Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

Python

am 31.05.2021

von Leon Zander

Proseminar SoSe 2021

Softwareentwicklung in der Wissenschaft

Gliederung

- Merkmale von Python
- Anwendungsgebiete
- Codebeispiele
  - Variablen
  - Kontrollstrukturen
  - Funktionen
  - Klassen
  - Dekoratoren
- Performance von Python
- Glue It All Together With Python
- Ein Beispiel aus der Wissenschaft
- Fazit

Merkmale von Python 1/2

- Universell einsetzbar
  - Unterstützt viele Programmierparadigma
- Üblicherweise interpretiert
  - Verschiedene Implementationen verfügbar
    - CPython, PyPy, Jython
- Open source
- Verwaltet von der Python Software Foundation (PSF)

Merkmale von Python 2/2

- Dynamische Typisierung
- Organisiert in Modulen
  - Python Package Index (PyPI)
  - Installation über pip oder bspw. Anaconda

- Grundprinzip: „Alles ist ein Objekt“
- Python Enhancement Proposals
  - PEP 8 – Style Guide for Python Code
  - PEP 257 – Docstring Conventions

Anwendungsgebiete (Auswahl)

- Machine learning (Tensorflow, PyTorch)
- Data Science (Pandas, NumPy)
- Web Development (Django, Flask)
- Datenvisualisierung (Matplotlib)

- Scripting/ Plug-Ins für andere Programme (Blender, Photoshop, Gimp)

Codebeispiele

```
print("Hello World!")
```

Codebeispiele - Variablen

```
1  # Keine Typinformation notwendig
2  x = 42
3
4  # Eine Liste mit verschiedenen Datentypen
5  x = [1, 2, 3j, 4.2, "Hallo", 'c']
6  x.append("Hallo Welt")
7
8  print(x) # [1, 2, 3j, 4.2, 'Hallo', 'c', 'Hallo Welt']
9
10 # Ein Tupel
11 x = ("Unveränderlich")
12
13 # Ein Dictionary
14 y = {'a':1, 'b':2, 'c':3}
15
16 print(y['a']) # 1
17
```

Codebeispiele - Kontrollstrukturen

```
1  # Wir initialisieren eine Liste mit einigen Quadratzahlen
2  zahlen1 = []
3
4  for i in range(10):
5      zahlen1.append(i*i)
6
7  print(zahlen1) # [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
8
9  # List-Comprehension
10 zahlen2 = [i*i for i in range(10)]
11
12 quadrat = lambda x: x*x
13 zahlen3 = [quadrat(i) for i in range(10)]
14
15 # Comprehensions auch für andere Datentypen möglich
16 nachfolger = {i:i+1 for i in range(5)}
17
18 print(nachfolger) # {0: 1, 1: 2, 2: 3, 3: 4, 4: 5}
19
```

Codebeispiele - Funktionen

```
1  def fibonacci(n):
2      if n < 2:
3          return 1
4      return fibonacci(n-1) + fibonacci(n-2)
5
6  print(fibonacci(5)) # 8
7
8  def fibonacci2(n: int) -> int:
9      if n < 2:
10         return 1
11     return fibonacci2(n-1) + fibonacci2(n-2)
12
13 print(fibonacci2(5)) # 8
14
```

- Type-Hints ermöglichen statische Code-Analyse
- Zur Laufzeit aber keine Überprüfung

Codebeispiele - Klassen

```python
class Vector:
    """
        Eine Klasse für Vektoren mit 2 Komponenten
    """
    def __init__(self, x=0, y=0):
        self._x = x
        self._y = y

    def __add__(self, other):
        x = self._x + other._x
        y = self._y + other._y

        return Vector(x, y)

    def __str__(self):
        return f"({self._x} | {self._y})"

v1 = Vector(7, 15)
v2 = Vector(y=10, x=5)

print(v1 + v2) # (12 | 25)

```

Codebeispiele – Decorators 1/2

```python
import logging

logging.basicConfig(level=logging.DEBUG)

_logger = logging.getLogger(__name__)

def log(level=logging.DEBUG, logger=_logger):
    def inner(func):
        def wrapper(*args, **kwargs):

            val = func(*args, **kwargs)

            logger.log(level, f" {func.__name__} wurde mit {args} aufgerufen und ergab {val}")

            return val
        return wrapper
    return inner

```

Codebeispiele – Decorators 2/2

```python
from Decorators import log


@log()
def fibonacci(n: int) -> int:
    if n < 2:
        return 1
    return fibonacci(n-1) + fibonacci(n-2)

print(fibonacci(3))

```

```
DEBUG:Decorators: fibonacci wurde mit (1,) aufgerufen und ergab 1
DEBUG:Decorators: fibonacci wurde mit (0,) aufgerufen und ergab 1
DEBUG:Decorators: fibonacci wurde mit (2,) aufgerufen und ergab 2
DEBUG:Decorators: fibonacci wurde mit (1,) aufgerufen und ergab 1
DEBUG:Decorators: fibonacci wurde mit (3,) aufgerufen und ergab 3
3
```

Python hat eine geringe Performance

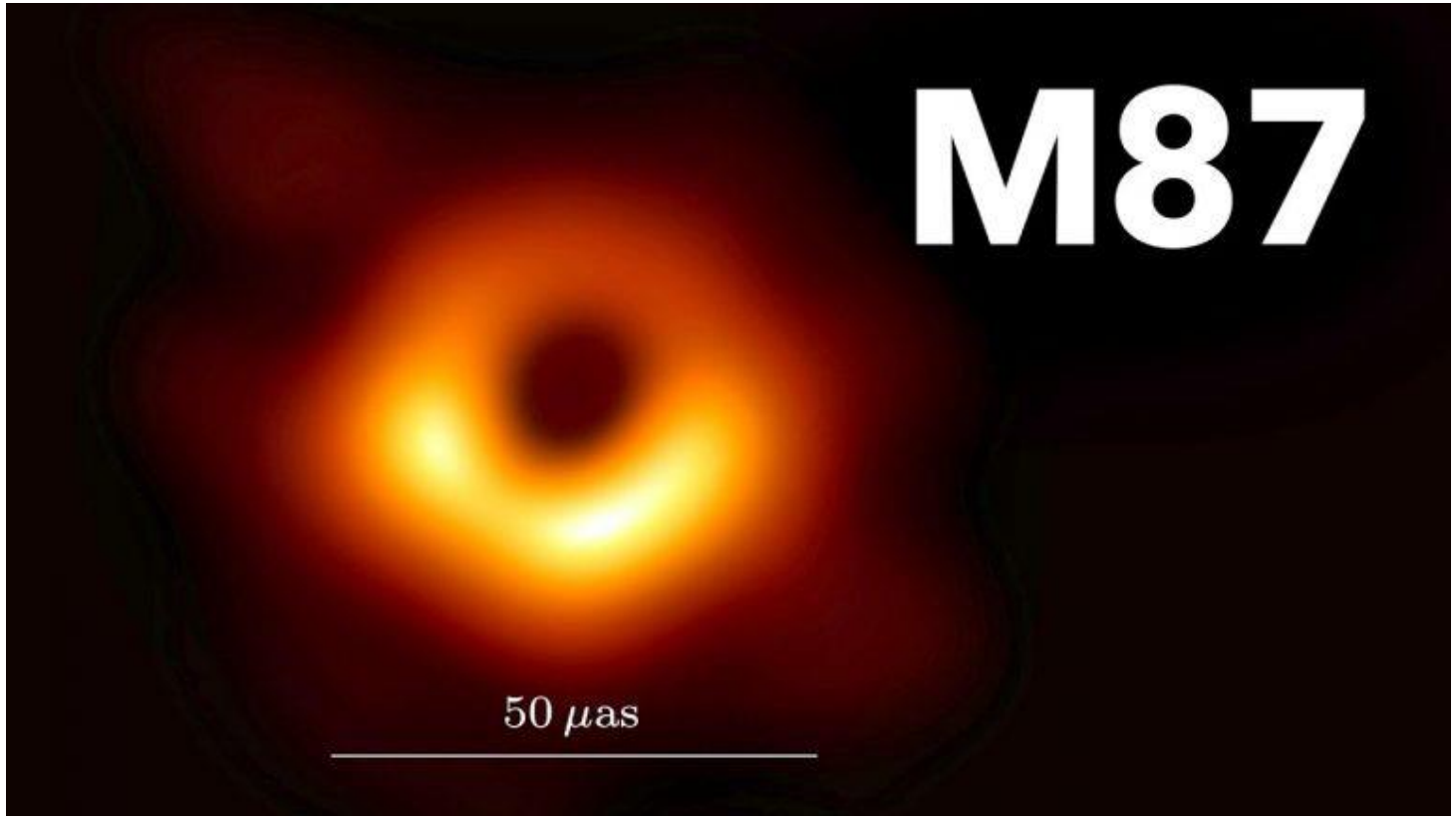
- Performance hängt vom spezifischen Compiler/Interpreter ab
  - Kompatibilität vs. Performance
- Global Interpreter Lock (GIL)
  - Nur ein Thread hat Zugriff auf Interpreter
- Dynamische Typisierung erschwert Optimierungen seitens Python selbst

Glue It All Together With Python

- Prototypen in Python
- Performancekritischer Code wird in Module ausgelagert
  - Module bieten Schnittstelle zu Code in bspw. C/C++
  - So z.B. NumPy und Tensorflow

- => Insgesamt kann so der Overhead durch den Interpreter gemildert werden

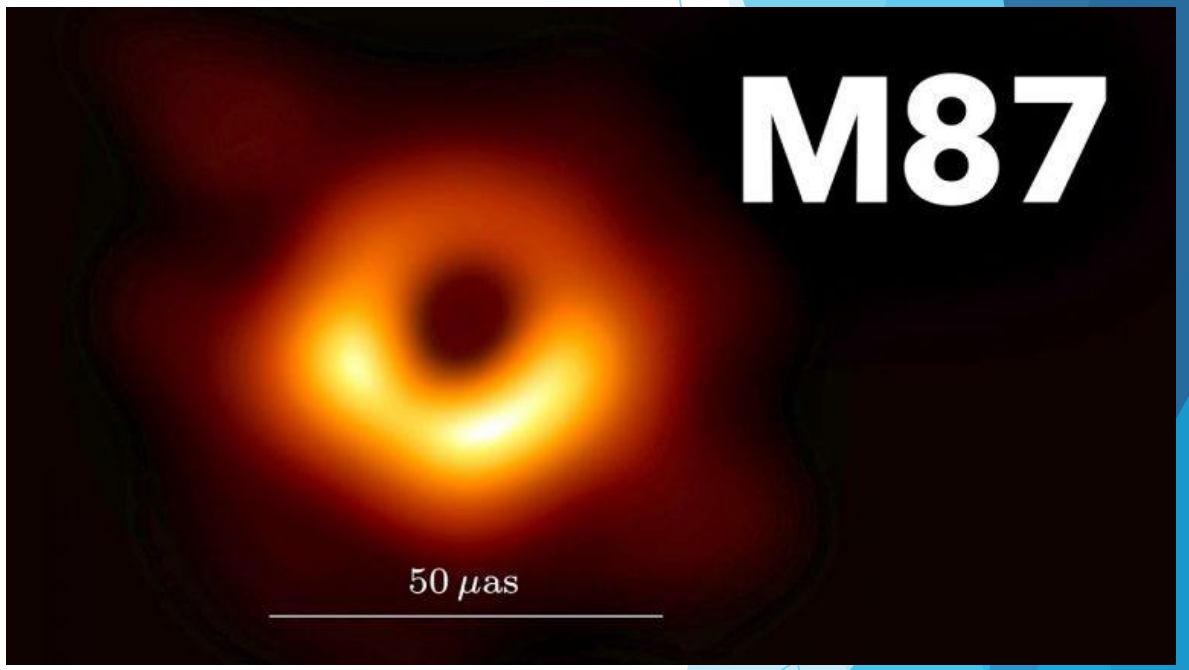
Ein Beispiel aus der Wissenschaft 1/3



Das erste Bild von einem schwarzen Loch
Quelle: https://analyticsindiamag.com/wp-content/uploads/2019/04/maxresdefault-768x432.jpg

Ein Beispiel aus der Wissenschaft 2/3

- Vergleichbar mit einem Foto einer Orange auf der Mondoberfläche
- 8 Teleskope über den Globus verteilt
- Messung über 5 Nächte
- Insgesamt 14 Petabyte an VLBI Messdaten

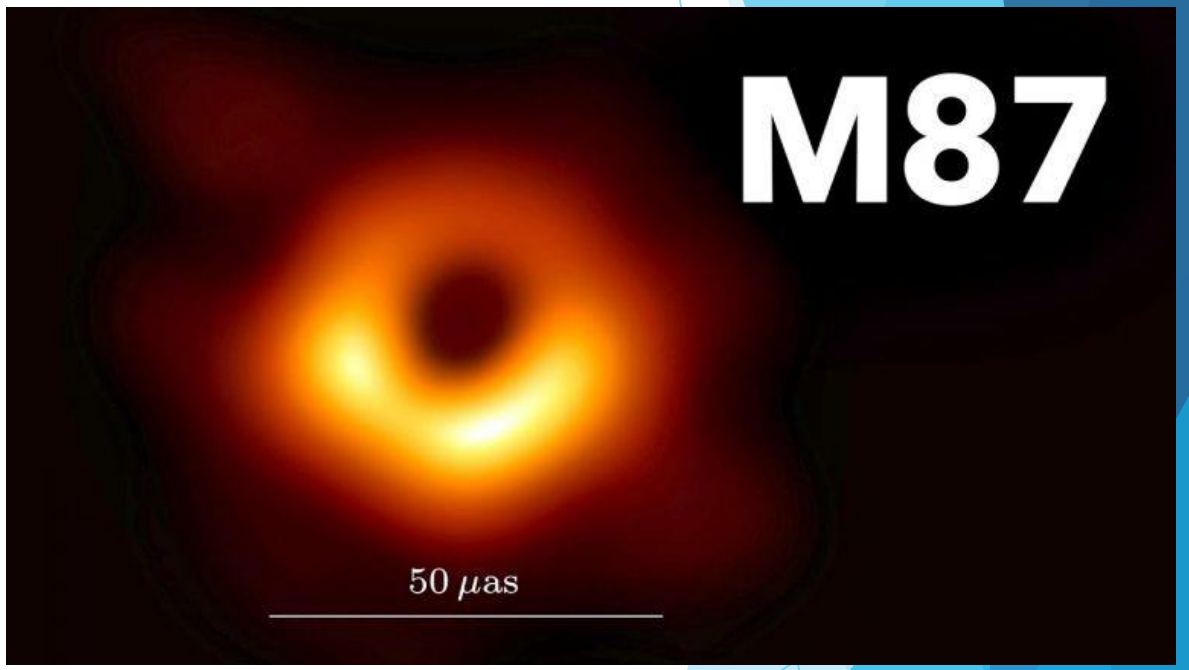


Das erste Bild von einem schwarzen Loch
Quelle: https://analyticsindiamag.com/wp-content/uploads/2019/04/maxresdefault-768x432.jpg

Ein Beispiel aus der Wissenschaft 3/3

- Ein Python Skript basierend auf AIPS
- Rekonstruktion basierend auf Messungen und Wahrscheinlichkeiten

- Module:
  - Scikitlearn
  - Pandas
  - Astropy
  - Matplotlib
  - NumPy



Das erste Bild von einem schwarzen Loch
Quelle: https://analyticsindiamag.com/wp-content/uploads/2019/04/maxresdefault-768x432.jpg

Github: https://github.com/eventhorizontelescope

Fazit

- Eine flache Lernkurve
- Schnelle Entwicklung
- Module können Performance ausgleichen

Quellen

- https://de.wikipedia.org/wiki/Python_(Programmiersprache) (25.05.21)
- https://www.python.org/doc/essays/omg-darpa-mcc-position/ (25.05.21)
- https://numpy.org/case-studies/blackhole-image/ (25.05.21)
- https://www.welcometothejungle.com/en/articles/btc-performance-python (27.05.21)
- https://hackernoon.com/why-is-python-so-slow-e5074b6fe55b (23.05.21)
- https://www.python.org/dev/peps/ (28.05.21)