0, 1)=1, math_gcd=1.
gcd(765, 273)=3, math_gcd=3.
gcd(24359573700, 35943207300)=2148300, math_gcd=2148300.
```

Euklidischer Algorithmus für den Größten Gemeinsamen Teiler

- Natürlich wollen wir das Ergebnis unserer Funktion mit ihr vergleichen.
- Natürlich können wir nicht zwei Funktionen mit dem gleichen Namen (`gcd`) im selben Kontext haben.
- Deshalb importieren wir die Funktion aus dem Modul `math` *unter einem anderen Namen*:
- `from math import gcd as math_gcd` importiert die Funktion `gcd` aus dem Module `math` und stellt sie unter dem Namen `math_gcd` zur Verfügung.
- Und wir benutzen sie im f-String in unserer Funktion `print_gcd` unter diesem Namen.

```python
"""Euclidian Algorithm for the Greatest Common Divisor as a function."""

from math import gcd as math_gcd  # Use math's gcd under name `math_gcd`.


def gcd(a: int, b: int) -> int:  # 2 `int` parameters and `int` result
    """
    Compute the greatest common divisor of two numbers `a` and `b`.

    :param a: the first number
    :param b: the second number
    :return: the greatest common divisor of `a` and `b`
    """
    while b != 0:  # Repeat in a loop until `b == 0`.
        a, b = b, a % b  # the same as `t = b`; `b = a % b`; `b = t`.
    return a  # If `b` becomes `0`, then the gcd is in `a`.


def print_gcd(a: int, b: int) -> None:  # `-> None` == returns nothing
    """
    Print the result of the gcd of `a` and `b`.

    :param a: the first number
    :param b: the second number
    """
    print(f"gcd({a}, {b})={gcd(a, b)}, math_gcd={math_gcd(a, b)}.")
    # Notice: no `return` statement. Because we return nothing.


print_gcd(1, 0)
print_gcd(0, 1)
print_gcd(765, 273)
print_gcd(24359573700, 35943207300)
```

↓ `python3 def_gcd.py` ↓

```
gcd(1, 0)=1, math_gcd=1.
gcd(0, 1)=1, math_gcd=1.
gcd(765, 273)=3, math_gcd=3.
gcd(24359573700, 35943207300)=2148300, math_gcd=2148300.
```

Euklidischer Algorithmus für den Größten Gemeinsamen Teiler

- Deshalb importieren wir die Funktion aus dem Modul `math` *unter einem anderen Namen*:
- `from math import gcd as math_gcd` importiert die Funktion `gcd` aus dem Module `math` und stellt sie unter dem Namen `math_gcd` zur Verfügung.
- Und wir benutzen sie im f-String in unserer Funktion `print_gcd` unter diesem Namen.
- Wir bestätigen, dass `gcd` und `math_gcd` die gleichen Ergebnisse liefern für vier Testfälle am Ende unseres Programms.

```python
"""Euclidian Algorithm for the Greatest Common Divisor as a function."""

from math import gcd as math_gcd  # Use math's gcd under name `math_gcd`.


def gcd(a: int, b: int) -> int:  # 2 `int` parameters and `int` result
    """
    Compute the greatest common divisor of two numbers `a` and `b`.

    :param a: the first number
    :param b: the second number
    :return: the greatest common divisor of `a` and `b`
    """
    while b != 0:  # Repeat in a loop until `b == 0`.
        a, b = b, a % b  # the same as `t = b`; `b = a % b`; `b = t`.
    return a  # If `b` becomes `0`, then the gcd is in `a`.


def print_gcd(a: int, b: int) -> None:  # `-> None` == returns nothing
    """
    Print the result of the gcd of `a` and `b`.

    :param a: the first number
    :param b: the second number
    """
    print(f"gcd({a}, {b})={gcd(a, b)}, math_gcd={math_gcd(a, b)}.")
    # Notice: no `return` statement. Because we return nothing.


print_gcd(1, 0)
print_gcd(0, 1)
print_gcd(765, 273)
print_gcd(24359573700, 35943207300)
```

↓ `python3 def_gcd.py` ↓

```
gcd(1, 0)=1, math_gcd=1.
gcd(0, 1)=1, math_gcd=1.
gcd(765, 273)=3, math_gcd=3.
gcd(24359573700, 35943207300)=2148300, math_gcd=2148300.
```

Euklidischer Algorithmus für den Größten Gemeinsamen Teiler

- Und wir benutzen sie im f-String in unserer Funktion `print_gcd` unter diesem Namen.
- Wir bestätigen, dass `gcd` und `math_gcd` die gleichen Ergebnisse liefern für vier Testfälle am Ende unseres Programms.
- Nun da wir fertig sind, lassen Sie mich noch erwähnen, dass es eine besonders effiziente binäre Variante des Euklidischen Algorithmus gibt, die scheller als unsere Implementierung ist und bereits in China im ersten Jahrhundert Common Era (CE) entwickelt wurde[1] und zwar steht die im *Jiu Zhang Suanshu* (九章算术)[5,6,15,23,28].

```python
"""Euclidian Algorithm for the Greatest Common Divisor as a function."""

from math import gcd as math_gcd  # Use math's gcd under name `math_gcd
    ↪ `.


def gcd(a: int, b: int) -> int:  # 2 `int` parameters and `int` result
    """
    Compute the greatest common divisor of two numbers `a` and `b`.

    :param a: the first number
    :param b: the second number
    :return: the greatest common divisor of `a` and `b`
    """
    while b != 0:  # Repeat in a loop until `b == 0`.
        a, b = b, a % b  # the same as `t = b`; `b = a % b`; `b = t`.
    return a  # If `b` becomes `0`, then the gcd is in `a`.


def print_gcd(a: int, b: int) -> None:  # `-> None` == returns nothing
    """
    Print the result of the gcd of `a` and `b`.

    :param a: the first number
    :param b: the second number
    """
    print(f"gcd({a}, {b})={gcd(a, b)}, math_gcd={math_gcd(a, b)}.")
    # Notice: no `return` statement. Because we return nothing.


print_gcd(1, 0)
print_gcd(0, 1)
print_gcd(765, 273)
print_gcd(24359573700, 35943207300)
```

↓ `python3 def_gcd.py` ↓

```
gcd(1, 0)=1, math_gcd=1.
gcd(0, 1)=1, math_gcd=1.
gcd(765, 273)=3, math_gcd=3.
gcd(24359573700, 35943207300)=2148300, math_gcd=2148300.
```

Zusammenfassung

Zusammenfassung

- Wir sind also nun in der Lage, unsere eigenen Funktionen zu implementieren.

Zusammenfassung

- Wir sind also nun in der Lage, unsere eigenen Funktionen zu implementieren.
- Wir können diese ordentlich mit Type-Hints und Docstrings annotieren.

Zusammenfassung

- Wir sind also nun in der Lage, unsere eigenen Funktionen zu implementieren.
- Wir können diese ordentlich mit Type-Hints und Docstrings annotieren.
- Wir können damit wiederverwendbare Stücke von Kode definieren, die von verschiedenen anderen Stellen im Kode heraus aufgerufen werden können.

Zusammenfassung

- Wir sind also nun in der Lage, unsere eigenen Funktionen zu implementieren.
- Wir können diese ordentlich mit Type-Hints und Docstrings annotieren.
- Wir können damit wiederverwendbare Stücke von Kode definieren, die von verschiedenen anderen Stellen im Kode heraus aufgerufen werden können.
- Wir können sie als Einheiten von Programmkode auch mit anderen Entwicklern teilen, die dann unsere Dokumentation lesen und verstehen können.

谢谢您们！

Thank you!

Vielen Dank!

References I

[1] Richard P. Brent. *Further Analysis of the Binary Euclidean Algorithm*. arXiv.org: Computing Research Repository (CoRR) abs/1303.2772. Ithaca, NY, USA: Cornell Universiy Library, Nov. 1999–12. März 2013. doi:10.48550/arXiv.1303.2772. URL: https://arxiv.org/abs/1303.2772 (besucht am 2024-09-28). arXiv:1303.2772v1 [cs.DS] 12 Mar 2013. Report number PRG TR-7-99 of Oxford, Oxfordshire, England, UK: Oxford University Computing Laboratory, 11 1999, see https://maths-people.anu.edu.au/~brent/pd/rpb183tr.pdf (siehe S. 93–108, 119–156).

[2] Florian Bruhin. *Python f-Strings*. Winterthur, Switzerland: Bruhin Software, 31. Mai 2023. URL: https://fstring.help (besucht am 2024-07-25) (siehe S. 167).

[3] Antonio Cavacini. "Is the CE/BCE notation becoming a standard in scholarly literature?" *Scientometrics* 102(2):1661–1668, Juli 2015. London, England, UK: Springer Nature Limited. ISSN: 0138-9130. doi:10.1007/s11192-014-1352-1 (siehe S. 167).

[4] Noureddine Chabini und Rachid Beguenane. "FPGA-Based Designs of the Factorial Function". In: *IEEE Canadian Conference on Electrical and Computer Engineering (CCECE'2022)*. 18.–20. Sep. 2022, Halifax, NS, Canada. Piscataway, NJ, USA: Institute of Electrical and Electronics Engineers (IEEE), 2022, S. 16–20. ISBN: 978-1-6654-8432-9. doi:10.1109/CCECE49351.2022.9918302 (siehe S. 66–72, 168).

[5] Christopher Cullen. "Learning from Liu Hui? A Different Way to Do Mathematics". *Notices of the American Mathematical Society* 49(7):783–790, Aug. 2002. Providence, RI, USA: American Mathematical Society (AMS). ISSN: 1088-9477. URL: https://www.ams.org/notices/200207/comm-cullen.pdf (besucht am 2024-08-09) (siehe S. 119–156).

[6] Joseph W. Dauben. "Archimedes and Liu Hui on Circles and Spheres". *Ontology Studies (Cuadernos de Ontología)* 10:21–38, 2010. Leioa, Bizkaia, Spain: Universidad del País Vasco / Euskal Herriko Unibertsitatea. ISSN: 1576-2270. URL: https://ddd.uab.cat/pub/ontstu/15762270n10/15762270n10p21.pdf (besucht am 2024-08-10) (siehe S. 119–156).

[7] Jacques Dutka. "The Early History of the Factorial Function". *Archive for History of Exact Sciences* 43(3):225–249, Sep. 1991. Berlin/Heidelberg, Germany: Springer-Verlag GmbH Germany. ISSN: 0003-9519. doi:10.1007/BF00389433. Communicated by Umberto Bottazzini (siehe S. 66–72, 168).

References II

[8] Euclid of Alexandria ($E\acute{\upsilon}\kappa\lambda\epsilon\acute{\iota}\delta\eta\varsigma$). *Euclid's Elements of Geometry* ($\Sigma\tau o\iota\chi\epsilon\tilde{\iota}\alpha$)*. The Greek Text of J.L. Heiberg (1883-1885) from Euclidis Elementa, Edidit et Latine Interpretatus est I.L. Heiberg in Aedibus B. G. Teubneri, 1883-1885. Edited, and provided with a modern English translation, by Richard Fitzpatrick*. Bd. 7. *Elementary Number Theory*. Hrsg. von Richard Fitzpatrick. Übers. von Johan Ludvig Heiberg. revised and corrected. Austin, TX, USA: The University of Texas at Austin, 2008. ISBN: 978-0-615-17984-1. URL: https://farside.ph.utexas.edu/Books/Euclid/Elements.pdf (besucht am 2024-09-30) (siehe S. 93–108).

[9] "Formatted String Literals". In: *Python 3 Documentation. The Python Tutorial*. Beaverton, OR, USA: Python Software Foundation (PSF), 2001–2025. Kap. 7.1.1. URL: https://docs.python.org/3/tutorial/inputoutput.html#formatted-string-literals (besucht am 2024-07-25) (siehe S. 167).

[10] Toru Fujita, Koji Nakano und Yasuaki Ito. "Bulk Execution of Euclidean Algorithms on the CUDA-Enabled GPU". *International Journal of Networking and Computing (IJNC)* 6(1):42–63, Jan. 2016. Higashi-Hiroshima, Japan: Department of Information Engineering, Hiroshima University. ISSN: 2185-2839. URL: http://www.ijnc.org (besucht am 2024-09-28) (siehe S. 93–108).

[11] Bhavesh Gawade. "Mastering F-Strings in Python: Efficient String Handling in Python Using Smart F-Strings". In: *C O D E B*. Mumbai, Maharashtra, India: Code B Solutions Pvt Ltd, 25. Apr.–3. Juni 2025. URL: https://code-b.dev/blog/f-strings-in-python (besucht am 2025-08-04) (siehe S. 167).

[12] David Goodger und Guido van Rossum. *Docstring Conventions*. Python Enhancement Proposal (PEP) 257. Beaverton, OR, USA: Python Software Foundation (PSF), 29. Mai–13. Juni 2001. URL: https://peps.python.org/pep-0257 (besucht am 2024-07-27) (siehe S. 167).

[13] Olaf Górski. "Why f-strings are awesome: Performance of different string concatenation methods in Python". In: *DEV Community*. Sacramento, CA, USA: DEV Community Inc., 8. Nov. 2022. URL: https://dev.to/grski/performance-of-different-string-concatenation-methods-in-python-why-f-strings-are-awesome-2e97 (besucht am 2025-08-04) (siehe S. 167).

[14] John Hunt. *A Beginners Guide to Python 3 Programming*. 2. Aufl. Undergraduate Topics in Computer Science (UTICS). Cham, Switzerland: Springer, 2023. ISBN: 978-3-031-35121-1. doi:10.1007/978-3-031-35122-8 (siehe S. 168).

[15] Shen Kangshen, John Newsome Crossley und Anthony W.-C. Lun. *The Nine Chapters on the Mathematical Art: Companion and Commentary*. Oxford, Oxfordshire, England, UK: Oxford University Press, 7. Okt. 1999. ISBN: 978-0-19-853936-0. doi:10.1093/oso/9780198539360.001.0001 (siehe S. 119–156).

References III

[16] Łukasz Langa. *Literature Overview for Type Hints*. Python Enhancement Proposal (PEP) 482. Beaverton, OR, USA: Python Software Foundation (PSF), 8. Jan. 2015. URL: https://peps.python.org/pep-0482 (besucht am 2024-10-09) (siehe S. 168).

[17] Kent D. Lee und Steve Hubbard. *Data Structures and Algorithms with Python*. Undergraduate Topics in Computer Science (UTICS). Cham, Switzerland: Springer, 2015. ISBN: 978-3-319-13071-2. doi:10.1007/978-3-319-13072-9 (siehe S. 168).

[18] Jukka Lehtosalo, Ivan Levkivskyi, Jared Hance, Ethan Smith, Guido van Rossum, Jelle „JelleZijlstra“ Zijlstra, Michael J. Sullivan, Shantanu Jain, Xuanda Yang, Jingchen Ye, Nikita Sobolev und Mypy Contributors. *Mypy – Static Typing for Python*. San Francisco, CA, USA: GitHub Inc, 2024. URL: https://github.com/python/mypy (besucht am 2024-08-17) (siehe S. 167).

[19] Peter Luschny. *A New Kind of Factorial Function*. Highland Park, NJ, USA: The OEIS Foundation Inc., 4. Okt. 2015. URL: https://oeis.org/A000142/a000142.pdf (besucht am 2024-09-29) (siehe S. 66–92, 168).

[20] Mark Lutz. *Learning Python*. 6. Aufl. Sebastopol, CA, USA: O'Reilly Media, Inc., März 2025. ISBN: 978-1-0981-7130-8 (siehe S. 168).

[21] Aaron Maxwell. *What are f-strings in Python and how can I use them?* Oakville, ON, Canada: Infinite Skills Inc, Juni 2017. ISBN: 978-1-4919-9486-3 (siehe S. 167).

[22] "More Control Flow Tools". In: *Python 3 Documentation. The Python Tutorial*. Beaverton, OR, USA: Python Software Foundation (PSF), 2001–2025. Kap. 4. URL: https://docs.python.org/3/tutorial/controlflow.html (besucht am 2025-09-03) (siehe S. 5–10).

[23] John J. O'Connor und Edmund F. Robertson. *Liu Hui*. St Andrews, Scotland, UK: University of St Andrews, School of Mathematics and Statistics, Dez. 2003. URL: https://mathshistory.st-andrews.ac.uk/Biographies/Liu_Hui (besucht am 2024-08-10) (siehe S. 119–156).

[24] Yasset Pérez-Riverol, Laurent Gatto, Rui Wang, Timo Sachsenberg, Julian Uszkoreit, Felipe da Veiga Leprevost, Christian Fufezan, Tobias Ternent, Stephen J. Eglen, Daniel S. Katz, Tom J. Pollard, Alexander Konovalov, Robert M. Flight, Kai Blin und Juan Antonio Vizcaíno. "Ten Simple Rules for Taking Advantage of Git and GitHub". *PLOS Computational Biology* 12(7), 14. Juli 2016. San Francisco, CA, USA: Public Library of Science (PLOS). ISSN: 1553-7358. doi:10.1371/JOURNAL.PCBI.1004947 (siehe S. 167).

[25] Syamal K. Sen und Ravi P. Agarwal. "Existence of year zero in astronomical counting is advantageous and preserves compatibility with significance of AD, BC, CE, and BCE". In: *Zero – A Landmark Discovery, the Dreadful Void, and the Ultimate Mind*. Amsterdam, The Netherlands: Elsevier B.V., 2016. Kap. 5.5, S. 94–95. ISBN: 978-0-08-100774-7. doi:10.1016/C2015-0-02299-7 (siehe S. 167).

References IV

[26] Anna Skoulikari. *Learning Git*. Sebastopol, CA, USA: O'Reilly Media, Inc., Mai 2023. ISBN: 978-1-0981-3391-7 (siehe S. 167).

[27] Eric V. „ericvsmith" Smith. *Literal String Interpolation*. Python Enhancement Proposal (PEP) 498. Beaverton, OR, USA: Python Software Foundation (PSF), 6. Nov. 2016–9. Sep. 2023. URL: https://peps.python.org/pep-0498 (besucht am 2024-07-25) (siehe S. 167).

[28] Philip D. Straffin Jr. "Liu Hui and the First Golden Age of Chinese Mathematics". *Mathematics Magazine* 71(3):163–181, Juni 1998. London, England, UK: Taylor and Francis Ltd. ISSN: 0025-570X. doi:10.2307/2691200. URL: https://www.researchgate.net/publication/237334342 (besucht am 2024-08-10) (siehe S. 119–156).

[29] *Python 3 Documentation. The Python Tutorial*. Beaverton, OR, USA: Python Software Foundation (PSF), 2001–2025. URL: https://docs.python.org/3/tutorial (besucht am 2025-04-26).

[30] Mariot Tsitoara. *Beginning Git and GitHub: Version Control, Project Management and Teamwork for the New Developer*. New York, NY, USA: Apress Media, LLC, März 2024. ISBN: 979-8-8688-0215-7 (siehe S. 167, 168).

[31] Guido van Rossum und Łukasz Langa. *Type Hints*. Python Enhancement Proposal (PEP) 484. Beaverton, OR, USA: Python Software Foundation (PSF), 29. Sep. 2014. URL: https://peps.python.org/pep-0484 (besucht am 2024-08-22) (siehe S. 168).

[32] Guido van Rossum, Barry Warsaw und Alyssa Coghlan. *Style Guide for Python Code*. Python Enhancement Proposal (PEP) 8. Beaverton, OR, USA: Python Software Foundation (PSF), 5. Juli 2001. URL: https://peps.python.org/pep-0008 (besucht am 2024-07-27) (siehe S. 14, 63, 167).

[33] Thomas Weise (汤卫思). *Programming with Python*. Hefei, Anhui, China (中国安徽省合肥市): Hefei University (合肥大学), School of Artificial Intelligence and Big Data (人工智能与大数据学院), Institute of Applied Optimization (应用优化研究所, IAO), 2024–2025. URL: https://thomasweise.github.io/programmingWithPython (besucht am 2025-01-05) (siehe S. 167, 168).

[34] Collin Winter und Tony Lownds. *Function Annotations*. Python Enhancement Proposal (PEP). Beaverton, OR, USA: Python Software Foundation (PSF), 2. Dez. 2006. URL: https://peps.python.org/pep-3107 (besucht am 2024-12-12) (siehe S. 24–30, 39).

[35] Nicola Abdo Ziadeh, Michael B. Rowton, A. Geoffrey Woodhead, Wolfgang Helck, Jean L.A. Filliozat, Hiroyuki Momo, Eric Thompson, E.J. Wiesenberg und Shih-ch'ang Wu. "Chronology – Christian History, Dates, Events". In: *Encyclopaedia Britannica*. Hrsg. von The Editors of Encyclopaedia Britannica. Chicago, IL, USA: Encyclopædia Britannica, Inc., 26. Juli 1999–20. März 2024. URL: https://www.britannica.com/topic/chronology/Christian (besucht am 2025-08-27) (siehe S. 167).

Glossary (in English) I

BCE The time notation *before Common Era* is a non-religious but chronological equivalent alternative to the traditional *Before Christ (BC)* notation, which refers to the years *before* the birth of Jesus Christ[3]. The years BCE are counted down, i.e., the larger the year, the farther in the past. The year 1 BCE comes directly before the year 1 CE[25,35].

CE The time notation *Common Era* is a non-religious but chronological equivalent alternative to the traditional *Anno Domini (AD)* notation, which refers to the years *after* the birth of Jesus Christ[3]. The years CE are counted upwards, i.e., the smaller they are, the farther they are in the past. The year 1 CE comes directly after the year 1 BCE[25,35].

docstring Docstrings are special string constants in Python that contain documentation for modules or functions[12]. They must be delimited by `"""..."""`[12,32].

f-string let you include the results of expressions in strings[2,9,11,13,21,27]. They can contain expressions (in curly braces) like `f"a{6-1}b"` that are then transformed to text via (string) interpolation, which turns the string to `"a5b"`. F-strings are delimited by `f"..."`.

Git is a distributed Version Control Systems (VCS) which allows multiple users to work on the same code while preserving the history of the code changes[26,30]. Learn more at https://git-scm.com.

GitHub is a website where software projects can be hosted and managed via the Git VCS[24,30]. Learn more at https://github.com.

modulo division is, in Python, done by the operator `%` that computes the remainder of a division. `15 % 6` gives us `3`.

Mypy is a static type checking tool for Python[18] that makes use of type hints. Learn more at https://github.com/python/mypy and in[33].

Glossary (in English) II

Python The Python programming language[14,17,20,33], i.e., what you will learn about in our book[33]. Learn more at https://python.org.

(string) interpolation In Python, string interpolation is the process where all the expressions in an f-string are evaluated and the final string is constructed. An example for string interpolation is turning `f"Rounded {1.234:.2f}"` to `"Rounded 1.23"`.

type hint are annotations that help programmers and static code analysis tools such as Mypy to better understand what type a variable or function parameter is supposed to be[16,31]. Python is a dynamically typed programming language where you do not need to specify the type of, e.g., a variable. This creates problems for code analysis, both automated as well as manual: For example, it may not always be clear whether a variable or function parameter should be an integer or floating point number. The annotations allow us to explicitly state which type is expected. They are *ignored* during the program execution. They are a basically a piece of documentation.

VCS A *Version Control System* is a software which allows you to manage and preserve the historical development of your program code[30]. A distributed VCS allows multiple users to work on the same code and upload their changes to the server, which then preserves the change history. The most popular distributed VCS is Git.

$i!$ The factorial $a!$ of a natural number $a \in \mathbb{N}_1$ is the product of all positive natural numbers less than or equal to a, i.e., $a! = 1 * 2 * 3 * 4 * \cdots * (a-1) * a$[4,7,19].

$\mathbb{N}_1$ the set of the natural numbers *excluding* 0, i.e., 1, 2, 3, 4, and so on. It holds that $\mathbb{N}_1 \subset \mathbb{Z}$.

$\mathbb{R}$ the set of the real numbers.

$\mathbb{Z}$ the set of the integers numbers including positive and negative numbers and 0, i.e., …, -3, -2, -1, 0, 1, 2, 3, …, and so on. It holds that $\mathbb{Z} \subset \mathbb{R}$.