



合肥大學
HEFEI UNIVERSITY



Programming with Python

27. Funktionen in Modulen

Thomas Weise (汤卫思)
tweise@hfuu.edu.cn

Institute of Applied Optimization (IAO)
School of Artificial Intelligence and Big Data
Hefei University
Hefei, Anhui, China

应用优化研究所
人工智能与大数据学院
合肥大学
中国安徽省合肥市

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-Code 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



1. Einleitung
2. Beispiel
3. Zusammenfassung



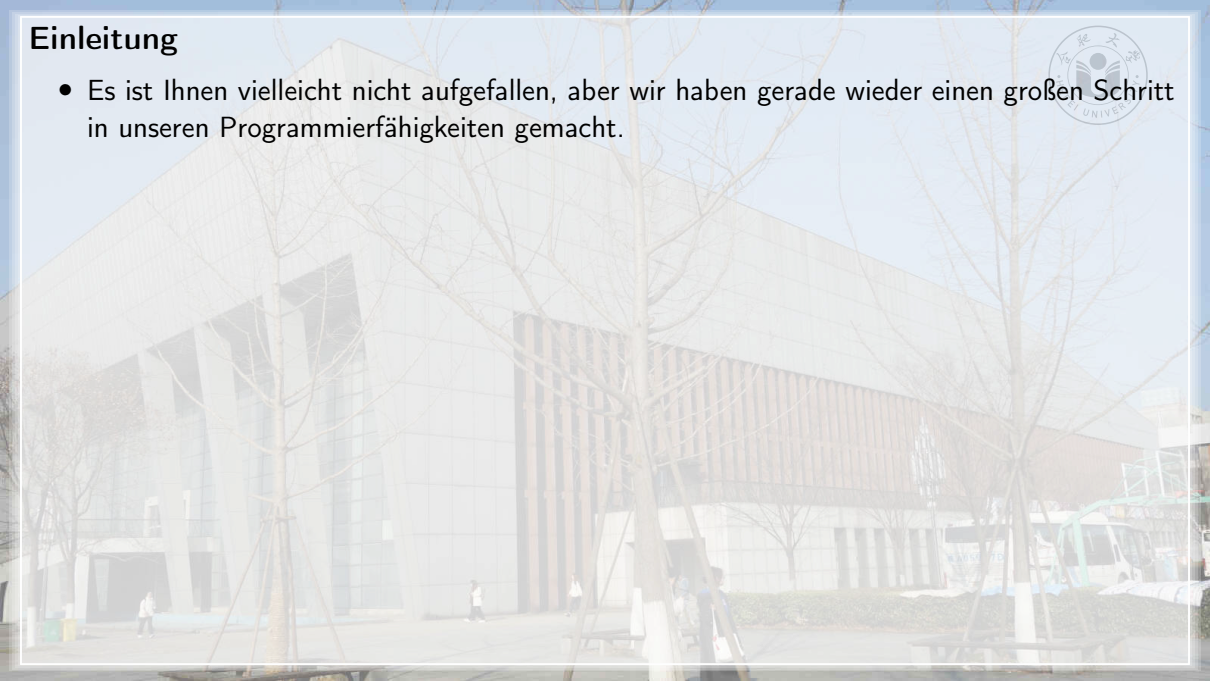


Einleitung



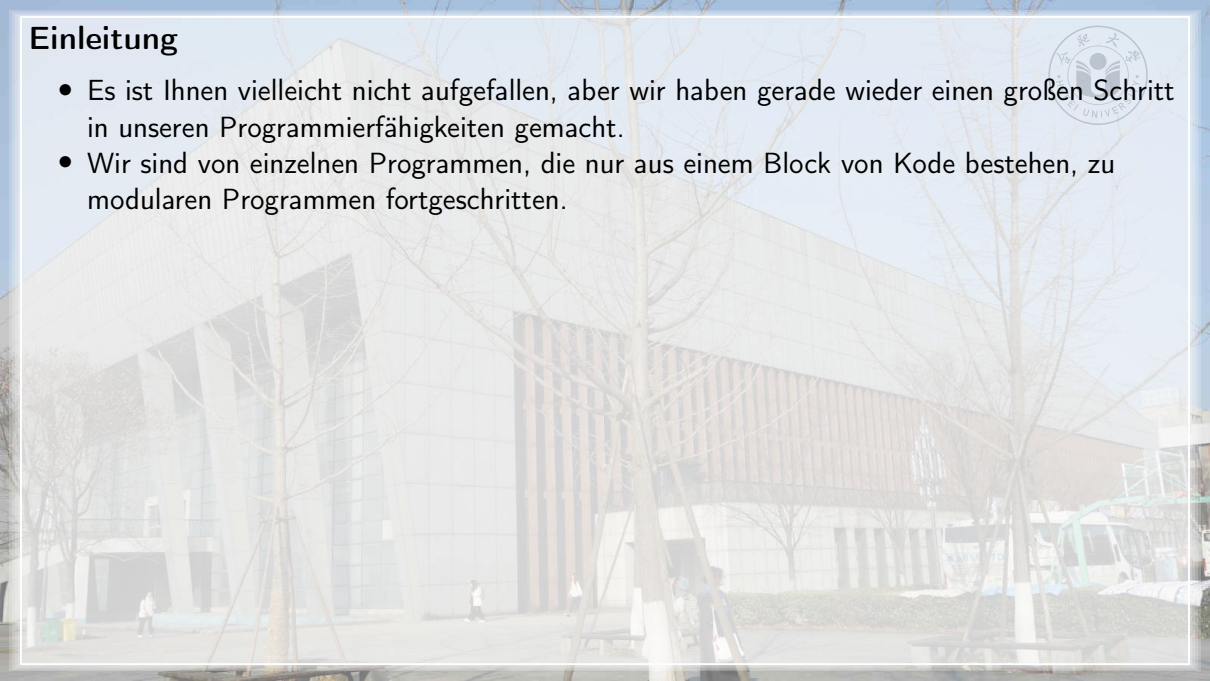
Einleitung

- Es ist Ihnen vielleicht nicht aufgefallen, aber wir haben gerade wieder einen großen Schritt in unseren Programmierfähigkeiten gemacht.



Einleitung

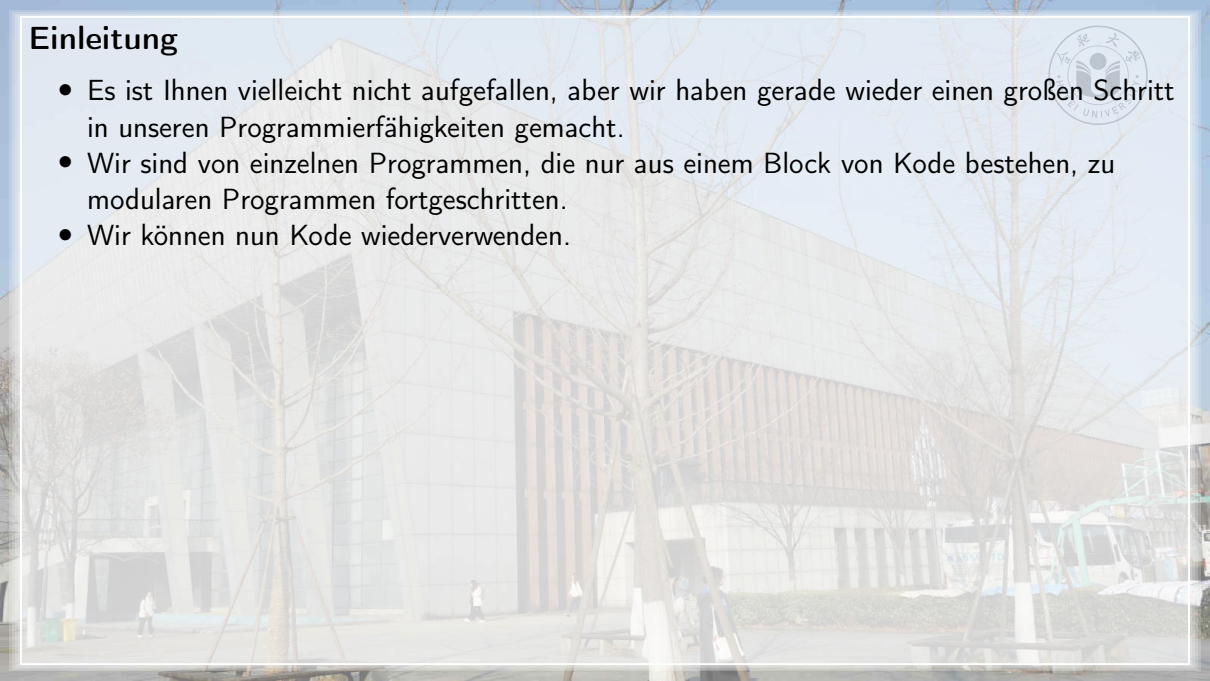
- Es ist Ihnen vielleicht nicht aufgefallen, aber wir haben gerade wieder einen großen Schritt in unseren Programmierfähigkeiten gemacht.
- Wir sind von einzelnen Programmen, die nur aus einem Block von Code bestehen, zu modularen Programmen fortgeschritten.



Einleitung



- Es ist Ihnen vielleicht nicht aufgefallen, aber wir haben gerade wieder einen großen Schritt in unseren Programmierfähigkeiten gemacht.
- Wir sind von einzelnen Programmen, die nur aus einem Block von Code bestehen, zu modularen Programmen fortgeschritten.
- Wir können nun Code wiederverwenden.



Einleitung



- Es ist Ihnen vielleicht nicht aufgefallen, aber wir haben gerade wieder einen großen Schritt in unseren Programmierfähigkeiten gemacht.
- Wir sind von einzelnen Programmen, die nur aus einem Block von Code bestehen, zu modularen Programmen fortgeschritten.
- Wir können nun Code wiederverwenden.
- Als wir unsere Reise begonnen haben, haben wir unsere Befehle einen nach dem anderen in den Python-Interpreter eingetippt und ausgeführt.

Einleitung



- Es ist Ihnen vielleicht nicht aufgefallen, aber wir haben gerade wieder einen großen Schritt in unseren Programmierfähigkeiten gemacht.
- Wir sind von einzelnen Programmen, die nur aus einem Block von Code bestehen, zu modularen Programmen fortgeschritten.
- Wir können nun Code wiederverwenden.
- Als wir unsere Reise begonnen haben, haben wir unsere Befehle einen nach dem anderen in den Python-Interpreter eingetippt und ausgeführt.
- Dann haben wir den Code in Dateien geschrieben, wodurch wir dann unsere Programme mehrmals ausführen konnten.

Einleitung



- Es ist Ihnen vielleicht nicht aufgefallen, aber wir haben gerade wieder einen großen Schritt in unseren Programmierfähigkeiten gemacht.
- Wir sind von einzelnen Programmen, die nur aus einem Block von Code bestehen, zu modularen Programmen fortgeschritten.
- Wir können nun Code wiederverwenden.
- Als wir unsere Reise begonnen haben, haben wir unsere Befehle einen nach dem anderen in den Python-Interpreter eingetippt und ausgeführt.
- Dann haben wir den Code in Dateien geschrieben, wodurch wir dann unsere Programme mehrmals ausführen konnten.
- Jetzt können wir unseren Code in Funktionen strukturieren, die wir mit Docstrings und Type Hints erklären können.

Einleitung



- Es ist Ihnen vielleicht nicht aufgefallen, aber wir haben gerade wieder einen großen Schritt in unseren Programmierfähigkeiten gemacht.
- Wir sind von einzelnen Programmen, die nur aus einem Block von Code bestehen, zu modularen Programmen fortgeschritten.
- Wir können nun Code wiederverwenden.
- Als wir unsere Reise begonnen haben, haben wir unsere Befehle einen nach dem anderen in den Python-Interpreter eingetippt und ausgeführt.
- Dann haben wir den Code in Dateien geschrieben, wodurch wir dann unsere Programme mehrmals ausführen konnten.
- Jetzt können wir unseren Code in Funktionen strukturieren, die wir mit Docstrings und Type Hints erklären können.
- Jetzt können wir schon nützliche und vernünftig große Programme schreiben.

Einleitung



- Es ist Ihnen vielleicht nicht aufgefallen, aber wir haben gerade wieder einen großen Schritt in unseren Programmierfähigkeiten gemacht.
- Wir sind von einzelnen Programmen, die nur aus einem Block von Code bestehen, zu modularen Programmen fortgeschritten.
- Wir können nun Code wiederverwenden.
- Als wir unsere Reise begonnen haben, haben wir unsere Befehle einen nach dem anderen in den Python-Interpreter eingetippt und ausgeführt.
- Dann haben wir den Code in Dateien geschrieben, wodurch wir dann unsere Programme mehrmals ausführen konnten.
- Jetzt können wir unseren Code in Funktionen strukturieren, die wir mit Docstrings und Type Hints erklären können.
- Jetzt können wir schon nützliche und vernünftig große Programme schreiben.
- Aber bis jetzt sind unsere gesamten Programme noch in einzelnen Datei gespeichert.

Einleitung



- Es ist Ihnen vielleicht nicht aufgefallen, aber wir haben gerade wieder einen großen Schritt in unseren Programmierfähigkeiten gemacht.
- Wir sind von einzelnen Programmen, die nur aus einem Block von Code bestehen, zu modularen Programmen fortgeschritten.
- Wir können nun Code wiederverwenden.
- Als wir unsere Reise begonnen haben, haben wir unsere Befehle einen nach dem anderen in den Python-Interpreter eingetippt und ausgeführt.
- Dann haben wir den Code in Dateien geschrieben, wodurch wir dann unsere Programme mehrmals ausführen konnten.
- Jetzt können wir unseren Code in Funktionen strukturieren, die wir mit Docstrings und Type Hints erklären können.
- Jetzt können wir schon nützliche und vernünftig große Programme schreiben.
- Aber bis jetzt sind unsere gesamten Programme noch in einzelnen Datei gespeichert.
- Das bringt uns an eine gewisse Grenze, für die Komplexität von den Applikationen, die wir bauen können.

Einleitung



- Wir sind von einzelnen Programmen, die nur aus einem Block von Code bestehen, zu modularen Programmen fortgeschritten.
- Wir können nun Code wiederverwenden.
- Als wir unsere Reise begonnen haben, haben wir unsere Befehle einen nach dem anderen in den Python-Interpreter eingetippt und ausgeführt.
- Dann haben wir den Code in Dateien geschrieben, wodurch wir dann unsere Programme mehrmals ausführen konnten.
- Jetzt können wir unseren Code in Funktionen strukturieren, die wir mit Docstrings und Type Hints erklären können.
- Jetzt können wir schon nützliche und vernünftig große Programme schreiben.
- Aber bis jetzt sind unsere gesamten Programme noch in einzelnen Dateien gespeichert.
- Das bringt uns an eine gewisse Grenze, für die Komplexität von den Applikationen, die wir bauen können.
- Nach ein paar Tausend Zeilen Code in einer einzelnen Datei und vielleicht ein paar Duzend Funktionen wird es sehr schwer, irgendeine Übersicht zu behalten.

Einleitung



- Wir können nun Code wiederverwenden.
- Als wir unsere Reise begonnen haben, haben wir unsere Befehle einen nach dem anderen in den Python-Interpreter eingetippt und ausgeführt.
- Dann haben wir den Code in Dateien geschrieben, wodurch wir dann unsere Programme mehrmals ausführen konnten.
- Jetzt können wir unseren Code in Funktionen strukturieren, die wir mit Docstrings und Type Hints erklären können.
- Jetzt können wir schon nützliche und vernünftig große Programme schreiben.
- Aber bis jetzt sind unsere gesamten Programme noch in einzelnen Datei gespeichert.
- Das bringt uns an eine gewisse Grenze, für die Komplexität von den Applikationen, die wir bauen können.
- Nach ein paar Tausend Zeile Code in einer einzelnen Datei und vielleicht ein paar Duzend Funktionen wird es sehr schwer, irgendeine Übersicht zu behalten.
- Wir könnten diese Begrenzung leicht aufbrechen, wenn wir Code in verschiedene Dateien aufteilen könnten.

Module und Pakete

- Wir lernen jetzt also, wie wir Code auf mehrere Dateien verteilen können.



Module und Pakete

- Wir lernen jetzt also, wie wir Code auf mehrere Dateien verteilen können.
- Das beantwortet zwei wichtige Fragen.



Module und Pakete



- Wir lernen jetzt also, wie wir Code auf mehrere Dateien verteilen können.
- Das beantwortet zwei wichtige Fragen
 1. Wie können wir es vermeiden, unsere Applikationen als einzelne große, unstrukturierte Dateien zu schreiben, die wir unmöglich auf lange Zeit warten können?

Module und Pakete



- Wir lernen jetzt also, wie wir Code auf mehrere Dateien verteilen können.
- Das beantwortet zwei wichtige Fragen
 1. Wie können wir es vermeiden, unsere Applikationen als einzelne große, unstrukturierte Dateien zu schreiben, die wir unmöglich auf lange Zeit warten können?
 2. Wie können wir unsere Applikationen in kleinere Einheiten zerlegen, die wir einzeln Testen, Verbessern, und Warten können und vielleicht sogar in verschiedenen Kontexten verwenden können?

Module und Pakete



- Wir lernen jetzt also, wie wir Code auf mehrere Dateien verteilen können.
- Das beantwortet zwei wichtige Fragen
 1. Wie können wir es vermeiden, unsere Applikationen als einzelne große, unstrukturierte Dateien zu schreiben, die wir unmöglich auf lange Zeit warten können?
 2. Wie können wir unsere Applikationen in kleinere Einheiten zerlegen, die wir einzeln Testen, Verbessern, und Warten können und vielleicht sogar in verschiedenen Kontexten verwenden können?
- Ein großer Teil der Antwort sind *Module* und *Pakete*⁶².

Module und Pakete



- Wir lernen jetzt also, wie wir Code auf mehrere Dateien verteilen können.
- Das beantwortet zwei wichtige Fragen
 1. Wie können wir es vermeiden, unsere Applikationen als einzelne große, unstrukturierte Dateien zu schreiben, die wir unmöglich auf lange Zeit warten können?
 2. Wie können wir unsere Applikationen in kleinere Einheiten zerlegen, die wir einzeln Testen, Verbessern, und Warten können und vielleicht sogar in verschiedenen Kontexten verwenden können?
- Ein großer Teil der Antwort sind *Module* und *Pakete*⁶².
- Hier, ist ein *Modul* im Grunde eine Python-Datei und ein *Paket* ist ein Verzeichnis, das solche Dateien beinhaltet.

Module und Pakete



- Wir lernen jetzt also, wie wir Code auf mehrere Dateien verteilen können.
- Das beantwortet zwei wichtige Fragen
 1. Wie können wir es vermeiden, unsere Applikationen als einzelne große, unstrukturierte Dateien zu schreiben, die wir unmöglich auf lange Zeit warten können?
 2. Wie können wir unsere Applikationen in kleinere Einheiten zerlegen, die wir einzeln Testen, Verbessern, und Warten können und vielleicht sogar in verschiedenen Kontexten verwenden können?
- Ein großer Teil der Antwort sind *Module* und *Pakete*⁶².
- Hier, ist ein *Modul* im Grunde eine Python-Datei und ein *Paket* ist ein Verzeichnis, das solche Dateien beinhaltet.
- Wie in [62] steht, müssen Module nicht unbedingt Dateien seien und Pakete können auch anders erstellt werden ... wir bleiben hier aber bei den einfachen Definitionen.

Module und Pakete



- Wir lernen jetzt also, wie wir Kode auf mehrere Dateien verteilen können.
- Das beantwortet zwei wichtige Fragen
 1. Wie können wir es vermeiden, unsere Applikationen als einzelne große, unstrukturierte Dateien zu schreiben, die wir unmöglich auf lange Zeit warten können?
 2. Wie können wir unsere Applikationen in kleinere Einheiten zerlegen, die wir einzeln Testen, Verbessern, und Warten können und vielleicht sogar in verschiedenen Kontexten verwenden können?
- Ein großer Teil der Antwort sind *Module* und *Pakete*⁶².
- Hier, ist ein *Modul* im Grunde eine Python-Datei und ein *Paket* ist ein Verzeichnis, das solche Dateien beinhaltet.
- Wie in [62] steht, müssen Module nicht unbedingt Dateien seien und Pakete können auch anders erstellt werden ... wir bleiben hier aber bei den einfachen Definitionen.
- Wir haben ja auch schon mit Modulen gearbeitet, z. B. dem Modul `math`.



Beispiel



Unser Eigenes Modul

- Wir haben ja auch schon mit Modulen gearbeitet, z. B. dem Modul math.



Unser Eigenes Modul

- Wir haben ja auch schon mit Modulen gearbeitet, z. B. dem Modul `math`.
- Dieses Modul ist im Grunde eine Kollektion mathematischer Funktionen.



Unser Eigenes Modul

- Wir haben ja auch schon mit Modulen gearbeitet, z. B. dem Modul `math`.
- Dieses Modul ist im Grunde eine Kollektion mathematischer Funktionen.
- Wir haben ja auch selber schon ein paar mathematische Funktionen implementiert.



Unser Eigenes Modul



- Wir haben ja auch schon mit Modulen gearbeitet, z. B. dem Modul `math`.
- Dieses Modul ist im Grunde eine Kollektion mathematischer Funktionen.
- Wir haben ja auch selber schon ein paar mathematische Funktionen implementiert.
- Tuen wir also ein paar davon in ein Modul!

Unser Eigenes Modul

- Wir haben ja auch schon mit Modulen gearbeitet, z. B. dem Modul `math`.
- Dieses Modul ist im Grunde eine Kollektion mathematischer Funktionen.
- Wir haben ja auch selber schon ein paar mathematische Funktionen implementiert.
- Tuen wir also ein paar davon in ein Modul!
- Wir erstellen die Datei `my_math.py` und tun zwei Funktionen hinein.

```
1  """A module with mathematics routines."""
2
3  from math import isclose # Checks if two float numbers are similar.
4
5
6  def factorial(a: int) -> int: # 1 `int` parameter and `int` result
7      """
8      Compute the factorial of a positive integer `a`.
9
10     :param a: the number to compute the factorial of
11     :return: the factorial of `a`, i.e., `a!`.
12     """
13     product: int = 1 # Initialize `product` as `1`.
14     for i in range(2, a + 1): # `i` goes from `2` to `a`.
15         product *= i # Multiply `i` to the product.
16     return product # Return the product, which now is the factorial.
17
18
19 def sqrt(number: float) -> float:
20     """
21     Compute the square root of a given `number`.
22
23     :param number: The number to compute the square root of.
24     :return: A value `v` such that `v * v` is approximately `number`.
25     """
26     guess: float = 1.0 # This will hold the current guess.
27     old_guess: float = 0.0 # 0.0 is just a dummy value != guess.
28     while not isclose(old_guess, guess): # Repeat until no change.
29         old_guess = guess # The current guess becomes the old guess.
30         guess = 0.5 * (guess + number / guess) # The new guess.
31     return guess
```

Unser Eigenes Modul



- Dieses Modul ist im Grunde eine Kollektion mathematischer Funktionen.
- Wir haben ja auch selber schon ein paar mathematische Funktionen implementiert.
- Tuen wir also ein paar davon in ein Modul!
- Wir erstellen die Datei `my_math.py` und tuen zwei Funktionen hinein.

Gute Praxis

Die Namen von Paketen und Modulen sollen kurz und in Kleinbuchstaben geschrieben werden. Unterstriche können verwendet werden, um die Lesbarkeit zu verbessern.⁶⁸

Unser Eigenes Modul

- Wir haben ja auch selber schon ein paar mathematische Funktionen implementiert.
- Tuen wir also ein paar davon in ein Modul!
- Wir erstellen die Datei `my_math.py` und tuen zwei Funktionen hinein:
- Die Funktion `factorial` und eine neue Funktion namens `sqrt`.

```
1  """A module with mathematics routines."""
2
3  from math import isclose # Checks if two float numbers are similar.
4
5
6  def factorial(a: int) -> int: # 1 `int` parameter and `int` result
7      """
8      Compute the factorial of a positive integer `a`.
9
10     :param a: the number to compute the factorial of
11     :return: the factorial of `a`, i.e., `a!`.
12     """
13     product: int = 1 # Initialize `product` as `1`.
14     for i in range(2, a + 1): # `i` goes from `2` to `a`.
15         product *= i # Multiply `i` to the product.
16     return product # Return the product, which now is the factorial.
17
18
19 def sqrt(number: float) -> float:
20     """
21     Compute the square root of a given `number`.
22
23     :param number: The number to compute the square root of.
24     :return: A value `v` such that `v * v` is approximately `number`.
25     """
26     guess: float = 1.0 # This will hold the current guess.
27     old_guess: float = 0.0 # 0.0 is just a dummy value != guess.
28     while not isclose(old_guess, guess): # Repeat until no change.
29         old_guess = guess # The current guess becomes the old guess.
30         guess = 0.5 * (guess + number / guess) # The new guess.
31     return guess
```

Unser Eigenes Modul

- Tuen wir also ein paar davon in ein Modul!
- Wir erstellen die Datei `my_math.py` und tuen zwei Funktionen hinein:
- Die Funktion `factorial` und eine neue Funktion namens `sqrt`.
- Die `sqrt`-Funktion basically ist im Grunde unser Kode von Einheit 25 mit unserer Implementierung von Heron's Methode zur Berechnung der Quadratwurzel.

```
1  """A module with mathematics routines."""
2
3  from math import isclose # Checks if two float numbers are similar.
4
5
6  def factorial(a: int) -> int: # 1 `int` parameter and `int` result
7      """
8      Compute the factorial of a positive integer `a`.
9
10     :param a: the number to compute the factorial of
11     :return: the factorial of `a`, i.e., `a!`.
12     """
13     product: int = 1 # Initialize `product` as `1`.
14     for i in range(2, a + 1): # `i` goes from `2` to `a`.
15         product *= i # Multiply `i` to the product.
16     return product # Return the product, which now is the factorial.
17
18
19 def sqrt(number: float) -> float:
20     """
21     Compute the square root of a given `number`.
22
23     :param number: The number to compute the square root of.
24     :return: A value `v` such that `v * v` is approximately `number`.
25     """
26     guess: float = 1.0 # This will hold the current guess.
27     old_guess: float = 0.0 # 0.0 is just a dummy value != guess.
28     while not isclose(old_guess, guess): # Repeat until no change.
29         old_guess = guess # The current guess becomes the old guess.
30         guess = 0.5 * (guess + number / guess) # The new guess.
31     return guess
```


Unser Eigenes Modul

- Wir erstellen die Datei `my_math.py` und tun zwei Funktionen hinein:
- Die Funktion `factorial` und eine neue Funktion namens `sqrt`.
- Die `sqrt`-Funktion basically ist im Grunde unser Kode von Einheit 25 mit unserer Implementierung von Heron's Methode zur Berechnung der Quadratwurzel.
- Nun kommt jedoch `number` als Parameter in unsere Funktion hinein.

```
1  """A module with mathematics routines."""
2
3  from math import isclose # Checks if two float numbers are similar.
4
5
6  def factorial(a: int) -> int: # 1 `int` parameter and `int` result
7      """
8      Compute the factorial of a positive integer `a`.
9
10     :param a: the number to compute the factorial of
11     :return: the factorial of `a`, i.e., `a!`.
12     """
13     product: int = 1 # Initialize `product` as `1`.
14     for i in range(2, a + 1): # `i` goes from `2` to `a`.
15         product *= i # Multiply `i` to the product.
16     return product # Return the product, which now is the factorial.
17
18
19 def sqrt(number: float) -> float:
20     """
21     Compute the square root of a given `number`.
22
23     :param number: The number to compute the square root of.
24     :return: A value `v` such that `v * v` is approximately `number`.
25     """
26     guess: float = 1.0 # This will hold the current guess.
27     old_guess: float = 0.0 # 0.0 is just a dummy value != guess.
28     while not isclose(old_guess, guess): # Repeat until no change.
29         old_guess = guess # The current guess becomes the old guess.
30         guess = 0.5 * (guess + number / guess) # The new guess.
31     return guess
```

Unser Eigenes Modul

- Die Funktion `factorial` und eine neue Funktion namens `sqrt`.
- Die `sqrt`-Funktion basically ist im Grunde unser Kode von Einheit 25 mit unserer Implementierung von Heron's Methode zur Berechnung der Quadratwurzel.
- Nun kommt jedoch `number` als Parameter in unsere Funktion hinein.
- Unser neues Modul has the name `my_math`, weil es in der Datei `my_math.py` gespeichert ist.

```
1  """A module with mathematics routines."""
2
3  from math import isclose # Checks if two float numbers are similar.
4
5
6  def factorial(a: int) -> int: # 1 `int` parameter and `int` result
7      """
8      Compute the factorial of a positive integer `a`.
9
10     :param a: the number to compute the factorial of
11     :return: the factorial of `a`, i.e., `a!`.
12     """
13     product: int = 1 # Initialize `product` as `1`.
14     for i in range(2, a + 1): # `i` goes from `2` to `a`.
15         product *= i # Multiply `i` to the product.
16     return product # Return the product, which now is the factorial.
17
18
19 def sqrt(number: float) -> float:
20     """
21     Compute the square root of a given `number`.
22
23     :param number: The number to compute the square root of.
24     :return: A value `v` such that `v * v` is approximately `number`.
25     """
26     guess: float = 1.0 # This will hold the current guess.
27     old_guess: float = 0.0 # 0.0 is just a dummy value != guess.
28     while not isclose(old_guess, guess): # Repeat until no change.
29         old_guess = guess # The current guess becomes the old guess.
30         guess = 0.5 * (guess + number / guess) # The new guess.
31     return guess
```

Unser Eigenes Modul

- Die Funktion `factorial` und eine neue Funktion namens `sqrt`.
- Die `sqrt`-Funktion basically ist im Grunde unser Kode von Einheit 25 mit unserer Implementierung von Heron's Methode zur Berechnung der Quadratwurzel.
- Nun kommt jedoch `number` als Parameter in unsere Funktion hinein.
- Unser neues Modul has the name `my_math`, weil es in der Datei `my_math.py` gespeichert ist.
- Es sieht im Grunde nicht anders aus, als was wir bisher gemacht haben.

```
1  """A module with mathematics routines."""
2
3  from math import isclose # Checks if two float numbers are similar.
4
5
6  def factorial(a: int) -> int: # 1 `int` parameter and `int` result
7      """
8      Compute the factorial of a positive integer `a`.
9
10     :param a: the number to compute the factorial of
11     :return: the factorial of `a`, i.e., `a!`.
12     """
13     product: int = 1 # Initialize `product` as `1`.
14     for i in range(2, a + 1): # `i` goes from `2` to `a`.
15         product *= i # Multiply `i` to the product.
16     return product # Return the product, which now is the factorial.
17
18
19 def sqrt(number: float) -> float:
20     """
21     Compute the square root of a given `number`.
22
23     :param number: The number to compute the square root of.
24     :return: A value `v` such that `v * v` is approximately `number`.
25     """
26     guess: float = 1.0 # This will hold the current guess.
27     old_guess: float = 0.0 # 0.0 is just a dummy value != guess.
28     while not isclose(old_guess, guess): # Repeat until no change.
29         old_guess = guess # The current guess becomes the old guess.
30         guess = 0.5 * (guess + number / guess) # The new guess.
31     return guess
```

Unser Eigenes Modul

- Die `sqrt`-Funktion basically ist im Grunde unser Kode von Einheit 25 mit unserer Implementierung von Heron's Methode zur Berechnung der Quadratwurzel.
- Nun kommt jedoch `number` als Parameter in unsere Funktion hinein.
- Unser neues Modul has the name `my_math`, weil es in der Datei `my_math.py` gespeichert ist.
- Es sieht im Grunde nicht anders aus, als was wir bisher gemacht haben.
- Ein Unterschied ist, dass es *nichts* macht.

```
1  """A module with mathematics routines."""
2
3  from math import isclose # Checks if two float numbers are similar.
4
5
6  def factorial(a: int) -> int: # 1 `int` parameter and `int` result
7      """
8      Compute the factorial of a positive integer `a`.
9
10     :param a: the number to compute the factorial of
11     :return: the factorial of `a`, i.e., `a!`.
12     """
13     product: int = 1 # Initialize `product` as `1`.
14     for i in range(2, a + 1): # `i` goes from `2` to `a`.
15         product *= i # Multiply `i` to the product.
16     return product # Return the product, which now is the factorial.
17
18
19 def sqrt(number: float) -> float:
20     """
21     Compute the square root of a given `number`.
22
23     :param number: The number to compute the square root of.
24     :return: A value `v` such that `v * v` is approximately `number`.
25     """
26     guess: float = 1.0 # This will hold the current guess.
27     old_guess: float = 0.0 # 0.0 is just a dummy value != guess.
28     while not isclose(old_guess, guess): # Repeat until no change.
29         old_guess = guess # The current guess becomes the old guess.
30         guess = 0.5 * (guess + number / guess) # The new guess.
31     return guess
```


Unser Eigenes Modul

- Nun kommt jedoch `number` als Parameter in unsere Funktion hinein.
- Unser neues Modul has the name `my_math`, weil es in der Datei `my_math.py` gespeichert ist.
- Es sieht im Grunde nicht anders aus, als was wir bisher gemacht haben.
- Ein Unterschied ist, dass es *nichts* macht.
- In der Datei erstellen wir zwei Funktionen.

```
1  """A module with mathematics routines."""
2
3  from math import isclose # Checks if two float numbers are similar.
4
5
6  def factorial(a: int) -> int: # 1 `int` parameter and `int` result
7      """
8      Compute the factorial of a positive integer `a`.
9
10     :param a: the number to compute the factorial of
11     :return: the factorial of `a`, i.e., `a!`.
12     """
13     product: int = 1 # Initialize `product` as `1`.
14     for i in range(2, a + 1): # `i` goes from `2` to `a`.
15         product *= i # Multiply `i` to the product.
16     return product # Return the product, which now is the factorial.
17
18
19 def sqrt(number: float) -> float:
20     """
21     Compute the square root of a given `number`.
22
23     :param number: The number to compute the square root of.
24     :return: A value `v` such that `v * v` is approximately `number`.
25     """
26     guess: float = 1.0 # This will hold the current guess.
27     old_guess: float = 0.0 # 0.0 is just a dummy value != guess.
28     while not isclose(old_guess, guess): # Repeat until no change.
29         old_guess = guess # The current guess becomes the old guess.
30         guess = 0.5 * (guess + number / guess) # The new guess.
31     return guess
```

Unser Eigenes Modul

- Unser neues Modul has the name `my_math`, weil es in der Datei `my_math.py` gespeichert ist.
- Es sieht im Grunde nicht anders aus, als was wir bisher gemacht haben.
- Ein Unterschied ist, dass es *nichts* macht.
- In der Datei erstellen wir zwei Funktionen.
- Wir rufen sie aber nicht auf und führen auch sonst keinen Code aktiv aus.

```
1  """A module with mathematics routines."""
2
3  from math import isclose # Checks if two float numbers are similar.
4
5
6  def factorial(a: int) -> int: # 1 `int` parameter and `int` result
7      """
8      Compute the factorial of a positive integer `a`.
9
10     :param a: the number to compute the factorial of
11     :return: the factorial of `a`, i.e., `a!`.
12     """
13     product: int = 1 # Initialize `product` as `1`.
14     for i in range(2, a + 1): # `i` goes from `2` to `a`.
15         product *= i # Multiply `i` to the product.
16     return product # Return the product, which now is the factorial.
17
18
19 def sqrt(number: float) -> float:
20     """
21     Compute the square root of a given `number`.
22
23     :param number: The number to compute the square root of.
24     :return: A value `v` such that `v * v` is approximately `number`.
25     """
26     guess: float = 1.0 # This will hold the current guess.
27     old_guess: float = 0.0 # 0.0 is just a dummy value != guess.
28     while not isclose(old_guess, guess): # Repeat until no change.
29         old_guess = guess # The current guess becomes the old guess.
30         guess = 0.5 * (guess + number / guess) # The new guess.
31     return guess
```

Unser Eigenes Modul

- Es sieht im Grunde nicht anders aus, als was wir bisher gemacht haben.
- Ein Unterschied ist, dass es *nichts* macht.
- In der Datei erstellen wir zwei Funktionen.
- Wir rufen sie aber nicht auf und führen auch sonst keinen Code aktiv aus.
- Das ist der Sinn des Moduls.

```
1  """A module with mathematics routines."""
2
3  from math import isclose # Checks if two float numbers are similar.
4
5
6  def factorial(a: int) -> int: # 1 `int` parameter and `int` result
7      """
8      Compute the factorial of a positive integer `a`.
9
10     :param a: the number to compute the factorial of
11     :return: the factorial of `a`, i.e., `a!`.
12     """
13     product: int = 1 # Initialize `product` as `1`.
14     for i in range(2, a + 1): # `i` goes from `2` to `a`.
15         product *= i # Multiply `i` to the product.
16     return product # Return the product, which now is the factorial.
17
18
19 def sqrt(number: float) -> float:
20     """
21     Compute the square root of a given `number`.
22
23     :param number: The number to compute the square root of.
24     :return: A value `v` such that `v * v` is approximately `number`.
25     """
26     guess: float = 1.0 # This will hold the current guess.
27     old_guess: float = 0.0 # 0.0 is just a dummy value != guess.
28     while not isclose(old_guess, guess): # Repeat until no change.
29         old_guess = guess # The current guess becomes the old guess.
30         guess = 0.5 * (guess + number / guess) # The new guess.
31     return guess
```

Unser Eigenes Modul

- Ein Unterschied ist, dass es *nichts* macht.
- In der Datei erstellen wir zwei Funktionen.
- Wir rufen sie aber nicht auf und führen auch sonst keinen Code aktiv aus.
- Das ist der Sinn des Moduls.
- Es stellt zwei Funktionen zur Verfügung.

```
1  """A module with mathematics routines."""
2
3  from math import isclose # Checks if two float numbers are similar.
4
5
6  def factorial(a: int) -> int: # 1 `int` parameter and `int` result
7      """
8      Compute the factorial of a positive integer `a`.
9
10     :param a: the number to compute the factorial of
11     :return: the factorial of `a`, i.e., `a!`.
12     """
13     product: int = 1 # Initialize `product` as `1`.
14     for i in range(2, a + 1): # `i` goes from `2` to `a`.
15         product *= i # Multiply `i` to the product.
16     return product # Return the product, which now is the factorial.
17
18
19 def sqrt(number: float) -> float:
20     """
21     Compute the square root of a given `number`.
22
23     :param number: The number to compute the square root of.
24     :return: A value `v` such that `v * v` is approximately `number`.
25     """
26     guess: float = 1.0 # This will hold the current guess.
27     old_guess: float = 0.0 # 0.0 is just a dummy value != guess.
28     while not isclose(old_guess, guess): # Repeat until no change.
29         old_guess = guess # The current guess becomes the old guess.
30         guess = 0.5 * (guess + number / guess) # The new guess.
31     return guess
```


Unser Eigenes Modul

- In der Datei erstellen wir zwei Funktionen.
- Wir rufen sie aber nicht auf und führen auch sonst keinen Code aktiv aus.
- Das ist der Sinn des Moduls.
- Es stellt zwei Funktionen zur Verfügung.
- Diese können wir nun an anderer Stelle verwenden.

```
1  """A module with mathematics routines."""
2
3  from math import isclose # Checks if two float numbers are similar.
4
5
6  def factorial(a: int) -> int: # 1 `int` parameter and `int` result
7      """
8      Compute the factorial of a positive integer `a`.
9
10     :param a: the number to compute the factorial of
11     :return: the factorial of `a`, i.e., `a!`.
12     """
13     product: int = 1 # Initialize `product` as `1`.
14     for i in range(2, a + 1): # `i` goes from `2` to `a`.
15         product *= i # Multiply `i` to the product.
16     return product # Return the product, which now is the factorial.
17
18
19 def sqrt(number: float) -> float:
20     """
21     Compute the square root of a given `number`.
22
23     :param number: The number to compute the square root of.
24     :return: A value `v` such that `v * v` is approximately `number`.
25     """
26     guess: float = 1.0 # This will hold the current guess.
27     old_guess: float = 0.0 # 0.0 is just a dummy value != guess.
28     while not isclose(old_guess, guess): # Repeat until no change.
29         old_guess = guess # The current guess becomes the old guess.
30         guess = 0.5 * (guess + number / guess) # The new guess.
31     return guess
```

Unser Eigenes Modul

- Wir rufen sie aber nicht auf und führen auch sonst keinen Code aktiv aus.
- Das ist der Sinn des Moduls.
- Es stellt zwei Funktionen zur Verfügung.
- Diese können wir nun an anderer Stelle verwenden.
- Und Datei `use_my_math.py` ist wo wir sie verwenden.

```
1  """Using the mathematics module."""
2
3  from my_math import factorial, sqrt # Import our two functions.
4
5  print(f"6!={factorial(6)}") # Use the `factorial` function.
6  print(f"\u221A3={sqrt(3.0)}") # Use the `sqrt` function.
7  print(f"\u221A(6!*1.5)={sqrt(factorial(6) * 1.5)}") # Use both.
8
9  print("We now use Liu Hui's Method to Approximate \u03c0.")
10 e: int = 6 # the number of edges: We start with a hexagon, i.e., e=6.
11 s: float = 1.0 # the side length: Initially 1, i.e., radius is also 1.
12 for _ in range(6):
13     print(f"{e} edges, side length={s} give us \u03c0\u2248{e * s / 2}.")
14     e *= 2 # We double the number of edges...
15     s = sqrt(2 - sqrt(4 - (s ** 2))) # ...and recompute the side length
```

↓ python3 use_my_math.py ↓

```
1  6!=720
2  √3=1.7320508075688772
3  √(6!*1.5)=32.863353450309965
4  We now use Liu Hui's Method to Approximate π.
5  6 edges, side length=1.0 give us π≈3.0.
6  12 edges, side length=0.5176380902050417 give us π≈3.10582854123025.
7  24 edges, side length=0.2610523844401035 give us π≈3.1326286132812418.
8  48 edges, side length=0.13080625846028637 give us π≈3.139350203046873.
9  96 edges, side length=0.0654381656435527 give us π≈3.14103195089053.
10 192 edges, side length=0.03272346325297234 give us π≈3.1414524722853443.
```

Unser Eigenes Modul

- Das ist der Sinn des Moduls.
- Es stellt zwei Funktionen zur Verfügung.
- Diese können wir nun an anderer Stelle verwenden.
- Und Datei `use_my_math.py` ist wo wir sie verwenden.
- In dieser Datei wollen wir unsere beiden Funktionen `factorial` und `sqrt` vom Modul `my_math` benutzen.

```
1  """Using the mathematics module."""
2
3  from my_math import factorial, sqrt # Import our two functions.
4
5  print(f"6!={factorial(6)}") # Use the `factorial` function.
6  print(f"\u221A3={sqrt(3.0)}") # Use the `sqrt` function.
7  print(f"\u221A(6!*1.5)={sqrt(factorial(6) * 1.5)}") # Use both.
8
9  print("We now use Liu Hui's Method to Approximate \u03c0.")
10 e: int = 6 # the number of edges: We start with a hexagon, i.e., e=6.
11 s: float = 1.0 # the side length: Initially 1, i.e., radius is also 1.
12 for _ in range(6):
13     print(f"{e} edges, side length={s} give us \u03c0\u2248{e * s / 2}.")
14     e *= 2 # We double the number of edges...
15     s = sqrt(2 - sqrt(4 - (s ** 2))) # ...and recompute the side length
```

↓ python3 use_my_math.py ↓

```
1  6!=720
2  √3=1.7320508075688772
3  √(6!*1.5)=32.863353450309965
4  We now use Liu Hui's Method to Approximate π.
5  6 edges, side length=1.0 give us π≈3.0.
6  12 edges, side length=0.5176380902050417 give us π≈3.10582854123025.
7  24 edges, side length=0.2610523844401035 give us π≈3.1326286132812418.
8  48 edges, side length=0.13080625846028637 give us π≈3.139350203046873.
9  96 edges, side length=0.0654381656435527 give us π≈3.14103195089053.
10 192 edges, side length=0.03272346325297234 give us π≈3.1414524722853443.
```

Unser Eigenes Modul

- Es stellt zwei Funktionen zur Verfügung.
- Diese können wir nun an anderer Stelle verwenden.
- Und Datei `use_my_math.py` ist wo wir sie verwenden.
- In dieser Datei wollen wir unsere beiden Funktionen `factorial` und `sqrt` vom Modul `my_math` benutzen.
- Wir müssen dem Python-Interpreter also sagen, wo es diese finden kann.

```
1  """Using the mathematics module."""
2
3  from my_math import factorial, sqrt # Import our two functions.
4
5  print(f"6!={factorial(6)}") # Use the `factorial` function.
6  print(f"\u221A3={sqrt(3.0)}") # Use the `sqrt` function.
7  print(f"\u221A(6!*1.5)={sqrt(factorial(6) * 1.5)}") # Use both.
8
9  print("We now use Liu Hui's Method to Approximate \u03c0.")
10 e: int = 6 # the number of edges: We start with a hexagon, i.e., e=6.
11 s: float = 1.0 # the side length: Initially 1, i.e., radius is also 1.
12 for _ in range(6):
13     print(f"{e} edges, side length={s} give us \u03c0\u2248{e * s / 2}.")
14     e *= 2 # We double the number of edges...
15     s = sqrt(2 - sqrt(4 - (s ** 2))) # ...and recompute the side length
```

↓ python3 use_my_math.py ↓

```
1  6!=720
2  √3=1.7320508075688772
3  √(6!*1.5)=32.863353450309965
4  We now use Liu Hui's Method to Approximate π.
5  6 edges, side length=1.0 give us π≈3.0.
6  12 edges, side length=0.5176380902050417 give us π≈3.10582854123025.
7  24 edges, side length=0.2610523844401035 give us π≈3.1326286132812418.
8  48 edges, side length=0.13080625846028637 give us π≈3.139350203046873.
9  96 edges, side length=0.0654381656435527 give us π≈3.14103195089053.
10 192 edges, side length=0.03272346325297234 give us π≈3.1414524722853443.
```


Unser Eigenes Modul

- Diese können wir nun an anderer Stelle verwenden.
- Und Datei `use_my_math.py` ist wo wir sie verwenden.
- In dieser Datei wollen wir unsere beiden Funktionen `factorial` und `sqrt` vom Modul `my_math` benutzen.
- Wir müssen dem Python-Interpreter also sagen, wo es diese finden kann.
- Wir tun das, in dem wir schreiben `from my_math`
`import factorial, sqrt.`

```
1  """Using the mathematics module."""
2
3  from my_math import factorial, sqrt # Import our two functions.
4
5  print(f"6!={factorial(6)}") # Use the `factorial` function.
6  print(f"\u221A3={sqrt(3.0)}") # Use the `sqrt` function.
7  print(f"\u221A(6!*1.5)={sqrt(factorial(6) * 1.5)}") # Use both.
8
9  print("We now use Liu Hui's Method to Approximate \u03c0.")
10 e: int = 6 # the number of edges: We start with a hexagon, i.e., e=6.
11 s: float = 1.0 # the side length: Initially 1, i.e., radius is also 1.
12 for _ in range(6):
13     print(f"{e} edges, side length={s} give us \u03c0\u2248{e * s / 2}.")
14     e *= 2 # We double the number of edges...
15     s = sqrt(2 - sqrt(4 - (s ** 2))) # ...and recompute the side length
```

↓ python3 use_my_math.py ↓

```
1  6!=720
2  √3=1.7320508075688772
3  √(6!*1.5)=32.863353450309965
4  We now use Liu Hui's Method to Approximate π.
5  6 edges, side length=1.0 give us π≈3.0.
6  12 edges, side length=0.5176380902050417 give us π≈3.10582854123025.
7  24 edges, side length=0.2610523844401035 give us π≈3.1326286132812418.
8  48 edges, side length=0.13080625846028637 give us π≈3.139350203046873.
9  96 edges, side length=0.0654381656435527 give us π≈3.14103195089053.
10 192 edges, side length=0.03272346325297234 give us π≈3.1414524722853443.
```

Unser Eigenes Modul

- Und Datei `use_my_math.py` ist wo wir sie verwenden.
- In dieser Datei wollen wir unsere beiden Funktionen `factorial` und `sqrt` vom Modul `my_math` benutzen.
- Wir müssen dem Python-Interpreter also sagen, wo es diese finden kann.
- Wir tun das, in dem wir schreiben `from my_math`
`import factorial, sqrt`.
- Die Bedeutung ist ziemlich klar.

```
1  """Using the mathematics module."""
2
3  from my_math import factorial, sqrt # Import our two functions.
4
5  print(f"6!={factorial(6)}") # Use the `factorial` function.
6  print(f"\u221A3={sqrt(3.0)}") # Use the `sqrt` function.
7  print(f"\u221A(6!*1.5)={sqrt(factorial(6) * 1.5)}") # Use both.
8
9  print("We now use Liu Hui's Method to Approximate \u03c0.")
10 e: int = 6 # the number of edges: We start with a hexagon, i.e., e=6.
11 s: float = 1.0 # the side length: Initially 1, i.e., radius is also 1.
12 for _ in range(6):
13     print(f"{e} edges, side length={s} give us \u03c0\u2248{e * s / 2}.")
14     e *= 2 # We double the number of edges...
15     s = sqrt(2 - sqrt(4 - (s ** 2))) # ...and recompute the side length
```

↓ python3 use_my_math.py ↓

```
1  6!=720
2  √3=1.7320508075688772
3  √(6!*1.5)=32.863353450309965
4  We now use Liu Hui's Method to Approximate π.
5  6 edges, side length=1.0 give us π≈3.0.
6  12 edges, side length=0.5176380902050417 give us π≈3.10582854123025.
7  24 edges, side length=0.2610523844401035 give us π≈3.1326286132812418.
8  48 edges, side length=0.13080625846028637 give us π≈3.139350203046873.
9  96 edges, side length=0.0654381656435527 give us π≈3.14103195089053.
10 192 edges, side length=0.03272346325297234 give us π≈3.1414524722853443.
```

Unser Eigenes Modul

- In dieser Datei wollen wir unsere beiden Funktionen `factorial` und `sqrt` vom Modul `my_math` benutzen.
- Wir müssen dem Python-Interpreter also sagen, wo es diese finden kann.
- Wir tun das, in dem wir schreiben `from my_math`
`import factorial, sqrt`.
- Die Bedeutung ist ziemlich klar.
- Es gibt ein Modul `my_math` von dem wir die Funktionen `factorial` holen wollen `sqrt`.

```
1  """Using the mathematics module."""
2
3  from my_math import factorial, sqrt # Import our two functions.
4
5  print(f"6!={factorial(6)}") # Use the `factorial` function.
6  print(f"\u221A3={sqrt(3.0)}") # Use the `sqrt` function.
7  print(f"\u221A(6!*1.5)={sqrt(factorial(6) * 1.5)}") # Use both.
8
9  print("We now use Liu Hui's Method to Approximate \u03c0.")
10 e: int = 6 # the number of edges: We start with a hexagon, i.e., e=6.
11 s: float = 1.0 # the side length: Initially 1, i.e., radius is also 1.
12 for _ in range(6):
13     print(f"{e} edges, side length={s} give us \u03c0\u2248{e * s / 2}.")
14     e *= 2 # We double the number of edges...
15     s = sqrt(2 - sqrt(4 - (s ** 2))) # ...and recompute the side length
```

↓ python3 use_my_math.py ↓

```
1  6!=720
2  √3=1.7320508075688772
3  √(6!*1.5)=32.863353450309965
4  We now use Liu Hui's Method to Approximate π.
5  6 edges, side length=1.0 give us π≈3.0.
6  12 edges, side length=0.5176380902050417 give us π≈3.10582854123025.
7  24 edges, side length=0.2610523844401035 give us π≈3.1326286132812418.
8  48 edges, side length=0.13080625846028637 give us π≈3.139350203046873.
9  96 edges, side length=0.0654381656435527 give us π≈3.14103195089053.
10 192 edges, side length=0.03272346325297234 give us π≈3.1414524722853443.
```

Unser Eigenes Modul

- Wir müssen dem Python-Interpreter also sagen, wo es diese finden kann.
- Wir tun das, in dem wir schreiben

```
from my_math
import factorial, sqrt.
```
- Die Bedeutung ist ziemlich klar.
- Es gibt ein Modul `my_math` von dem wir die Funktionen `factorial` holen wollen `sqrt`.
- Der Python-Interpreter kennt viele Module.

```
1  """Using the mathematics module."""
2
3  from my_math import factorial, sqrt  # Import our two functions.
4
5  print(f"6!={factorial(6)}")  # Use the `factorial` function.
6  print(f"\u221A3={sqrt(3.0)}")  # Use the `sqrt` function.
7  print(f"\u221A(6!*1.5)={sqrt(factorial(6) * 1.5)}")  # Use both.
8
9  print("We now use Liu Hui's Method to Approximate \u03c0.")
10 e: int = 6  # the number of edges: We start with a hexagon, i.e., e=6.
11 s: float = 1.0  # the side length: Initially 1, i.e., radius is also 1.
12 for _ in range(6):
13     print(f"{e} edges, side length={s} give us \u03c0\u2248{e * s / 2}.")
14     e *= 2  # We double the number of edges...
15     s = sqrt(2 - sqrt(4 - (s ** 2)))  # ...and recompute the side length
```

↓ python3 use_my_math.py ↓

```
1  6!=720
2  √3=1.7320508075688772
3  √(6!*1.5)=32.863353450309965
4  We now use Liu Hui's Method to Approximate π.
5  6 edges, side length=1.0 give us π≈3.0.
6  12 edges, side length=0.5176380902050417 give us π≈3.10582854123025.
7  24 edges, side length=0.2610523844401035 give us π≈3.1326286132812418.
8  48 edges, side length=0.13080625846028637 give us π≈3.139350203046873.
9  96 edges, side length=0.0654381656435527 give us π≈3.14103195089053.
10 192 edges, side length=0.03272346325297234 give us π≈3.1414524722853443.
```


Unser Eigenes Modul

- Wir tun das, in dem wir schreiben
`from my_math`
`import factorial, sqrt.`
- Die Bedeutung ist ziemlich klar.
- Es gibt ein Modul `my_math` von dem wir die Funktionen `factorial` holen wollen `sqrt`.
- Der Python-Interpreter kennt viele Module.
- Mehrere Module kommen schon mit der Python-Installation mit, wie z. B. `math`.

```
1  """Using the mathematics module."""
2
3  from my_math import factorial, sqrt # Import our two functions.
4
5  print(f"6!={factorial(6)}") # Use the `factorial` function.
6  print(f"\u221A3={sqrt(3.0)}") # Use the `sqrt` function.
7  print(f"\u221A(6!*1.5)={sqrt(factorial(6) * 1.5)}") # Use both.
8
9  print("We now use Liu Hui's Method to Approximate \u03c0.")
10 e: int = 6 # the number of edges: We start with a hexagon, i.e., e=6.
11 s: float = 1.0 # the side length: Initially 1, i.e., radius is also 1.
12 for _ in range(6):
13     print(f"{e} edges, side length={s} give us \u03c0\u2248{e * s / 2}.")
14     e *= 2 # We double the number of edges...
15     s = sqrt(2 - sqrt(4 - (s ** 2))) # ...and recompute the side length
```

↓ python3 use_my_math.py ↓

```
1  6!=720
2  √3=1.7320508075688772
3  √(6!*1.5)=32.863353450309965
4  We now use Liu Hui's Method to Approximate π.
5  6 edges, side length=1.0 give us π≈3.0.
6  12 edges, side length=0.5176380902050417 give us π≈3.10582854123025.
7  24 edges, side length=0.2610523844401035 give us π≈3.1326286132812418.
8  48 edges, side length=0.13080625846028637 give us π≈3.139350203046873.
9  96 edges, side length=0.0654381656435527 give us π≈3.14103195089053.
10 192 edges, side length=0.03272346325297234 give us π≈3.1414524722853443.
```

Unser Eigenes Modul

- Die Bedeutung ist ziemlich klar.
- Es gibt ein Modul `my_math` von dem wir die Funktionen `factorial` holen wollen `sqrt`.
- Der Python-Interpreter kennt viele Module.
- Mehrere Module kommen schon mit der Python-Installation mit, wie z. B. `math`.
- Andere werden mit dem Paketmanager `pip`³¹ installiert.

```
1  """Using the mathematics module."""
2
3  from my_math import factorial, sqrt # Import our two functions.
4
5  print(f"6!={factorial(6)}") # Use the `factorial` function.
6  print(f"\u221A3={sqrt(3.0)}") # Use the `sqrt` function.
7  print(f"\u221A(6!*1.5)={sqrt(factorial(6) * 1.5)}") # Use both.
8
9  print("We now use Liu Hui's Method to Approximate \u03c0.")
10 e: int = 6 # the number of edges: We start with a hexagon, i.e., e=6.
11 s: float = 1.0 # the side length: Initially 1, i.e., radius is also 1.
12 for _ in range(6):
13     print(f"{e} edges, side length={s} give us \u03c0\u2248{e * s / 2}.")
14     e *= 2 # We double the number of edges...
15     s = sqrt(2 - sqrt(4 - (s ** 2))) # ...and recompute the side length
```

↓ python3 use_my_math.py ↓

```
1  6!=720
2  √3=1.7320508075688772
3  √(6!*1.5)=32.863353450309965
4  We now use Liu Hui's Method to Approximate π.
5  6 edges, side length=1.0 give us π≈3.0.
6  12 edges, side length=0.5176380902050417 give us π≈3.10582854123025.
7  24 edges, side length=0.2610523844401035 give us π≈3.1326286132812418.
8  48 edges, side length=0.13080625846028637 give us π≈3.139350203046873.
9  96 edges, side length=0.0654381656435527 give us π≈3.14103195089053.
10 192 edges, side length=0.03272346325297234 give us π≈3.1414524722853443.
```

Unser Eigenes Modul

- Es gibt ein Modul `my_math` von dem wir die Funktionen `factorial` holen wollen `sqrt`.
- Der Python-Interpreter kennt viele Module.
- Mehrere Module kommen schon mit der Python-Installation mit, wie z. B. `math`.
- Andere werden mit dem Paketmanager `pip`³¹ installiert.
- Das Modul `my_math` wird gefunden, weil wir es in das selbe Verzeichnis tun wie Program `use_my_math.py`.

```
1  """Using the mathematics module."""
2
3  from my_math import factorial, sqrt # Import our two functions.
4
5  print(f"6!={factorial(6)}") # Use the `factorial` function.
6  print(f"\u221A3={sqrt(3.0)}") # Use the `sqrt` function.
7  print(f"\u221A(6!*1.5)={sqrt(factorial(6) * 1.5)}") # Use both.
8
9  print("We now use Liu Hui's Method to Approximate \u03c0.")
10 e: int = 6 # the number of edges: We start with a hexagon, i.e., e=6.
11 s: float = 1.0 # the side length: Initially 1, i.e., radius is also 1.
12 for _ in range(6):
13     print(f"{e} edges, side length={s} give us \u03c0\u2248{e * s / 2}.")
14     e *= 2 # We double the number of edges...
15     s = sqrt(2 - sqrt(4 - (s ** 2))) # ...and recompute the side length
```

↓ python3 use_my_math.py ↓

```
1  6!=720
2  √3=1.7320508075688772
3  √(6!*1.5)=32.863353450309965
4  We now use Liu Hui's Method to Approximate π.
5  6 edges, side length=1.0 give us π≈3.0.
6  12 edges, side length=0.5176380902050417 give us π≈3.10582854123025.
7  24 edges, side length=0.2610523844401035 give us π≈3.1326286132812418.
8  48 edges, side length=0.13080625846028637 give us π≈3.139350203046873.
9  96 edges, side length=0.0654381656435527 give us π≈3.14103195089053.
10 192 edges, side length=0.03272346325297234 give us π≈3.1414524722853443.
```

Unser Eigenes Modul

- Der Python-Interpreter kennt viele Module.
- Mehrere Module kommen schon mit der Python-Installation mit, wie z. B. `math`.
- Andere werden mit dem Paketmanager `pip`³¹ installiert.
- Das Modul `my_math` wird gefunden, weil wir es in das selbe Verzeichnis tun wie Program `use_my_math.py`.
- Wir hätten die Datei `my_math.py` auch in ein Unterverzeichnis namens `math_pack` tun können.

```
1  """Using the mathematics module."""
2
3  from my_math import factorial, sqrt # Import our two functions.
4
5  print(f"6!={factorial(6)}") # Use the `factorial` function.
6  print(f"\u221A3={sqrt(3.0)}") # Use the `sqrt` function.
7  print(f"\u221A(6!*1.5)={sqrt(factorial(6) * 1.5)}") # Use both.
8
9  print("We now use Liu Hui's Method to Approximate \u03c0.")
10 e: int = 6 # the number of edges: We start with a hexagon, i.e., e=6.
11 s: float = 1.0 # the side length: Initially 1, i.e., radius is also 1.
12 for _ in range(6):
13     print(f"{e} edges, side length={s} give us \u03c0\u2248{e * s / 2}.")
14     e *= 2 # We double the number of edges...
15     s = sqrt(2 - sqrt(4 - (s ** 2))) # ...and recompute the side length
```

↓ python3 use_my_math.py ↓

```
1  6!=720
2  \u221A3=1.7320508075688772
3  \u221A(6!*1.5)=32.863353450309965
4  We now use Liu Hui's Method to Approximate \u03c0.
5  6 edges, side length=1.0 give us \u03c0\u22483.0.
6  12 edges, side length=0.5176380902050417 give us \u03c0\u22483.10582854123025.
7  24 edges, side length=0.2610523844401035 give us \u03c0\u22483.1326286132812418.
8  48 edges, side length=0.13080625846028637 give us \u03c0\u22483.139350203046873.
9  96 edges, side length=0.0654381656435527 give us \u03c0\u22483.14103195089053.
10 192 edges, side length=0.03272346325297234 give us \u03c0\u22483.1414524722853443.
```


Unser Eigenes Modul

- Mehrere Module kommen schon mit der Python-Installation mit, wie z. B. `math`.
- Andere werden mit dem Paketmanager `pip`³¹ installiert.
- Das Modul `my_math` wird gefunden, weil wir es in das selbe Verzeichnis tun wie Program `use_my_math.py`.
- Wir hätten die Datei `my_math.py` auch in ein Unterverzeichnis namens `math_pack` tun können.
- Dann würden wir unsere Funktionen von `math_pack.my_math` importieren, wobei `math_pack` dann *Paket* genannt werden würde.

```
1  """Using the mathematics module."""
2
3  from my_math import factorial, sqrt # Import our two functions.
4
5  print(f"6!={factorial(6)}") # Use the `factorial` function.
6  print(f"\u221A3={sqrt(3.0)}") # Use the `sqrt` function.
7  print(f"\u221A(6!*1.5)={sqrt(factorial(6) * 1.5)}") # Use both.
8
9  print("We now use Liu Hui's Method to Approximate \u03c0.")
10 e: int = 6 # the number of edges: We start with a hexagon, i.e., e=6.
11 s: float = 1.0 # the side length: Initially 1, i.e., radius is also 1.
12 for _ in range(6):
13     print(f"{e} edges, side length={s} give us \u03c0\u2248{e * s / 2}.")
14     e *= 2 # We double the number of edges...
15     s = sqrt(2 - sqrt(4 - (s ** 2))) # ...and recompute the side length
```

↓ python3 use_my_math.py ↓

```
1  6!=720
2  √3=1.7320508075688772
3  √(6!*1.5)=32.863353450309965
4  We now use Liu Hui's Method to Approximate π.
5  6 edges, side length=1.0 give us π≈3.0.
6  12 edges, side length=0.5176380902050417 give us π≈3.10582854123025.
7  24 edges, side length=0.2610523844401035 give us π≈3.1326286132812418.
8  48 edges, side length=0.13080625846028637 give us π≈3.139350203046873.
9  96 edges, side length=0.0654381656435527 give us π≈3.14103195089053.
10 192 edges, side length=0.03272346325297234 give us π≈3.1414524722853443.
```

Unser Eigenes Modul

- Andere werden mit dem Paketmanager pip³¹ installiert.
- Das Modul `my_math` wird gefunden, weil wir es in das selbe Verzeichnis tun wie Program `use_my_math.py`.
- Wir hätten die Datei `my_math.py` auch in ein Unterverzeichnis namens `math_pack` tun können.
- Dann würden wir unsere Funktionen von `math_pack.my_math` importieren, wobei `math_pack` dann *Paket* genannt werden würde.
- Natürlich könnten wir die Verzeichnisse auch tiefer Schachteln.

```
1  """Using the mathematics module."""
2
3  from my_math import factorial, sqrt # Import our two functions.
4
5  print(f"6!={factorial(6)}") # Use the `factorial` function.
6  print(f"\u221A3={sqrt(3.0)}") # Use the `sqrt` function.
7  print(f"\u221A(6!*1.5)={sqrt(factorial(6) * 1.5)}") # Use both.
8
9  print("We now use Liu Hui's Method to Approximate \u03c0.")
10 e: int = 6 # the number of edges: We start with a hexagon, i.e., e=6.
11 s: float = 1.0 # the side length: Initially 1, i.e., radius is also 1.
12 for _ in range(6):
13     print(f"{e} edges, side length={s} give us \u03c0\u2248{e * s / 2}.")
14     e *= 2 # We double the number of edges...
15     s = sqrt(2 - sqrt(4 - (s ** 2))) # ...and recompute the side length
```

↓ python3 use_my_math.py ↓

```
1  6!=720
2  √3=1.7320508075688772
3  √(6!*1.5)=32.863353450309965
4  We now use Liu Hui's Method to Approximate π.
5  6 edges, side length=1.0 give us π≈3.0.
6  12 edges, side length=0.5176380902050417 give us π≈3.10582854123025.
7  24 edges, side length=0.2610523844401035 give us π≈3.1326286132812418.
8  48 edges, side length=0.13080625846028637 give us π≈3.139350203046873.
9  96 edges, side length=0.0654381656435527 give us π≈3.14103195089053.
10 192 edges, side length=0.03272346325297234 give us π≈3.1414524722853443.
```

Unser Eigenes Modul

- Das Modul `my_math` wird gefunden, weil wir es in das selbe Verzeichnis tun wie Program `use_my_math.py`.
- Wir hätten die Datei `my_math.py` auch in ein Unterverzeichnis namens `math_pack` tun können.
- Dann würden wir unsere Funktionen von `math_pack.my_math` importieren, wobei `math_pack` dann *Paket* genannt werden würde.
- Natürlich könnten wir die Verzeichnisse auch tiefer Schachteln.
- Wir könnten Verzeichnis `utils` haben und das Verzeichnis `math_pack` mit unserer Datei `my_math.py` dort hinein tun.

```
1  """Using the mathematics module."""
2
3  from my_math import factorial, sqrt # Import our two functions.
4
5  print(f"6!={factorial(6)}") # Use the `factorial` function.
6  print(f"\u221A3={sqrt(3.0)}") # Use the `sqrt` function.
7  print(f"\u221A(6!*1.5)={sqrt(factorial(6) * 1.5)}") # Use both.
8
9  print("We now use Liu Hui's Method to Approximate \u03c0.")
10 e: int = 6 # the number of edges: We start with a hexagon, i.e., e=6.
11 s: float = 1.0 # the side length: Initially 1, i.e., radius is also 1.
12 for _ in range(6):
13     print(f"{e} edges, side length={s} give us \u03c0\u2248{e * s / 2}.")
14     e *= 2 # We double the number of edges...
15     s = sqrt(2 - sqrt(4 - (s ** 2))) # ...and recompute the side length
```

↓ python3 use_my_math.py ↓

```
1  6!=720
2  √3=1.7320508075688772
3  √(6!*1.5)=32.863353450309965
4  We now use Liu Hui's Method to Approximate π.
5  6 edges, side length=1.0 give us π≈3.0.
6  12 edges, side length=0.5176380902050417 give us π≈3.10582854123025.
7  24 edges, side length=0.2610523844401035 give us π≈3.1326286132812418.
8  48 edges, side length=0.13080625846028637 give us π≈3.139350203046873.
9  96 edges, side length=0.0654381656435527 give us π≈3.14103195089053.
10 192 edges, side length=0.03272346325297234 give us π≈3.1414524722853443.
```

Unser Eigenes Modul

- Dann würden wir unsere Funktionen von `math_pack.my_math` importieren, wobei `math_pack` dann *Paket* genannt werden würde.
- Natürlich könnten wir die Verzeichnisse auch tiefer Schachteln.
- Wir könnten Verzeichnis `utils` haben und das Verzeichnis `math_pack` mit unserer Datei `my_math.py` dort hinein tun.
- Dann würden wir unsere Funktionen so importieren: `from utils.math_pack.my_math import`....

```
1  """Using the mathematics module."""
2
3  from my_math import factorial, sqrt # Import our two functions.
4
5  print(f"6!={factorial(6)}") # Use the `factorial` function.
6  print(f"\u221A3={sqrt(3.0)}") # Use the `sqrt` function.
7  print(f"\u221A(6!*1.5)={sqrt(factorial(6) * 1.5)}") # Use both.
8
9  print("We now use Liu Hui's Method to Approximate \u03c0.")
10 e: int = 6 # the number of edges: We start with a hexagon, i.e., e=6.
11 s: float = 1.0 # the side length: Initially 1, i.e., radius is also 1.
12 for _ in range(6):
13     print(f"{e} edges, side length={s} give us \u03c0\u2248{e * s / 2}.")
14     e *= 2 # We double the number of edges...
15     s = sqrt(2 - sqrt(4 - (s ** 2))) # ...and recompute the side length
```

↓ python3 use_my_math.py ↓

```
1  6!=720
2  √3=1.7320508075688772
3  √(6!*1.5)=32.863353450309965
4  We now use Liu Hui's Method to Approximate π.
5  6 edges, side length=1.0 give us π≈3.0.
6  12 edges, side length=0.5176380902050417 give us π≈3.10582854123025.
7  24 edges, side length=0.2610523844401035 give us π≈3.1326286132812418.
8  48 edges, side length=0.13080625846028637 give us π≈3.139350203046873.
9  96 edges, side length=0.0654381656435527 give us π≈3.14103195089053.
10 192 edges, side length=0.03272346325297234 give us π≈3.1414524722853443.
```


Unser Eigenes Modul

- Wir könnten Verzeichnis `utils` haben und das Verzeichnis `math_pack` mit unserer Datei `my_math.py` dort hinein tun.
- Dann würden wir unsere Funktionen so importieren: `from utils.math_pack.my_math import`....
- Die Paket- und Modul-Namen werden beim Importieren mit Punkten (`.`) getrennt.

```
1  """Using the mathematics module."""
2
3  from my_math import factorial, sqrt # Import our two functions.
4
5  print(f"6!={factorial(6)}") # Use the `factorial` function.
6  print(f"\u221A3={sqrt(3.0)}") # Use the `sqrt` function.
7  print(f"\u221A(6!*1.5)={sqrt(factorial(6) * 1.5)}") # Use both.
8
9  print("We now use Liu Hui's Method to Approximate \u03c0.")
10 e: int = 6 # the number of edges: We start with a hexagon, i.e., e=6.
11 s: float = 1.0 # the side length: Initially 1, i.e., radius is also 1.
12 for _ in range(6):
13     print(f"{e} edges, side length={s} give us \u03c0\u2248{e * s / 2}.")
14     e *= 2 # We double the number of edges...
15     s = sqrt(2 - sqrt(4 - (s ** 2))) # ...and recompute the side length
```

↓ python3 use_my_math.py ↓

```
1  6!=720
2  √3=1.7320508075688772
3  √(6!*1.5)=32.863353450309965
4  We now use Liu Hui's Method to Approximate π.
5  6 edges, side length=1.0 give us π≈3.0.
6  12 edges, side length=0.5176380902050417 give us π≈3.10582854123025.
7  24 edges, side length=0.2610523844401035 give us π≈3.1326286132812418.
8  48 edges, side length=0.13080625846028637 give us π≈3.139350203046873.
9  96 edges, side length=0.0654381656435527 give us π≈3.14103195089053.
10 192 edges, side length=0.03272346325297234 give us π≈3.1414524722853443.
```

Unser Eigenes Modul

- Dann würden wir unsere Funktionen so importieren: `from utils.math_pack.my_math import`....
- Die Paket- und Modul-Namen werden beim Importieren mit Punkten (.) getrennt.
- So können wir unser Projekt in Module und Pakete für verschiedene Aufgaben strukturieren.

```
1  """Using the mathematics module."""
2
3  from my_math import factorial, sqrt # Import our two functions.
4
5  print(f"6!={factorial(6)}") # Use the `factorial` function.
6  print(f"\u221A3={sqrt(3.0)}") # Use the `sqrt` function.
7  print(f"\u221A(6!*1.5)={sqrt(factorial(6) * 1.5)}") # Use both.
8
9  print("We now use Liu Hui's Method to Approximate \u03c0.")
10 e: int = 6 # the number of edges: We start with a hexagon, i.e., e=6.
11 s: float = 1.0 # the side length: Initially 1, i.e., radius is also 1.
12 for _ in range(6):
13     print(f"{e} edges, side length={s} give us \u03c0\u2248{e * s / 2}.")
14     e *= 2 # We double the number of edges...
15     s = sqrt(2 - sqrt(4 - (s ** 2))) # ...and recompute the side length
```

↓ python3 use_my_math.py ↓

```
1  6!=720
2  √3=1.7320508075688772
3  √(6!*1.5)=32.863353450309965
4  We now use Liu Hui's Method to Approximate π.
5  6 edges, side length=1.0 give us π≈3.0.
6  12 edges, side length=0.5176380902050417 give us π≈3.10582854123025.
7  24 edges, side length=0.2610523844401035 give us π≈3.1326286132812418.
8  48 edges, side length=0.13080625846028637 give us π≈3.139350203046873.
9  96 edges, side length=0.0654381656435527 give us π≈3.14103195089053.
10 192 edges, side length=0.03272346325297234 give us π≈3.1414524722853443.
```

Unser Eigenes Modul

- Dann würden wir unsere Funktionen so importieren: `from utils.math_pack.my_math import`....
- Die Paket- und Modul-Namen werden beim Importieren mit Punkten (.) getrennt.
- So können wir unser Projekt in Module und Pakete für verschiedene Aufgaben strukturieren.
- In `use_my_math.py` können wir nun `sqrt` und `factorial` genauso verwenden, als ob wir sie dort drin definiert hätten.

```
1  """Using the mathematics module."""
2
3  from my_math import factorial, sqrt # Import our two functions.
4
5  print(f"6!={factorial(6)}") # Use the `factorial` function.
6  print(f"\u221A3={sqrt(3.0)}") # Use the `sqrt` function.
7  print(f"\u221A(6!*1.5)={sqrt(factorial(6) * 1.5)}") # Use both.
8
9  print("We now use Liu Hui's Method to Approximate \u03c0.")
10 e: int = 6 # the number of edges: We start with a hexagon, i.e., e=6.
11 s: float = 1.0 # the side length: Initially 1, i.e., radius is also 1.
12 for _ in range(6):
13     print(f"{e} edges, side length={s} give us \u03c0\u2248{e * s / 2}.")
14     e *= 2 # We double the number of edges...
15     s = sqrt(2 - sqrt(4 - (s ** 2))) # ...and recompute the side length
```

↓ python3 use_my_math.py ↓

```
1  6!=720
2  √3=1.7320508075688772
3  √(6!*1.5)=32.863353450309965
4  We now use Liu Hui's Method to Approximate π.
5  6 edges, side length=1.0 give us π≈3.0.
6  12 edges, side length=0.5176380902050417 give us π≈3.10582854123025.
7  24 edges, side length=0.2610523844401035 give us π≈3.1326286132812418.
8  48 edges, side length=0.13080625846028637 give us π≈3.139350203046873.
9  96 edges, side length=0.0654381656435527 give us π≈3.14103195089053.
10 192 edges, side length=0.03272346325297234 give us π≈3.1414524722853443.
```

Unser Eigenes Modul

- Die Paket- und Modul-Namen werden beim Importieren mit Punkten (.) getrennt.
- So können wir unser Projekt in Module und Pakete für verschiedene Aufgaben strukturieren.
- In `use_my_math.py` können wir nun `sqrt` und `factorial` genauso verwenden, als ob wir sie dort drin definiert hätten.
- Wir drucken erstmal ein paar Ergebnisse von `sqrt` und `factorial`.

```
1  """Using the mathematics module."""
2
3  from my_math import factorial, sqrt # Import our two functions.
4
5  print(f"6!={factorial(6)}") # Use the `factorial` function.
6  print(f"\u221A3={sqrt(3.0)}") # Use the `sqrt` function.
7  print(f"\u221A(6!*1.5)={sqrt(factorial(6) * 1.5)}") # Use both.
8
9  print("We now use Liu Hui's Method to Approximate \u03c0.")
10 e: int = 6 # the number of edges: We start with a hexagon, i.e., e=6.
11 s: float = 1.0 # the side length: Initially 1, i.e., radius is also 1.
12 for _ in range(6):
13     print(f"{e} edges, side length={s} give us \u03c0\u2248{e * s / 2}.")
14     e *= 2 # We double the number of edges...
15     s = sqrt(2 - sqrt(4 - (s ** 2))) # ...and recompute the side length
```

↓ python3 use_my_math.py ↓

```
1  6!=720
2  √3=1.7320508075688772
3  √(6!*1.5)=32.863353450309965
4  We now use Liu Hui's Method to Approximate π.
5  6 edges, side length=1.0 give us π≈3.0.
6  12 edges, side length=0.5176380902050417 give us π≈3.10582854123025.
7  24 edges, side length=0.2610523844401035 give us π≈3.1326286132812418.
8  48 edges, side length=0.13080625846028637 give us π≈3.139350203046873.
9  96 edges, side length=0.0654381656435527 give us π≈3.14103195089053.
10 192 edges, side length=0.03272346325297234 give us π≈3.1414524722853443.
```


Unser Eigenes Modul

- So können wir unser Projekt in Module und Pakete für verschiedene Aufgaben strukturieren.
- In `use_my_math.py` können wir nun `sqrt` und `factorial` genauso verwenden, als ob wir sie dort drin definiert hätten.
- Wir drucken erstmal ein paar Ergebnisse von `sqrt` und `factorial`.
- Wir kopieren auch etwas Kode von einem alten Beispiel, wo wir LIU Hui (刘徽) seine Methode zum Annähern von π verwendet haben.

```
1  """Using the mathematics module."""
2
3  from my_math import factorial, sqrt # Import our two functions.
4
5  print(f"6!={factorial(6)}") # Use the `factorial` function.
6  print(f"\u221A3={sqrt(3.0)}") # Use the `sqrt` function.
7  print(f"\u221A(6!*1.5)={sqrt(factorial(6) * 1.5)}") # Use both.
8
9  print("We now use Liu Hui's Method to Approximate \u03c0.")
10 e: int = 6 # the number of edges: We start with a hexagon, i.e., e=6.
11 s: float = 1.0 # the side length: Initially 1, i.e., radius is also 1.
12 for _ in range(6):
13     print(f"{e} edges, side length={s} give us \u03c0\u2248{e * s / 2}.")
14     e *= 2 # We double the number of edges...
15     s = sqrt(2 - sqrt(4 - (s ** 2))) # ...and recompute the side length
```

↓ python3 use_my_math.py ↓

```
1  6!=720
2  √3=1.7320508075688772
3  √(6!*1.5)=32.863353450309965
4  We now use Liu Hui's Method to Approximate π.
5  6 edges, side length=1.0 give us π≈3.0.
6  12 edges, side length=0.5176380902050417 give us π≈3.10582854123025.
7  24 edges, side length=0.2610523844401035 give us π≈3.1326286132812418.
8  48 edges, side length=0.13080625846028637 give us π≈3.139350203046873.
9  96 edges, side length=0.0654381656435527 give us π≈3.14103195089053.
10 192 edges, side length=0.03272346325297234 give us π≈3.1414524722853443.
```

Unser Eigenes Modul

- In `use_my_math.py` können wir nun `sqrt` und `factorial` genauso verwenden, als ob wir sie dort drin definiert hätten.
- Wir drucken erstmal ein paar Ergebnisse von `sqrt` und `factorial`.
- Wir kopieren auch etwas Kode von einem alten Beispiel, wo wir LIU Hui (刘徽) seine Methode zum Annähern von π verwendet haben.
- Diesmal nehmen wir unsere eigene Methode zum Berechnen der Quadratwurzel und nicht die Funktion aus dem `math`-Modul.

```
1  """Using the mathematics module."""
2
3  from my_math import factorial, sqrt # Import our two functions.
4
5  print(f"6!={factorial(6)}") # Use the `factorial` function.
6  print(f"\u221A3={sqrt(3.0)}") # Use the `sqrt` function.
7  print(f"\u221A(6!*1.5)={sqrt(factorial(6) * 1.5)}") # Use both.
8
9  print("We now use Liu Hui's Method to Approximate \u03c0.")
10 e: int = 6 # the number of edges: We start with a hexagon, i.e., e=6.
11 s: float = 1.0 # the side length: Initially 1, i.e., radius is also 1.
12 for _ in range(6):
13     print(f"{e} edges, side length={s} give us \u03c0\u2248{e * s / 2}.")
14     e *= 2 # We double the number of edges...
15     s = sqrt(2 - sqrt(4 - (s ** 2))) # ...and recompute the side length
```

↓ python3 use_my_math.py ↓

```
1  6!=720
2  √3=1.7320508075688772
3  √(6!*1.5)=32.863353450309965
4  We now use Liu Hui's Method to Approximate π.
5  6 edges, side length=1.0 give us π≈3.0.
6  12 edges, side length=0.5176380902050417 give us π≈3.10582854123025.
7  24 edges, side length=0.2610523844401035 give us π≈3.1326286132812418.
8  48 edges, side length=0.13080625846028637 give us π≈3.139350203046873.
9  96 edges, side length=0.0654381656435527 give us π≈3.14103195089053.
10 192 edges, side length=0.03272346325297234 give us π≈3.1414524722853443.
```

Unser Eigenes Modul

- Wir drucken erstmal ein paar Ergebnisse von `sqrt` und `factorial`.
- Wir kopieren auch etwas Kode von einem alten Beispiel, wo wir LIU Hui (刘徽) seine Methode zum Annähern von π verwendet haben.
- Diesmal nehmen wir unsere eigene Methode zum Berechnen der Quadratwurzel und nicht die Funktion aus dem `math`-Modul.
- Interessanterweise haben die letzten beiden Annäherungsschritte die selben Ergebnisse wie damals.

```
1  """Using the mathematics module."""
2
3  from my_math import factorial, sqrt # Import our two functions.
4
5  print(f"6!={factorial(6)}") # Use the `factorial` function.
6  print(f"\u221A3={sqrt(3.0)}") # Use the `sqrt` function.
7  print(f"\u221A(6!*1.5)={sqrt(factorial(6) * 1.5)}") # Use both.
8
9  print("We now use Liu Hui's Method to Approximate \u03c0.")
10 e: int = 6 # the number of edges: We start with a hexagon, i.e., e=6.
11 s: float = 1.0 # the side length: Initially 1, i.e., radius is also 1.
12 for _ in range(6):
13     print(f"{e} edges, side length={s} give us \u03c0\u2248{e * s / 2}.")
14     e *= 2 # We double the number of edges...
15     s = sqrt(2 - sqrt(4 - (s ** 2))) # ...and recompute the side length
```

↓ python3 use_my_math.py ↓

```
1  6!=720
2  √3=1.7320508075688772
3  √(6!*1.5)=32.863353450309965
4  We now use Liu Hui's Method to Approximate π.
5  6 edges, side length=1.0 give us π≈3.0.
6  12 edges, side length=0.5176380902050417 give us π≈3.10582854123025.
7  24 edges, side length=0.2610523844401035 give us π≈3.1326286132812418.
8  48 edges, side length=0.13080625846028637 give us π≈3.139350203046873.
9  96 edges, side length=0.0654381656435527 give us π≈3.14103195089053.
10 192 edges, side length=0.03272346325297234 give us π≈3.1414524722853443.
```



Zusammenfassung



Zusammenfassung



- Seit wir `while`-Schleifen benutzen können, waren wir in der Lage, im Grunde jede durch einen Computer berechenbare Funktion zu implementieren.

Zusammenfassung



- Seit wir `while`-Schleifen benutzen können, waren wir in der Lage, im Grunde jede durch einen Computer berechenbare Funktion zu implementieren.
- Wir waren aber auf der technischen Ebene eingeschränkt.

Zusammenfassung



- Seit wir `while`-Schleifen benutzen können, waren wir in der Lage, im Grunde jede durch einen Computer berechenbare Funktion zu implementieren.
- Wir waren aber auf der technischen Ebene eingeschränkt.
- Wir könnten zwar, theoretisch, beliebig komplexe Programme implementieren.

Zusammenfassung



- Seit wir `while`-Schleifen benutzen können, waren wir in der Lage, im Grunde jede durch einen Computer berechenbare Funktion zu implementieren.
- Wir waren aber auf der technischen Ebene eingeschränkt.
- Wir könnten zwar, theoretisch, beliebig komplexe Programme implementieren.
- Ohne Funktionen hätte das aber sicherlich zu sich oftmals wiederholenden Code geführt.

Zusammenfassung



- Seit wir `while`-Schleifen benutzen können, waren wir in der Lage, im Grunde jede durch einen Computer berechenbare Funktion zu implementieren.
- Wir waren aber auf der technischen Ebene eingeschränkt.
- Wir könnten zwar, theoretisch, beliebig komplexe Programme implementieren.
- Ohne Funktionen hätte das aber sicherlich zu sich oftmals wiederholenden Code geführt.
- Und ohne Module hätten wir irgendwann eine riesige, nicht wartbare und unleserliche Datei gehabt.

Zusammenfassung



- Seit wir `while`-Schleifen benutzen können, waren wir in der Lage, im Grunde jede durch einen Computer berechenbare Funktion zu implementieren.
- Wir waren aber auf der technischen Ebene eingeschränkt.
- Wir könnten zwar, theoretisch, beliebig komplexe Programme implementieren.
- Ohne Funktionen hätte das aber sicherlich zu sich oftmals wiederholenden Code geführt.
- Und ohne Module hätten wir irgendwann eine riesige, nicht wartbare und unleserliche Datei gehabt.
- Diese Beschränkungen haben wir jetzt durchbrochen.

Zusammenfassung



- Seit wir `while`-Schleifen benutzen können, waren wir in der Lage, im Grunde jede durch einen Computer berechenbare Funktion zu implementieren.
- Wir waren aber auf der technischen Ebene eingeschränkt.
- Wir könnten zwar, theoretisch, beliebig komplexe Programme implementieren.
- Ohne Funktionen hätte das aber sicherlich zu sich oftmals wiederholenden Code geführt.
- Und ohne Module hätten wir irgendwann eine riesige, nicht wartbare und unleserliche Datei gehabt.
- Diese Beschränkungen haben wir jetzt durchbrochen.
- Ab jetzt können wir beliebig große und beliebig komplexe Applikationen mit Python programmieren.

Zusammenfassung



- Seit wir `while`-Schleifen benutzen können, waren wir in der Lage, im Grunde jede durch einen Computer berechenbare Funktion zu implementieren.
- Wir waren aber auf der technischen Ebene eingeschränkt.
- Wir könnten zwar, theoretisch, beliebig komplexe Programme implementieren.
- Ohne Funktionen hätte das aber sicherlich zu sich oftmals wiederholenden Code geführt.
- Und ohne Module hätten wir irgendwann eine riesige, nicht wartbare und unleserliche Datei gehabt.
- Diese Beschränkungen haben wir jetzt durchbrochen.
- Ab jetzt können wir beliebig große und beliebig komplexe Applikationen mit Python programmieren.
- Wir können verschiedene Funktionen für verschiedene Teilaufgaben implementieren.

Zusammenfassung



- Seit wir `while`-Schleifen benutzen können, waren wir in der Lage, im Grunde jede durch einen Computer berechenbare Funktion zu implementieren.
- Wir waren aber auf der technischen Ebene eingeschränkt.
- Wir könnten zwar, theoretisch, beliebig komplexe Programme implementieren.
- Ohne Funktionen hätte das aber sicherlich zu sich oftmals wiederholenden Code geführt.
- Und ohne Module hätten wir irgendwann eine riesige, nicht wartbare und unleserliche Datei gehabt.
- Diese Beschränkungen haben wir jetzt durchbrochen.
- Ab jetzt können wir beliebig große und beliebig komplexe Applikationen mit Python programmieren.
- Wir können verschiedene Funktionen für verschiedene Teilaufgaben implementieren.
- Wir können Funktionen für ähnliche Themengebiete in verschiedene Module gruppieren.

Zusammenfassung



- Seit wir `while`-Schleifen benutzen können, waren wir in der Lage, im Grunde jede durch einen Computer berechenbare Funktion zu implementieren.
- Wir waren aber auf der technischen Ebene eingeschränkt.
- Wir könnten zwar, theoretisch, beliebig komplexe Programme implementieren.
- Ohne Funktionen hätte das aber sicherlich zu sich oftmals wiederholenden Code geführt.
- Und ohne Module hätten wir irgendwann eine riesige, nicht wartbare und unleserliche Datei gehabt.
- Diese Beschränkungen haben wir jetzt durchbrochen.
- Ab jetzt können wir beliebig große und beliebig komplexe Applikationen mit Python programmieren.
- Wir können verschiedene Funktionen für verschiedene Teilaufgaben implementieren.
- Wir können Funktionen für ähnliche Themengebiete in verschiedene Module gruppieren.
- Wir können Module sogar hierarchisch in Pakete gruppieren.

Zusammenfassung



- Seit wir `while`-Schleifen benutzen können, waren wir in der Lage, im Grunde jede durch einen Computer berechenbare Funktion zu implementieren.
- Wir waren aber auf der technischen Ebene eingeschränkt.
- Wir könnten zwar, theoretisch, beliebig komplexe Programme implementieren.
- Ohne Funktionen hätte das aber sicherlich zu sich oftmals wiederholenden Code geführt.
- Und ohne Module hätten wir irgendwann eine riesige, nicht wartbare und unleserliche Datei gehabt.
- Diese Beschränkungen haben wir jetzt durchbrochen.
- Ab jetzt können wir beliebig große und beliebig komplexe Applikationen mit Python programmieren.
- Wir können verschiedene Funktionen für verschiedene Teilaufgaben implementieren.
- Wir können Funktionen für ähnliche Themengebiete in verschiedene Module gruppieren.
- Wir können Module sogar hierarchisch in Pakete gruppieren.
- Damit können wir wohlstrukturierte und wartbare Software entwickeln.



谢谢你们！
Thank you!
Vielen Dank!



References I



- [1] Adam Aspin und Karine Aspin. *Query Answers with MariaDB – Volume I: Introduction to SQL Queries*. Tetras Publishing, Okt. 2018. ISBN: 978-1-9996172-4-0. See also² (siehe S. 77, 86).
- [2] Adam Aspin und Karine Aspin. *Query Answers with MariaDB – Volume II: In-Depth Querying*. Tetras Publishing, Okt. 2018. ISBN: 978-1-9996172-5-7. See also¹ (siehe S. 77, 86).
- [3] Daniel J. Barrett. *Efficient Linux at the Command Line*. Sebastopol, CA, USA: O'Reilly Media, Inc., Feb. 2022. ISBN: 978-1-0981-1340-7 (siehe S. 86, 87).
- [4] Daniel Bartholomew. *Learning the MariaDB Ecosystem: Enterprise-level Features for Scalability and Availability*. New York, NY, USA: Apress Media, LLC, Okt. 2019. ISBN: 978-1-4842-5514-8 (siehe S. 86).
- [5] Tim Berners-Lee. *Re: Qualifiers on Hypertext links...* Geneva, Switzerland: World Wide Web project, European Organization for Nuclear Research (CERN) und Newsgroups: alt.hypertext, 6. Aug. 1991. URL: <https://www.w3.org/People/Berners-Lee/1991/08/art-6484.txt> (besucht am 2025-02-05) (siehe S. 88).
- [6] Alex Berson. *Client/Server Architecture*. 2. Aufl. Computer Communications Series. New York, NY, USA: McGraw-Hill, 29. März 1996. ISBN: 978-0-07-005664-0 (siehe S. 85).
- [7] Ethan Bommarito und Michael Bommarito. *An Empirical Analysis of the Python Package Index (PyPI)*. arXiv.org: Computing Research Repository (CoRR) abs/1907.11073. Ithaca, NY, USA: Cornell University Library, 26. Juli 2019. doi:10.48550/arXiv.1907.11073. URL: <https://arxiv.org/abs/1907.11073> (besucht am 2024-08-17). arXiv:1907.11073v2 [cs.SE] 26 Jul 2019 (siehe S. 86).
- [8] Silvia Botros und Jeremy Tinley. *High Performance MySQL*. 4. Aufl. Sebastopol, CA, USA: O'Reilly Media, Inc., Nov. 2021. ISBN: 978-1-4920-8051-0 (siehe S. 86).
- [9] Ed Bott. *Windows 11 Inside Out*. Hoboken, NJ, USA: Microsoft Press, Pearson Education, Inc., Feb. 2023. ISBN: 978-0-13-769132-6 (siehe S. 86).
- [10] Ron Brash und Ganesh Naik. *Bash Cookbook*. Birmingham, England, UK: Packt Publishing Ltd, Juli 2018. ISBN: 978-1-78862-936-2 (siehe S. 85).
- [11] Jason Cannon. *High Availability for the LAMP Stack*. Shelter Island, NY, USA: Manning Publications, Juni 2022 (siehe S. 86, 87).

References II



- [12] Donald D. Chamberlin. "50 Years of Queries". *Communications of the ACM (CACM)* 67(8):110–121, Aug. 2024. New York, NY, USA: Association for Computing Machinery (ACM). ISSN: 0001-0782. doi:10.1145/3649887. URL: <https://cacm.acm.org/research/50-years-of-queries> (besucht am 2025-01-09) (siehe S. 87).
- [13] David Clinton und Christopher Negus. *Ubuntu Linux Bible*. 10. Aufl. Bible Series. Chichester, West Sussex, England, UK: John Wiley and Sons Ltd., 10. Nov. 2020. ISBN: 978-1-119-72233-5 (siehe S. 87).
- [14] Edgar Frank „Ted“ Codd. "A Relational Model of Data for Large Shared Data Banks". *Communications of the ACM (CACM)* 13(6):377–387, Juni 1970. New York, NY, USA: Association for Computing Machinery (ACM). ISSN: 0001-0782. doi:10.1145/362384.362685. URL: <https://www.seas.upenn.edu/~zives/03f/cis550/codd.pdf> (besucht am 2025-01-05) (siehe S. 87).
- [15] *Database Language SQL*. Techn. Ber. ANSI X3.135-1986. Washington, D.C., USA: American National Standards Institute (ANSI), 1986 (siehe S. 87).
- [16] Matt David und Blake Barnhill. *How to Teach People SQL*. San Francisco, CA, USA: The Data School, Chart.io, Inc., 10. Dez. 2019–10. Apr. 2023. URL: <https://dataschool.com/how-to-teach-people-sql> (besucht am 2025-02-27) (siehe S. 87).
- [17] *Database Language SQL*. International Standard ISO 9075-1987. Geneva, Switzerland: International Organization for Standardization (ISO), 1987 (siehe S. 87).
- [18] Paul Deitel, Harvey Deitel und Abbey Deitel. *Internet & World Wide WebW[?]: How to Program*. 5. Aufl. Hoboken, NJ, USA: Pearson Education, Inc., Nov. 2011. ISBN: 978-0-13-299045-5 (siehe S. 88).
- [19] Russell J.T. Dyer. *Learning MySQL and MariaDB*. Sebastopol, CA, USA: O'Reilly Media, Inc., März 2015. ISBN: 978-1-4493-6290-4 (siehe S. 86).
- [20] Leonhard Euler. "An Essay on Continued Fractions". Übers. von Myra F. Wyman und Bostwick F. Wyman. *Mathematical Systems Theory* 18(1):295–328, Dez. 1985. New York, NY, USA: Springer Science+Business Media, LLC. ISSN: 1432-4350. doi:10.1007/BF01699475. URL: <https://www.researchgate.net/publication/301720080> (besucht am 2024-09-24). Translation of²¹. (Siehe S. 79).

References III



- [21] Leonhard Euler. "De Fractionibus Continuis Dissertation". *Commentarii Academiae Scientiarum Petropolitanae* 9:98–137, 1737–1744. Petropolis (St. Petersburg), Russia: Typis Academiae. URL: <https://scholarlycommons.pacific.edu/cgi/viewcontent.cgi?article=1070> (besucht am 2024-09-24). See²⁰ for a translation. (Siehe S. 78, 88).
- [22] Luca Ferrari und Enrico Pirozzi. *Learn PostgreSQL*. 2. Aufl. Birmingham, England, UK: Packt Publishing Ltd, Okt. 2023. ISBN: 978-1-83763-564-1 (siehe S. 86).
- [23] Michael Filaseta. "The Transcendence of e and π ". In: *Math 785: Transcendental Number Theory*. Columbia, SC, USA: University of South Carolina, Frühling 2011. Kap. 6. URL: <https://people.math.sc.edu/filaseta/gradcourses/Math785/Math785Notes6.pdf> (besucht am 2024-07-05) (siehe S. 88).
- [24] 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. 85).
- [25] Terry Halpin und Tony Morgan. *Information Modeling and Relational Databases*. 3. Aufl. Burlington, MA, USA/San Mateo, CA, USA: Morgan Kaufmann Publishers, Juli 2024. ISBN: 978-0-443-23791-1 (siehe S. 87).
- [26] Jan L. Harrington. *Relational Database Design and Implementation*. 4. Aufl. Burlington, MA, USA/San Mateo, CA, USA: Morgan Kaufmann Publishers, Apr. 2016. ISBN: 978-0-12-849902-3 (siehe S. 87).
- [27] Michael Hausenblas. *Learning Modern Linux*. Sebastopol, CA, USA: O'Reilly Media, Inc., Apr. 2022. ISBN: 978-1-0981-0894-6 (siehe S. 86).
- [28] Matthew Helmke. *Ubuntu Linux Unleashed 2021 Edition*. 14. Aufl. Reading, MA, USA: Addison-Wesley Professional, Aug. 2020. ISBN: 978-0-13-668539-5 (siehe S. 86, 87).
- [29] 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. 86).

References IV



- [30] *Information Technology – Database Languages – SQL – Part 1: Framework (SQL/Framework), Part 1*. International Standard ISO/IEC 9075-1:2023(E), Sixth Edition, (ANSI X3.135). Geneva, Switzerland: International Organization for Standardization (ISO) und International Electrotechnical Commission (IEC), Juni 2023. URL: [https://standards.iso.org/ittf/PubliclyAvailableStandards/ISO_IEC_9075-1_2023_ed_6_-_id_76583_Publication_PDF_\(en\).zip](https://standards.iso.org/ittf/PubliclyAvailableStandards/ISO_IEC_9075-1_2023_ed_6_-_id_76583_Publication_PDF_(en).zip) (besucht am 2025-01-08). Consists of several parts, see <https://modern-sql.com/standard> for information where to obtain them. (Siehe S. 87).
- [31] *Python 3 Documentation. Installing Python Modules*. Beaverton, OR, USA: Python Software Foundation (PSF), 2001–2025. URL: <https://docs.python.org/3/installing> (besucht am 2024-08-17) (siehe S. 31–54, 86).
- [32] Arthur Jones, Kenneth R. Pearson und Sidney A. Morris. "Transcendence of e and π ". In: *Abstract Algebra and Famous Impossibilities*. Universitext (UTX). New York, NY, USA: Springer New York, 1991. Kap. 9, S. 115–161. ISSN: 0172-5939. ISBN: 978-1-4419-8552-1. doi:10.1007/978-1-4419-8552-1_8 (siehe S. 88).
- [33] Jay LaCroix. *Mastering Ubuntu Server*. 4. Aufl. Birmingham, England, UK: Packt Publishing Ltd, Sep. 2022. ISBN: 978-1-80323-424-3 (siehe S. 87).
- [34] 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. 87).
- [35] 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. 86).
- [36] 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. 86).
- [37] Gloria Lotha, Aakanksha Gaur, Erik Gregersen, Swati Chopra und William L. Hosch. "Client-Server Architecture". In: *Encyclopaedia Britannica*. Hrsg. von The Editors of Encyclopaedia Britannica. Chicago, IL, USA: Encyclopædia Britannica, Inc., 3. Jan. 2025. URL: <https://www.britannica.com/technology/client-server-architecture> (besucht am 2025-01-20) (siehe S. 85).
- [38] Mark Lutz. *Learning Python*. 6. Aufl. Sebastopol, CA, USA: O'Reilly Media, Inc., März 2025. ISBN: 978-1-0981-7130-8 (siehe S. 86).

References V



- [39] *MariaDB Server Documentation*. Milpitas, CA, USA: MariaDB, 2025. URL: <https://mariadb.com/kb/en/documentation> (besucht am 2025-04-24) (siehe S. 86).
- [40] "Mathematical Functions and Operators". In: *PostgreSQL Documentation*. 17.4. The PostgreSQL Global Development Group (PGDG), 20. Feb. 2025. Kap. 9.3. URL: <https://www.postgresql.org/docs/17/functions-math.html> (besucht am 2025-02-27) (siehe S. 88).
- [41] Jim Melton und Alan R. Simon. *SQL: 1999 – Understanding Relational Language Components*. The Morgan Kaufmann Series in Data Management Systems. Burlington, MA, USA/San Mateo, CA, USA: Morgan Kaufmann Publishers, Juni 2001. ISBN: 978-1-55860-456-8 (siehe S. 87).
- [42] Cameron Newham und Bill Rosenblatt. *Learning the Bash Shell – Unix Shell Programming: Covers Bash 3.0*. 3. Aufl. Sebastopol, CA, USA: O'Reilly Media, Inc., 2005. ISBN: 978-0-596-00965-6 (siehe S. 85).
- [43] Ivan Niven. "The Transcendence of π ". *The American Mathematical Monthly* 46(8):469–471, Okt. 1939. London, England, UK: Taylor and Francis Ltd. ISSN: 1930-0972. doi:10.2307/2302515 (siehe S. 88).
- [44] Regina O. Obe und Leo S. Hsu. *PostgreSQL: Up and Running*. 3. Aufl. Sebastopol, CA, USA: O'Reilly Media, Inc., Okt. 2017. ISBN: 978-1-4919-6336-4 (siehe S. 86).
- [45] Robert Orfali, Dan Harkey und Jeri Edwards. *Client/Server Survival Guide*. 3. Aufl. Chichester, West Sussex, England, UK: John Wiley and Sons Ltd., 25. Jan. 1999. ISBN: 978-0-471-31615-2 (siehe S. 85).
- [46] 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 Kononov, 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. 85).
- [47] pip Developers. *pip Documentation v24.3.1*. Beaverton, OR, USA: Python Software Foundation (PSF), 27. Okt. 2024. URL: <https://pip.pypa.io> (besucht am 2024-12-25) (siehe S. 86).
- [48] *PostgreSQL Documentation*. 17.4. The PostgreSQL Global Development Group (PGDG), Feb. 2025. URL: <https://www.postgresql.org/docs/17/index.html> (besucht am 2025-02-25).

References VI



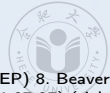
- [49] *PostgreSQL Essentials: Leveling Up Your Data Work*. Sebastopol, CA, USA: O'Reilly Media, Inc., März 2024 (siehe S. 86).
- [50] Abhishek Ratan, Eric Chou, Pradeeban Kathiravelu und Dr. M.O. Faruque Sarker. *Python Network Programming*. Birmingham, England, UK: Packt Publishing Ltd, Jan. 2019. ISBN: 978-1-78883-546-6 (siehe S. 85).
- [51] Federico Razzoli. *Mastering MariaDB*. Birmingham, England, UK: Packt Publishing Ltd, Sep. 2014. ISBN: 978-1-78398-154-0 (siehe S. 86).
- [52] Mike Reichardt, Michael Gundall und Hans D. Schotten. "Benchmarking the Operation Times of NoSQL and MySQL Databases for Python Clients". In: *47th Annual Conference of the IEEE Industrial Electronics Society (IECON'2021)*. 13.–15. Okt. 2021, Toronto, ON, Canada. Piscataway, NJ, USA: Institute of Electrical and Electronics Engineers (IEEE), 2021, S. 1–8. ISSN: 2577-1647. ISBN: 978-1-6654-3554-3. doi:10.1109/IECON48115.2021.9589382 (siehe S. 86).
- [53] Mark Richards und Neal Ford. *Fundamentals of Software Architecture: An Engineering Approach*. Sebastopol, CA, USA: O'Reilly Media, Inc., Jan. 2020. ISBN: 978-1-4920-4345-4 (siehe S. 85).
- [54] Ellen Siever, Stephen Figgins, Robert Love und Arnold Robbins. *Linux in a Nutshell*. 6. Aufl. Sebastopol, CA, USA: O'Reilly Media, Inc., Sep. 2009. ISBN: 978-0-596-15448-6 (siehe S. 86).
- [55] Anna Skoulikari. *Learning Git*. Sebastopol, CA, USA: O'Reilly Media, Inc., Mai 2023. ISBN: 978-1-0981-3391-7 (siehe S. 85).
- [56] John Miles Smith und Philip Yen-Tang Chang. "Optimizing the Performance of a Relational Algebra Database Interface". *Communications of the ACM (CACM)* 18(10):568–579, Okt. 1975. New York, NY, USA: Association for Computing Machinery (ACM). ISSN: 0001-0782. doi:10.1145/361020.361025 (siehe S. 87).
- [57] "SQL Commands". In: *PostgreSQL Documentation*. 17.4. The PostgreSQL Global Development Group (PGDG), 20. Feb. 2025. Kap. Part VI. Reference. URL: <https://www.postgresql.org/docs/17/sql-commands.html> (besucht am 2025-02-25) (siehe S. 87).
- [58] Ryan K. Stephens und Ronald R. Plew. *Sams Teach Yourself SQL in 21 Days*. 4. Aufl. Sams Tech Yourself. Indianapolis, IN, USA: SAMS Technical Publishing und Hoboken, NJ, USA: Pearson Education, Inc., Okt. 2002. ISBN: 978-0-672-32451-2 (siehe S. 83, 87).

References VII



- [59] Ryan K. Stephens, Ronald R. Plew, Bryan Morgan und Jeff Perkins. *SQL in 21 Tagen. Die Datenbank-Abfragesprache SQL vollständig erklärt (in 14/21 Tagen)*. 6. Aufl. Burgthann, Bayern, Germany: Markt+Technik Verlag GmbH, Feb. 1998. ISBN: 978-3-8272-2020-2. Translation of⁵⁸ (siehe S. 87).
- [60] Allen Taylor. *Introducing SQL and Relational Databases*. New York, NY, USA: Apress Media, LLC, Sep. 2018. ISBN: 978-1-4842-3841-7 (siehe S. 87).
- [61] Alkin Tezuysal und Ibrar Ahmed. *Database Design and Modeling with PostgreSQL and MySQL*. Birmingham, England, UK: Packt Publishing Ltd, Juli 2024. ISBN: 978-1-80323-347-5 (siehe S. 86).
- [62] "The Import System". In: *Python 3 Documentation. The Python Language Reference*. Beaverton, OR, USA: Python Software Foundation (PSF), 2001–2025. Kap. 5. URL: <https://docs.python.org/3/reference/import.html> (besucht am 2024-10-01) (siehe S. 16–23).
- [63] *The Python Package Index (PyPI)*. Beaverton, OR, USA: Python Software Foundation (PSF), 2024. URL: <https://pypi.org> (besucht am 2024-08-17) (siehe S. 86).
- [64] Linus Torvalds. "The Linux Edge". *Communications of the ACM (CACM)* 42(4):38–39, Apr. 1999. New York, NY, USA: Association for Computing Machinery (ACM). ISSN: 0001-0782. doi:10.1145/299157.299165 (siehe S. 86).
- [65] 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. 85, 88).
- [66] Marat Valiev, Bogdan Vasilescu und James D. Herbsleb. "Ecosystem-Level Determinants of Sustained Activity in Open-Source Projects: A Case Study of the PyPI Ecosystem". In: *ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/SIGSOFT FSE'2018)*. 4.–9. Nov. 2018, Lake Buena Vista, FL, USA. Hrsg. von Gary T. Leavens, Alessandro F. Garcia und Corina S. Păsăreanu. New York, NY, USA: Association for Computing Machinery (ACM), 2018, S. 644–655. ISBN: 978-1-4503-5573-5. doi:10.1145/3236024.3236062 (siehe S. 86).
- [67] 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. 87).

References VIII



- [68] 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. 30, 85).
- [69] Sander van Vugt. *Linux Fundamentals*. 2. Aufl. Hoboken, NJ, USA: Pearson IT Certification, Juni 2022. ISBN: 978-0-13-792931-3 (siehe S. 86).
- [70] Thomas Weise (汤卫思). *Databases*. 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), 2025. URL: <https://thomasweise.github.io/databases> (besucht am 2025-01-05) (siehe S. 85, 87).
- [71] 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. 86).
- [72] *What is a Relational Database?* Armonk, NY, USA: International Business Machines Corporation (IBM), 20. Okt. 2021–12. Dez. 2024. URL: <https://www.ibm.com/think/topics/relational-databases> (besucht am 2025-01-05) (siehe S. 87).
- [73] Ulf Michael „Monty“ Widenius, David Axmark und Uppsala, Sweden: MySQL AB. *MySQL Reference Manual – Documentation from the Source*. Sebastopol, CA, USA: O'Reilly Media, Inc., 9. Juli 2002. ISBN: 978-0-596-00265-7 (siehe S. 86).
- [74] Kinza Yasar und Craig S. Mullins. *Definition: Database Management System (DBMS)*. Newton, MA, USA: TechTarget, Inc., Juni 2024. URL: <https://www.techtarget.com/searchdatamanagement/definition/database-management-system> (besucht am 2025-01-11) (siehe S. 85).
- [75] Giorgio Zarrelli. *Mastering Bash*. Birmingham, England, UK: Packt Publishing Ltd, Juni 2017. ISBN: 978-1-78439-687-9 (siehe S. 85).

Glossary (in English) I



Bash is a the shell used under Ubuntu Linux, i.e., the program that „runs“ in the terminal and interprets your commands, allowing you to start and interact with other programs^{10,42,75}. Learn more at <https://www.gnu.org/software/bash>.

client In a client-server architecture, the client is a device or process that requests a service from the server. It initiates the communication with the server, sends a request, and receives the response with the result of the request. Typical examples for clients are web browsers in the internet as well as clients for database management systems (DBMSes), such as psql.

client-server architecture is a system design where a central server receives requests from one or multiple clients^{6,37,45,50,53}. These requests and responses are usually sent over network connections. A typical example for such a system is the World Wide Web (WWW), where web servers host websites and make them available to web browsers, the clients. Another typical example is the structure of database (DB) software, where a central server, the DBMS, offers access to the DB to the different clients. Here, the client can be some terminal software shipping with the DBMS, such as psql, or the different applications that access the DBs.

DB A *database* is an organized collection of structured information or data, typically stored electronically in a computer system. Databases are discussed in our book *Databases*⁷⁰.

DBMS A *database management system* is the software layer located between the user or application and the DB. The DBMS allows the user/application to create, read, write, update, delete, and otherwise manipulate the data in the DB⁷⁴.

docstring Docstrings are special string constants in Python that contain documentation for modules or functions²⁴. They must be delimited by `"""..."""`^{24,68}.

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^{55,65}. Learn more at <https://git-scm.com>.

GitHub is a website where software projects can be hosted and managed via the Git VCS^{46,65}. Learn more at <https://github.com>.

IT information technology

Glossary (in English) II



LAMP Stack A system setup for web applications: Linux, Apache (a web server), MySQL, and the server-side scripting language PHP^{11,28}.

Linux is the leading open source operating system, i.e., a free alternative for Microsoft Windows^{3,27,54,64,69}. We recommend using it for this course, for software development, and for research. Learn more at <https://www.linux.org>. Its variant Ubuntu is particularly easy to use and install.

MariaDB An open source relational database management system that has forked off from MySQL^{1,2,4,19,39,51}. See <https://mariadb.org> for more information.

Microsoft Windows is a commercial proprietary operating system⁹. It is widely spread, but we recommend using a Linux variant such as Ubuntu for software development and for our course. Learn more at <https://www.microsoft.com/windows>.

Mypy is a static type checking tool for Python³⁶ that makes use of type hints. Learn more at <https://github.com/python/mypy> and in⁷¹.

MySQL An open source relational database management system^{8,19,52,61,73}. MySQL is famous for its use in the LAMP Stack. See <https://www.mysql.com> for more information.

pip is the standard tool to install Python software packages from the PyPI repository^{31,47}. To install a package `thepackage` hosted on PyPI, type `pip install thepackage` into the terminal. Learn more at <https://packaging.python.org/installing>.

PostgreSQL An open source object-relational DBMS^{22,44,49,61}. See <https://postgresql.org> for more information.







psql is the client program used to access the PostgreSQL DBMS server.

PyPI The Python Package Index (PyPI) is an online repository that provides the software packages that you can install with `pip`^{7,63,66}. Learn more at <https://pypi.org>.

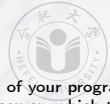
Python The Python programming language^{29,35,38,71}, i.e., what you will learn about in our book⁷¹. Learn more at <https://python.org>.

Glossary (in English) III



- relational database A relational DB is a database that organizes data into rows (tuples, records) and columns (attributes), which collectively form tables (relations) where the data points are related to each other^{14,25,26,56,60,70,72}.
- server In a client-server architecture, the server is a process that fulfills the requests of the clients. It usually waits for incoming communication carrying the requests from the clients. For each request, it takes the necessary actions, performs the required computations, and then sends a response with the result of the request. Typical examples for servers are web servers¹¹ in the internet as well as DBMSes. It is also common to refer to the computer running the server processes as server as well, i.e., to call it the „server computer“³³.
- SQL The *Structured Query Language* is basically a programming language for querying and manipulating relational databases^{12,15–17,30,41,57–60}. It is understood by many DBMSes. You find the Structured Query Language (SQL) commands supported by PostgreSQL in the reference⁵⁷.
- terminal A terminal is a text-based window where you can enter commands and execute them^{3,13}. Knowing what a terminal is and how to use it is very essential in any programming- or system administration-related task. If you want to open a terminal under Microsoft Windows, you can Druck auf  + , dann Schreiben von `cmd`, dann Druck auf . Under Ubuntu Linux,  +  +  opens a terminal, which then runs a Bash shell inside.
- 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^{34,67}. 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.
- Ubuntu is a variant of the open source operating system Linux^{13,28}. We recommend that you use this operating system to follow this class, for software development, and for research. Learn more at <https://ubuntu.com>. If you are in China, you can download it from <https://mirrors.ustc.edu.cn/ubuntu-releases>.

Glossary (in English) IV



VCS A *Version Control System* is a software which allows you to manage and preserve the historical development of your program code⁶⁵. 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.

WWW World Wide Web^{5,18}

π is the ratio of the circumference U of a circle and its diameter d , i.e., $\pi = U/d$. $\pi \in \mathbb{R}$ is an irrational and transcendental number^{23,32,43}, which is approximately $\pi \approx 3.141\,592\,653\,589\,793\,238\,462\,643$. In Python, it is provided by the `math` module as constant `pi` with value `3.141592653589793`. In PostgreSQL, it is provided by the SQL function `pi()` with value `3.141592653589793`⁴⁰.

e is Euler's number²¹, the base of the natural logarithm. $e \in \mathbb{R}$ is an irrational and transcendental number^{23,32}, which is approximately $e \approx 2.718\,281\,828\,459\,045\,235\,360$. In Python, it is provided by the `math` module as constant `e` with value `2.718281828459045`. In PostgreSQL, you can obtain it via the SQL function `exp(1)` as value `2.718281828459045`⁴⁰.

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