合肥大學
HEFEI UNIVERSITY

Programming with Python

39. Generator-Funktionen

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

Version: 2026-01-29

Programming with Python

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

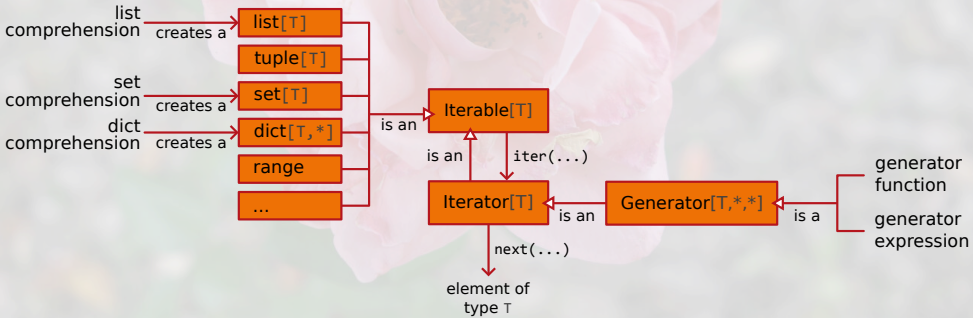
Die Webseite mit dem Lehrmaterial dieses Kurses ist https://thomasweise.github.io/programmingWithPython (siehe auch den QR-Kode unten rechts). Dort können Sie das Kursbuch (in Englisch) und diese Slides finden. Das Repository mit den Beispielprogrammen in Python finden Sie unter https://github.com/thomasWeise/programmingWithPythonCode.

Outline

Einleitung

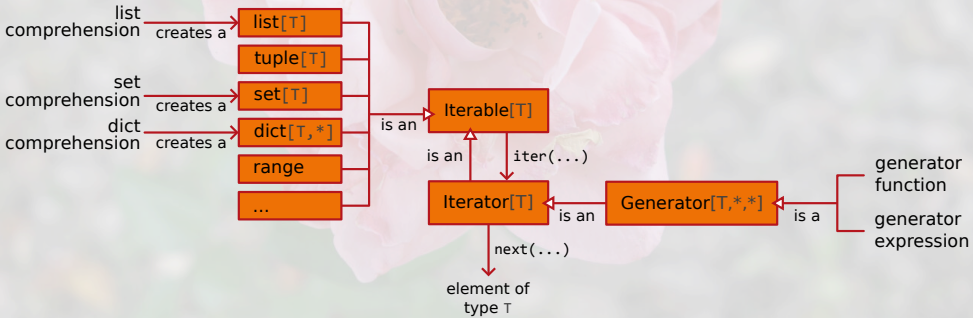
Einleitung

- Das letzte Element aus unserer Übersichtsgrafik, das wir noch nicht diskutiert haben, sind *Generator-Funktionen*[22].



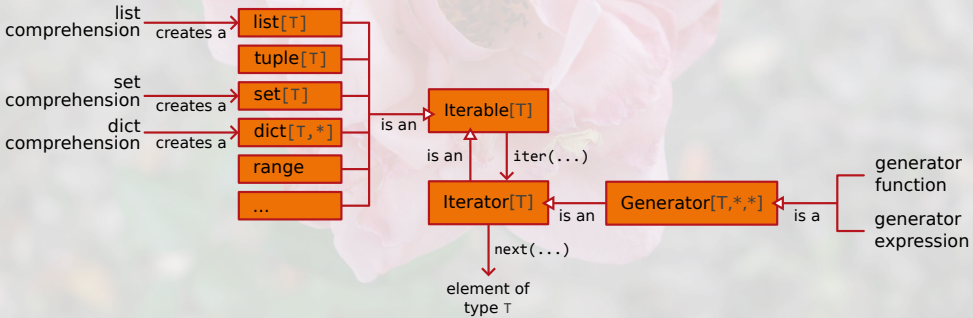
Einleitung

- Das letzte Element aus unserer Übersichtsgrafik, das wir noch nicht diskutiert haben, sind *Generator-Funktionen*[22].
- Aus der Sich eines Benutzers einer Generator-Funktion handelt es sich hierbei im Grunde um ein Funktion, die sich wie ein `Iterator` verhält.



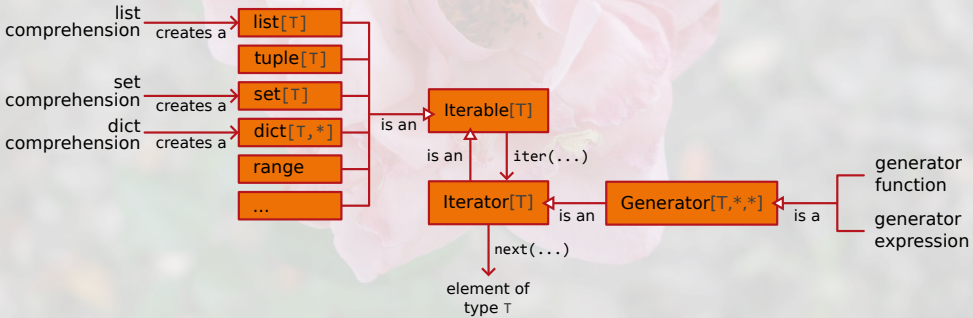
Einleitung

- Das letzte Element aus unserer Übersichtsgrafik, das wir noch nicht diskutiert haben, sind *Generator-Funktionen*[22].
- Aus der Sich eines Benutzers einer Generator-Funktion handelt es sich hierbei im Grunde um ein Funktion, die sich wie ein `Iterator` verhält.
- Wir können ganz normal eine Sequenz von Werten aus diesem `Iterator` herausholen.



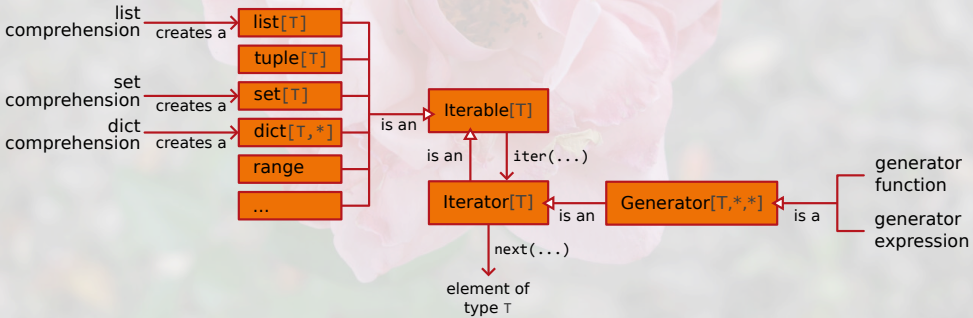
Einleitung

- Das letzte Element aus unserer Übersichtsgrafik, das wir noch nicht diskutiert haben, sind *Generator-Funktionen*[22].
- Aus der Sich eines Benutzers einer Generator-Funktion handelt es sich hierbei im Grunde um ein Funktion, die sich wie ein `Iterator` verhält.
- Wir können ganz normal eine Sequenz von Werten aus diesem `Iterator` herausholen.
- Wir können mit einer `for`-Schleife darüber iterieren.



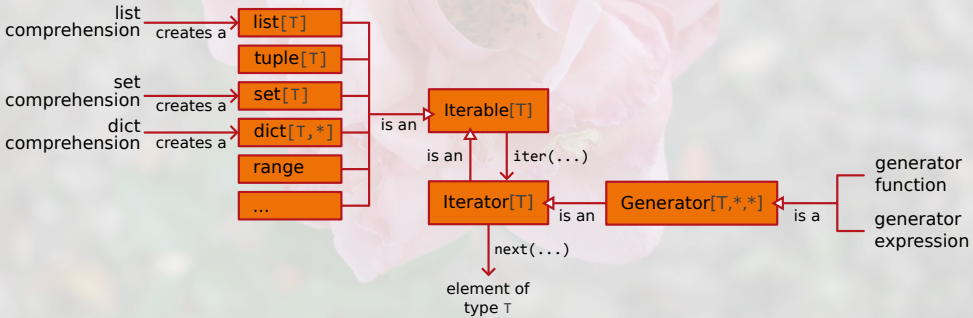
Einleitung

- Aus der Sich eines Benutzers einer Generator-Funktion handelt es sich hierbei im Grunde um ein Funktion, die sich wie ein `Iterator` verhält.
- Wir können ganz normal eine Sequenz von Werten aus diesem `Iterator` herausholen.
- Wir können mit einer `for`-Schleife darüber iterieren.
- Wir können ihn in einer Comprehension verwenden oder an den Konstruktor einer Kollektion übergeben.



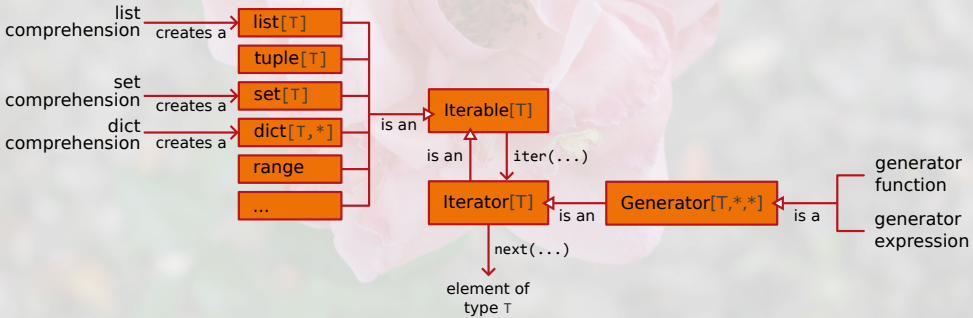
Einleitung

- Wir können ganz normal eine Sequenz von Werten aus diesem `Iterator` herausholen.
- Wir können mit einer `for`-Schleife darüber iterieren.
- Wir können ihn in einer Comprehension verwenden oder an den Konstruktor einer Kollektion übergeben.
- Aus Sicht des Implementierers dagegen sieht eine Generator-Funktion mehr wie eine Funktion aus, die mehrmals Werte zurückgeben kann.



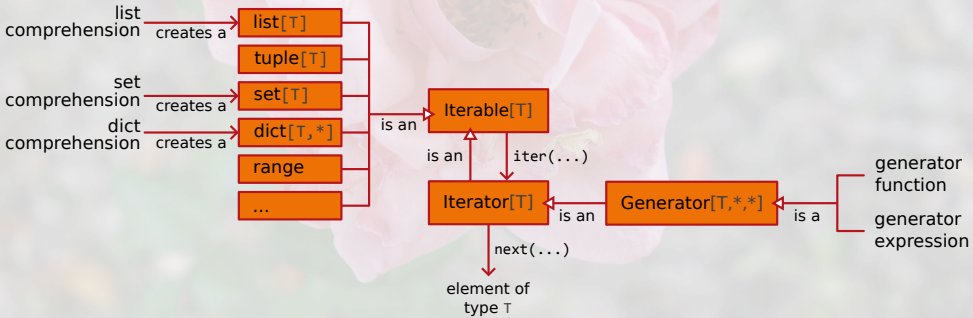
Einleitung

- Wir können mit einer `for`-Schleife darüber iterieren.
- Wir können ihn in einer Comprehension verwenden oder an den Konstruktor einer Kollektion übergeben.
- Aus Sicht des Implementierers dagegen sieht eine Generator-Funktion mehr wie eine Funktion aus, die mehrmals Werte zurückgeben kann.
- An Stelle des Schlüsselwortes `return` verwenden wir dazu das Schlüsselwort `yield`.



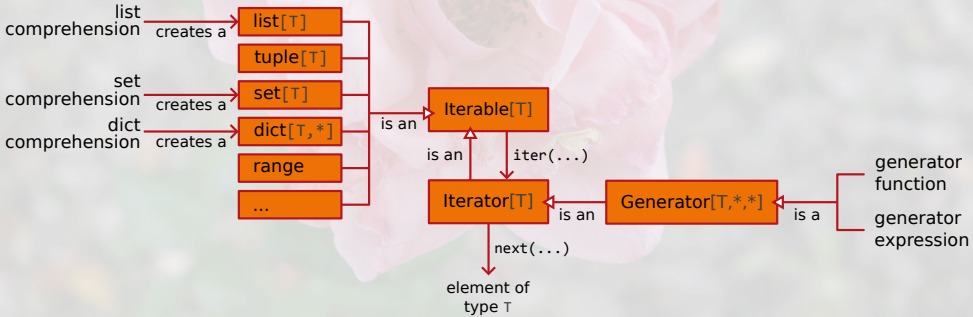
Einleitung

- Aus Sicht des Implementierers dagegen sieht eine Generator-Funktion mehr wie eine Funktion aus, die mehrmals Werte zurückgeben kann.
- An Stelle des Schlüsselwortes `return` verwenden wir dazu das Schlüsselwort `yield`.
- Jedes Element, dass die Sequenz die wir generieren zurückliefert, wird durch `yield` produziert.



Einleitung

- An Stelle des Schlüsselwortes `return` verwenden wir dazu das Schlüsselwort `yield`.
- Jedes Element, dass die Sequenz die wir generieren zurückliefert, wird durch `yield` produziert.
- Das fühlt sich an wie eine Funktion, die einen Wert zurückgibt, der dann von Kode „außen“ verarbeitet wird und dann wird die Funktion fortgesetzt und kann weitere Werte zurückliefern.



Beispiele

Einfaches Beispiel

- Das klingt erstmal sehr verwirrend.

Einfaches Beispiel

- Das klingt erstmal sehr verwirrend.
- Schauen wir uns einfaches Beispiel an.

```python
"""A simple example for generator functions."""

from typing import Generator  # The type hint for generators.


def generator_123() -> Generator[int, None, None]:
    """A generator function which yields 1, 2, 3."""
    yield 1  # The first time next(...) is called, the result is 1.
    yield 2  # The second time next(...) is called, the result is 2.
    yield 3  # The third time next(...) is called, the result is 3.


print(f"{list(generator_123()) = }")  # The list is [1, 2, 3].

gen: Generator[int, None, None] = generator_123()  # Use directly.
print(f"{next(gen) = }")               # First time next:  1
print(f"{next(gen) = }")               # Second time next: 2
print(f"{next(gen) = }", flush=True)   # Third time next:  3
print(f"{next(gen) = }", flush=True)   # raises StopIteration
```

↓ python3 simple_generator_function.py ↓

```
list(generator_123()) = [1, 2, 3]
next(gen) = 1
next(gen) = 2
next(gen) = 3
Traceback (most recent call last):
  File "{...}/iteration/simple_generator_function.py", line 19, in <
    ↪ module>
    print(f"{next(gen) = }", flush=True)  # raises StopIteration
          ^^^^^^^^^^^^^^^
StopIteration
# 'python3 simple_generator_function.py' failed with exit code 1.
```

Einfaches Beispiel

- Das klingt erstmal sehr verwirrend.
- Schauen wir uns einfaches Beispiel an.

```python
"""A simple example for generator functions."""

from typing import Generator  # The type hint for generators.


def generator_123() -> Generator[int, None, None]:
    """A generator function which yields 1, 2, 3."""
    yield 1  # The first time next(...) is called, the result is 1.
    yield 2  # The second time next(...) is called, the result is 2.
    yield 3  # The third time next(...) is called, the result is 3.


print(f"{list(generator_123()) = }")  # The list is [1, 2, 3].

gen: Generator[int, None, None] = generator_123()  # Use directly.
print(f"{next(gen) = }")               # First time next:  1
print(f"{next(gen) = }")               # Second time next: 2
print(f"{next(gen) = }", flush=True)   # Third time next:  3
print(f"{next(gen) = }", flush=True)   # raises StopIteration
```

↓ python3 simple_generator_function.py ↓

```
list(generator_123()) = [1, 2, 3]
next(gen) = 1
next(gen) = 2
next(gen) = 3
Traceback (most recent call last):
  File "{...}/iteration/simple_generator_function.py", line 19, in <
    ↪ module>
    print(f"{next(gen) = }", flush=True)  # raises StopIteration
          ^^^^^^^^^^^^^^^
StopIteration
# 'python3 simple_generator_function.py' failed with exit code 1.
```

Einfaches Beispiel

- Das klingt erstmal sehr verwirrend.
- Schauen wir uns einfaches Beispiel an.
- Wir erstellen eine einfache Generator-Funktion, die die Werte 1, 2, und 3 zurückliefern soll.

```python
"""A simple example for generator functions."""

from typing import Generator  # The type hint for generators.


def generator_123() -> Generator[int, None, None]:
    """A generator function which yields 1, 2, 3."""
    yield 1  # The first time next(...) is called, the result is 1.
    yield 2  # The second time next(...) is called, the result is 2.
    yield 3  # The third time next(...) is called, the result is 3.


print(f"{list(generator_123()) = }")  # The list is [1, 2, 3].

gen: Generator[int, None, None] = generator_123()  # Use directly.
print(f"{next(gen) = }")               # First time next:  1
print(f"{next(gen) = }")               # Second time next: 2
print(f"{next(gen) = }", flush=True)   # Third time next:  3
print(f"{next(gen) = }", flush=True)   # raises StopIteration
```

↓ python3 simple_generator_function.py ↓

```
list(generator_123()) = [1, 2, 3]
next(gen) = 1
next(gen) = 2
next(gen) = 3
Traceback (most recent call last):
  File "{...}/iteration/simple_generator_function.py", line 19, in <
    ↪ module>
    print(f"{next(gen) = }", flush=True)  # raises StopIteration
          ^^^^^^^^^^^^^^^
StopIteration
# 'python3 simple_generator_function.py' failed with exit code 1.
```

Einfaches Beispiel

- Das klingt erstmal sehr verwirrend.
- Schauen wir uns einfaches Beispiel an.
- Wir erstellen eine einfache Generator-Funktion, die die Werte 1, 2, und 3 zurückliefern soll.
- Wir nennen diese Funktion `generator_123` und deklarieren sie mit dem Schlüsselwort `def`, genau wie jede andere normale Funktion in Python.

```python
"""A simple example for generator functions."""

from typing import Generator  # The type hint for generators.


def generator_123() -> Generator[int, None, None]:
    """A generator function which yields 1, 2, 3."""
    yield 1  # The first time next(...) is called, the result is 1.
    yield 2  # The second time next(...) is called, the result is 2.
    yield 3  # The third time next(...) is called, the result is 3.


print(f"{list(generator_123()) = }")  # The list is [1, 2, 3].

gen: Generator[int, None, None] = generator_123()  # Use directly.
print(f"{next(gen) = }")               # First time next:  1
print(f"{next(gen) = }")               # Second time next: 2
print(f"{next(gen) = }", flush=True)   # Third time next:  3
print(f"{next(gen) = }", flush=True)   # raises StopIteration
```

↓ python3 simple_generator_function.py ↓

```
list(generator_123()) = [1, 2, 3]
next(gen) = 1
next(gen) = 2
next(gen) = 3
Traceback (most recent call last):
  File "{...}/iteration/simple_generator_function.py", line 19, in <
    ↪ module>
    print(f"{next(gen) = }", flush=True)  # raises StopIteration
          ^^^^^^^^^^
StopIteration
# 'python3 simple_generator_function.py' failed with exit code 1.
```

Einfaches Beispiel

- Schauen wir uns einfaches Beispiel an.
- Wir erstellen eine einfache Generator-Funktion, die die Werte 1, 2, und 3 zurückliefern soll.
- Wir nennen diese Funktion `generator_123` und deklarieren sie mit dem Schlüsselwort `def`, genau wie jede andere normale Funktion in Python.
- Den Rückgabewert wird mit dem Type Hint `Generator[int, None, None]` annotiert, was bedeutet, dass das hier eine Generator-Funktion ist, die `int`-Werte produziert.

```python
"""A simple example for generator functions."""

from typing import Generator  # The type hint for generators.


def generator_123() -> Generator[int, None, None]:
    """A generator function which yields 1, 2, 3."""
    yield 1  # The first time next(...) is called, the result is 1.
    yield 2  # The second time next(...) is called, the result is 2.
    yield 3  # The third time next(...) is called, the result is 3.


print(f"{list(generator_123()) = }")  # The list is [1, 2, 3].

gen: Generator[int, None, None] = generator_123()  # Use directly.
print(f"{next(gen) = }")               # First time next:  1
print(f"{next(gen) = }")               # Second time next: 2
print(f"{next(gen) = }", flush=True)   # Third time next:  3
print(f"{next(gen) = }", flush=True)   # raises StopIteration
```

↓ python3 simple_generator_function.py ↓

```
list(generator_123()) = [1, 2, 3]
next(gen) = 1
next(gen) = 2
next(gen) = 3
Traceback (most recent call last):
  File "{...}/iteration/simple_generator_function.py", line 19, in <
      ↪ module>
    print(f"{next(gen) = }", flush=True)  # raises StopIteration
          ^^^^^^^^^^
StopIteration
# 'python3 simple_generator_function.py' failed with exit code 1.
```

Einfaches Beispiel

- Wir nennen diese Funktion `generator_123` und deklarieren sie mit dem Schlüsselwort `def`, genau wie jede andere normale Funktion in Python.
- Den Rückgabewert wird mit dem Type Hint `Generator[int, None, None]` annotiert, was bedeutet, dass das hier eine Generator-Funktion ist, die `int`-Werte produziert.
- Der Körper der Funktion besteht aus nur drei Befehlen: `yield 1`, `yield 2`, und `yield 3`.

```python
"""A simple example for generator functions."""

from typing import Generator  # The type hint for generators.


def generator_123() -> Generator[int, None, None]:
    """A generator function which yields 1, 2, 3."""
    yield 1  # The first time next(...) is called, the result is 1.
    yield 2  # The second time next(...) is called, the result is 2.
    yield 3  # The third time next(...) is called, the result is 3.


print(f"{list(generator_123()) = }")  # The list is [1, 2, 3].

gen: Generator[int, None, None] = generator_123()  # Use directly.
print(f"{next(gen) = }")               # First time next:  1
print(f"{next(gen) = }")               # Second time next: 2
print(f"{next(gen) = }", flush=True)   # Third time next:  3
print(f"{next(gen) = }", flush=True)   # raises StopIteration
```

↓ `python3 simple_generator_function.py` ↓

```
list(generator_123()) = [1, 2, 3]
next(gen) = 1
next(gen) = 2
next(gen) = 3
Traceback (most recent call last):
  File "{...}/iteration/simple_generator_function.py", line 19, in <
    ↪ module>
    print(f"{next(gen) = }", flush=True)  # raises StopIteration
          ^^^^^^^^^^^^^^^
StopIteration
# 'python3 simple_generator_function.py' failed with exit code 1.
```

Einfaches Beispiel

- Den Rückgabewert wird mit dem Type Hint `Generator[int, None, None]` annotiert, was bedeutet, dass das hier eine Generator-Funktion ist, die `int`-Werte produziert.
- Der Körper der Funktion besteht aus nur drei Befehlen: `yield 1`, `yield 2`, und `yield 3`.
- Wir können den von der Funktion gelieferten `Generator` z. B. benutzen um eine Liste zu füllen.

```python
"""A simple example for generator functions."""

from typing import Generator  # The type hint for generators.


def generator_123() -> Generator[int, None, None]:
    """A generator function which yields 1, 2, 3."""
    yield 1  # The first time next(...) is called, the result is 1.
    yield 2  # The second time next(...) is called, the result is 2.
    yield 3  # The third time next(...) is called, the result is 3.


print(f"{list(generator_123()) = }")  # The list is [1, 2, 3].

gen: Generator[int, None, None] = generator_123()  # Use directly.
print(f"{next(gen) = }")               # First time next:  1
print(f"{next(gen) = }")               # Second time next: 2
print(f"{next(gen) = }", flush=True)   # Third time next:  3
print(f"{next(gen) = }", flush=True)   # raises StopIteration
```

↓ python3 simple_generator_function.py ↓

```
list(generator_123()) = [1, 2, 3]
next(gen) = 1
next(gen) = 2
next(gen) = 3
Traceback (most recent call last):
  File "{...}/iteration/simple_generator_function.py", line 19, in <
    ↪ module>
    print(f"{next(gen) = }", flush=True)  # raises StopIteration
          ^^^^^^^^^^^^^^^
StopIteration
# 'python3 simple_generator_function.py' failed with exit code 1.
```

Einfaches Beispiel

- Den Rückgabewert wird mit dem Type Hint `Generator[int, None, None]` annotiert, was bedeutet, dass das hier eine Generator-Funktion ist, die `int`-Werte produziert.
- Der Körper der Funktion besteht aus nur drei Befehlen: `yield 1`, `yield 2`, und `yield 3`.
- Wir können den von der Funktion gelieferten `Generator` z. B. benutzen um eine Liste zu füllen.
- `list(generator_123())` erstellt die Liste `[1, 2, 3]`.

```python
"""A simple example for generator functions."""

from typing import Generator  # The type hint for generators.


def generator_123() -> Generator[int, None, None]:
    """A generator function which yields 1, 2, 3."""
    yield 1  # The first time next(...) is called, the result is 1.
    yield 2  # The second time next(...) is called, the result is 2.
    yield 3  # The third time next(...) is called, the result is 3.


print(f"{list(generator_123()) = }")  # The list is [1, 2, 3].

gen: Generator[int, None, None] = generator_123()  # Use directly.
print(f"{next(gen) = }")               # First time next:  1
print(f"{next(gen) = }")               # Second time next: 2
print(f"{next(gen) = }", flush=True)   # Third time next:  3
print(f"{next(gen) = }", flush=True)   # raises StopIteration
```

↓ python3 simple_generator_function.py ↓

```
list(generator_123()) = [1, 2, 3]
next(gen) = 1
next(gen) = 2
next(gen) = 3
Traceback (most recent call last):
  File "{...}/iteration/simple_generator_function.py", line 19, in <
    ↪ module>
    print(f"{next(gen) = }", flush=True)  # raises StopIteration
          ^^^^^^^^^^
StopIteration
# 'python3 simple_generator_function.py' failed with exit code 1.
```

Einfaches Beispiel

- Der Körper der Funktion besteht aus nur drei Befehlen: `yield 1`, `yield 2`, und `yield 3`.
- Wir können den von der Funktion gelieferten `Generator` z. B. benutzen um eine Liste zu füllen.
- `list(generator_123())` erstellt die Liste `[1, 2, 3]`.
- Natürlich können wir auch manuell über die Generator-Funktionieren, genauso wie über jeden anderen `Iterator`.

```python
"""A simple example for generator functions."""

from typing import Generator  # The type hint for generators.


def generator_123() -> Generator[int, None, None]:
    """A generator function which yields 1, 2, 3."""
    yield 1  # The first time next(...) is called, the result is 1.
    yield 2  # The second time next(...) is called, the result is 2.
    yield 3  # The third time next(...) is called, the result is 3.


print(f"{list(generator_123()) = }")  # The list is [1, 2, 3].

gen: Generator[int, None, None] = generator_123()  # Use directly.
print(f"{next(gen) = }")               # First time next:  1
print(f"{next(gen) = }")               # Second time next: 2
print(f"{next(gen) = }", flush=True)   # Third time next:  3
print(f"{next(gen) = }", flush=True)   # raises StopIteration
```

↓ python3 simple_generator_function.py ↓

```
list(generator_123()) = [1, 2, 3]
next(gen) = 1
next(gen) = 2
next(gen) = 3
Traceback (most recent call last):
  File "{...}/iteration/simple_generator_function.py", line 19, in <
    ↪ module>
    print(f"{next(gen) = }", flush=True)  # raises StopIteration
          ^^^^^^^^^^
StopIteration
# 'python3 simple_generator_function.py' failed with exit code 1.
```

Einfaches Beispiel

- Wir können den von der Funktion gelieferten `Generator` z. B. benutzen um eine Liste zu füllen.
- `list(generator_123())` erstellt die Liste `[1, 2, 3]`.
- Natürlich können wir auch manuell über die Generator-Funktionieren, genauso wie über jeden anderen `Iterator`.
- Dafür setzen wir `gen = generator_123()`.

```python
"""A simple example for generator functions."""

from typing import Generator  # The type hint for generators.


def generator_123() -> Generator[int, None, None]:
    """A generator function which yields 1, 2, 3."""
    yield 1  # The first time next(...) is called, the result is 1.
    yield 2  # The second time next(...) is called, the result is 2.
    yield 3  # The third time next(...) is called, the result is 3.


print(f"{list(generator_123()) = }")  # The list is [1, 2, 3].

gen: Generator[int, None, None] = generator_123()  # Use directly.
print(f"{next(gen) = }")               # First time next:  1
print(f"{next(gen) = }")               # Second time next: 2
print(f"{next(gen) = }", flush=True)   # Third time next:  3
print(f"{next(gen) = }", flush=True)   # raises StopIteration
```

↓ python3 simple_generator_function.py ↓

```
list(generator_123()) = [1, 2, 3]
next(gen) = 1
next(gen) = 2
next(gen) = 3
Traceback (most recent call last):
  File "{...}/iteration/simple_generator_function.py", line 19, in <
    ↪ module>
    print(f"{next(gen) = }", flush=True)  # raises StopIteration
          ^^^^^^^^^^
StopIteration
# 'python3 simple_generator_function.py' failed with exit code 1.
```

Einfaches Beispiel

- `list(generator_123())` erstellt die Liste `[1, 2, 3]`.
- Natürlich können wir auch manuell über die Generator-Funktionieren, genauso wie über jeden anderen `Iterator`.
- Dafür setzen wir `gen = generator_123()`.
- Wenn wir das erste Mal `next(gen)` aufrufen, bekommen wir `1`.

```python
"""A simple example for generator functions."""

from typing import Generator  # The type hint for generators.


def generator_123() -> Generator[int, None, None]:
    """A generator function which yields 1, 2, 3."""
    yield 1  # The first time next(...) is called, the result is 1.
    yield 2  # The second time next(...) is called, the result is 2.
    yield 3  # The third time next(...) is called, the result is 3.


print(f"{list(generator_123()) = }")  # The list is [1, 2, 3].

gen: Generator[int, None, None] = generator_123()  # Use directly.
print(f"{next(gen) = }")               # First time next:  1
print(f"{next(gen) = }")               # Second time next: 2
print(f"{next(gen) = }", flush=True)   # Third time next:  3
print(f"{next(gen) = }", flush=True)   # raises StopIteration
```

↓ python3 simple_generator_function.py ↓

```
list(generator_123()) = [1, 2, 3]
next(gen) = 1
next(gen) = 2
next(gen) = 3
Traceback (most recent call last):
  File "{...}/iteration/simple_generator_function.py", line 19, in <
    ↪ module>
    print(f"{next(gen) = }", flush=True)  # raises StopIteration
          ^^^^^^^^^^
StopIteration
# 'python3 simple_generator_function.py' failed with exit code 1.
```

Einfaches Beispiel

- Natürlich können wir auch manuell über die Generator-Funktionieren, genauso wie über jeden anderen `Iterator`.
- Dafür setzen wir `gen = generator_123()`.
- Wenn wir das erste Mal `next(gen)` aufrufen, bekommen wir `1`.
- Wenn wir das zweite Mal `next(gen)` aufrufen, bekommen wir `2`.

```python
"""A simple example for generator functions."""

from typing import Generator  # The type hint for generators.


def generator_123() -> Generator[int, None, None]:
    """A generator function which yields 1, 2, 3."""
    yield 1  # The first time next(...) is called, the result is 1.
    yield 2  # The second time next(...) is called, the result is 2.
    yield 3  # The third time next(...) is called, the result is 3.


print(f"{list(generator_123()) = }")  # The list is [1, 2, 3].

gen: Generator[int, None, None] = generator_123()  # Use directly.
print(f"{next(gen) = }")               # First time next:  1
print(f"{next(gen) = }")               # Second time next: 2
print(f"{next(gen) = }", flush=True)   # Third time next:  3
print(f"{next(gen) = }", flush=True)   # raises StopIteration
```

↓ python3 simple_generator_function.py ↓

```
list(generator_123()) = [1, 2, 3]
next(gen) = 1
next(gen) = 2
next(gen) = 3
Traceback (most recent call last):
  File "{...}/iteration/simple_generator_function.py", line 19, in <
    ↪ module>
    print(f"{next(gen) = }", flush=True)  # raises StopIteration
          ^^^^^^^^^^^^^^^
StopIteration
# 'python3 simple_generator_function.py' failed with exit code 1.
```

Einfaches Beispiel

- Dafür setzen wir `gen = generator_123()`.
- Wenn wir das erste Mal `next(gen)` aufrufen, bekommen wir `1`.
- Wenn wir das zweite Mal `next(gen)` aufrufen, bekommen wir `2`.
- Wenn wir das dritte Mal `next(gen)` aufrufen, bekommen wir `3`.

```python
"""A simple example for generator functions."""

from typing import Generator  # The type hint for generators.


def generator_123() -> Generator[int, None, None]:
    """A generator function which yields 1, 2, 3."""
    yield 1  # The first time next(...) is called, the result is 1.
    yield 2  # The second time next(...) is called, the result is 2.
    yield 3  # The third time next(...) is called, the result is 3.


print(f"{list(generator_123()) = }")  # The list is [1, 2, 3].

gen: Generator[int, None, None] = generator_123()  # Use directly.
print(f"{next(gen) = }")               # First time next:  1
print(f"{next(gen) = }")               # Second time next: 2
print(f"{next(gen) = }", flush=True)   # Third time next:  3
print(f"{next(gen) = }", flush=True)   # raises StopIteration
```

↓ python3 simple_generator_function.py ↓

```
list(generator_123()) = [1, 2, 3]
next(gen) = 1
next(gen) = 2
next(gen) = 3
Traceback (most recent call last):
  File "{...}/iteration/simple_generator_function.py", line 19, in <
    ↪ module>
    print(f"{next(gen) = }", flush=True)  # raises StopIteration
          ^^^^^^^^^^^^^^^
StopIteration
# 'python3 simple_generator_function.py' failed with exit code 1.
```

Einfaches Beispiel

- Wenn wir das erste Mal `next(gen)` aufrufen, bekommen wir `1`.
- Wenn wir das zweite Mal `next(gen)` aufrufen, bekommen wir `2`.
- Wenn wir das dritte Mal `next(gen)` aufrufen, bekommen wir `3`.
- Der vierte Aufruf von `next(gen)` erzeugt dann eine `StopIteration`-Ausnahme.

```python
"""A simple example for generator functions."""

from typing import Generator  # The type hint for generators.


def generator_123() -> Generator[int, None, None]:
    """A generator function which yields 1, 2, 3."""
    yield 1  # The first time next(...) is called, the result is 1.
    yield 2  # The second time next(...) is called, the result is 2.
    yield 3  # The third time next(...) is called, the result is 3.


print(f"{list(generator_123()) = }")  # The list is [1, 2, 3].

gen: Generator[int, None, None] = generator_123()  # Use directly.
print(f"{next(gen) = }")               # First time next:  1
print(f"{next(gen) = }")               # Second time next: 2
print(f"{next(gen) = }", flush=True)   # Third time next:  3
print(f"{next(gen) = }", flush=True)   # raises StopIteration
```

↓ python3 simple_generator_function.py ↓

```
list(generator_123()) = [1, 2, 3]
next(gen) = 1
next(gen) = 2
next(gen) = 3
Traceback (most recent call last):
  File "{...}/iteration/simple_generator_function.py", line 19, in <
    ↪ module>
    print(f"{next(gen) = }", flush=True)  # raises StopIteration
          ^^^^^^^^^^^^^^^
StopIteration
# 'python3 simple_generator_function.py' failed with exit code 1.
```

Einfaches Beispiel

- Wenn wir das erste Mal `next(gen)` aufrufen, bekommen wir `1`.
- Wenn wir das zweite Mal `next(gen)` aufrufen, bekommen wir `2`.
- Wenn wir das dritte Mal `next(gen)` aufrufen, bekommen wir `3`.
- Der vierte Aufruf von `next(gen)` erzeugt dann eine `StopIteration`-Ausnahme.
- Damit ist das Ende der Sequenz erreicht.

```python
"""A simple example for generator functions."""

from typing import Generator  # The type hint for generators.


def generator_123() -> Generator[int, None, None]:
    """A generator function which yields 1, 2, 3."""
    yield 1  # The first time next(...) is called, the result is 1.
    yield 2  # The second time next(...) is called, the result is 2.
    yield 3  # The third time next(...) is called, the result is 3.


print(f"{list(generator_123()) = }")  # The list is [1, 2, 3].

gen: Generator[int, None, None] = generator_123()  # Use directly.
print(f"{next(gen) = }")               # First time next:  1
print(f"{next(gen) = }")               # Second time next: 2
print(f"{next(gen) = }", flush=True)   # Third time next:  3
print(f"{next(gen) = }", flush=True)   # raises StopIteration
```

↓ python3 simple_generator_function.py ↓

```
list(generator_123()) = [1, 2, 3]
next(gen) = 1
next(gen) = 2
next(gen) = 3
Traceback (most recent call last):
  File "{...}/iteration/simple_generator_function.py", line 19, in <
    ↪ module>
    print(f"{next(gen) = }", flush=True)  # raises StopIteration
          ^^^^^^^^^^^^^^^
StopIteration
# 'python3 simple_generator_function.py' failed with exit code 1.
```

Einfaches Beispiel

- Wenn wir das zweite Mal `next(gen)` aufrufen, bekommen wir 2.
- Wenn wir das dritte Mal `next(gen)` aufrufen, bekommen wir 3.
- Der vierte Aufruf von `next(gen)` erzeugt dann eine `StopIteration`-Ausnahme.
- Damit ist das Ende der Sequenz erreicht.
- Wir können Generator-Funktionen also wirklich genau wie normale Iteratoren verwenden.

```python
"""A simple example for generator functions."""

from typing import Generator  # The type hint for generators.


def generator_123() -> Generator[int, None, None]:
    """A generator function which yields 1, 2, 3."""
    yield 1  # The first time next(...) is called, the result is 1.
    yield 2  # The second time next(...) is called, the result is 2.
    yield 3  # The third time next(...) is called, the result is 3.


print(f"{list(generator_123()) = }")  # The list is [1, 2, 3].

gen: Generator[int, None, None] = generator_123()  # Use directly.
print(f"{next(gen) = }")               # First time next:  1
print(f"{next(gen) = }")               # Second time next: 2
print(f"{next(gen) = }", flush=True)   # Third time next:  3
print(f"{next(gen) = }", flush=True)   # raises StopIteration
```

↓ python3 simple_generator_function.py ↓

```
list(generator_123()) = [1, 2, 3]
next(gen) = 1
next(gen) = 2
next(gen) = 3
Traceback (most recent call last):
  File "{...}/iteration/simple_generator_function.py", line 19, in <
    ↪ module>
    print(f"{next(gen) = }", flush=True)  # raises StopIteration
          ^^^^^^^^^^^^^^^
StopIteration
# 'python3 simple_generator_function.py' failed with exit code 1.
```

Einfaches Beispiel

- Wenn wir das dritte Mal `next(gen)` aufrufen, bekommen wir `3`.
- Der vierte Aufruf von `next(gen)` erzeugt dann eine `StopIteration`-Ausnahme.
- Damit ist das Ende der Sequenz erreicht.
- Wir können Generator-Funktionen also wirklich genau wie normale Iteratoren verwenden.
- Das Interessante ist, dass die Funktion wirklich bei jedem `yield` unterbrochen wird.

```python
"""A simple example for generator functions."""

from typing import Generator  # The type hint for generators.


def generator_123() -> Generator[int, None, None]:
    """A generator function which yields 1, 2, 3."""
    yield 1  # The first time next(...) is called, the result is 1.
    yield 2  # The second time next(...) is called, the result is 2.
    yield 3  # The third time next(...) is called, the result is 3.


print(f"{list(generator_123()) = }")  # The list is [1, 2, 3].

gen: Generator[int, None, None] = generator_123()  # Use directly.
print(f"{next(gen) = }")               # First time next:  1
print(f"{next(gen) = }")               # Second time next: 2
print(f"{next(gen) = }", flush=True)   # Third time next:  3
print(f"{next(gen) = }", flush=True)   # raises StopIteration
```

↓ python3 simple_generator_function.py ↓

```
list(generator_123()) = [1, 2, 3]
next(gen) = 1
next(gen) = 2
next(gen) = 3
Traceback (most recent call last):
  File "{...}/iteration/simple_generator_function.py", line 19, in <
      ↪ module>
    print(f"{next(gen) = }", flush=True)  # raises StopIteration
          ^^^^^^^^^^
StopIteration
# 'python3 simple_generator_function.py' failed with exit code 1.
```

Einfaches Beispiel

- Der vierte Aufruf von `next(gen)` erzeugt dann eine `StopIteration`-Ausnahme.
- Damit ist das Ende der Sequenz erreicht.
- Wir können Generator-Funktionen also wirklich genau wie normale Iteratoren verwenden.
- Das Interessante ist, dass die Funktion wirklich bei jedem `yield` unterbrochen wird.
- Der von `yield` zurückgelieferte Wert wird dann vom Kode außerhalb der Funktion erhalten, der damit machen kann, was immer er will.

```python
"""A simple example for generator functions."""

from typing import Generator  # The type hint for generators.


def generator_123() -> Generator[int, None, None]:
    """A generator function which yields 1, 2, 3."""
    yield 1  # The first time next(...) is called, the result is 1.
    yield 2  # The second time next(...) is called, the result is 2.
    yield 3  # The third time next(...) is called, the result is 3.


print(f"{list(generator_123()) = }")  # The list is [1, 2, 3].

gen: Generator[int, None, None] = generator_123()  # Use directly.
print(f"{next(gen) = }")               # First time next:  1
print(f"{next(gen) = }")               # Second time next: 2
print(f"{next(gen) = }", flush=True)   # Third time next:  3
print(f"{next(gen) = }", flush=True)   # raises StopIteration
```

↓ python3 simple_generator_function.py ↓

```
list(generator_123()) = [1, 2, 3]
next(gen) = 1
next(gen) = 2
next(gen) = 3
Traceback (most recent call last):
  File "{...}/iteration/simple_generator_function.py", line 19, in <
      ↪ module>
    print(f"{next(gen) = }", flush=True)  # raises StopIteration
          ^^^^^^^^^^
StopIteration
# 'python3 simple_generator_function.py' failed with exit code 1.
```

Einfaches Beispiel

- Damit ist das Ende der Sequenz erreicht.
- Wir können Generator-Funktionen also wirklich genau wie normale Iteratoren verwenden.
- Das Interessante ist, dass die Funktion wirklich bei jedem `yield` unterbrochen wird.
- Der von `yield` zurückgelieferte Wert wird dann vom Kode außerhalb der Funktion erhalten, der damit machen kann, was immer er will.
- Die Funktion wird nach dem `yield` fortgesetzt, wenn `next` aufgerufen wird.

```python
"""A simple example for generator functions."""

from typing import Generator  # The type hint for generators.


def generator_123() -> Generator[int, None, None]:
    """A generator function which yields 1, 2, 3."""
    yield 1  # The first time next(...) is called, the result is 1.
    yield 2  # The second time next(...) is called, the result is 2.
    yield 3  # The third time next(...) is called, the result is 3.


print(f"{list(generator_123()) = }")  # The list is [1, 2, 3].

gen: Generator[int, None, None] = generator_123()  # Use directly.
print(f"{next(gen) = }")               # First time next:  1
print(f"{next(gen) = }")               # Second time next: 2
print(f"{next(gen) = }", flush=True)   # Third time next:  3
print(f"{next(gen) = }", flush=True)   # raises StopIteration
```

↓ python3 simple_generator_function.py ↓

```
list(generator_123()) = [1, 2, 3]
next(gen) = 1
next(gen) = 2
next(gen) = 3
Traceback (most recent call last):
  File "{...}/iteration/simple_generator_function.py", line 19, in <
      ↪ module>
    print(f"{next(gen) = }", flush=True)  # raises StopIteration
          ^^^^^^^^^^
StopIteration
# 'python3 simple_generator_function.py' failed with exit code 1.
```

Einfaches Beispiel

- Wir können Generator-Funktionen also wirklich genau wie normale Iteratoren verwenden.
- Das Interessante ist, dass die Funktion wirklich bei jedem `yield` unterbrochen wird.
- Der von `yield` zurückgelieferte Wert wird dann vom Kode außerhalb der Funktion erhalten, der damit machen kann, was immer er will.
- Die Funktion wird nach dem `yield` fortgesetzt, wenn `next` aufgerufen wird.
- Erreicht der Kontrollfluss das Ende der Funktion, dann endet auch die Sequenz.

```python
"""A simple example for generator functions."""

from typing import Generator  # The type hint for generators.


def generator_123() -> Generator[int, None, None]:
    """A generator function which yields 1, 2, 3."""
    yield 1  # The first time next(...) is called, the result is 1.
    yield 2  # The second time next(...) is called, the result is 2.
    yield 3  # The third time next(...) is called, the result is 3.


print(f"{list(generator_123()) = }")  # The list is [1, 2, 3].

gen: Generator[int, None, None] = generator_123()  # Use directly.
print(f"{next(gen) = }")               # First time next:  1
print(f"{next(gen) = }")               # Second time next: 2
print(f"{next(gen) = }", flush=True)   # Third time next:  3
print(f"{next(gen) = }", flush=True)   # raises StopIteration
```

↓ python3 simple_generator_function.py ↓

```
list(generator_123()) = [1, 2, 3]
next(gen) = 1
next(gen) = 2
next(gen) = 3
Traceback (most recent call last):
  File "{...}/iteration/simple_generator_function.py", line 19, in <
      ↪ module>
    print(f"{next(gen) = }", flush=True)  # raises StopIteration
          ^^^^^^^^^^
StopIteration
# 'python3 simple_generator_function.py' failed with exit code 1.
```

Einfaches Beispiel

- Das Interessante ist, dass die Funktion wirklich bei jedem `yield` unterbrochen wird.
- Der von `yield` zurückgelieferte Wert wird dann vom Kode außerhalb der Funktion erhalten, der damit machen kann, was immer er will.
- Die Funktion wird nach dem `yield` fortgesetzt, wenn `next` aufgerufen wird.
- Erreicht der Kontrollfluss das Ende der Funktion, dann endet auch die Sequenz.
- Das wird sichtbar, wenn wir eine Funktion implementieren, die eine unendliche Sequenz zurückliefert.

```python
"""A simple example for generator functions."""

from typing import Generator  # The type hint for generators.


def generator_123() -> Generator[int, None, None]:
    """A generator function which yields 1, 2, 3."""
    yield 1  # The first time next(...) is called, the result is 1.
    yield 2  # The second time next(...) is called, the result is 2.
    yield 3  # The third time next(...) is called, the result is 3.


print(f"{list(generator_123()) = }")  # The list is [1, 2, 3].

gen: Generator[int, None, None] = generator_123()  # Use directly.
print(f"{next(gen) = }")               # First time next:  1
print(f"{next(gen) = }")               # Second time next: 2
print(f"{next(gen) = }", flush=True)   # Third time next:  3
print(f"{next(gen) = }", flush=True)   # raises StopIteration
```

↓ python3 simple_generator_function.py ↓

```
list(generator_123()) = [1, 2, 3]
next(gen) = 1
next(gen) = 2
next(gen) = 3
Traceback (most recent call last):
  File "{...}/iteration/simple_generator_function.py", line 19, in <
    ↪ module>
    print(f"{next(gen) = }", flush=True)  # raises StopIteration
          ^^^^^^^^^^
StopIteration
# 'python3 simple_generator_function.py' failed with exit code 1.
```

Fibonacci Sequenz

- Fibonacci war ein mittelalterlicher italienischer Mathematiker, der im 12. und 13. Jahrhundert Common Era (CE) gelebt hat[2].

Sculpture by Bertel Thorvaldsen, 1834/1838. Source: Thorvaldsens Museum A187, photographer Jakob Faurvig.CC0

Fibonacci Sequenz

- Fibonacci war ein mittelalterlicher italienischer Mathematiker, der im 12. und 13. Jahrhundert Common Era (CE) gelebt hat[2].
- Er hat das Buch *Liber Abaci*[24] geschrieben, mit dem arabische Zahlen in Europa eingeführt wurden.

Sculpture by Bertel Thorvaldsen, 1834/1838. Source: Thorvaldsens Museum A187, photographer Jakob Faurvig,CC0

Fibonacci Sequenz

- Fibonacci war ein mittelalterlicher italienischer Mathematiker, der im 12. und 13. Jahrhundert Common Era (CE) gelebt hat[2].
- Er hat das Buch *Liber Abaci*[24] geschrieben, mit dem arabische Zahlen in Europa eingeführt wurden.
- Er ist auch für die Fibonacci-Zahlen bekannt, die der Sequenz $F_n = F_{n-1} + F_{n-2}$ folgen, wobei $F_0 = 0$ und $F_1 = 1$[24,32].

Sculpture by Bertel Thorvaldsen, 1834/1838. Source: Thorvaldsens Museum A187, photographer Jakob Faurvig.CC0

Fibonacci Sequenz

- Fibonacci war ein mittelalterlicher italienischer Mathematiker, der im 12. und 13. Jahrhundert Common Era (CE) gelebt hat[2].
- Er hat das Buch *Liber Abaci*[24] geschrieben, mit dem arabische Zahlen in Europa eingeführt wurden.
- Er ist auch für die Fibonacci-Zahlen bekannt, die der Sequenz $F_n = F_{n-1} + F_{n-2}$ folgen, wobei $F_0 = 0$ und $F_1 = 1$[24,32].
- Wir wollen über diese Zahlenfolge iterieren.

Sculpture by Bertel Thorvaldsen, 1834/1838. Source: Thorvaldsens Museum A187, photographer Jakob Faurvig,CC0

Fibonacci Sequenz

- Er hat das Buch *Liber Abaci*[24] geschrieben, mit dem arabische Zahlen in Europa eingeführt wurden.
- Er ist auch für die Fibonacci-Zahlen bekannt, die der Sequenz $F_n = F_{n-1} + F_{n-2}$ folgen, wobei $F_0 = 0$ und $F_1 = 1$[24,32].
- Wir wollen über diese Zahlenfolge iterieren.
- Dafür definieren wir die Funktion `fibonacci`, die wir wieder mit `Generator` als Rückgabewert annotieren.

```python
"""A generator functions iterating over the Fibonacci Sequence."""

from typing import Generator  # The type hint for generators.


def fibonacci() -> Generator[int, None, None]:
    """A generator returning Fibonacci numbers."""
    i: int = 0   # Initialize i.
    j: int = 1   # Initialize j.
    while True:  # Loop forever, i.e., generator can continue forever.
        yield i  # Return the Fibonacci number.
        i, j = j, i + j  # i = old_j and j = old_i + old_j


for a in fibonacci():  # Loop over the generated sequence.
    print(a)  # Print the sequence element.
    if a > 30:  # If a > 300, then
        break    # we stop the iteration.

# list(fibonacci()) <-- This would fail!!
```

↓ `python3 fibonacci_generator.py` ↓

```
0
1
1
2
3
5
8
13
21
34
# 'python3 fibonacci_generator.py' succeeded with exit code 0.
```

Fibonacci Sequenz

- Er hat das Buch *Liber Abaci*[24] geschrieben, mit dem arabische Zahlen in Europa eingeführt wurden.
- Er ist auch für die Fibonacci-Zahlen bekannt, die der Sequenz $F_n = F_{n-1} + F_{n-2}$ folgen, wobei $F_0 = 0$ und $F_1 = 1$[24,32].
- Wir wollen über diese Zahlenfolge iterieren.
- Dafür definieren wir die Funktion `fibonacci`, die wir wieder mit `Generator` als Rückgabewert annotieren.
- Die Funktion beginnt damit, `i` $= F_0 = 0$ und `j` $= F_1 = 1$ zu setzen.

```python
"""A generator functions iterating over the Fibonacci Sequence."""

from typing import Generator  # The type hint for generators.


def fibonacci() -> Generator[int, None, None]:
    """A generator returning Fibonacci numbers."""
    i: int = 0   # Initialize i.
    j: int = 1   # Initialize j.
    while True:  # Loop forever, i.e., generator can continue forever.
        yield i  # Return the Fibonacci number.
        i, j = j, i + j  # i = old_j and j = old_i + old_j


for a in fibonacci():  # Loop over the generated sequence.
    print(a)  # Print the sequence element.
    if a > 30:  # If a > 300, then
        break    # we stop the iteration.

# list(fibonacci()) <-- This would fail!!
```

↓ `python3 fibonacci_generator.py` ↓

```
0
1
1
2
3
5
8
13
21
34
# 'python3 fibonacci_generator.py' succeeded with exit code 0.
```

Fibonacci Sequenz

- Wir wollen über diese Zahlenfolge iterieren.
- Dafür definieren wir die Funktion `fibonacci`, die wir wieder mit `Generator` als Rückgabewert annotieren.
- Die Funktion beginnt damit, `i` $= F_0 = 0$ und `j` $= F_1 = 1$ zu setzen.
- Danach kommt eine `while`-Schleife, die niemals aufhört, da ihre Schleifenbedingung einfach auf `True`) gesetzt ist.

```python
"""A generator functions iterating over the Fibonacci Sequence."""

from typing import Generator  # The type hint for generators.


def fibonacci() -> Generator[int, None, None]:
    """A generator returning Fibonacci numbers."""
    i: int = 0   # Initialize i.
    j: int = 1   # Initialize j.
    while True:  # Loop forever, i.e., generator can continue forever.
        yield i  # Return the Fibonacci number.
        i, j = j, i + j  # i = old_j and j = old_i + old_j


for a in fibonacci():  # Loop over the generated sequence.
    print(a)  # Print the sequence element.
    if a > 30:  # If a > 300, then
        break    # we stop the iteration.

# list(fibonacci()) <-- This would fail!!
```

↓ `python3 fibonacci_generator.py` ↓

```
0
1
1
2
3
5
8
13
21
34
# 'python3 fibonacci_generator.py' succeeded with exit code 0.
```

Fibonacci Sequenz

- Wir wollen über diese Zahlenfolge iterieren.
- Dafür definieren wir die Funktion `fibonacci`, die wir wieder mit `Generator` als Rückgabewert annotieren.
- Die Funktion beginnt damit, `i` $= F_0 = 0$ und `j` $= F_1 = 1$ zu setzen.
- Danach kommt eine `while`-Schleife, die niemals aufhört, da ihre Schleifenbedingung einfach auf `True`) gesetzt ist.
- In der Schleife folgt dann zuerst ein `yield i`.

```python
"""A generator functions iterating over the Fibonacci Sequence."""

from typing import Generator  # The type hint for generators.


def fibonacci() -> Generator[int, None, None]:
    """A generator returning Fibonacci numbers."""
    i: int = 0   # Initialize i.
    j: int = 1   # Initialize j.
    while True:  # Loop forever, i.e., generator can continue forever.
        yield i  # Return the Fibonacci number.
        i, j = j, i + j  # i = old_j and j = old_i + old_j


for a in fibonacci():  # Loop over the generated sequence.
    print(a)  # Print the sequence element.
    if a > 30:  # If a > 300, then
        break    # we stop the iteration.

# list(fibonacci()) <-- This would fail!!
```

↓ `python3 fibonacci_generator.py` ↓

```
0
1
1
2
3
5
8
13
21
34
# 'python3 fibonacci_generator.py' succeeded with exit code 0.
```

Fibonacci Sequenz

- Dafür definieren wir die Funktion `fibonacci`, die wir wieder mit `Generator` als Rückgabewert annotieren.
- Die Funktion beginnt damit, `i` $= F_0 = 0$ und `j` $= F_1 = 1$ zu setzen.
- Danach kommt eine `while`-Schleife, die niemals aufhört, da ihre Schleifenbedingung einfach auf `True`) gesetzt ist.
- In der Schleife folgt dann zuerst ein `yield i`.
- Das bedeutet, dass die Ausführung der Funktion unterbrochen wird.

```python
"""A generator functions iterating over the Fibonacci Sequence."""

from typing import Generator  # The type hint for generators.


def fibonacci() -> Generator[int, None, None]:
    """A generator returning Fibonacci numbers."""
    i: int = 0   # Initialize i.
    j: int = 1   # Initialize j.
    while True:  # Loop forever, i.e., generator can continue forever.
        yield i  # Return the Fibonacci number.
        i, j = j, i + j  # i = old_j and j = old_i + old_j


for a in fibonacci():  # Loop over the generated sequence.
    print(a)  # Print the sequence element.
    if a > 30:  # If a > 300, then
        break    # we stop the iteration.

# list(fibonacci()) <-- This would fail!!
```

↓ `python3 fibonacci_generator.py` ↓

```
0
1
1
2
3
5
8
13
21
34
# 'python3 fibonacci_generator.py' succeeded with exit code 0.
```

Fibonacci Sequenz

- Die Funktion beginnt damit, `i` $= F_0 = 0$ und `j` $= F_1 = 1$ zu setzen.
- Danach kommt eine `while`-Schleife, die niemals aufhört, da ihre Schleifenbedingung einfach auf `True`) gesetzt ist.
- In der Schleife folgt dann zuerst ein `yield i`.
- Das bedeutet, dass die Ausführung der Funktion unterbrochen wird.
- Der Wert von `i` wird dann an den Kode außerhalb der Funktion übergeben, der über die Sequenz iteriert.

```python
"""A generator functions iterating over the Fibonacci Sequence."""

from typing import Generator  # The type hint for generators.


def fibonacci() -> Generator[int, None, None]:
    """A generator returning Fibonacci numbers."""
    i: int = 0   # Initialize i.
    j: int = 1   # Initialize j.
    while True:  # Loop forever, i.e., generator can continue forever.
        yield i  # Return the Fibonacci number.
        i, j = j, i + j  # i = old_j and j = old_i + old_j


for a in fibonacci():  # Loop over the generated sequence.
    print(a)  # Print the sequence element.
    if a > 30:  # If a > 300, then
        break    # we stop the iteration.

# list(fibonacci()) <-- This would fail!!
```

↓ `python3 fibonacci_generator.py` ↓

```
0
1
1
2
3
5
8
13
21
34
# 'python3 fibonacci_generator.py' succeeded with exit code 0.
```

Fibonacci Sequenz

- Danach kommt eine `while`-Schleife, die niemals aufhört, da ihre Schleifenbedingung einfach auf `True` gesetzt ist.
- In der Schleife folgt dann zuerst ein `yield i`.
- Das bedeutet, dass die Ausführung der Funktion unterbrochen wird.
- Der Wert von `i` wird dann an den Kode außerhalb der Funktion übergeben, der über die Sequenz iteriert.
- Wenn dieser Kode an `next` aufruft, läuft die Funktion weiter.

```python
"""A generator functions iterating over the Fibonacci Sequence."""

from typing import Generator  # The type hint for generators.


def fibonacci() -> Generator[int, None, None]:
    """A generator returning Fibonacci numbers."""
    i: int = 0   # Initialize i.
    j: int = 1   # Initialize j.
    while True:  # Loop forever, i.e., generator can continue forever.
        yield i  # Return the Fibonacci number.
        i, j = j, i + j  # i = old_j and j = old_i + old_j


for a in fibonacci():  # Loop over the generated sequence.
    print(a)  # Print the sequence element.
    if a > 30:  # If a > 300, then
        break    # we stop the iteration.

# list(fibonacci()) <-- This would fail!!
```

↓ `python3 fibonacci_generator.py` ↓

```
0
1
1
2
3
5
8
13
21
34
# 'python3 fibonacci_generator.py' succeeded with exit code 0.
```

Fibonacci Sequenz

- In der Schleife folgt dann zuerst ein `yield i`.
- Das bedeutet, dass die Ausführung der Funktion unterbrochen wird.
- Der Wert von `i` wird dann an den Kode außerhalb der Funktion übergeben, der über die Sequenz iteriert.
- Wenn dieser Kode an `next` aufruft, läuft die Funktion weiter.
- Dann weist sie `j` and `i + j` auf `i` und `j` zu.

```python
"""A generator functions iterating over the Fibonacci Sequence."""

from typing import Generator  # The type hint for generators.


def fibonacci() -> Generator[int, None, None]:
    """A generator returning Fibonacci numbers."""
    i: int = 0   # Initialize i.
    j: int = 1   # Initialize j.
    while True:  # Loop forever, i.e., generator can continue forever.
        yield i  # Return the Fibonacci number.
        i, j = j, i + j  # i = old_j and j = old_i + old_j


for a in fibonacci():  # Loop over the generated sequence.
    print(a)  # Print the sequence element.
    if a > 30:  # If a > 300, then
        break    # we stop the iteration.

# list(fibonacci()) <-- This would fail!!
```

↓ `python3 fibonacci_generator.py` ↓

```
0
1
1
2
3
5
8
13
21
34
# 'python3 fibonacci_generator.py' succeeded with exit code 0.
```

Fibonacci Sequenz

- Das bedeutet, dass die Ausführung der Funktion unterbrochen wird.
- Der Wert von `i` wird dann an den Kode außerhalb der Funktion übergeben, der über die Sequenz iteriert.
- Wenn dieser Kode an `next` aufruft, läuft die Funktion weiter.
- Dann weist sie `j` and `i + j` auf `i` und `j` zu.
- Damit wird der alte Wert von `j` in `i` gespeichert.

```python
"""A generator functions iterating over the Fibonacci Sequence."""

from typing import Generator  # The type hint for generators.


def fibonacci() -> Generator[int, None, None]:
    """A generator returning Fibonacci numbers."""
    i: int = 0   # Initialize i.
    j: int = 1   # Initialize j.
    while True:  # Loop forever, i.e., generator can continue forever.
        yield i  # Return the Fibonacci number.
        i, j = j, i + j  # i = old_j and j = old_i + old_j


for a in fibonacci():  # Loop over the generated sequence.
    print(a)  # Print the sequence element.
    if a > 30:  # If a > 300, then
        break    # we stop the iteration.

# list(fibonacci()) <-- This would fail!!
```

↓ `python3 fibonacci_generator.py` ↓

```
0
1
1
2
3
5
8
13
21
34
# 'python3 fibonacci_generator.py' succeeded with exit code 0.
```

Fibonacci Sequenz

- Der Wert von `i` wird dann an den Kode außerhalb der Funktion übergeben, der über die Sequenz iteriert.
- Wenn dieser Kode an `next` aufruft, läuft die Funktion weiter.
- Dann weist sie `j` and `i + j` auf `i` und `j` zu.
- Damit wird der alte Wert von `j` in `i` gespeichert.
- Gleichzeitig wird die Summe der alten Werte von `i` und `j` in `j` gespeichert.

```python
"""A generator functions iterating over the Fibonacci Sequence."""

from typing import Generator  # The type hint for generators.


def fibonacci() -> Generator[int, None, None]:
    """A generator returning Fibonacci numbers."""
    i: int = 0   # Initialize i.
    j: int = 1   # Initialize j.
    while True:  # Loop forever, i.e., generator can continue forever.
        yield i  # Return the Fibonacci number.
        i, j = j, i + j  # i = old_j and j = old_i + old_j


for a in fibonacci():  # Loop over the generated sequence.
    print(a)  # Print the sequence element.
    if a > 30:  # If a > 300, then
        break    # we stop the iteration.

# list(fibonacci()) <-- This would fail!!
```

↓ `python3 fibonacci_generator.py` ↓

```
0
1
1
2
3
5
8
13
21
34
# 'python3 fibonacci_generator.py' succeeded with exit code 0.
```

Fibonacci Sequenz

- Wenn dieser Kode an `next` aufruft, läuft die Funktion weiter.
- Dann weist sie `j` and `i + j` auf `i` und `j` zu.
- Damit wird der alte Wert von `j` in `i` gespeichert.
- Gleichzeitig wird die Summe der alten Werte von `i` und `j` in `j` gespeichert.
- In der nächsten Iteration wird `yield i` dann die nächste Fibonacci-Nummer zurückliefern.

```python
"""A generator functions iterating over the Fibonacci Sequence."""

from typing import Generator  # The type hint for generators.


def fibonacci() -> Generator[int, None, None]:
    """A generator returning Fibonacci numbers."""
    i: int = 0   # Initialize i.
    j: int = 1   # Initialize j.
    while True:  # Loop forever, i.e., generator can continue forever.
        yield i  # Return the Fibonacci number.
        i, j = j, i + j  # i = old_j and j = old_i + old_j


for a in fibonacci():  # Loop over the generated sequence.
    print(a)  # Print the sequence element.
    if a > 30:  # If a > 300, then
        break    # we stop the iteration.

# list(fibonacci()) <-- This would fail!!
```

↓ `python3 fibonacci_generator.py` ↓

```
0
1
1
2
3
5
8
13
21
34
# 'python3 fibonacci_generator.py' succeeded with exit code 0.
```

Fibonacci Sequenz

- Dann weist sie `j` and `i + j` auf `i` und `j` zu.
- Damit wird der alte Wert von `j` in `i` gespeichert.
- Gleichzeitig wird die Summe der alten Werte von `i` und `j` in `j` gespeichert.
- In der nächsten Iteration wird `yield i` dann die nächste Fibonacci-Nummer zurückliefern.
- Wir können nun über den `Generator`, den `fibonacci()` zurückliefert, mit einer normalen `for`-Schleife iterieren.

```python
"""A generator functions iterating over the Fibonacci Sequence."""

from typing import Generator  # The type hint for generators.


def fibonacci() -> Generator[int, None, None]:
    """A generator returning Fibonacci numbers."""
    i: int = 0   # Initialize i.
    j: int = 1   # Initialize j.
    while True:  # Loop forever, i.e., generator can continue forever.
        yield i  # Return the Fibonacci number.
        i, j = j, i + j  # i = old_j and j = old_i + old_j


for a in fibonacci():  # Loop over the generated sequence.
    print(a)  # Print the sequence element.
    if a > 30:  # If a > 300, then
        break    # we stop the iteration.

# list(fibonacci()) <-- This would fail!!
```

↓ `python3 fibonacci_generator.py` ↓

```
0
1
1
2
3
5
8
13
21
34
# 'python3 fibonacci_generator.py' succeeded with exit code 0.
```

Fibonacci Sequenz

- Damit wird der alte Wert von `j` in `i` gespeichert.
- Gleichzeitig wird die Summe der alten Werte von `i` und `j` in `j` gespeichert.
- In der nächsten Iteration wird `yield i` dann die nächste Fibonacci-Nummer zurückliefern.
- Wir können nun über den `Generator`, den `fibonacci()` zurückliefert, mit einer normalen `for`-Schleife iterieren.
- Das wird dann eine Endlosschleife werden, es sei denn, wir fügen ein zusätzliches Abbruchkriterium ein.

```python
"""A generator functions iterating over the Fibonacci Sequence."""

from typing import Generator  # The type hint for generators.


def fibonacci() -> Generator[int, None, None]:
    """A generator returning Fibonacci numbers."""
    i: int = 0   # Initialize i.
    j: int = 1   # Initialize j.
    while True:  # Loop forever, i.e., generator can continue forever.
        yield i  # Return the Fibonacci number.
        i, j = j, i + j  # i = old_j and j = old_i + old_j


for a in fibonacci():  # Loop over the generated sequence.
    print(a)  # Print the sequence element.
    if a > 30:  # If a > 300, then
        break    # we stop the iteration.

# list(fibonacci()) <-- This would fail!!
```

↓ `python3 fibonacci_generator.py` ↓

```
0
1
1
2
3
5
8
13
21
34
# 'python3 fibonacci_generator.py' succeeded with exit code 0.
```

Fibonacci Sequenz

- Gleichzeitig wird die Summe der alten Werte von `i` und `j` in `j` gespeichert.
- In der nächsten Iteration wird `yield i` dann die nächste Fibonacci-Nummer zurückliefern.
- Wir können nun über den `Generator`, den `fibonacci()` zurückliefert, mit einer normalen `for`-Schleife iterieren.
- Das wird dann eine Endlosschleife werden, es sei denn, wir fügen ein zusätzliches Abbruchkriterium ein.
- In unserer Schleife drucken wir die Fibonacci-Zahlen `a`, die wir bekommen, immer aus.

```python
"""A generator functions iterating over the Fibonacci Sequence."""

from typing import Generator  # The type hint for generators.


def fibonacci() -> Generator[int, None, None]:
    """A generator returning Fibonacci numbers."""
    i: int = 0   # Initialize i.
    j: int = 1   # Initialize j.
    while True:  # Loop forever, i.e., generator can continue forever.
        yield i  # Return the Fibonacci number.
        i, j = j, i + j  # i = old_j and j = old_i + old_j


for a in fibonacci():  # Loop over the generated sequence.
    print(a)  # Print the sequence element.
    if a > 30:  # If a > 300, then
        break    # we stop the iteration.

# list(fibonacci()) <-- This would fail!!
```

↓ `python3 fibonacci_generator.py` ↓

```
0
1
1
2
3
5
8
13
21
34
# 'python3 fibonacci_generator.py' succeeded with exit code 0.
```

Fibonacci Sequenz

- In der nächsten Iteration wird `yield i` dann die nächste Fibonacci-Nummer zurückliefern.
- Wir können nun über den `Generator`, den `fibonacci()` zurückliefert, mit einer normalen `for`-Schleife iterieren.
- Das wird dann eine Endlosschleife werden, es sei denn, wir fügen ein zusätzliches Abbruchkriterium ein.
- In unserer Schleife drucken wir die Fibonacci-Zahlen `a`, die wir bekommen, immer aus.
- Wir brechen die Iteration aber mit `break` ab, so bald `a > 30` eintritt.

```python
"""A generator functions iterating over the Fibonacci Sequence."""

from typing import Generator  # The type hint for generators.


def fibonacci() -> Generator[int, None, None]:
    """A generator returning Fibonacci numbers."""
    i: int = 0   # Initialize i.
    j: int = 1   # Initialize j.
    while True:  # Loop forever, i.e., generator can continue forever.
        yield i  # Return the Fibonacci number.
        i, j = j, i + j  # i = old_j and j = old_i + old_j


for a in fibonacci():  # Loop over the generated sequence.
    print(a)  # Print the sequence element.
    if a > 30:  # If a > 300, then
        break    # we stop the iteration.

# list(fibonacci()) <-- This would fail!!
```

↓ `python3 fibonacci_generator.py` ↓

```
0
1
1
2
3
5
8
13
21
34
# 'python3 fibonacci_generator.py' succeeded with exit code 0.
```

Fibonacci Sequenz

- Wir können nun über den `Generator`, den `fibonacci()` zurückliefert, mit einer normalen `for`-Schleife iterieren.
- Das wird dann eine Endlosschleife werden, es sei denn, wir fügen ein zusätzliches Abbruchkriterium ein.
- In unserer Schleife drucken wir die Fibonacci-Zahlen `a`, die wir bekommen, immer aus.
- Wir brechen die Iteration aber mit `break` ab, so bald `a > 30` eintritt.
- Es soll erwähnt werden, dass so etwas wie `list(fibonacci())` eine ganz schlechte Idee wäre.

```python
"""A generator functions iterating over the Fibonacci Sequence."""

from typing import Generator  # The type hint for generators.


def fibonacci() -> Generator[int, None, None]:
    """A generator returning Fibonacci numbers."""
    i: int = 0   # Initialize i.
    j: int = 1   # Initialize j.
    while True:  # Loop forever, i.e., generator can continue forever.
        yield i  # Return the Fibonacci number.
        i, j = j, i + j  # i = old_j and j = old_i + old_j


for a in fibonacci():  # Loop over the generated sequence.
    print(a)  # Print the sequence element.
    if a > 30:  # If a > 300, then
        break    # we stop the iteration.

# list(fibonacci()) <-- This would fail!!
```

↓ python3 fibonacci_generator.py ↓

```
0
1
1
2
3
5
8
13
21
34
# 'python3 fibonacci_generator.py' succeeded with exit code 0.
```

Fibonacci Sequenz

- Das wird dann eine Endlosschleife werden, es sei denn, wir fügen ein zusätzliches Abbruchkriterium ein.
- In unserer Schleife drucken wir die Fibonacci-Zahlen `a`, die wir bekommen, immer aus.
- Wir brechen die Iteration aber mit `break` ab, so bald `a > 30` eintritt.
- Es soll erwähnt werden, dass so etwas wie `list(fibonacci())` eine ganz schlechte Idee wäre.
- Es würde versuchen, eine unendlich große Liste zu produzieren, was dann unweigerlich zu einer `MemoryError`-Ausnahme führt.

```python
"""A generator functions iterating over the Fibonacci Sequence."""

from typing import Generator  # The type hint for generators.


def fibonacci() -> Generator[int, None, None]:
    """A generator returning Fibonacci numbers."""
    i: int = 0   # Initialize i.
    j: int = 1   # Initialize j.
    while True:  # Loop forever, i.e., generator can continue forever.
        yield i  # Return the Fibonacci number.
        i, j = j, i + j  # i = old_j and j = old_i + old_j


for a in fibonacci():  # Loop over the generated sequence.
    print(a)  # Print the sequence element.
    if a > 30:  # If a > 300, then
        break    # we stop the iteration.

# list(fibonacci()) <-- This would fail!!
```

↓ `python3 fibonacci_generator.py` ↓

```
0
1
1
2
3
5
8
13
21
34
# 'python3 fibonacci_generator.py' succeeded with exit code 0.
```

Primzahlen generieren

- Als letztes Beispiel wollen wir nun den Kode um Primzahlen[4,20,34] aufzuzählen aus Einheit 23 in eine Generator-Funktion gießen.

```python
"""A generator function for prime numbers."""

from math import isqrt  # The integer square root function.
from typing import Generator  # The type hint for generators.


def primes() -> Generator[int, None, None]:
    """
    Provide a sequence of prime numbers.

    >>> gen = primes()
    >>> next(gen)
    2
    >>> next(gen)
    3
    >>> next(gen)
    5
    >>> next(gen)
    7
    >>> next(gen)
    11
    """
    yield 2  # The first and only even prime number.

    found: list[int] = []  # The list of already discovered primes.
    candidate: int = 1     # The current prime candidate
    while True:            # Loop over candidates.
        candidate += 2  # Move to the next odd number as prime candidate
        is_prime: bool = True  # Let us assume that `candidate` is prime
        limit: int = isqrt(candidate)  # Get maximum possible divisor.
        for check in found:  # We only test with the odd primes we got.
            if check > limit:  # If the potential divisor is too big,
                break          # then we can stop the inner loop here.
            if candidate % check == 0:  # division without remainder
                is_prime = False  # check divides candidate evenly, so
                break  # candidate is not a prime. Stop the inner loop.

        if is_prime:  # If True: no smaller number divides candidate.
            yield candidate  # Return the prime number as next element.
            found.append(candidate)  # Store candidate in primes list.
```

Primzahlen generieren

- Als letztes Beispiel wollen wir nun den Kode um Primzahlen[4,20,34] aufzuzählen aus Einheit 23 in eine Generator-Funktion gießen.
- Wir tun das in Programm `prime_generator.py`.

```python
"""A generator function for prime numbers."""

from math import isqrt  # The integer square root function.
from typing import Generator  # The type hint for generators.


def primes() -> Generator[int, None, None]:
    """
    Provide a sequence of prime numbers.

    >>> gen = primes()
    >>> next(gen)
    2
    >>> next(gen)
    3
    >>> next(gen)
    5
    >>> next(gen)
    7
    >>> next(gen)
    11
    """
    yield 2  # The first and only even prime number.

    found: list[int] = []  # The list of already discovered primes.
    candidate: int = 1     # The current prime candidate
    while True:            # Loop over candidates.
        candidate += 2  # Move to the next odd number as prime candidate
        is_prime: bool = True  # Let us assume that `candidate` is prime
        limit: int = isqrt(candidate)  # Get maximum possible divisor.
        for check in found:  # We only test with the odd primes we got.
            if check > limit:  # If the potential divisor is too big,
                break          # then we can stop the inner loop here.
            if candidate % check == 0:  # division without remainder
                is_prime = False  # check divides candidate evenly, so
                break  # candidate is not a prime. Stop the inner loop.

        if is_prime:  # If True: no smaller number divides candidate.
            yield candidate  # Return the prime number as next element.
            found.append(candidate)  # Store candidate in primes list.
```

Primzahlen generieren

- Als letztes Beispiel wollen wir nun den Kode um Primzahlen[4,20,34] aufzuzählen aus Einheit 23 in eine Generator-Funktion gießen.
- Wir tun das in Programm `prime_generator.py`.
- Damals hatten wir Primzahlen mit zwei verschachtelten `for`-Schleifen produziert.

```
"""A generator function for prime numbers."""

from math import isqrt  # The integer square root function.
from typing import Generator  # The type hint for generators.


def primes() -> Generator[int, None, None]:
    """
    Provide a sequence of prime numbers.

    >>> gen = primes()
    >>> next(gen)
    2
    >>> next(gen)
    3
    >>> next(gen)
    5
    >>> next(gen)
    7
    >>> next(gen)
    11
    """
    yield 2  # The first and only even prime number.

    found: list[int] = []  # The list of already discovered primes.
    candidate: int = 1     # The current prime candidate
    while True:            # Loop over candidates.
        candidate += 2  # Move to the next odd number as prime candidate
        is_prime: bool = True  # Let us assume that `candidate` is prime
        limit: int = isqrt(candidate)  # Get maximum possible divisor.
        for check in found:  # We only test with the odd primes we got.
            if check > limit:  # If the potential divisor is too big,
                break          # then we can stop the inner loop here.
            if candidate % check == 0:  # division without remainder
                is_prime = False  # check divides candidate evenly, so
                break  # candidate is not a prime. Stop the inner loop.

        if is_prime:  # If True: no smaller number divides candidate.
            yield candidate  # Return the prime number as next element.
            found.append(candidate)  # Store candidate in primes list.
```

Primzahlen generieren

- Als letztes Beispiel wollen wir nun den Kode um Primzahlen[4,20,34] aufzuzählen aus Einheit 23 in eine Generator-Funktion gießen.
- Wir tun das in Programm `prime_generator.py`.
- Damals hatten wir Primzahlen mit zwei verschachtelten `for`-Schleifen produziert.
- Wir hatten die äußere Schleife bis maximal 199 iterieren lassen.

```python
"""A generator function for prime numbers."""

from math import isqrt  # The integer square root function.
from typing import Generator  # The type hint for generators.


def primes() -> Generator[int, None, None]:
    """
    Provide a sequence of prime numbers.

    >>> gen = primes()
    >>> next(gen)
    2
    >>> next(gen)
    3
    >>> next(gen)
    5
    >>> next(gen)
    7
    >>> next(gen)
    11
    """
    yield 2  # The first and only even prime number.

    found: list[int] = []  # The list of already discovered primes.
    candidate: int = 1     # The current prime candidate
    while True:            # Loop over candidates.
        candidate += 2  # Move to the next odd number as prime candidate
        is_prime: bool = True  # Let us assume that `candidate` is prime
        limit: int = isqrt(candidate)  # Get maximum possible divisor.
        for check in found:  # We only test with the odd primes we got.
            if check > limit:  # If the potential divisor is too big,
                break          # then we can stop the inner loop here.
            if candidate % check == 0:  # division without remainder
                is_prime = False  # check divides candidate evenly, so
                break  # candidate is not a prime. Stop the inner loop.

        if is_prime:  # If True: no smaller number divides candidate.
            yield candidate  # Return the prime number as next element.
            found.append(candidate)  # Store candidate in primes list.
```

Primzahlen generieren

- Als letztes Beispiel wollen wir nun den Kode um Primzahlen[4,20,34] aufzuzählen aus Einheit 23 in eine Generator-Funktion gießen.
- Wir tun das in Programm `prime_generator.py`.
- Damals hatten wir Primzahlen mit zwei verschachtelten `for`-Schleifen produziert.
- Wir hatten die äußere Schleife bis maximal 199 iterieren lassen.
- Jetzt brauchen wir so eine Begrenzung nicht mehr.

```
"""A generator function for prime numbers."""

from math import isqrt  # The integer square root function.
from typing import Generator  # The type hint for generators.


def primes() -> Generator[int, None, None]:
    """
    Provide a sequence of prime numbers.

    >>> gen = primes()
    >>> next(gen)
    2
    >>> next(gen)
    3
    >>> next(gen)
    5
    >>> next(gen)
    7
    >>> next(gen)
    11
    """
    yield 2  # The first and only even prime number.

    found: list[int] = []  # The list of already discovered primes.
    candidate: int = 1     # The current prime candidate
    while True:            # Loop over candidates.
        candidate += 2  # Move to the next odd number as prime candidate
        is_prime: bool = True  # Let us assume that `candidate` is prime
        limit: int = isqrt(candidate)  # Get maximum possible divisor.
        for check in found:  # We only test with the odd primes we got.
            if check > limit:  # If the potential divisor is too big,
                break          # then we can stop the inner loop here.
            if candidate % check == 0:  # division without remainder
                is_prime = False  # check divides candidate evenly, so
                break  # candidate is not a prime. Stop the inner loop.

        if is_prime:  # If True: no smaller number divides candidate.
            yield candidate  # Return the prime number as next element.
            found.append(candidate)  # Store candidate in primes list.
```

Primzahlen generieren

- Wir tun das in Programm `prime_generator.py`.
- Damals hatten wir Primzahlen mit zwei verschachtelten `for`-Schleifen produziert.
- Wir hatten die äußere Schleife bis maximal 199 iterieren lassen.
- Jetzt brauchen wir so eine Begrenzung nicht mehr.
- Wir können einfach annehmen, dass wer auch immer unsere Generator-Funktion verwendet, einfach aufhören wird, die Zahlen zu iterieren, wenn sie genug Primzahlen haben.

```python
"""A generator function for prime numbers."""

from math import isqrt  # The integer square root function.
from typing import Generator  # The type hint for generators.


def primes() -> Generator[int, None, None]:
    """
    Provide a sequence of prime numbers.

    >>> gen = primes()
    >>> next(gen)
    2
    >>> next(gen)
    3
    >>> next(gen)
    5
    >>> next(gen)
    7
    >>> next(gen)
    11
    """
    yield 2  # The first and only even prime number.

    found: list[int] = []  # The list of already discovered primes.
    candidate: int = 1     # The current prime candidate
    while True:            # Loop over candidates.
        candidate += 2  # Move to the next odd number as prime candidate
        is_prime: bool = True  # Let us assume that `candidate` is prime
        limit: int = isqrt(candidate)  # Get maximum possible divisor.
        for check in found:  # We only test with the odd primes we got.
            if check > limit:  # If the potential divisor is too big,
                break          # then we can stop the inner loop here.
            if candidate % check == 0:  # division without remainder
                is_prime = False  # check divides candidate evenly, so
                break  # candidate is not a prime. Stop the inner loop.

        if is_prime:  # If True: no smaller number divides candidate.
            yield candidate  # Return the prime number as next element.
            found.append(candidate)  # Store candidate in primes list.
```

Primzahlen generieren

- Damals hatten wir Primzahlen mit zwei verschachtelten `for`-Schleifen produziert.
- Wir hatten die äußere Schleife bis maximal 199 iterieren lassen.
- Jetzt brauchen wir so eine Begrenzung nicht mehr.
- Wir können einfach annehmen, dass wer auch immer unsere Generator-Funktion verwendet, einfach aufhören wird, die Zahlen zu iterieren, wenn sie genug Primzahlen haben.
- Wir nehmen deshalb nun eine `while True`-Schleife.

```python
"""A generator function for prime numbers."""

from math import isqrt  # The integer square root function.
from typing import Generator  # The type hint for generators.


def primes() -> Generator[int, None, None]:
    """
    Provide a sequence of prime numbers.

    >>> gen = primes()
    >>> next(gen)
    2
    >>> next(gen)
    3
    >>> next(gen)
    5
    >>> next(gen)
    7
    >>> next(gen)
    11
    """
    yield 2  # The first and only even prime number.

    found: list[int] = []  # The list of already discovered primes.
    candidate: int = 1     # The current prime candidate
    while True:            # Loop over candidates.
        candidate += 2  # Move to the next odd number as prime candidate
        is_prime: bool = True  # Let us assume that `candidate` is prime
        limit: int = isqrt(candidate)  # Get maximum possible divisor.
        for check in found:  # We only test with the odd primes we got.
            if check > limit:  # If the potential divisor is too big,
                break          # then we can stop the inner loop here.
            if candidate % check == 0:  # division without remainder
                is_prime = False  # check divides candidate evenly, so
                break  # candidate is not a prime. Stop the inner loop.

        if is_prime:  # If True: no smaller number divides candidate.
            yield candidate  # Return the prime number as next element.
            found.append(candidate)  # Store candidate in primes list.
```

Primzahlen generieren

- Wir hatten die äußere Schleife bis maximal 199 iterieren lassen.
- Jetzt brauchen wir so eine Begrenzung nicht mehr.
- Wir können einfach annehmen, dass wer auch immer unsere Generator-Funktion verwendet, einfach aufhören wird, die Zahlen zu iterieren, wenn sie genug Primzahlen haben.
- Wir nehmen deshalb nun eine `while True`-Schleife.
- Wir müssen auch keine Liste mehr zurückliefern, wodurch wir die einzige gerade Primzahl (2) nicht mehr speichern müssen.

```python
"""A generator function for prime numbers."""

from math import isqrt  # The integer square root function.
from typing import Generator  # The type hint for generators.


def primes() -> Generator[int, None, None]:
    """
    Provide a sequence of prime numbers.

    >>> gen = primes()
    >>> next(gen)
    2
    >>> next(gen)
    3
    >>> next(gen)
    5
    >>> next(gen)
    7
    >>> next(gen)
    11
    """
    yield 2  # The first and only even prime number.

    found: list[int] = []  # The list of already discovered primes.
    candidate: int = 1     # The current prime candidate
    while True:            # Loop over candidates.
        candidate += 2  # Move to the next odd number as prime candidate
        is_prime: bool = True  # Let us assume that `candidate` is prime
        limit: int = isqrt(candidate)  # Get maximum possible divisor.
        for check in found:  # We only test with the odd primes we got.
            if check > limit:  # If the potential divisor is too big,
                break          # then we can stop the inner loop here.
            if candidate % check == 0:  # division without remainder
                is_prime = False  # check divides candidate evenly, so
                break  # candidate is not a prime. Stop the inner loop.

        if is_prime:  # If True: no smaller number divides candidate.
            yield candidate  # Return the prime number as next element.
            found.append(candidate)  # Store candidate in primes list.
```

Primzahlen generieren

- Jetzt brauchen wir so eine Begrenzung nicht mehr.
- Wir können einfach annehmen, dass wer auch immer unsere Generator-Funktion verwendet, einfach aufhören wird, die Zahlen zu iterieren, wenn sie genug Primzahlen haben.
- Wir nehmen deshalb nun eine `while True`-Schleife.
- Wir müssen auch keine Liste mehr zurückliefern, wodurch wir die einzige gerade Primzahl (2) nicht mehr speichern müssen.
- Stattdessen machen wir einfach `yield 2` am Anfang unseres Kodes.

```python
"""A generator function for prime numbers."""

from math import isqrt  # The integer square root function.
from typing import Generator  # The type hint for generators.


def primes() -> Generator[int, None, None]:
    """
    Provide a sequence of prime numbers.

    >>> gen = primes()
    >>> next(gen)
    2
    >>> next(gen)
    3
    >>> next(gen)
    5
    >>> next(gen)
    7
    >>> next(gen)
    11
    """
    yield 2  # The first and only even prime number.

    found: list[int] = []  # The list of already discovered primes.
    candidate: int = 1     # The current prime candidate
    while True:            # Loop over candidates.
        candidate += 2  # Move to the next odd number as prime candidate
        is_prime: bool = True  # Let us assume that `candidate` is prime
        limit: int = isqrt(candidate)  # Get maximum possible divisor.
        for check in found:  # We only test with the odd primes we got.
            if check > limit:  # If the potential divisor is too big,
                break          # then we can stop the inner loop here.
            if candidate % check == 0:  # division without remainder
                is_prime = False  # check divides candidate evenly, so
                break  # candidate is not a prime. Stop the inner loop.

        if is_prime:  # If True: no smaller number divides candidate.
            yield candidate  # Return the prime number as next element.
            found.append(candidate)  # Store candidate in primes list.
```

Primzahlen generieren

- Wir nehmen deshalb nun eine `while True`-Schleife.
- Wir müssen auch keine Liste mehr zurückliefern, wodurch wir die einzige gerade Primzahl (2) nicht mehr speichern müssen.
- Stattdessen machen wir einfach `yield 2` am Anfang unseres Kodes.
- Der letzte wichtige Unterschied zwischen dem alten und dem neuen Kode ist das, nachdem wir eine Zahl als Primzahl bestätigt haben, wir diese nicht nur an die Liste `found` der ungeraden Primzahlen anhängen, sondern sie auch per `yield` zurückgeben.

```python
"""A generator function for prime numbers."""

from math import isqrt  # The integer square root function.
from typing import Generator  # The type hint for generators.


def primes() -> Generator[int, None, None]:
    """
    Provide a sequence of prime numbers.

    >>> gen = primes()
    >>> next(gen)
    2
    >>> next(gen)
    3
    >>> next(gen)
    5
    >>> next(gen)
    7
    >>> next(gen)
    11
    """
    yield 2  # The first and only even prime number.

    found: list[int] = []  # The list of already discovered primes.
    candidate: int = 1     # The current prime candidate
    while True:            # Loop over candidates.
        candidate += 2  # Move to the next odd number as prime candidate
        is_prime: bool = True  # Let us assume that `candidate` is prime
        limit: int = isqrt(candidate)  # Get maximum possible divisor.
        for check in found:  # We only test with the odd primes we got.
            if check > limit:  # If the potential divisor is too big,
                break          # then we can stop the inner loop here.
            if candidate % check == 0:  # division without remainder
                is_prime = False  # check divides candidate evenly, so
                break  # candidate is not a prime. Stop the inner loop.

        if is_prime:  # If True: no smaller number divides candidate.
            yield candidate  # Return the prime number as next element.
            found.append(candidate)  # Store candidate in primes list.
```

Primzahlen generieren

- Stattdessen machen wir einfach `yield 2` am Anfang unseres Kodes.
- Der letzte wichtige Unterschied zwischen dem alten und dem neuen Kode ist das, nachdem wir eine Zahl als Primzahl bestätigt haben, wir diese nicht nur an die Liste `found` der ungeraden Primzahlen anhängen, sondern sie auch per `yield` zurückgeben.
- Davon abgesehen funktioniert unsere Funktion im Grunde genauso wie damals.

```python
"""A generator function for prime numbers."""

from math import isqrt  # The integer square root function.
from typing import Generator  # The type hint for generators.


def primes() -> Generator[int, None, None]:
    """
    Provide a sequence of prime numbers.

    >>> gen = primes()
    >>> next(gen)
    2
    >>> next(gen)
    3
    >>> next(gen)
    5
    >>> next(gen)
    7
    >>> next(gen)
    11
    """
    yield 2  # The first and only even prime number.

    found: list[int] = []  # The list of already discovered primes.
    candidate: int = 1     # The current prime candidate
    while True:            # Loop over candidates.
        candidate += 2  # Move to the next odd number as prime candidate
        is_prime: bool = True  # Let us assume that `candidate` is prime
        limit: int = isqrt(candidate)  # Get maximum possible divisor.
        for check in found:  # We only test with the odd primes we got.
            if check > limit:  # If the potential divisor is too big,
                break          # then we can stop the inner loop here.
            if candidate % check == 0:  # division without remainder
                is_prime = False  # check divides candidate evenly, so
                break  # candidate is not a prime. Stop the inner loop.

        if is_prime:  # If True: no smaller number divides candidate.
            yield candidate  # Return the prime number as next element.
            found.append(candidate)  # Store candidate in primes list.
```

Primzahlen generieren

- Stattdessen machen wir einfach `yield 2` am Anfang unseres Kodes.
- Der letzte wichtige Unterschied zwischen dem alten und dem neuen Kode ist das, nachdem wir eine Zahl als Primzahl bestätigt haben, wir diese nicht nur an die Liste `found` der ungeraden Primzahlen anhängen, sondern sie auch per `yield` zurückgeben.
- Davon abgesehen funktioniert unsere Funktion im Grunde genauso wie damals.
- Wir iterieren über die ungeraden Zahlen `candidate`, die Primzahlen seien könnten.

```python
"""A generator function for prime numbers."""

from math import isqrt  # The integer square root function.
from typing import Generator  # The type hint for generators.


def primes() -> Generator[int, None, None]:
    """
    Provide a sequence of prime numbers.

    >>> gen = primes()
    >>> next(gen)
    2
    >>> next(gen)
    3
    >>> next(gen)
    5
    >>> next(gen)
    7
    >>> next(gen)
    11
    """
    yield 2  # The first and only even prime number.

    found: list[int] = []  # The list of already discovered primes.
    candidate: int = 1     # The current prime candidate
    while True:            # Loop over candidates.
        candidate += 2  # Move to the next odd number as prime candidate
        is_prime: bool = True  # Let us assume that `candidate` is prime
        limit: int = isqrt(candidate)  # Get maximum possible divisor.
        for check in found:  # We only test with the odd primes we got.
            if check > limit:  # If the potential divisor is too big,
                break          # then we can stop the inner loop here.
            if candidate % check == 0:  # division without remainder
                is_prime = False  # check divides candidate evenly, so
                break  # candidate is not a prime. Stop the inner loop.

        if is_prime:  # If True: no smaller number divides candidate.
            yield candidate  # Return the prime number as next element.
            found.append(candidate)  # Store candidate in primes list.
```

Primzahlen generieren

- Davon abgesehen funktioniert unsere Funktion im Grunde genauso wie damals.
- Wir iterieren über die ungeraden Zahlen `candidate`, die Primzahlen seien könnten.
- Für jeden `candidate` testen wir alle vorher identifizierten Primzahlen `check` (außer 2, natürlich), ob sie `candidate` mit Rest 0 teilen können.

```python
"""A generator function for prime numbers."""

from math import isqrt  # The integer square root function.
from typing import Generator  # The type hint for generators.


def primes() -> Generator[int, None, None]:
    """
    Provide a sequence of prime numbers.

    >>> gen = primes()
    >>> next(gen)
    2
    >>> next(gen)
    3
    >>> next(gen)
    5
    >>> next(gen)
    7
    >>> next(gen)
    11
    """
    yield 2  # The first and only even prime number.

    found: list[int] = []  # The list of already discovered primes.
    candidate: int = 1     # The current prime candidate
    while True:            # Loop over candidates.
        candidate += 2  # Move to the next odd number as prime candidate
        is_prime: bool = True  # Let us assume that `candidate` is prime
        limit: int = isqrt(candidate)  # Get maximum possible divisor.
        for check in found:  # We only test with the odd primes we got.
            if check > limit:  # If the potential divisor is too big,
                break          # then we can stop the inner loop here.
            if candidate % check == 0:  # division without remainder
                is_prime = False  # check divides candidate evenly, so
                break  # candidate is not a prime. Stop the inner loop.

        if is_prime:  # If True: no smaller number divides candidate.
            yield candidate  # Return the prime number as next element.
            found.append(candidate)  # Store candidate in primes list.
```

Primzahlen generieren

- Davon abgesehen funktioniert unsere Funktion im Grunde genauso wie damals.
- Wir iterieren über die ungeraden Zahlen `candidate`, die Primzahlen seien könnten.
- Für jeden `candidate` testen wir alle vorher identifizierten Primzahlen `check` (außer 2, natürlich), ob sie `candidate` mit Rest 0 teilen können.
- Bei der ersten solchen Zahl, auf die das zutrifft, wissen wir das `candidate` keine Primzahl seien kann.

```python
"""A generator function for prime numbers."""

from math import isqrt  # The integer square root function.
from typing import Generator  # The type hint for generators.


def primes() -> Generator[int, None, None]:
    """
    Provide a sequence of prime numbers.

    >>> gen = primes()
    >>> next(gen)
    2
    >>> next(gen)
    3
    >>> next(gen)
    5
    >>> next(gen)
    7
    >>> next(gen)
    11
    """
    yield 2  # The first and only even prime number.

    found: list[int] = []  # The list of already discovered primes.
    candidate: int = 1     # The current prime candidate
    while True:            # Loop over candidates.
        candidate += 2  # Move to the next odd number as prime candidate
        is_prime: bool = True  # Let us assume that `candidate` is prime
        limit: int = isqrt(candidate)  # Get maximum possible divisor.
        for check in found:  # We only test with the odd primes we got.
            if check > limit:  # If the potential divisor is too big,
                break          # then we can stop the inner loop here.
            if candidate % check == 0:  # division without remainder
                is_prime = False  # check divides candidate evenly, so
                break  # candidate is not a prime. Stop the inner loop.

        if is_prime:  # If True: no smaller number divides candidate.
            yield candidate  # Return the prime number as next element.
            found.append(candidate)  # Store candidate in primes list.
```

Primzahlen generieren

- Wir iterieren über die ungeraden Zahlen `candidate`, die Primzahlen seien könnten.
- Für jeden `candidate` testen wir alle vorher identifizierten Primzahlen `check` (außer 2, natürlich), ob sie `candidate` mit Rest 0 teilen können.
- Bei der ersten solchen Zahl, auf die das zutrifft, wissen wir das `candidate` keine Primzahl seien kann.
- Wenn keine Zahl `check` existiert, die ein Teiler von `candidate` ist, dann ist `candidate` eine Primzahl.

```python
"""A generator function for prime numbers."""

from math import isqrt  # The integer square root function.
from typing import Generator  # The type hint for generators.


def primes() -> Generator[int, None, None]:
    """
    Provide a sequence of prime numbers.

    >>> gen = primes()
    >>> next(gen)
    2
    >>> next(gen)
    3
    >>> next(gen)
    5
    >>> next(gen)
    7
    >>> next(gen)
    11
    """
    yield 2  # The first and only even prime number.

    found: list[int] = []  # The list of already discovered primes.
    candidate: int = 1     # The current prime candidate
    while True:            # Loop over candidates.
        candidate += 2  # Move to the next odd number as prime candidate
        is_prime: bool = True  # Let us assume that `candidate` is prime
        limit: int = isqrt(candidate)  # Get maximum possible divisor.
        for check in found:  # We only test with the odd primes we got.
            if check > limit:  # If the potential divisor is too big,
                break          # then we can stop the inner loop here.
            if candidate % check == 0:  # division without remainder
                is_prime = False  # check divides candidate evenly, so
                break  # candidate is not a prime. Stop the inner loop.

        if is_prime:  # If True: no smaller number divides candidate.
            yield candidate  # Return the prime number as next element.
            found.append(candidate)  # Store candidate in primes list.
```

Primzahlen generieren

- Für jeden `candidate` testen wir alle vorher identifizierten Primzahlen `check` (außer 2, natürlich), ob sie `candidate` mit Rest 0 teilen können.
- Bei der ersten solchen Zahl, auf die das zutrifft, wissen wir das `candidate` keine Primzahl seien kann.
- Wenn keine Zahl `check` existiert, die ein Teiler von `candidate` ist, dann ist `candidate` eine Primzahl.
- Wir müssen aber nur Zahlen `check` prüfen, die kleiner oder gleich `isqrt(candidate)` sind, also $\leq \lfloor \sqrt{\texttt{candidate}} \rfloor$.

```python
"""A generator function for prime numbers."""

from math import isqrt  # The integer square root function.
from typing import Generator  # The type hint for generators.


def primes() -> Generator[int, None, None]:
    """
    Provide a sequence of prime numbers.

    >>> gen = primes()
    >>> next(gen)
    2
    >>> next(gen)
    3
    >>> next(gen)
    5
    >>> next(gen)
    7
    >>> next(gen)
    11
    """
    yield 2  # The first and only even prime number.

    found: list[int] = []  # The list of already discovered primes.
    candidate: int = 1     # The current prime candidate
    while True:            # Loop over candidates.
        candidate += 2  # Move to the next odd number as prime candidate
        is_prime: bool = True  # Let us assume that `candidate` is prime
        limit: int = isqrt(candidate)  # Get maximum possible divisor.
        for check in found:  # We only test with the odd primes we got.
            if check > limit:  # If the potential divisor is too big,
                break          # then we can stop the inner loop here.
            if candidate % check == 0:  # division without remainder
                is_prime = False  # check divides candidate evenly, so
                break  # candidate is not a prime. Stop the inner loop.

        if is_prime:  # If True: no smaller number divides candidate.
            yield candidate  # Return the prime number as next element.
            found.append(candidate)  # Store candidate in primes list.
```

Primzahlen generieren

- Bei der ersten solchen Zahl, auf die das zutrifft, wissen wir das `candidate` keine Primzahl seien kann.
- Wenn keine Zahl `check` existiert, die ein Teiler von `candidate` ist, dann ist `candidate` eine Primzahl.
- Wir müssen aber nur Zahlen `check` prüfen, die kleiner oder gleich `isqrt(candidate)` sind, also $\leq \lfloor \sqrt{\texttt{candidate}} \rfloor$.
- Zahlen, die größer als das sind, brauchen nicht geprüft werden.

```python
"""A generator function for prime numbers."""

from math import isqrt  # The integer square root function.
from typing import Generator  # The type hint for generators.


def primes() -> Generator[int, None, None]:
    """
    Provide a sequence of prime numbers.

    >>> gen = primes()
    >>> next(gen)
    2
    >>> next(gen)
    3
    >>> next(gen)
    5
    >>> next(gen)
    7
    >>> next(gen)
    11
    """
    yield 2  # The first and only even prime number.

    found: list[int] = []  # The list of already discovered primes.
    candidate: int = 1     # The current prime candidate
    while True:            # Loop over candidates.
        candidate += 2  # Move to the next odd number as prime candidate
        is_prime: bool = True  # Let us assume that `candidate` is prime
        limit: int = isqrt(candidate)  # Get maximum possible divisor.
        for check in found:  # We only test with the odd primes we got.
            if check > limit:  # If the potential divisor is too big,
                break          # then we can stop the inner loop here.
            if candidate % check == 0:  # division without remainder
                is_prime = False  # check divides candidate evenly, so
                break  # candidate is not a prime. Stop the inner loop.

        if is_prime:  # If True: no smaller number divides candidate.
            yield candidate  # Return the prime number as next element.
            found.append(candidate)  # Store candidate in primes list.
```

Primzahlen generieren

- Bei der ersten solchen Zahl, auf die das zutrifft, wissen wir das `candidate` keine Primzahl seien kann.
- Wenn keine Zahl `check` existiert, die ein Teiler von `candidate` ist, dann ist `candidate` eine Primzahl.
- Wir müssen aber nur Zahlen `check` prüfen, die kleiner oder gleich `isqrt(candidate)` sind, also $\leq \lfloor \sqrt{\texttt{candidate}} \rfloor$.
- Zahlen, die größer als das sind, brauchen nicht geprüft werden.
- Sie würden ja zu einem Quotient kleiner als sie selbst führen – und den hätten wir ja vorher schon probiert.

```python
"""A generator function for prime numbers."""

from math import isqrt  # The integer square root function.
from typing import Generator  # The type hint for generators.


def primes() -> Generator[int, None, None]:
    """
    Provide a sequence of prime numbers.

    >>> gen = primes()
    >>> next(gen)
    2
    >>> next(gen)
    3
    >>> next(gen)
    5
    >>> next(gen)
    7
    >>> next(gen)
    11
    """
    yield 2  # The first and only even prime number.

    found: list[int] = []  # The list of already discovered primes.
    candidate: int = 1     # The current prime candidate
    while True:            # Loop over candidates.
        candidate += 2  # Move to the next odd number as prime candidate
        is_prime: bool = True  # Let us assume that `candidate` is prime
        limit: int = isqrt(candidate)  # Get maximum possible divisor.
        for check in found:  # We only test with the odd primes we got.
            if check > limit:  # If the potential divisor is too big,
                break          # then we can stop the inner loop here.
            if candidate % check == 0:  # division without remainder
                is_prime = False  # check divides candidate evenly, so
                break  # candidate is not a prime. Stop the inner loop.

        if is_prime:  # If True: no smaller number divides candidate.
            yield candidate  # Return the prime number as next element.
            found.append(candidate)  # Store candidate in primes list.
```

Primzahlen generieren

- Wenn keine Zahl `check` existiert, die ein Teiler von `candidate` ist, dann ist `candidate` eine Primzahl.
- Wir müssen aber nur Zahlen `check` prüfen, die kleiner oder gleich `isqrt(candidate)` sind, also $\leq \lfloor \sqrt{\texttt{candidate}} \rfloor$.
- Zahlen, die größer als das sind, brauchen nicht geprüft werden.
- Sie würden ja zu einem Quotient kleiner als sie selbst führen – und den hätten wir ja vorher schon probiert.
- Unsere Funktion wird also eine Primzahl nach der anderen finden via `yield` zurückliefern.

```python
"""A generator function for prime numbers."""

from math import isqrt  # The integer square root function.
from typing import Generator  # The type hint for generators.


def primes() -> Generator[int, None, None]:
    """
    Provide a sequence of prime numbers.

    >>> gen = primes()
    >>> next(gen)
    2
    >>> next(gen)
    3
    >>> next(gen)
    5
    >>> next(gen)
    7
    >>> next(gen)
    11
    """
    yield 2  # The first and only even prime number.

    found: list[int] = []  # The list of already discovered primes.
    candidate: int = 1     # The current prime candidate
    while True:            # Loop over candidates.
        candidate += 2  # Move to the next odd number as prime candidate
        is_prime: bool = True  # Let us assume that `candidate` is prime
        limit: int = isqrt(candidate)  # Get maximum possible divisor.
        for check in found:  # We only test with the odd primes we got.
            if check > limit:  # If the potential divisor is too big,
                break          # then we can stop the inner loop here.
            if candidate % check == 0:  # division without remainder
                is_prime = False  # check divides candidate evenly, so
                break  # candidate is not a prime. Stop the inner loop.

        if is_prime:  # If True: no smaller number divides candidate.
            yield candidate  # Return the prime number as next element.
            found.append(candidate)  # Store candidate in primes list.
```

Primzahlen generieren

- Wir müssen aber nur Zahlen `check` prüfen, die kleiner oder gleich `isqrt(candidate)` sind, also $\leq \lfloor \sqrt{\texttt{candidate}} \rfloor$.
- Zahlen, die größer als das sind, brauchen nicht geprüft werden.
- Sie würden ja zu einem Quotient kleiner als sie selbst führen – und den hätten wir ja vorher schon probiert.
- Unsere Funktion wird also eine Primzahl nach der anderen finden via `yield` zurückliefern.
- Jedes Mal, wenn wir mit `yield` einen Wert zurückliefern, wird die Ausführung unsere Funktion unterbrochen.

```python
"""A generator function for prime numbers."""

from math import isqrt  # The integer square root function.
from typing import Generator  # The type hint for generators.


def primes() -> Generator[int, None, None]:
    """
    Provide a sequence of prime numbers.

    >>> gen = primes()
    >>> next(gen)
    2
    >>> next(gen)
    3
    >>> next(gen)
    5
    >>> next(gen)
    7
    >>> next(gen)
    11
    """
    yield 2  # The first and only even prime number.

    found: list[int] = []  # The list of already discovered primes.
    candidate: int = 1     # The current prime candidate
    while True:            # Loop over candidates.
        candidate += 2  # Move to the next odd number as prime candidate
        is_prime: bool = True  # Let us assume that `candidate` is prime
        limit: int = isqrt(candidate)  # Get maximum possible divisor.
        for check in found:  # We only test with the odd primes we got.
            if check > limit:  # If the potential divisor is too big,
                break          # then we can stop the inner loop here.
            if candidate % check == 0:  # division without remainder
                is_prime = False  # check divides candidate evenly, so
                break  # candidate is not a prime. Stop the inner loop.

        if is_prime:  # If True: no smaller number divides candidate.
            yield candidate  # Return the prime number as next element.
            found.append(candidate)  # Store candidate in primes list.
```

Primzahlen generieren

- Zahlen, die größer als das sind, brauchen nicht geprüft werden.
- Sie würden ja zu einem Quotient kleiner als sie selbst führen – und den hätten wir ja vorher schon probiert.
- Unsere Funktion wird also eine Primzahl nach der anderen finden via `yield` zurückliefern.
- Jedes Mal, wenn wir mit `yield` einen Wert zurückliefern, wird die Ausführung unsere Funktion unterbrochen.
- Das passiert zum ersten Mal wenn wir `yield 2` aufrufen.

```python
"""A generator function for prime numbers."""

from math import isqrt  # The integer square root function.
from typing import Generator  # The type hint for generators.


def primes() -> Generator[int, None, None]:
    """
    Provide a sequence of prime numbers.

    >>> gen = primes()
    >>> next(gen)
    2
    >>> next(gen)
    3
    >>> next(gen)
    5
    >>> next(gen)
    7
    >>> next(gen)
    11
    """
    yield 2  # The first and only even prime number.

    found: list[int] = []  # The list of already discovered primes.
    candidate: int = 1     # The current prime candidate
    while True:            # Loop over candidates.
        candidate += 2  # Move to the next odd number as prime candidate
        is_prime: bool = True  # Let us assume that `candidate` is prime
        limit: int = isqrt(candidate)  # Get maximum possible divisor.
        for check in found:  # We only test with the odd primes we got.
            if check > limit:  # If the potential divisor is too big,
                break          # then we can stop the inner loop here.
            if candidate % check == 0:  # division without remainder
                is_prime = False  # check divides candidate evenly, so
                break  # candidate is not a prime. Stop the inner loop.

        if is_prime:  # If True: no smaller number divides candidate.
            yield candidate  # Return the prime number as next element.
            found.append(candidate)  # Store candidate in primes list.
```

Primzahlen generieren

- Sie würden ja zu einem Quotient kleiner als sie selbst führen – und den hätten wir ja vorher schon probiert.
- Unsere Funktion wird also eine Primzahl nach der anderen finden via `yield` zurückliefern.
- Jedes Mal, wenn wir mit `yield` einen Wert zurückliefern, wird die Ausführung unsere Funktion unterbrochen.
- Das passiert zum ersten Mal wenn wir `yield 2` aufrufen.
- Danach passiert es bei jeder Iteration der äußeren Schleife, wenn diese eine Primzahl findet.

```python
"""A generator function for prime numbers."""

from math import isqrt  # The integer square root function.
from typing import Generator  # The type hint for generators.


def primes() -> Generator[int, None, None]:
    """
    Provide a sequence of prime numbers.

    >>> gen = primes()
    >>> next(gen)
    2
    >>> next(gen)
    3
    >>> next(gen)
    5
    >>> next(gen)
    7
    >>> next(gen)
    11
    """
    yield 2  # The first and only even prime number.

    found: list[int] = []  # The list of already discovered primes.
    candidate: int = 1     # The current prime candidate
    while True:            # Loop over candidates.
        candidate += 2  # Move to the next odd number as prime candidate
        is_prime: bool = True  # Let us assume that `candidate` is prime
        limit: int = isqrt(candidate)  # Get maximum possible divisor.
        for check in found:  # We only test with the odd primes we got.
            if check > limit:  # If the potential divisor is too big,
                break          # then we can stop the inner loop here.
            if candidate % check == 0:  # division without remainder
                is_prime = False  # check divides candidate evenly, so
                break  # candidate is not a prime. Stop the inner loop.

        if is_prime:  # If True: no smaller number divides candidate.
            yield candidate  # Return the prime number as next element.
            found.append(candidate)  # Store candidate in primes list.
```

Primzahlen generieren

- Jedes Mal, wenn wir mit `yield` einen Wert zurückliefern, wird die Ausführung unsere Funktion unterbrochen.
- Das passiert zum ersten Mal wenn wir `yield 2` aufrufen.
- Danach passiert es bei jeder Iteration der äußeren Schleife, wenn diese eine Primzahl findet.
- Jedes Mal, wenn unsere Funktion auf diese Art unterbrochen wird, wird der entsprechende Wert an den Kode außerhalb der Funktion übergeben, der über unsere Sequenz iteriert.

```python
"""A generator function for prime numbers."""

from math import isqrt  # The integer square root function.
from typing import Generator  # The type hint for generators.


def primes() -> Generator[int, None, None]:
    """
    Provide a sequence of prime numbers.

    >>> gen = primes()
    >>> next(gen)
    2
    >>> next(gen)
    3
    >>> next(gen)
    5
    >>> next(gen)
    7
    >>> next(gen)
    11
    """
    yield 2  # The first and only even prime number.

    found: list[int] = []  # The list of already discovered primes.
    candidate: int = 1     # The current prime candidate
    while True:            # Loop over candidates.
        candidate += 2  # Move to the next odd number as prime candidate
        is_prime: bool = True  # Let us assume that `candidate` is prime
        limit: int = isqrt(candidate)  # Get maximum possible divisor.
        for check in found:  # We only test with the odd primes we got.
            if check > limit:  # If the potential divisor is too big,
                break          # then we can stop the inner loop here.
            if candidate % check == 0:  # division without remainder
                is_prime = False  # check divides candidate evenly, so
                break  # candidate is not a prime. Stop the inner loop.

        if is_prime:  # If True: no smaller number divides candidate.
            yield candidate  # Return the prime number as next element.
            found.append(candidate)  # Store candidate in primes list.
```

Primzahlen generieren

- Das passiert zum ersten Mal wenn wir `yield 2` aufrufen.
- Danach passiert es bei jeder Iteration der äußeren Schleife, wenn diese eine Primzahl findet.
- Jedes Mal, wenn unsere Funktion auf diese Art unterbrochen wird, wird der entsprechende Wert an den Kode außerhalb der Funktion übergeben, der über unsere Sequenz iteriert.
- Während wir unsere Funktion implementieren, wissen wir gar nicht, was dieser Kode tut (außer, dass er `next` auf unseren Generator anwendet).

```python
"""A generator function for prime numbers."""

from math import isqrt  # The integer square root function.
from typing import Generator  # The type hint for generators.


def primes() -> Generator[int, None, None]:
    """
    Provide a sequence of prime numbers.

    >>> gen = primes()
    >>> next(gen)
    2
    >>> next(gen)
    3
    >>> next(gen)
    5
    >>> next(gen)
    7
    >>> next(gen)
    11
    """
    yield 2  # The first and only even prime number.

    found: list[int] = []  # The list of already discovered primes.
    candidate: int = 1     # The current prime candidate
    while True:            # Loop over candidates.
        candidate += 2  # Move to the next odd number as prime candidate
        is_prime: bool = True  # Let us assume that `candidate` is prime
        limit: int = isqrt(candidate)  # Get maximum possible divisor.
        for check in found:  # We only test with the odd primes we got.
            if check > limit:  # If the potential divisor is too big,
                break          # then we can stop the inner loop here.
            if candidate % check == 0:  # division without remainder
                is_prime = False  # check divides candidate evenly, so
                break  # candidate is not a prime. Stop the inner loop.

        if is_prime:  # If True: no smaller number divides candidate.
            yield candidate  # Return the prime number as next element.
            found.append(candidate)  # Store candidate in primes list.
```

Primzahlen generieren

- Danach passiert es bei jeder Iteration der äußeren Schleife, wenn diese eine Primzahl findet.
- Jedes Mal, wenn unsere Funktion auf diese Art unterbrochen wird, wird der entsprechende Wert an den Kode außerhalb der Funktion übergeben, der über unsere Sequenz iteriert.
- Während wir unsere Funktion implementieren, wissen wir gar nicht, was dieser Kode tut (außer, dass er `next` auf unseren Generator anwendet).
- Und wir müssen es auch gar nicht wissen.

```python
"""A generator function for prime numbers."""

from math import isqrt  # The integer square root function.
from typing import Generator  # The type hint for generators.


def primes() -> Generator[int, None, None]:
    """
    Provide a sequence of prime numbers.

    >>> gen = primes()
    >>> next(gen)
    2
    >>> next(gen)
    3
    >>> next(gen)
    5
    >>> next(gen)
    7
    >>> next(gen)
    11
    """
    yield 2  # The first and only even prime number.

    found: list[int] = []  # The list of already discovered primes.
    candidate: int = 1     # The current prime candidate
    while True:            # Loop over candidates.
        candidate += 2  # Move to the next odd number as prime candidate
        is_prime: bool = True  # Let us assume that `candidate` is prime
        limit: int = isqrt(candidate)  # Get maximum possible divisor.
        for check in found:  # We only test with the odd primes we got.
            if check > limit:  # If the potential divisor is too big,
                break          # then we can stop the inner loop here.
            if candidate % check == 0:  # division without remainder
                is_prime = False  # check divides candidate evenly, so
                break  # candidate is not a prime. Stop the inner loop.

        if is_prime:  # If True: no smaller number divides candidate.
            yield candidate  # Return the prime number as next element.
            found.append(candidate)  # Store candidate in primes list.
```

Primzahlen generieren

- Während wir unsere Funktion implementieren, wissen wir gar nicht, was dieser Kode tut (außer, dass er `next` auf unseren Generator anwendet).
- Und wir müssen es auch gar nicht wissen.
- Alles, was uns interessiert, ist das wenn er `next` aufruft, dass dann unsere Funktion mit der Instruktion nach dem `yield` weitermacht und läuft, bis sie die nächste Primzahl findet and mit `yield` zurückgibt.

```python
"""A generator function for prime numbers."""

from math import isqrt  # The integer square root function.
from typing import Generator  # The type hint for generators.


def primes() -> Generator[int, None, None]:
    """
    Provide a sequence of prime numbers.

    >>> gen = primes()
    >>> next(gen)
    2
    >>> next(gen)
    3
    >>> next(gen)
    5
    >>> next(gen)
    7
    >>> next(gen)
    11
    """
    yield 2  # The first and only even prime number.

    found: list[int] = []  # The list of already discovered primes.
    candidate: int = 1     # The current prime candidate
    while True:            # Loop over candidates.
        candidate += 2  # Move to the next odd number as prime candidate
        is_prime: bool = True  # Let us assume that `candidate` is prime
        limit: int = isqrt(candidate)  # Get maximum possible divisor.
        for check in found:  # We only test with the odd primes we got.
            if check > limit:  # If the potential divisor is too big,
                break          # then we can stop the inner loop here.
            if candidate % check == 0:  # division without remainder
                is_prime = False  # check divides candidate evenly, so
                break  # candidate is not a prime. Stop the inner loop.

        if is_prime:  # If True: no smaller number divides candidate.
            yield candidate  # Return the prime number as next element.
            found.append(candidate)  # Store candidate in primes list.
```

Primzahlen generieren

- Während wir unsere Funktion implementieren, wissen wir gar nicht, was dieser Kode tut (außer, dass er `next` auf unseren Generator anwendet).
- Und wir müssen es auch gar nicht wissen.
- Alles, was uns interessiert, ist das wenn er `next` aufruft, dass dann unsere Funktion mit der Instruktion nach dem `yield` weitermacht und läuft, bis sie die nächste Primzahl findet and mit `yield` zurückgibt.
- Wie demonstrieren mit einem Doctest, wie unsere Generator-Funktion funktioniert.

```python
"""A generator function for prime numbers."""

from math import isqrt  # The integer square root function.
from typing import Generator  # The type hint for generators.


def primes() -> Generator[int, None, None]:
    """
    Provide a sequence of prime numbers.

    >>> gen = primes()
    >>> next(gen)
    2
    >>> next(gen)
    3
    >>> next(gen)
    5
    >>> next(gen)
    7
    >>> next(gen)
    11
    """
    yield 2  # The first and only even prime number.

    found: list[int] = []  # The list of already discovered primes.
    candidate: int = 1     # The current prime candidate
    while True:            # Loop over candidates.
        candidate += 2  # Move to the next odd number as prime candidate
        is_prime: bool = True  # Let us assume that `candidate` is prime
        limit: int = isqrt(candidate)  # Get maximum possible divisor.
        for check in found:  # We only test with the odd primes we got.
            if check > limit:  # If the potential divisor is too big,
                break          # then we can stop the inner loop here.
            if candidate % check == 0:  # division without remainder
                is_prime = False  # check divides candidate evenly, so
                break  # candidate is not a prime. Stop the inner loop.

        if is_prime:  # If True: no smaller number divides candidate.
            yield candidate  # Return the prime number as next element.
            found.append(candidate)  # Store candidate in primes list.
```

Primzahlen generieren

- Und wir müssen es auch gar nicht wissen.
- Alles, was uns interessiert, ist das wenn er `next` aufruft, dass dann unsere Funktion mit der Instruktion nach dem `yield` weitermacht und läuft, bis sie die nächste Primzahl findet and mit `yield` zurückgibt.
- Wie demonstrieren mit einem Doctest, wie unsere Generator-Funktion funktioniert.
- Der Test beginnt damit, den `Generator` als `gen = primes()` zu instantiieren.

```python
"""A generator function for prime numbers."""

from math import isqrt  # The integer square root function.
from typing import Generator  # The type hint for generators.


def primes() -> Generator[int, None, None]:
    """
    Provide a sequence of prime numbers.

    >>> gen = primes()
    >>> next(gen)
    2
    >>> next(gen)
    3
    >>> next(gen)
    5
    >>> next(gen)
    7
    >>> next(gen)
    11
    """
    yield 2  # The first and only even prime number.

    found: list[int] = []  # The list of already discovered primes.
    candidate: int = 1     # The current prime candidate
    while True:            # Loop over candidates.
        candidate += 2  # Move to the next odd number as prime candidate
        is_prime: bool = True  # Let us assume that `candidate` is prime
        limit: int = isqrt(candidate)  # Get maximum possible divisor.
        for check in found:  # We only test with the odd primes we got.
            if check > limit:  # If the potential divisor is too big,
                break          # then we can stop the inner loop here.
            if candidate % check == 0:  # division without remainder
                is_prime = False  # check divides candidate evenly, so
                break  # candidate is not a prime. Stop the inner loop.

        if is_prime:  # If True: no smaller number divides candidate.
            yield candidate  # Return the prime number as next element.
            found.append(candidate)  # Store candidate in primes list.
```

Primzahlen generieren

- Und wir müssen es auch gar nicht wissen.
- Alles, was uns interessiert, ist das wenn er `next` aufruft, dass dann unsere Funktion mit der Instruktion nach dem `yield` weitermacht und läuft, bis sie die nächste Primzahl findet and mit `yield` zurückgibt.
- Wie demonstrieren mit einem Doctest, wie unsere Generator-Funktion funktioniert.
- Der Test beginnt damit, den `Generator` als `gen = primes()` zu instantiieren.
- Das erste `next(gen)` soll dann `2` liefern.

```python
"""A generator function for prime numbers."""

from math import isqrt  # The integer square root function.
from typing import Generator  # The type hint for generators.


def primes() -> Generator[int, None, None]:
    """
    Provide a sequence of prime numbers.

    >>> gen = primes()
    >>> next(gen)
    2
    >>> next(gen)
    3
    >>> next(gen)
    5
    >>> next(gen)
    7
    >>> next(gen)
    11
    """
    yield 2  # The first and only even prime number.

    found: list[int] = []  # The list of already discovered primes.
    candidate: int = 1     # The current prime candidate
    while True:            # Loop over candidates.
        candidate += 2  # Move to the next odd number as prime candidate
        is_prime: bool = True  # Let us assume that `candidate` is prime
        limit: int = isqrt(candidate)  # Get maximum possible divisor.
        for check in found:  # We only test with the odd primes we got.
            if check > limit:  # If the potential divisor is too big,
                break          # then we can stop the inner loop here.
            if candidate % check == 0:  # division without remainder
                is_prime = False  # check divides candidate evenly, so
                break  # candidate is not a prime. Stop the inner loop.

        if is_prime:  # If True: no smaller number divides candidate.
            yield candidate  # Return the prime number as next element.
            found.append(candidate)  # Store candidate in primes list.
```

Primzahlen generieren

- Alles, was uns interessiert, ist das wenn er `next` aufruft, dass dann unsere Funktion mit der Instruktion nach dem `yield` weitermacht und läuft, bis sie die nächste Primzahl findet and mit `yield` zurückgibt.
- Wie demonstrieren mit einem Doctest, wie unsere Generator-Funktion funktioniert.
- Der Test beginnt damit, den `Generator` als `gen = primes()` zu instantiieren.
- Das erste `next(gen)` soll dann `2` liefern.
- Das zweite soll `3` ergeben, das dritte `5`, und der vierte Aufruf ergibt `7`.

```python
"""A generator function for prime numbers."""

from math import isqrt  # The integer square root function.
from typing import Generator  # The type hint for generators.


def primes() -> Generator[int, None, None]:
    """
    Provide a sequence of prime numbers.

    >>> gen = primes()
    >>> next(gen)
    2
    >>> next(gen)
    3
    >>> next(gen)
    5
    >>> next(gen)
    7
    >>> next(gen)
    11
    """
    yield 2  # The first and only even prime number.

    found: list[int] = []  # The list of already discovered primes.
    candidate: int = 1     # The current prime candidate
    while True:            # Loop over candidates.
        candidate += 2  # Move to the next odd number as prime candidate
        is_prime: bool = True  # Let us assume that `candidate` is prime
        limit: int = isqrt(candidate)  # Get maximum possible divisor.
        for check in found:  # We only test with the odd primes we got.
            if check > limit:  # If the potential divisor is too big,
                break          # then we can stop the inner loop here.
            if candidate % check == 0:  # division without remainder
                is_prime = False  # check divides candidate evenly, so
                break  # candidate is not a prime. Stop the inner loop.

        if is_prime:  # If True: no smaller number divides candidate.
            yield candidate  # Return the prime number as next element.
            found.append(candidate)  # Store candidate in primes list.
```

Primzahlen generieren

- Wie demonstrieren mit einem Doctest, wie unsere Generator-Funktion funktioniert.
- Der Test beginnt damit, den `Generator` als `gen = primes()` zu instantiieren.
- Das erste `next(gen)` soll dann `2` liefern.
- Das zweite soll `3` ergeben, das dritte `5`, und der vierte Aufruf ergibt `7`.
- Das fünfte und letztes `next(gen)` im Doctest soll dann `11` liefern.

```python
"""A generator function for prime numbers."""

from math import isqrt  # The integer square root function.
from typing import Generator  # The type hint for generators.


def primes() -> Generator[int, None, None]:
    """
    Provide a sequence of prime numbers.

    >>> gen = primes()
    >>> next(gen)
    2
    >>> next(gen)
    3
    >>> next(gen)
    5
    >>> next(gen)
    7
    >>> next(gen)
    11
    """
    yield 2  # The first and only even prime number.

    found: list[int] = []  # The list of already discovered primes.
    candidate: int = 1     # The current prime candidate
    while True:            # Loop over candidates.
        candidate += 2  # Move to the next odd number as prime candidate
        is_prime: bool = True  # Let us assume that `candidate` is prime
        limit: int = isqrt(candidate)  # Get maximum possible divisor.
        for check in found:  # We only test with the odd primes we got.
            if check > limit:  # If the potential divisor is too big,
                break          # then we can stop the inner loop here.
            if candidate % check == 0:  # division without remainder
                is_prime = False  # check divides candidate evenly, so
                break  # candidate is not a prime. Stop the inner loop.

        if is_prime:  # If True: no smaller number divides candidate.
            yield candidate  # Return the prime number as next element.
            found.append(candidate)  # Store candidate in primes list.
```

Primzahlen generieren

- Der Test beginnt damit, den `Generator` als `gen = primes()` zu instantiieren.
- Das erste `next(gen)` soll dann `2` liefern.
- Das zweite soll `3` ergeben, das dritte `5`, und der vierte Aufruf ergibt `7`.
- Das fünfte und letztes `next(gen)` im Doctest soll dann `11` liefern.
- Und diese Tests sind auch erfolgreich, die Ausgaben stimmen also.

```
1 $ pytest --timeout=10 --no-header --tb=short --doctest-modules
     ↪ prime_generator.py
2 ========================== test session starts
     ↪ ==========================
3 collected 1 item
4
5 prime_generator.py .                                                    [100%]
6
7 ========================== 1 passed in 0.01s
     ↪ ===========================
8 # pytest 9.0.2 with pytest-timeout 2.4.0 succeeded with exit code 0.
```

Zusammenfassung

Zusammenfassung

- Aus der Perspektive eines Benutzers verhalten sich Generator-Funktionen wie Generator-Ausdrücke, welche wiederum einfache Iteratoren sind.

Zusammenfassung

- Aus der Perspektive eines Benutzers verhalten sich Generator-Funktionen wie Generator-Ausdrücke, welche wiederum einfache Iteratoren sind.
- Verglichen mit Generator-Ausdrücken sind Generator-Funktion viel mächtiger.

Zusammenfassung

- Aus der Perspektive eines Benutzers verhalten sich Generator-Funktionen wie Generator-Ausdrücke, welche wiederum einfache Iteratoren sind.
- Verglichen mit Generator-Ausdrücken sind Generator-Funktion viel mächtiger.
- Wir können beliebig komplexen Kode in so eine Funktion packen.

Zusammenfassung

- Aus der Perspektive eines Benutzers verhalten sich Generator-Funktionen wie Generator-Ausdrücke, welche wiederum einfache Iteratoren sind.
- Verglichen mit Generator-Ausdrücken sind Generator-Funktion viel mächtiger.
- Wir können beliebig komplexen Kode in so eine Funktion packen.
- Der Kode kann eine oder mehrere Stellen haben, an der Werte an die Außenwelt übergeben werden.

Zusammenfassung

- Aus der Perspektive eines Benutzers verhalten sich Generator-Funktionen wie Generator-Ausdrücke, welche wiederum einfache Iteratoren sind.
- Verglichen mit Generator-Ausdrücken sind Generator-Funktion viel mächtiger.
- Wir können beliebig komplexen Kode in so eine Funktion packen.
- Der Kode kann eine oder mehrere Stellen haben, an der Werte an die Außenwelt übergeben werden.
- Eine normale Funktion kann ein oder mehrere `return`-Statements haben.

Zusammenfassung

- Aus der Perspektive eines Benutzers verhalten sich Generator-Funktionen wie Generator-Ausdrücke, welche wiederum einfache Iteratoren sind.
- Verglichen mit Generator-Ausdrücken sind Generator-Funktion viel mächtiger.
- Wir können beliebig komplexen Kode in so eine Funktion packen.
- Der Kode kann eine oder mehrere Stellen haben, an der Werte an die Außenwelt übergeben werden.
- Eine normale Funktion kann ein oder mehrere `return`-Statements haben.
- Die Ausführung einer normalen Funktion ist zuende, wenn das erste `return`-Statement ausgeführt wird.

Zusammenfassung

- Aus der Perspektive eines Benutzers verhalten sich Generator-Funktionen wie Generator-Ausdrücke, welche wiederum einfache Iteratoren sind.
- Verglichen mit Generator-Ausdrücken sind Generator-Funktion viel mächtiger.
- Wir können beliebig komplexen Kode in so eine Funktion packen.
- Der Kode kann eine oder mehrere Stellen haben, an der Werte an die Außenwelt übergeben werden.
- Eine normale Funktion kann ein oder mehrere `return`-Statements haben.
- Die Ausführung einer normalen Funktion ist zuende, wenn das erste `return`-Statement ausgeführt wird.
- Eine Generator-Funktion kann ein oder mehrere `yield`-Statements haben.

Zusammenfassung

- Aus der Perspektive eines Benutzers verhalten sich Generator-Funktionen wie Generator-Ausdrücke, welche wiederum einfache Iteratoren sind.
- Verglichen mit Generator-Ausdrücken sind Generator-Funktion viel mächtiger.
- Wir können beliebig komplexen Kode in so eine Funktion packen.
- Der Kode kann eine oder mehrere Stellen haben, an der Werte an die Außenwelt übergeben werden.
- Eine normale Funktion kann ein oder mehrere `return`-Statements haben.
- Die Ausführung einer normalen Funktion ist zuende, wenn das erste `return`-Statement ausgeführt wird.
- Eine Generator-Funktion kann ein oder mehrere `yield`-Statements haben.
- Jedes davon übergibt einen Wert an die Außenwelt.

Zusammenfassung

- Aus der Perspektive eines Benutzers verhalten sich Generator-Funktionen wie Generator-Ausdrücke, welche wiederum einfache Iteratoren sind.
- Verglichen mit Generator-Ausdrücken sind Generator-Funktion viel mächtiger.
- Wir können beliebig komplexen Kode in so eine Funktion packen.
- Der Kode kann eine oder mehrere Stellen haben, an der Werte an die Außenwelt übergeben werden.
- Eine normale Funktion kann ein oder mehrere `return`-Statements haben.
- Die Ausführung einer normalen Funktion ist zuende, wenn das erste `return`-Statement ausgeführt wird.
- Eine Generator-Funktion kann ein oder mehrere `yield`-Statements haben.
- Jedes davon übergibt einen Wert an die Außenwelt.
- Eine Generator-Funktion wird nach `yield` fortgesetzt, und zwar so lange bis entweder ihr Ende erreicht wird, oder der Kode außerhalb aufhört, über ihre Sequenz zu iterieren.

谢谢您们！

Thank you!

Vielen Dank!

References I

[1] Kent L. Beck. *JUnit Pocket Guide*. Sebastopol, CA, USA: O'Reilly Media, Inc., Sep. 2004. ISBN: 978-0-596-00743-0 (siehe S. 106).

[2] Frances Carney Gies, Aakanksha Gaur, Erik Gregersen, Gloria Lotha, Emily Rodriguez, Veenu Setia, Gaurav Shukla und Grace Young. "Fibonacci". In: *Encyclopaedia Britannica*. Hrsg. von The Editors of Encyclopaedia Britannica. Chicago, IL, USA: Encyclopædia Britannica, Inc., 26. Juli 2025. URL: https://www.britannica.com/biography/Fibonacci (besucht am 2025-09-18) (siehe S. 37–40).

[3] Antonio Cavacini. "Is the CE/BCE notation becoming a standard in scholarly literature?" *Scientometrics* 102(2):1661–1668, Juli 2015. London, England, UK: Springer Nature Limited. ISSN: 0138-9130. doi:10.1007/s11192-014-1352-1 (siehe S. 105).

[4] Richard Crandall und Carl Pomerance. *Prime Numbers: A Computational Perspective*. 2. Aufl. New York, NY, USA: Springer New York, 4. Aug. 2005. ISBN: 978-0-387-25282-7. doi:10.1007/0-387-28979-8 (siehe S. 58–62).

[5] Alfredo Deza und Noah Gift. *Testing In Python*. San Francisco, CA, USA: Pragmatic AI Labs, Feb. 2020. ISBN: 979-8-6169-6064-1 (siehe S. 105).

[6] "Doctest – Test Interactive Python Examples". In: *Python 3 Documentation*. *The Python Standard Library*. Beaverton, OR, USA: Python Software Foundation (PSF), 2001–2025. URL: https://docs.python.org/3/library/doctest.html (besucht am 2024-11-07) (siehe S. 105).

[7] 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. 105).

[8] 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. 106).

[9] Holger Krekel und pytest-Dev Team. "How to Run Doctests". In: *pytest Documentation*. Release 8.4. Freiburg, Baden-Württemberg, Germany: merlinux GmbH. Kap. 2.8, S. 65–69. URL: https://docs.pytest.org/en/stable/how-to/doctest.html (besucht am 2024-11-07) (siehe S. 105).

[10] Holger Krekel und pytest-Dev Team. *pytest Documentation*. Release 8.4. Freiburg, Baden-Württemberg, Germany: merlinux GmbH. URL: https://readthedocs.org/projects/pytest/downloads/pdf/latest (besucht am 2024-11-07) (siehe S. 105).

[11] 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. 106).

References II

[12] 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. 106).

[13] 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. 105).

[14] Mark Lutz. *Learning Python*. 6. Aufl. Sebastopol, CA, USA: O'Reilly Media, Inc., März 2025. ISBN: 978-1-0981-7130-8 (siehe S. 106).

[15] A. Jefferson Offutt. "Unit Testing Versus Integration Testing". In: *Test: Faster, Better, Sooner – IEEE International Test Conference (ITC'1991)*. 26.–30. Okt. 1991, Nashville, TN, USA. Los Alamitos, CA, USA: IEEE Computer Society, 1991. Kap. Paper P2.3, S. 1108–1109. ISSN: 1089-3539. ISBN: 978-0-8186-9156-0. doi:10.1109/TEST.1991.519784 (siehe S. 106).

[16] Brian Okken. *Python Testing with pytest*. Flower Mound, TX, USA: Pragmatic Bookshelf by The Pragmatic Programmers, L.L.C., Feb. 2022. ISBN: 978-1-68050-860-4 (siehe S. 105).

[17] Michael Olan. "Unit Testing: Test Early, Test Often". *Journal of Computing Sciences in Colleges (JCSC)* 19(2):319–328, Dez. 2003. New York, NY, USA: Association for Computing Machinery (ACM). ISSN: 1937-4771. doi:10.5555/948785.948830. URL: https://www.researchgate.net/publication/255673967 (besucht am 2025-09-05) (siehe S. 106).

[18] Ashwin Pajankar. *Python Unit Test Automation: Automate, Organize, and Execute Unit Tests in Python*. New York, NY, USA: Apress Media, LLC, Dez. 2021. ISBN: 978-1-4842-7854-3 (siehe S. 105, 106).

[19] Yasset Pérez-Riverol, Laurent Gatto, Rui Wang, Timo Sachsenberg, Julian Uszkoreit, Felipe da Veiga Leprevost, Christian Fufezan, Tobias Ternent, Stephen J. Eglen, Daniel S. Katz, Tom J. Pollard, Alexander Konovalov, Robert M. Flight, Kai Blin und Juan Antonio Vizcaíno. "Ten Simple Rules for Taking Advantage of Git and GitHub". *PLOS Computational Biology* 12(7), 14. Juli 2016. San Francisco, CA, USA: Public Library of Science (PLOS). ISSN: 1553-7358. doi:10.1371/JOURNAL.PCBI.1004947 (siehe S. 105).

[20] Hans Riesel. *Prime Numbers and Computer Methods for Factorization*. 2. Aufl. Bd. 126 der Reihe Progress in Mathematics (PM). New York, NY, USA: Springer Science+Business Media, LLC, 1. Okt. 1994–30. Sep. 2012. ISSN: 0743-1643. ISBN: 978-0-8176-3743-9. doi:10.1007/978-1-4612-0251-6. Boston, MA, USA: Birkhäuser (siehe S. 58–62).

References III

[21] Per Runeson. "A Survey of Unit Testing Practices". *IEEE Software* 23(4):22–29, Juli–Aug. 2006. Piscataway, NJ, USA: Institute of Electrical and Electronics Engineers (IEEE). ISSN: 0740-7459. doi:10.1109/MS.2006.91 (siehe S. 106).

[22] Neil Schemenauer, Tim Peters und Magnus Lie Hetland. *Simple GeneratorS*. Python Enhancement Proposal (PEP) 255. Beaverton, OR, USA: Python Software Foundation (PSF), 18. Mai 2001. URL: https://peps.python.org/pep-0255 (besucht am 2024-11-08) (siehe S. 5–8).

[23] Syamal K. Sen und Ravi P. Agarwal. "Existence of year zero in astronomical counting is advantageous and preserves compatibility with significance of AD, BC, CE, and BCE". In: *Zero – A Landmark Discovery, the Dreadful Void, and the Ultimate Mind*. Amsterdam, The Netherlands: Elsevier B.V., 2016. Kap. 5.5, S. 94–95. ISBN: 978-0-08-100774-7. doi:10.1016/C2015-0-02299-7 (siehe S. 105).

[24] Laurence E. Sigler. *Fibonacci's Liber Abaci: A Translation into Modern English of Leonardo Pisano's Book of Calculation*. Sources and Studies in the History of Mathematics and Physical Sciences. New York, NY, USA: Springer New York, 10. Sep. 2002. ISSN: 2196-8810. ISBN: 978-0-387-95419-6. doi:10.1007/978-1-4613-0079-3 (siehe S. 37–42).

[25] Anna Skoulikari. *Learning Git*. Sebastopol, CA, USA: O'Reilly Media, Inc., Mai 2023. ISBN: 978-1-0981-3391-7 (siehe S. 105).

[26] The Editors of Encyclopaedia Britannica, Hrsg. *Encyclopaedia Britannica*. Chicago, IL, USA: Encyclopædia Britannica, Inc.

[27] George K. Thiruvathukal, Konstantin Läufer und Benjamin Gonzalez. "Unit Testing Considered Useful". *Computing in Science & Engineering* 8(6):76–87, Nov.–Dez. 2006. Piscataway, NJ, USA: Institute of Electrical and Electronics Engineers (IEEE). ISSN: 1521-9615. doi:10.1109/MCSE.2006.124. URL: https://www.researchgate.net/publication/220094077 (besucht am 2024-10-01) (siehe S. 106).

[28] 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. 105, 106).

[29] 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. 106).

[30] 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. 105).

References IV

[31] 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. 105, 106).

[32] Eric Wolfgang Weisstein. "Fibonacci Number". In: *MathWorld – A Wolfram Web Resource*. Champaign, IL, USA: Wolfram Research, Inc., 22. Aug. 2024. URL: https://mathworld.wolfram.com/FibonacciNumber.html (besucht am 2024-11-08) (siehe S. 37–42).

[33] Eric Wolfgang Weisstein. *MathWorld – A Wolfram Web Resource*. Champaign, IL, USA: Wolfram Research, Inc., 22. Aug. 2024. URL: https://mathworld.wolfram.com (besucht am 2024-09-24).

[34] Eric Wolfgang Weisstein. "Prime Number". In: *MathWorld – A Wolfram Web Resource*. Champaign, IL, USA: Wolfram Research, Inc., 22. Aug. 2024. URL: https://mathworld.wolfram.com/PrimeNumber.html (besucht am 2024-09-24) (siehe S. 58–62).

[35] Kevin Wilson. *Python Made Easy*. Birmingham, England, UK: Packt Publishing Ltd, Aug. 2024. ISBN: 978-1-83664-615-0 (siehe S. 105).

[36] Nicola Abdo Ziadeh, Michael B. Rowton, A. Geoffrey Woodhead, Wolfgang Helck, Jean L.A. Filliozat, Hiroyuki Momo, Eric Thompson, E.J. Wiesenberg und Shih-ch'ang Wu. "Chronology – Christian History, Dates, Events". In: *Encyclopaedia Britannica*. Hrsg. von The Editors of Encyclopaedia Britannica. Chicago, IL, USA: Encyclopædia Britannica, Inc., 26. Juli 1999–20. März 2024. URL: https://www.britannica.com/topic/chronology/Christian (besucht am 2025-08-27) (siehe S. 105).

Glossary (in English) I

BCE The time notation *before Common Era* is a non-religious but chronological equivalent alternative to the traditional *Before Christ (BC)* notation, which refers to the years *before* the birth of Jesus Christ[3]. The years BCE are counted down, i.e., the larger the year, the farther in the past. The year 1 BCE comes directly before the year 1 CE[23,36].

CE The time notation *Common Era* is a non-religious but chronological equivalent alternative to the traditional *Anno Domini (AD)* notation, which refers to the years *after* the birth of Jesus Christ[3]. The years CE are counted upwards, i.e., the smaller they are, the farther they are in the past. The year 1 CE comes directly after the year 1 before Common Era (BCE)[23,36].

docstring Docstrings are special string constants in Python that contain documentation for modules or functions[7]. They must be delimited by `"""..."""`[7,30].

doctest *doctests* are unit tests in the form of as small pieces of code in the docstrings that look like interactive Python sessions. The first line of a statement in such a Python snippet is indented with Python»> and the following lines by `...`. These snippets can be executed by modules like `doctest`[6] or tools such as pytest[9]. Their output is the compared to the text following the snippet in the docstring. If the output matches this text, the test succeeds. Otherwise it fails.

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[25,28]. Learn more at https://git-scm.com.

GitHub is a website where software projects can be hosted and managed via the Git VCS[19,28]. Learn more at https://github.com.

Mypy is a static type checking tool for Python[13] that makes use of type hints. Learn more at https://github.com/python/mypy and in[31].

pytest is a framework for writing and executing unit tests in Python[5,10,16,18,35]. Learn more at https://pytest.org.

Glossary (in English) II

Python The Python programming language[8,12,14,31], i.e., what you will learn about in our book[31]. Learn more at https://python.org.

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[11,29]. 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.

unit test Software development is centered around creating the program code of an application, library, or otherwise useful system. A *unit test* is an *additional* code fragment that is not part of that productive code. It exists to execute (a part of) the productive code in a certain scenario (e.g., with specific parameters), to observe the behavior of that code, and to compare whether this behavior meets the specification[1,15,17,18,21,27]. If not, the unit test fails. The use of unit tests is at least threefold: First, they help us to detect errors in the code. Second, program code is usually not developed only once and, from then on, used without change indefinitely. Instead, programs are often updated, improved, extended, and maintained over a long time. Unit tests can help us to detect whether such changes in the program code, maybe after years, violate the specification or, maybe, cause another, depending, module of the program to violate its specification. Third, they are part of the documentation or even specification of a program.

VCS A *Version Control System* is a software which allows you to manage and preserve the historical development of your program code[28]. 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.