Werkzeuge für das wissenschaftliche Arbeiten

Python for Machine Learning and Data Science

Magnus Bender
bender@ifis.uni-luebeck.de
Wintersemester 2023/24

Inhaltsübersicht

1. Programmiersprache Python
   a) *Einführung, Erste Schritte*
   b) *Grundlagen*
   c) Fortgeschritten
2. Auszeichnungssprachen
   a) LaTeX, Markdown
3. Benutzeroberflächen und Entwicklungsumgebungen
   a) Jupyter Notebooks lokal und in der Cloud (Google Colab)
4. Versionsverwaltung
   a) Git, GitHub
5. Wissenschaftliches Rechnen
   a) NumPy, SciPy
6. Datenverarbeitung und -visualisierung
   a) Pandas, matplotlib, NLTK
7. Machine Learning (scikit-learn)
   a) Grundlegende Ansätze (Datensätze, Auswertung)
   b) Einfache Verfahren (Clustering, …)
8. DeepLearning
   a) TensorFlow, PyTorch, HuggingFace Transformers

Themen

- Magic Methods
- Generatoren
- Erweiterte Schleifen
- Fehlerbehandlung
- Kontextmanager

- Lambda-Funktionen
- Typannotationen
- Variablen und Zeiger
- Dekoratoren

Magic Methods

```
class Vector():

  def __init__(self, a, b):
    self.a = a
    self.b = b

  def __add__(self, o):
    return Vector(self.a + o.a, self.b + o.b)

  def __sub__(self, o):
    return Vector(self.a - o.a, self.b - o.b)

  def __iadd__(self, o):
    self.a += o.a
    self.b += o.b
    return self

  # def __isub__(self, o):

  def __str__(self):
    return "({}, {})".format(self.a, self.b)
```

Magic Methods

```python
class Vector():

  def __init__(self, a, b):
    self.a = a
    self.b = b

  def __add__(self, o):
    return Vector(self.a + o.a, self.b + o.b)

  def __sub__(self, o):
    return Vector(self.a - o.a, self.b - o.b)

  def __iadd__(self, o):
    self.a += o.a
    self.b += o.b
    return self

  # def __isub__(self, o):

  def __str__(self):
    return "({}, {})".format(self.a, self.b)
```

```python
x = Vector(1, 2)
print(x, str(x), x.__str__())


y = Vector(2, 3)
print(y)


print(x + y, x.__add__(y))


print(x - y)


x += y
print(x)

x -= y
print(x)
```

Magic Methods

```python
class Vector():

  def __init__(self, a, b):
    self.a = a
    self.b = b

  def __add__(self, o):
    return Vector(self.a + o.a, self.b + o.b)

  def __sub__(self, o):
    return Vector(self.a - o.a, self.b - o.b)

  def __iadd__(self, o):
    self.a += o.a
    self.b += o.b
    return self

  # def __isub__(self, o):

  def __str__(self):
    return "({}, {})".format(self.a, self.b)
```

```python
x = Vector(1, 2)
print(x, str(x), x.__str__())
```

```
(1, 2) (1, 2) (1, 2)
```

```python
y = Vector(2, 3)
print(y)
```

```python
print(x + y, x.__add__(y))
```

```python
print(x - y)
```

```python
x += y
print(x)
```

```python
x -= y
print(x)
```

Magic Methods

```python
class Vector():

  def __init__(self, a, b):
    self.a = a
    self.b = b

  def __add__(self, o):
    return Vector(self.a + o.a, self.b + o.b)

  def __sub__(self, o):
    return Vector(self.a - o.a, self.b - o.b)

  def __iadd__(self, o):
    self.a += o.a
    self.b += o.b
    return self

  # def __isub__(self, o):

  def __str__(self):
    return "({}, {})".format(self.a, self.b)
```

```python
x = Vector(1, 2)
print(x, str(x), x.__str__())
```

```
(1, 2) (1, 2) (1, 2)
```

```python
y = Vector(2, 3)
print(y)
```

```
(2, 3)
```

```python
print(x + y, x.__add__(y))
```

```python
print(x - y)
```

```python
x += y
print(x)
```

```python
x -= y
print(x)
```

Magic Methods

```python
class Vector():

  def __init__(self, a, b):
    self.a = a
    self.b = b

  def __add__(self, o):
    return Vector(self.a + o.a, self.b + o.b)

  def __sub__(self, o):
    return Vector(self.a - o.a, self.b - o.b)

  def __iadd__(self, o):
    self.a += o.a
    self.b += o.b
    return self

  # def __isub__(self, o):

  def __str__(self):
    return "({}, {})".format(self.a, self.b)
```

```python
x = Vector(1, 2)
print(x, str(x), x.__str__())
```

```
(1, 2) (1, 2) (1, 2)
```

```python
y = Vector(2, 3)
print(y)
```

```
(2, 3)
```

```python
print(x + y, x.__add__(y))
```

```
(3, 5) (3, 5)
```

```python
print(x - y)
```

```python
x += y
print(x)

x -= y
print(x)
```

Magic Methods

```python
class Vector():

  def __init__(self, a, b):
    self.a = a
    self.b = b

  def __add__(self, o):
    return Vector(self.a + o.a, self.b + o.b)

  def __sub__(self, o):
    return Vector(self.a - o.a, self.b - o.b)

  def __iadd__(self, o):
    self.a += o.a
    self.b += o.b
    return self

  # def __isub__(self, o):

  def __str__(self):
    return "({}, {})".format(self.a, self.b)
```

```python
x = Vector(1, 2)
print(x, str(x), x.__str__())
```

```
(1, 2) (1, 2) (1, 2)
```

```python
y = Vector(2, 3)
print(y)
```

```
(2, 3)
```

```python
print(x + y, x.__add__(y))
```

```
(3, 5) (3, 5)
```

```python
print(x - y)
```

```
(-1, -1)
```

```python
x += y
print(x)

x -= y
print(x)
```

Magic Methods

```
class Vector():

  def __init__(self, a, b):
    self.a = a
    self.b = b

  def __add__(self, o):
    return Vector(self.a + o.a, self.b + o.b)

  def __sub__(self, o):
    return Vector(self.a - o.a, self.b - o.b)

  def __iadd__(self, o):
    self.a += o.a
    self.b += o.b
    return self

  # def __isub__(self, o):

  def __str__(self):
    return "({}, {})".format(self.a, self.b)
```

```
x = Vector(1, 2)
print(x, str(x), x.__str__())
```

```
(1, 2) (1, 2) (1, 2)
```

```
y = Vector(2, 3)
print(y)
```

```
(2, 3)
```

```
print(x + y, x.__add__(y))
```

```
(3, 5) (3, 5)
```

```
print(x - y)
```

```
(-1, -1)
```

```
x += y
print(x)
```

```
(3, 5)
```

```
x -= y
print(x)
```

Magic Methods

```
class Vector():

  def __init__(self, a, b):
    self.a = a
    self.b = b

  def __add__(self, o):
    return Vector(self.a + o.a, self.b + o.b)

  def __sub__(self, o):
    return Vector(self.a - o.a, self.b - o.b)

  def __iadd__(self, o):
    self.a += o.a
    self.b += o.b
    return self

  # def __isub__(self, o):

  def __str__(self):
    return "({}, {})".format(self.a, self.b)
```

```
x = Vector(1, 2)
print(x, str(x), x.__str__())
```

```
(1, 2) (1, 2) (1, 2)
```

```
y = Vector(2, 3)
print(y)
```

```
(2, 3)
```

```
print(x + y, x.__add__(y))
```

```
(3, 5) (3, 5)
```

```
print(x - y)
```

```
(-1, -1)
```

```
x += y
print(x)
```

```
(3, 5)
```

```
x -= y
print(x)
```

```
(1, 2)
```

Magic Methods

```python
class Vector():

  def __init__(self, a, b):
    self.a = a
    self.b = b

  def __add__(self, o):
    return Vector(self.a + o.a, self.b + o.b)

  def __sub__(self, o):
    return Vector(self.a - o.a, self.b - o.b)

  def __iadd__(self, o):
    self.a += o.a
    self.b += o.b
    return self

  # def __isub__(self, o):

  def __str__(self):
    return "({}, {})".format(self.a, self.b)
```

```python
x = Vector(1, 2)
print(x, str(x), x.__str__())
```

```
(1, 2) (1, 2) (1, 2)
```

```python
y = Vector(2, 3)
print(y)
```

```
(2, 3)
```

```python
print(x + y, x.__add__(y))
```

```
(3, 5) (3, 5)
```

```python
print(x - y)
```

```
(-1, -1)
```

```python
x += y
print(x)
```

```
(3, 5)
```

```python
x -= y
print(x)
```

```
(1, 2)
```

Warum geht es auch, obwohl __isub__ nicht definiert ist?

Magic Methods

```python
class Vector():

  def __init__(self, a, b):
    self.a = a
    self.b = b

  def __add__(self, o):
    return Vector(self.a + o.a, self.b + o.b)

  def __sub__(self, o):
    return Vector(self.a - o.a, self.b - o.b)

  def __iadd__(self, o):
    self.a += o.a
    self.b += o.b
    return self

  # def __isub__(self, o):

  def __str__(self):
    return "({}, {})".format(self.a, self.b)
```

Falls kein String, versucht `print()` einen mit `str()` zu erzeugen und bei einer Klasse geht dies per `__str__()`.

```python
x = Vector(1, 2)
print(x, str(x), x.__str__())
```

```
(1, 2) (1, 2) (1, 2)
```

```python
y = Vector(2, 3)
print(y)
```

```
(2, 3)
```

```python
print(x + y, x.__add__(y))
```

```
(3, 5) (3, 5)
```

```python
print(x - y)
```

```
(-1, -1)
```

```python
x += y
print(x)
```

```
(3, 5)
```

```python
x -= y
print(x)
```

```
(1, 2)
```

Warum geht es auch, obwohl `__isub__` nicht definiert ist?

Magic Methods

Mathe-matisch	__add__	+	Addition

https://docs.python.org/3/reference/datamodel.html
https://docs.python.org/3/library/operator.html#mapping-operators-to-functions

Magic Methods

Mathe-matisch	__add__	+	Addition
	__sub__	-	Subtraktion

https://docs.python.org/3/reference/datamodel.html
https://docs.python.org/3/library/operator.html#mapping-operators-to-functions

Magic Methods

Mathe-matisch	`__add__`	+	Addition
	`__sub__`	-	Subtraktion
	`__mul__`	*	Multiplikation

https://docs.python.org/3/reference/datamodel.html
https://docs.python.org/3/library/operator.html#mapping-operators-to-functions

Magic Methods

Mathe-matisch	`__add__`	`+`	Addition
	`__sub__`	`-`	Subtraktion
	`__mul__`	`*`	Multiplikation
	`__matmul__`	`@`	Matrixmultiplikation

https://docs.python.org/3/reference/datamodel.html
https://docs.python.org/3/library/operator.html#mapping-operators-to-functions

Magic Methods

Mathe-matisch	`__add__`	`+`	Addition
	`__sub__`	`-`	Subtraktion
	`__mul__`	`*`	Multiplikation
	`__matmul__`	`@`	Matrixmultiplikation
	`__truediv__`	`/`	Division

https://docs.python.org/3/reference/datamodel.html
https://docs.python.org/3/library/operator.html#mapping-operators-to-functions

Magic Methods

Mathe-matisch	`__add__`	`+`	Addition
	`__sub__`	`-`	Subtraktion
	`__mul__`	`*`	Multiplikation
	`__matmul__`	`@`	Matrixmultiplikation
	`__truediv__`	`/`	Division
	`__floordiv__`	`//`	Ganzzahldivision

https://docs.python.org/3/reference/datamodel.html
https://docs.python.org/3/library/operator.html#mapping-operators-to-functions

Magic Methods

Mathe-matisch	`__add__`	`+`	Addition
	`__sub__`	`-`	Subtraktion
	`__mul__`	`*`	Multiplikation
	`__matmul__`	`@`	Matrixmultiplikation
	`__truediv__`	`/`	Division
	`__floordiv__`	`//`	Ganzzahldivision
	`__pow__`	`**`	Potenz

https://docs.python.org/3/reference/datamodel.html
https://docs.python.org/3/library/operator.html#mapping-operators-to-functions

Magic Methods

Mathe-matisch	`__add__`	`+`	Addition
	`__sub__`	`-`	Subtraktion
	`__mul__`	`*`	Multiplikation
	`__matmul__`	`@`	Matrixmultiplikation
	`__truediv__`	`/`	Division
	`__floordiv__`	`//`	Ganzzahldivision
	`__pow__`	`**`	Potenz
Logisch	`__and__`	`&`	Bitweises Und

https://docs.python.org/3/reference/datamodel.html
https://docs.python.org/3/library/operator.html#mapping-operators-to-functions

Magic Methods

Mathe-matisch	`__add__`	`+`	Addition
	`__sub__`	`-`	Subtraktion
	`__mul__`	`*`	Multiplikation
	`__matmul__`	`@`	Matrixmultiplikation
	`__truediv__`	`/`	Division
	`__floordiv__`	`//`	Ganzzahldivision
	`__pow__`	`**`	Potenz
Logisch	`__and__`	`&`	Bitweises Und
	`__xor__`	`^`	Bitweises Entweder-Oder

https://docs.python.org/3/reference/datamodel.html
https://docs.python.org/3/library/operator.html#mapping-operators-to-functions

Magic Methods

Mathe-matisch	`__add__`	`+`	Addition
	`__sub__`	`-`	Subtraktion
	`__mul__`	`*`	Multiplikation
	`__matmul__`	`@`	Matrixmultiplikation
	`__truediv__`	`/`	Division
	`__floordiv__`	`//`	Ganzzahldivision
	`__pow__`	`**`	Potenz
Logisch	`__and__`	`&`	Bitweises Und
	`__xor__`	`^`	Bitweises Entweder-Oder
	`__or__`	`|`	Bitweises Oder

https://docs.python.org/3/reference/datamodel.html
https://docs.python.org/3/library/operator.html#mapping-operators-to-functions

Magic Methods

Mathe-matisch	`__add__`	+	Addition
	`__sub__`	-	Subtraktion
	`__mul__`	*	Multiplikation
	`__matmul__`	@	Matrixmultiplikation
	`__truediv__`	/	Division
	`__floordiv__`	//	Ganzzahldivision
	`__pow__`	**	Potenz
Logisch	`__and__`	&	Bitweises Und
	`__xor__`	^	Bitweises Entweder-Oder
	`__or__`	|	Bitweises Oder

Vergleichend	`__lt__`	<	Kleiner

https://docs.python.org/3/reference/datamodel.html
https://docs.python.org/3/library/operator.html#mapping-operators-to-functions

Magic Methods

Mathe-matisch	`__add__`	`+`	Addition
	`__sub__`	`-`	Subtraktion
	`__mul__`	`*`	Multiplikation
	`__matmul__`	`@`	Matrixmultiplikation
	`__truediv__`	`/`	Division
	`__floordiv__`	`//`	Ganzzahldivision
	`__pow__`	`**`	Potenz
Logisch	`__and__`	`&`	Bitweises Und
	`__xor__`	`^`	Bitweises Entweder-Oder
	`__or__`	`|`	Bitweises Oder

Vergleichend	`__lt__`	`<`	Kleiner
	`__le__`	`<=`	Kleiner gleich

https://docs.python.org/3/reference/datamodel.html
https://docs.python.org/3/library/operator.html#mapping-operators-to-functions

Magic Methods

Mathe-matisch	`__add__`	`+`	Addition
	`__sub__`	`-`	Subtraktion
	`__mul__`	`*`	Multiplikation
	`__matmul__`	`@`	Matrixmultiplikation
	`__truediv__`	`/`	Division
	`__floordiv__`	`//`	Ganzzahldivision
	`__pow__`	`**`	Potenz
Logisch	`__and__`	`&`	Bitweises Und
	`__xor__`	`^`	Bitweises Entweder-Oder
	`__or__`	`|`	Bitweises Oder

Vergleichend	`__lt__`	`<`	Kleiner
	`__le__`	`<=`	Kleiner gleich
	`__eq__`	`==`	Gleich

https://docs.python.org/3/reference/datamodel.html
https://docs.python.org/3/library/operator.html#mapping-operators-to-functions

Magic Methods

Mathe-matisch	`__add__`	`+`	Addition
	`__sub__`	`-`	Subtraktion
	`__mul__`	`*`	Multiplikation
	`__matmul__`	`@`	Matrixmultiplikation
	`__truediv__`	`/`	Division
	`__floordiv__`	`//`	Ganzzahldivision
	`__pow__`	`**`	Potenz
Logisch	`__and__`	`&`	Bitweises Und
	`__xor__`	`^`	Bitweises Entweder-Oder
	`__or__`	`|`	Bitweises Oder

Vergleichend	`__lt__`	`<`	Kleiner
	`__le__`	`<=`	Kleiner gleich
	`__eq__`	`==`	Gleich
	`__ne__`	`!=`	Ungleich

https://docs.python.org/3/reference/datamodel.html
https://docs.python.org/3/library/operator.html#mapping-operators-to-functions

Magic Methods

Mathe-matisch	`__add__`	`+`	Addition
	`__sub__`	`-`	Subtraktion
	`__mul__`	`*`	Multiplikation
	`__matmul__`	`@`	Matrixmultiplikation
	`__truediv__`	`/`	Division
	`__floordiv__`	`//`	Ganzzahldivision
	`__pow__`	`**`	Potenz
Logisch	`__and__`	`&`	Bitweises Und
	`__xor__`	`^`	Bitweises Entweder-Oder
	`__or__`	`|`	Bitweises Oder

Vergleichend	`__lt__`	`<`	Kleiner
	`__le__`	`<=`	Kleiner gleich
	`__eq__`	`==`	Gleich
	`__ne__`	`!=`	Ungleich
	`__gt__`	`>`	Größer

https://docs.python.org/3/reference/datamodel.html
https://docs.python.org/3/library/operator.html#mapping-operators-to-functions

Magic Methods

Mathe-matisch	`__add__`	`+`	Addition
	`__sub__`	`-`	Subtraktion
	`__mul__`	`*`	Multiplikation
	`__matmul__`	`@`	Matrixmultiplikation
	`__truediv__`	`/`	Division
	`__floordiv__`	`//`	Ganzzahldivision
	`__pow__`	`**`	Potenz
Logisch	`__and__`	`&`	Bitweises Und
	`__xor__`	`^`	Bitweises Entweder-Oder
	`__or__`	`|`	Bitweises Oder

Vergleichend	`__lt__`	`<`	Kleiner
	`__le__`	`<=`	Kleiner gleich
	`__eq__`	`==`	Gleich
	`__ne__`	`!=`	Ungleich
	`__gt__`	`>`	Größer
	`__ge__`	`>=`	Größer gleich

https://docs.python.org/3/reference/datamodel.html
https://docs.python.org/3/library/operator.html#mapping-operators-to-functions

Magic Methods

Mathematisch	`__add__`	`+`	Addition
	`__sub__`	`-`	Subtraktion
	`__mul__`	`*`	Multiplikation
	`__matmul__`	`@`	Matrixmultiplikation
	`__truediv__`	`/`	Division
	`__floordiv__`	`//`	Ganzzahldivision
	`__pow__`	`**`	Potenz
Logisch	`__and__`	`&`	Bitweises Und
	`__xor__`	`^`	Bitweises Entweder-Oder
	`__or__`	`|`	Bitweises Oder

Vergleichend	`__lt__`	`<`	Kleiner
	`__le__`	`<=`	Kleiner gleich
	`__eq__`	`==`	Gleich
	`__ne__`	`!=`	Ungleich
	`__gt__`	`>`	Größer
	`__ge__`	`>=`	Größer gleich
Klassen (-eigenschaften)	`__bool__`	Bool eines Objekts (`if obj:`)	

https://docs.python.org/3/reference/datamodel.html
https://docs.python.org/3/library/operator.html#mapping-operators-to-functions

Magic Methods

Mathe-matisch	`__add__`	`+`	Addition
	`__sub__`	`-`	Subtraktion
	`__mul__`	`*`	Multiplikation
	`__matmul__`	`@`	Matrixmultiplikation
	`__truediv__`	`/`	Division
	`__floordiv__`	`//`	Ganzzahldivision
	`__pow__`	`**`	Potenz
Logisch	`__and__`	`&`	Bitweises Und
	`__xor__`	`^`	Bitweises Entweder-Oder
	`__or__`	`|`	Bitweises Oder

Vergleichend	`__lt__`	`<`	Kleiner
	`__le__`	`<=`	Kleiner gleich
	`__eq__`	`==`	Gleich
	`__ne__`	`!=`	Ungleich
	`__gt__`	`>`	Größer
	`__ge__`	`>=`	Größer gleich
Klassen (-eigenschaften)	`__bool__`	Bool eines Objekts (`if obj:`)	
	`__hash__`	Eindeutiger Hash des Objekts	

https://docs.python.org/3/reference/datamodel.html
https://docs.python.org/3/library/operator.html#mapping-operators-to-functions

Magic Methods

Mathematisch	`__add__`	`+`	Addition
	`__sub__`	`-`	Subtraktion
	`__mul__`	`*`	Multiplikation
	`__matmul__`	`@`	Matrixmultiplikation
	`__truediv__`	`/`	Division
	`__floordiv__`	`//`	Ganzzahldivision
	`__pow__`	`**`	Potenz
Logisch	`__and__`	`&`	Bitweises Und
	`__xor__`	`^`	Bitweises Entweder-Oder
	`__or__`	`|`	Bitweises Oder

Vergleichend	`__lt__`	`<`	Kleiner
	`__le__`	`<=`	Kleiner gleich
	`__eq__`	`==`	Gleich
	`__ne__`	`!=`	Ungleich
	`__gt__`	`>`	Größer
	`__ge__`	`>=`	Größer gleich
Klassen (-eigenschaften)	`__bool__`	Bool eines Objekts (`if obj:`)	
	`__hash__`	Eindeutiger Hash des Objekts	
	`__len__`	Länge des Objekts	

https://docs.python.org/3/reference/datamodel.html
https://docs.python.org/3/library/operator.html#mapping-operators-to-functions

Magic Methods

Mathe-matisch	`__add__`	`+`	Addition
	`__sub__`	`-`	Subtraktion
	`__mul__`	`*`	Multiplikation
	`__matmul__`	`@`	Matrixmultiplikation
	`__truediv__`	`/`	Division
	`__floordiv__`	`//`	Ganzzahldivision
	`__pow__`	`**`	Potenz
Logisch	`__and__`	`&`	Bitweises Und
	`__xor__`	`^`	Bitweises Entweder-Oder
	`__or__`	`|`	Bitweises Oder

Vergleichend	`__lt__`	`<`	Kleiner
	`__le__`	`<=`	Kleiner gleich
	`__eq__`	`==`	Gleich
	`__ne__`	`!=`	Ungleich
	`__gt__`	`>`	Größer
	`__ge__`	`>=`	Größer gleich
Klassen (-eigenschaften)	`__bool__`	Bool eines Objekts (`if obj:`)	
	`__hash__`	Eindeutiger Hash des Objekts	
	`__len__`	Länge des Objekts	
	`__str__`	String (Ausgabe)	

https://docs.python.org/3/reference/datamodel.html
https://docs.python.org/3/library/operator.html#mapping-operators-to-functions

Magic Methods

Mathe-matisch	`__add__`	`+`	Addition
	`__sub__`	`-`	Subtraktion
	`__mul__`	`*`	Multiplikation
	`__matmul__`	`@`	Matrixmultiplikation
	`__truediv__`	`/`	Division
	`__floordiv__`	`//`	Ganzzahldivision
	`__pow__`	`**`	Potenz
Logisch	`__and__`	`&`	Bitweises Und
	`__xor__`	`^`	Bitweises Entweder-Oder
	`__or__`	`|`	Bitweises Oder

Vergleichend	`__lt__`	`<`	Kleiner
	`__le__`	`<=`	Kleiner gleich
	`__eq__`	`==`	Gleich
	`__ne__`	`!=`	Ungleich
	`__gt__`	`>`	Größer
	`__ge__`	`>=`	Größer gleich
Klassen (-eigenschaften)	`__bool__`	Bool eines Objekts (`if obj:`)	
	`__hash__`	Eindeutiger Hash des Objekts	
	`__len__`	Länge des Objekts	
	`__str__`	String (Ausgabe)	
	`__del__`	Objekt löschen (`del obj`)	

https://docs.python.org/3/reference/datamodel.html
https://docs.python.org/3/library/operator.html#mapping-operators-to-functions

Magic Methods: dict & list

```
class DictContainer():

  def __init__(self):
    self.dict = {}

  def __setitem__(self, key, value):
    self.dict[key] = value

  def __getitem__(self, key):
    return self.dict[key]

  def __delitem__(self, key):
    del self.dict[key]

  def __contains__(self, key):
    return key in self.dict

  def __str__(self):
    return str(self.dict)
```

Magic Methods: dict & list

```
class DictContainer():

  def __init__(self):
    self.dict = {}

  def __setitem__(self, key, value):
    self.dict[key] = value

  def __getitem__(self, key):
    return self.dict[key]

  def __delitem__(self, key):
    del self.dict[key]

  def __contains__(self, key):
    return key in self.dict

  def __str__(self):
    return str(self.dict)
```

```
dc = DictContainer()

dc["a"] = "A"
dc.__setitem__("b", "B")
print(dc)

print(dc["a"])
print(dc.__getitem__("b"))

del dc["a"]
print(dc)

print("2" in dc, "b" in dc)
```

Magic Methods: dict & list

```
class DictContainer():

  def __init__(self):
    self.dict = {}

  def __setitem__(self, key, value):
    self.dict[key] = value

  def __getitem__(self, key):
    return self.dict[key]

  def __delitem__(self, key):
    del self.dict[key]

  def __contains__(self, key):
    return key in self.dict

  def __str__(self):
    return str(self.dict)
```

```
dc = DictContainer()

dc["a"] = "A"
dc.__setitem__("b", "B")
print(dc)

print(dc["a"])
print(dc.__getitem__("b"))

del dc["a"]
print(dc)

print("2" in dc, "b" in dc)
```

```
{'a': 'A', 'b': 'B'}
```

Magic Methods: dict & list

```python
class DictContainer():

  def __init__(self):
    self.dict = {}

  def __setitem__(self, key, value):
    self.dict[key] = value

  def __getitem__(self, key):
    return self.dict[key]

  def __delitem__(self, key):
    del self.dict[key]

  def __contains__(self, key):
    return key in self.dict

  def __str__(self):
    return str(self.dict)
```

```python
dc = DictContainer()

dc["a"] = "A"
dc.__setitem__("b", "B")
print(dc)

print(dc["a"])
print(dc.__getitem__("b"))

del dc["a"]
print(dc)

print("2" in dc, "b" in dc)
```

```
{'a': 'A', 'b': 'B'}

A
```

Magic Methods: dict & list

```python
class DictContainer():

  def __init__(self):
    self.dict = {}

  def __setitem__(self, key, value):
    self.dict[key] = value

  def __getitem__(self, key):
    return self.dict[key]

  def __delitem__(self, key):
    del self.dict[key]

  def __contains__(self, key):
    return key in self.dict

  def __str__(self):
    return str(self.dict)
```

```python
dc = DictContainer()

dc["a"] = "A"
dc.__setitem__("b", "B")
print(dc)

print(dc["a"])
print(dc.__getitem__("b"))

del dc["a"]
print(dc)

print("2" in dc, "b" in dc)
```

```
{'a': 'A', 'b': 'B'}
A
B
```

Magic Methods: dict & list

```python
class DictContainer():

  def __init__(self):
    self.dict = {}

  def __setitem__(self, key, value):
    self.dict[key] = value

  def __getitem__(self, key):
    return self.dict[key]

  def __delitem__(self, key):
    del self.dict[key]

  def __contains__(self, key):
    return key in self.dict

  def __str__(self):
    return str(self.dict)
```

```python
dc = DictContainer()

dc["a"] = "A"
dc.__setitem__("b", "B")
print(dc)

print(dc["a"])
print(dc.__getitem__("b"))

del dc["a"]
print(dc)

print("2" in dc, "b" in dc)
```

```
{'a': 'A', 'b': 'B'}
A
B
{'b': 'B'}
```

Magic Methods: dict & list

```python
class DictContainer():

  def __init__(self):
    self.dict = {}

  def __setitem__(self, key, value):
    self.dict[key] = value

  def __getitem__(self, key):
    return self.dict[key]

  def __delitem__(self, key):
    del self.dict[key]

  def __contains__(self, key):
    return key in self.dict

  def __str__(self):
    return str(self.dict)
```

```python
dc = DictContainer()

dc["a"] = "A"
dc.__setitem__("b", "B")
print(dc)
```

```
{'a': 'A', 'b': 'B'}
```

```python
print(dc["a"])
print(dc.__getitem__("b"))
```

```
A
B
```

```python
del dc["a"]
print(dc)
```

```
{'b': 'B'}
```

```python
print("2" in dc, "b" in dc)
```

```
False True
```

Generatoren

```
import timeit

lc_a = """
[i for i in range(2**10)]
"""
print(timeit.timeit(lc_a, number=20))
```

```
import timeit

g_a = """
(i for i in range(2**10))
"""
print(timeit.timeit(g_a, number=20))
```

```
lc_b = """
[i for i in range(2**20)]
"""
print(timeit.timeit(lc_b, number=20))
```

```
g_b = """
(i for i in range(2**20))
"""
print(timeit.timeit(g_b, number=20))
```

Generatoren

```
import timeit

lc_a = """
[i for i in range(2**10)]
"""
print(timeit.timeit(lc_a, number=20))
```

```
0.0007693090010434389
```

```
lc_b = """
[i for i in range(2**20)]
"""
print(timeit.timeit(lc_b, number=20))
```

```
0.9470873650006979
```

```
import timeit

g_a = """
(i for i in range(2**10))
"""
print(timeit.timeit(g_a, number=20))
```

```
1.0978999853250 57e-05
```

```
g_b = """
(i for i in range(2**20))
"""
print(timeit.timeit(g_b, number=20))
```

```
2.2750000425730832e-05
```

Generatoren

```
import timeit

lc_a = """
[i for i in range(2**10)]
"""
print(timeit.timeit(lc_a, number=20))
```

```
0.0007693090010434389
```

```
lc_b = """
[i for i in range(2**20)]
"""
print(timeit.timeit(lc_b, number=20))
```

```
0.9470873650006979
```

```
import timeit

g_a = """
(i for i in range(2**10))
"""
print(timeit.timeit(g_a, number=20))
```

```
1.0978999853250 57e-05
```

```
g_b = """
(i for i in range(2**20))
"""
print(timeit.timeit(g_b, number=20))
```

```
2.2750000425730832e-05
```

Links hängt die Dauer stark von der Anzahl ab, rechts nicht. Warum?

Generatoren

Generator Expression ( )
List Comprehension [ ]

```
import timeit

lc_a = """
[i for i in range(2**10)]
"""
print(timeit.timeit(lc_a, number=20))
```

```
0.0007693090010434389
```

```
lc_b = """
[i for i in range(2**20)]
"""
print(timeit.timeit(lc_b, number=20))
```

```
0.9470873650006979
```

```
import timeit

g_a = """
(i for i in range(2**10))
"""
print(timeit.timeit(g_a, number=20))
```

```
1.07899985325057e-05
```

```
g_b = """
(i for i in range(2**20))
"""
print(timeit.timeit(g_b, number=20))
```

```
2.2750000425730832e-05
```

Links hängt die Dauer stark von der Anzahl ab, rechts nicht. Warum?

Generatoren: `yield`

```
def list_range(until):
  l = []
  i = 0
  while i < until:
    l.append(i)
    i += 1
  return l

print(list_range(20))
for i in list_range(5):
  print(i)
```

Generatoren: `yield`

```
def list_range(until):
  l = []
  i = 0
  while i < until:
    l.append(i)
    i += 1
  return l

print(list_range(20))
for i in list_range(5):
  print(i)
```

```
[0, 1, 2, 3, 4, 5, … 18, 19]
0
1
2
3
4
```

Generatoren: `yield`

```python
def list_range(until):
  l = []
  i = 0
  while i < until:
    l.append(i)
    i += 1
  return l

print(list_range(20))
for i in list_range(5):
  print(i)
```

```
[0, 1, 2, 3, 4, 5, … 18, 19]
0
1
2
3
4
```

```python
def generate_range(until):
  i = 0
  while i < until:
    yield i
    i += 1

print(generate_range(20))
for i in generate_range(5):
  print(i)
```

Generatoren: `yield`

```python
def list_range(until):
  l = []
  i = 0
  while i < until:
    l.append(i)
    i += 1
  return l

print(list_range(20))
for i in list_range(5):
  print(i)
```

```
[0, 1, 2, 3, 4, 5, … 18, 19]
0
1
2
3
4
```

```python
def generate_range(until):
  i = 0
  while i < until:
    yield i
    i += 1

print(generate_range(20))
for i in generate_range(5):
  print(i)
```

```
<generator object … at 0x10a3bc040>
0
1
2
3
4
```

neratoren: `yield`

Eine ganze Liste mit den einzelnen Zahlen wird erstellt und danach zurückgegeben.

```
def list_range(until):
  l = []
  i = 0
  while i < until:
    l.append(i)
    i += 1
  return l

print(list_range(20))
for i in list_range(5):
  print(i)
```

```
[0, 1, 2, 3, 4, 5, … 18, 19]
0
1
2
3
4
```

Es wird immer nur der nächste Wert erzeugt, dieser zurückgegben, verarbeitet und dann der nächste erzeigt.

```
def generate_range(until):
  i = 0
  while i < until:
    yield i
    i += 1

print(generate_range(20))
for i in generate_range(5):
  print(i)
```

```
<generator object … at 0x10a3bc040>
0
1
2
3
4
```

Iteration mit Generatoren

```
class FileReader():

  def __init__(self, filename):
    self.filename = filename

  def __iter__(self):
    t = open(self.filename + ".titles.txt", "r")
    c = open(self.filename + ".contents.txt", "r")

    t_l = t.readline()
    c_l = c.readline()

    while t_l and c_l:
      yield t_l.strip(), c_l.strip().split(",")

      t_l = t.readline()
      c_l = c.readline()

    t.close()
    c.close()
```

Iteration mit Generatoren

```
class FileReader():

  def __init__(self, filename):
    self.filename = filename

  def __iter__(self):
    t = open(self.filename + ".titles.txt", "r")
    c = open(self.filename + ".contents.txt", "r")

    t_l = t.readline()
    c_l = c.readline()

    while t_l and c_l:
      yield t_l.strip(), c_l.strip().split(",")

      t_l = t.readline()
      c_l = c.readline()

    t.close()
    c.close()
```

e.contents.txt

```
Audi,BMW,Ford
Apple,Banana,Pear
Mouse,Screen,Keyboard
```

e.titles.txt

```
Car
Fruit
Computer
```

```
fr = FileReader("e")
for t, c in fr:
  print(t, c)
```

Iteration mit Generatoren

```
class FileReader():

  def __init__(self, filename):
    self.filename = filename

  def __iter__(self):
    t = open(self.filename + ".titles.txt", "r")
    c = open(self.filename + ".contents.txt", "r")

    t_l = t.readline()
    c_l = c.readline()

    while t_l and c_l:
      yield t_l.strip(), c_l.strip().split(",")

      t_l = t.readline()
      c_l = c.readline()

    t.close()
    c.close()
```

e.contents.txt

```
Audi,BMW,Ford
Apple,Banana,Pear
Mouse,Screen,Keyboard
```

e.titles.txt

```
Car
Fruit
Computer
```

```
fr = FileReader("e")
for t, c in fr:
  print(t, c)
```

```
Car ['Audi', 'BMW', 'Ford']
Fruit ['Apple', 'Banana', 'Pear']
Computer ['Mouse', 'Screen', 'Keyboard']
```

Iteration mit Generatoren

```
class FileReader():

  def __init__(self, filename):
    self.filename = filename

  def __iter__(self):
    t = open(self.filename + ".titles.txt", "r")
    c = open(self.filename + ".contents.txt", "r")

    t_l = t.readline()
    c_l = c.readline()

    while t_l and c_l:
      yield t_l.strip(), c_l.strip().split(",")

      t_l = t.readline()
      c_l = c.readline()

    t.close()
    c.close()
```

Iteration über ein Objekt durch Nutzung von `yield`.

Auch bei großen Dateien bleibt der Speicherverbrauch konstant. Laufzeit erhöht sich aber leicht.

Implizierer Aufruf der *Magic Method* `__iter__()`.

e.contents.txt

```
Audi,BMW,Ford
Apple,Banana,Pear
Mouse,Screen,Keyboard
```

e.titles.txt

```
Car
Fruit
Computer
```

```
fr = FileReader("e")
for t, c in fr:
  print(t, c)
```

```
Car ['Audi', 'BMW', 'Ford']
Fruit ['Apple', 'Banana', 'Pear']
Computer ['Mouse', 'Screen', 'Keyboard']
```

Erweiterte Schleifen

- List Comprehensions für Wörterbücher

Erweiterte Schleifen

- List Comprehensions für Wörterbücher

```
{i : i**2 for i in range(5)}
```

```
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16}
```

Erweiterte Schleifen

- List Comprehensions für Wörterbücher

```
{i : i**2 for i in range(5)}
```

```
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16}
```

- „Break“ und „Continue“

Erweiterte Schleifen

- List Comprehensions für Wörterbücher

```
{i : i**2 for i in range(5)}
```

```
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16}
```

- „Break“ und „Continue“

```
for i in range(20):
  if i < 2:
    continue
  elif i > 5:
    break

  print(i)
```

```
2
3
4
5
```

Erweiterte Schleifen

- List Comprehensions für Wörterbücher

```
{i : i**2 for i in range(5)}
```

```
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16}
```

- „Else“ nach Schleifen

- „Break“ und „Continue“

```
for i in range(20):
  if i < 2:
    continue
  elif i > 5:
    break

  print(i)
```

```
2
3
4
5
```

Erweiterte Schleifen

- List Comprehensions für Wörterbücher

```
{i : i**2 for i in range(5)}
```

```
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16}
```

- „Else“ nach Schleifen

```
for i in range(4):
  if i == 5:
    print("Found")
    break
else:
  print("Not found!")
```

```
Not found!
```

- „Break“ und „Continue“

```
for i in range(20):
  if i < 2:
    continue
  elif i > 5:
    break

  print(i)
```

```
2
3
4
5
```

Erweiterte Schleifen

- List Comprehensions für Wörterbücher

```
{i : i**2 for i in range(5)}
```

```
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16}
```

- „Else“ nach Schleifen

```
for i in range(4):
  if i == 5:
    print("Found")
    break
else:
  print("Not found!")
```

```
Not found!
```

```
for i in range(8):
  if i == 5:
    print("Found")
    break
else:
  print("Not found!")
```

```
Found!
```

- „Break“ und „Continue“

```
for i in range(20):
  if i < 2:
    continue
  elif i > 5:
    break

  print(i)
```

```
2
3
4
5
```

Erweiterte Schleifen

- List Comprehensions für Wörterbücher

```
{i : i**2 for i in range(5)}
```

```
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16}
```

- „Break“ und „Continue“

```
for i in range(20):
  if i < 2:
    continue
  elif i > 5:
    break

  print(i)
```

```
2
3
4
5
```

- „Else“ nach Schleifen

```
for i in range(4):
  if i == 5:
    print("Found")
    break
else:
  print("Not found!")
```

```
Not found!
```

```
for i in range(8):
  if i == 5:
    print("Found")
    break
else:
  print("Not found!")
```

```
Found!
```

Else unter einer Schleife wird also genau dann ausgeführt, wenn die Schleife durchgelaufen ist und nicht abgebrochen wurde.

Fehlerbehandlung

```
def divide(x, y):
  try:
    r = x / y
  except ZeroDivisionError:
    print("Division by zero!")
  except TypeError as e:
    print("TypeError:", e)
  else:
    print("{x} / {y} = {r}".format(
     x=x, y=y, r=r
    ))
  finally:
    print("Finally done ;)")
```


https://docs.python.org/3/tutorial/errors.html#defining-clean-up-actions

Fehlerbehandlung

```python
def divide(x, y):
  try:
    r = x / y
  except ZeroDivisionError:
    print("Division by zero!")
  except TypeError as e:
    print("TypeError:", e)
  else:
    print("{x} / {y} = {r}".format(
     x=x, y=y, r=r
    ))
  finally:
    print("Finally done ;)")
```

```python
divide(1, 2)
```

```python
divide(1, 0)
```

```python
divide("A", 2)
```

https://docs.python.org/3/tutorial/errors.html#defining-clean-up-actions

Fehlerbehandlung

```python
def divide(x, y):
  try:
    r = x / y
  except ZeroDivisionError:
    print("Division by zero!")
  except TypeError as e:
    print("TypeError:", e)
  else:
    print("{x} / {y} = {r}".format(
     x=x, y=y, r=r
    ))
  finally:
    print("Finally done ;)")
```

```python
divide(1, 2)
```

```
1 / 2 = 0.5
Finally done ;)
```

```python
divide(1, 0)
```

```
Division by zero!
Finally done ;)
```

```python
divide("A", 2)
```

```
TypeError: unsupported operand type(s) for /: 'str' and 'int'
Finally done ;)
```

Fehlerbehandlung

```python
def divide(x, y):
  try:
    r = x / y
  except ZeroDivisionError:
    print("Division by zero!")
  except TypeError as e:
    print("TypeError:", e)
  else:
    print("{x} / {y} = {r}".format(
     x=x, y=y, r=r
    ))
  finally:
    print("Finally done ;)")
```

```python
divide(1, 2)
```

```
1 / 2 = 0.5
Finally done ;)
```

```python
divide(1, 0)
```

```
Division by zero!
Finally done ;)
```

```python
divide("A", 2)
```

Fehlerbehandlung

```python
def divide(x, y):
  try:
    r = x / y
  except ZeroDivisionError:
    print("Division by zero!")
  except TypeError as e:
    print("TypeError:", e)
  else:
    print("{x} / {y} = {r}".format(
     x=x, y=y, r=r
    ))
  finally:
    print("Finally done ;)")
```

```
divide(1, 2)
```

```
1 / 2 = 0.5
Finally done ;)
```

```
divide(1, 0)
```

```
Division by zero!
Finally done ;)
```

```
divide("A", 2)
```

```
TypeError: unsupported operand type(s)
for /: 'str' and 'int'
Finally done ;)
```

Fehlerbehandlung

```python
def divide(x, y):
    try:
        r = x / y
    except ZeroDivisionError:
        print("Division by zero!")
    except TypeError as e:
        print("TypeError:", e)
    else:
        print("{x} / {y} = {r}".format(
            x=x, y=y, r=r
        ))
    finally:
        print("Finally done ;)")
```

```
divide(1, 2)
```

```
1 / 2 = 0.5
Finally done ;)
```

```
divide(1, 0)
```

```
Division by zero!
Finally done ;)
```

```
divide("A", 2)
```

```
TypeError: unsupported operand type(s)
for /: 'str' and 'int'
Finally done ;)
```

Wozu könnte `finally` nützlich sein?

Fehlerbehandlung

```python
def divide(x, y):
    try:
        r = x / y
    except ZeroDivisionError:
        print("Division by zero!")
    except TypeError as e:
        print("TypeError:", e)
    else:
        print("{x} / {y} = {r}".format(
            x=x, y=y, r=r
        ))
    finally:
        print("Finally done ;)")
```

```
divide(1, 2)
```

```
1 / 2 = 0.5
Finally done ;)
```

```
divide(1, 0)
```

```
Division by zero!
Finally done ;)
```

```
divide("A", 2)
```

```
TypeError: unsupported operand type(s)
for /: 'str' and 'int'
Finally done ;)
```

Verschiedene Fehler werden unterschieden und könne als Variable abgegriffen werden.

Falls kein Fehler auftrat.

Immer, egal ob Fehler oder nicht.

Wozu könnte `finally` nützlich sein?

Fehlerbehandlung

- Fehler können mit `raise` ausgelöst werden
- Fehler sind Objekte der Klasse (einer Unterklasse von) `Exception`
  - Vordefinierte Fehler: https://docs.python.org/3/library/exceptions.html#concrete-exceptions
  - Eigene Fehler können als Unterklasse erstellt werden

Fehlerbehandlung

- Fehler können mit `raise` ausgelöst werden
- Fehler sind Objekte der Klasse (einer Unterklasse von) `Exception`
  - Vordefinierte Fehler: https://docs.python.org/3/library/exceptions.html#concrete-exceptions
  - Eigene Fehler können als Unterklasse erstellt werden

```
class MyError(Exception):
  pass

raise MyError("Stopp")
```

```
Traceback (most recent call last):
  File "name.py", line 4, in <module>
    raise MyError("Stopp")
__main__.MyError: Stopp
```

Fehlerbehandlung

- Fehler können mit `raise` ausgelöst werden
- Fehler sind Objekte der Klasse (einer Unterklasse von) `Exception`
  - Vordefinierte Fehler: https://docs.python.org/3/library/exceptions.html#concrete-exceptions
  - Eigene Fehler können als Unterklasse erstellt werden

Wofür könnte eine solcher Fehlerklasse ohne weitere Implementierung nützlich sein?

```
class MyError(Exception):
  pass

raise MyError("Stopp")
```

```
Traceback (most recent call last):
  File "name.py", line 4, in <module>
    raise MyError("Stopp")
__main__.MyError: Stopp
```

Kontextmanager

```
try:

  t = open("example.titles.txt", "r")
  t.write("huhu")
  t.close()

except BaseException as e:
  print(e)
```

Kontextmanager

Wo ist hier der Fehler?

```
try:

  t = open("example.titles.txt", "r")
  t.write("huhu")
  t.close()

except BaseException as e:
  print(e)
```

Kontextmanager

Wo ist hier der Fehler?

```
try:

  t = open("example.titles.txt", "r")
  t.write("huhu")
  t.close()

except BaseException as e:
  print(e)

print(t.readline())
```

Kontextmanager

Wo ist hier der Fehler?

```
try:

  t = open("example.titles.txt", "r")
  t.write("huhu")
  t.close()

except BaseException as e:
  print(e)

print(t.readline())
```

```
not writable

Car
```

Kontextmanager

Wo ist hier der Fehler?

```
try:

  t = open("example.titles.txt", "r")
  t.write("huhu")
  t.close()

except BaseException as e:
  print(e)

print(t.readline())
```

```
not writable

Car
```

```
try:

  with open("example.contents.txt", "r") as c:
    c.write("huhu")


except BaseException as e:
  print(e)

print(c.readline())
```

Wo ist hier der Fehler?

Kontextmanager

```
try:

  t = open("example.titles.txt", "r")
  t.write("huhu")
  t.close()

except BaseException as e:
  print(e)

print(t.readline())
```

```
not writable

Car
```

```
try:

  with open("example.contents.txt", "r") as c:
    c.write("huhu")


except BaseException as e:
  print(e)

print(c.readline())
```

```
not writable

Traceback (most recent call last):
  File "name.py", line 18, in <module>
    print(c.readline())
ValueError: I/O operation on closed file.
```

Kontextmanager

Wo ist hier der Fehler?

```
try:

  t = open("example.titles.txt", "r")
  t.write("huhu")
  t.close()

except BaseException as e:
  print(e)

print(t.readline())
```

```
not writable

Car
```

```
try:

  with open("example.contents.txt", "r") as c:
    c.write("huhu")


except BaseException as e:
  print(e)

print(c.readline
```

```
not writable

Traceback (most recent call last):
  File "name.py", line 18, in <module>
    print(c.readline())
ValueError: I/O operation on closed file.
```

Intern durch die Magic Methods __enter__() und __exit__() realisiert.

Kontextmanager

Wo ist hier der Fehler?

```
try:

  t = open("example.titles.txt", "r")
  t.write("huhu")
  t.close()

except BaseException as e:
  print(e)

print(t.readline())
```

```
not writable

Car
```

Wenn `t.write()` einen Fehler auslöst, dann wird `t.close()` nicht ausgeführt und die Datei bleibt offen!

```
try:

  with open("example.contents.txt", "r") as c:
    c.write("huhu")


except BaseException as e:
  print(e)

print(c.readline
```

```
not writable

Traceback (most recent call last):
  File "name.py", line 18, in <module>
    print(c.readline())
ValueError: I/O operation on closed file.
```

Intern durch die Magic Methods `__enter__()` und `__exit__()` realisiert.

Lambda-Funktionen

```
def pow2(number):
  return number ** 2

numbers = [1, 2, 3, 4]
for i, n in enumerate(numbers):
  numbers[i] = pow2(n)
print(numbers)
```

Lambda-Funktionen

```
def pow2(number):
  return number ** 2

numbers = [1, 2, 3, 4]
for i, n in enumerate(numbers):
  numbers[i] = pow2(n)
print(numbers)
```

```
[1, 4, 9, 16]
```

Lambda-Funktionen

```
def pow2(number):
  return number ** 2

numbers = [1, 2, 3, 4]
for i, n in enumerate(numbers):
  numbers[i] = pow2(n)
print(numbers)
```

```
[1, 4, 9, 16]
```

```
numbers = [1, 2, 3, 4]
numbers = [ pow2(n) for n in numbers ]
print(numbers)
```

```
[1, 4, 9, 16]
```

Lambda-Funktionen

```
def pow2(number):
  return number ** 2

numbers = [1, 2, 3, 4]
for i, n in enumerate(numbers):
  numbers[i] = pow2(n)
print(numbers)
```

```
[1, 4, 9, 16]
```

```
numbers = [1, 2, 3, 4]
numbers = [ pow2(n) for n in numbers ]
print(numbers)
```

```
[1, 4, 9, 16]
```

```
numbers = [1, 2, 3, 4]
numbers = map(pow2, numbers)
print(numbers, list(numbers))
```

```
<map object at 0x10e0a2830> [1, 4, 9, 16]
```

Lambda-Funktionen

```
def pow2(number):
  return number ** 2

numbers = [1, 2, 3, 4]
for i, n in enumerate(numbers):
  numbers[i] = pow2(n)
print(numbers)
```

```
[1, 4, 9, 16]
```

```
numbers = [1, 2, 3, 4]
numbers = [ pow2(n) for n in numbers ]
print(numbers)
```

```
[1, 4, 9, 16]
```

```
numbers = [1, 2, 3, 4]
numbers = map(pow2, numbers)
print(numbers, list(numbers))
```

```
<map object at 0x10e0a2830> [1, 4, 9, 16]
```

```
numbers = [1, 2, 3, 4]
numbers = map(lambda n : n**2, numbers)
print(numbers, list(numbers))
```

```
<map object at 0x10e0a3280> [1, 4, 9, 16]
```

Lambda-Funktionen

```python
def pow2(number):
  return number ** 2

numbers = [1, 2, 3, 4]
for i, n in enumerate(numbers):
  numbers[i] = pow2(n)
print(numbers)
```

```
[1, 4, 9, 16]
```

```python
numbers = [1, 2, 3, 4]
numbers = [ pow2(n) for n in numbers ]
print(numbers)
```

```
[1, 4, 9, 16]
```

```python
numbers = [1, 2, 3, 4]
numbers = map(pow2, numbers)
print(numbers, list(numbers))
```

```
<map object at 0x10e0a2830> [1, 4, 9, 16]
```

```python
numbers = [1, 2, 3, 4]
numbers = map(lambda n : n**2, numbers)
print(numbers, list(numbers))
```

```
<map object at 0x10e0a3280> [1, 4, 9, 16]
```

```python
def pow2_f(number):
  return number ** 2

pow2_l = lambda n : n**2
```

Lambda-Funktionen

```python
def pow2(number):
  return number ** 2

numbers = [1, 2, 3, 4]
for i, n in enumerate(numbers):
  numbers[i] = pow2(n)
print(numbers)
```

```
[1, 4, 9, 16]
```

```python
numbers = [1, 2, 3, 4]
numbers = [ pow2(n) for n in numbers ]
print(numbers)
```

```
[1, 4, 9, 16]
```

```python
numbers = [1, 2, 3, 4]
numbers = map(pow2, numbers)
print(numbers, list(numbers))
```

```
<map object at 0x10e0a2830> [1, 4, 9, 16]
```

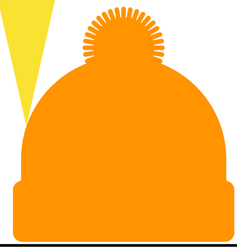
```python
numbers = [1, 2, 3, 4]
numbers = map(lambda n : n**2, numbers)
print(numbers, list(numbers))
```

```
<map object at 0x10e0a3280> [1, 4, 9, 16]
```

```python
def pow2_f(number):
  return number ** 2

pow2_l = lambda n : n**2
```

Sehr praktisch für z.B.

```
sort(key=lambda o: o.name)
```



Lambda-Funktionen

```
def pow2(number):
  return number ** 2

numbers = [1, 2, 3, 4]
for i, n in enumerate(numbers):
  numbers[i] = pow2(n)
print(numbers)
```

```
[1, 4, 9, 16]
```

```
def pow2_f(number):
  return number ** 2

pow2_l = lambda n : n**2
```

Zwei verschiedene Möglichkeiten eine Funktion zu erstellen!

Map nutzt intern einen Iterator und erzeugt somit nicht direkt eine Liste, sondern jeweils das nächste Element während der Iteration.

```
numbers = map(pow2, numbers)
print(numbers, list(numbers))
```

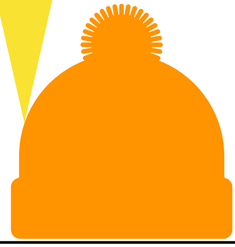
```
<map object at 0x10e0a2830> [1, 4, 9, 16]
```

```
numbers = [1, 2, 3, 4]
numbers = map(lambda n : n**2, numbers)
print(numbers, list(numbers))
```

```
<map object at 0x10e0a3280> [1, 4, 9, 16]
```

Sehr praktisch für z.B.

```
sort(key=lambda o: o.name)
```



Map, Filter & Reduce

```python
numbers = [1, 2, 3, 4]

m = map(lambda n: n ** 3 if n < 3 else n ** 2, numbers)
print(list(m))



f = filter(lambda n: n % 2 == 0, numbers)
print(list(f))



from functools import reduce

r = reduce(lambda c, n: c + n, numbers, 0)
print(r)
```

Map, Filter & Reduce

```python
numbers = [1, 2, 3, 4]

m = map(lambda n: n ** 3 if n < 3 else n ** 2, numbers)
print(list(m))
```

```
[1, 8, 9, 16]
```

```python
f = filter(lambda n: n % 2 == 0, numbers)
print(list(f))
```

```python
from functools import reduce

r = reduce(lambda c, n: c + n, numbers, 0)
print(r)
```

Map, Filter & Reduce

```
numbers = [1, 2, 3, 4]

m = map(lambda n: n ** 3 if n < 3 else n ** 2, numbers)
print(list(m))
```

```
[1, 8, 9, 16]
```

```
[n ** 3 if n < 3 else n ** 2 for n in numbers]
```

```
f = filter(lambda n: n % 2 == 0, numbers)
print(list(f))
```

```
from functools import reduce

r = reduce(lambda c, n: c + n, numbers, 0)
print(r)
```

Map, Filter & Reduce

```
numbers = [1, 2, 3, 4]

m = map(lambda n: n ** 3 if n < 3 else n ** 2, numbers)
print(list(m))
```

```
[1, 8, 9, 16]
```

```
[n ** 3 if n < 3 else n ** 2 for n in numbers]
```

```
f = filter(lambda n: n % 2 == 0, numbers)
print(list(f))
```

```
[2, 4]
```

```
from functools import reduce

r = reduce(lambda c, n: c + n, numbers, 0)
print(r)
```

Map, Filter & Reduce

```
numbers = [1, 2, 3, 4]

m = map(lambda n: n ** 3 if n < 3 else n ** 2, numbers)
print(list(m))
```

```
[1, 8, 9, 16]
```

```
[n ** 3 if n < 3 else n ** 2 for n in numbers]
```

```
f = filter(lambda n: n % 2 == 0, numbers)
print(list(f))
```

```
[2, 4]
```

```
[n for n in numbers if n % 2 == 0]
```

```
from functools import reduce

r = reduce(lambda c, n: c + n, numbers, 0)
print(r)
```

Map, Filter & Reduce

```
numbers = [1, 2, 3, 4]

m = map(lambda n: n ** 3 if n < 3 else n ** 2, numbers)
print(list(m))
```

```
[1, 8, 9, 16]
```

```
[n ** 3 if n < 3 else n ** 2 for n in numbers]
```

```
f = filter(lambda n: n % 2 == 0, numbers)
print(list(f))
```

```
[2, 4]
```

```
[n for n in numbers if n % 2 == 0]
```

```
from functools import reduce

r = reduce(lambda c, n: c + n, numbers, 0)
print(r)
```

```
10
```

Map, Filter & Reduce

```python
numbers = [1, 2, 3, 4]

m = map(lambda n: n ** 3 if n < 3 else n ** 2, numbers)
print(list(m))
```

```
[1, 8, 9, 16]
```

```python
[n ** 3 if n < 3 else n ** 2 for n in numbers]
```

```python
f = filter(lambda n: n % 2 == 0, numbers)
print(list(f))
```

```
[2, 4]
```

```python
[n for n in numbers if n % 2 == 0]
```

```python
from functools import reduce

r = reduce(lambda c, n: c + n, numbers, 0)
print(r)
```

```
10
```

```python
c = 0
for n in numbers:
  c = c + n
  # c = (lambda c, n: c + n)(c, n)
```

Map, Filter & Reduce

```
numbers = [1, 2, 3, 4]

m = map(lambda n: n ** 3 if n < 3 else n ** 2, numbers)
print(list(m))
```

```
[1, 8, 9, 16]
```

```
[n ** 3 if n < 3 else n ** 2 for n in numbers]
```

Alternative Schreibweisen mit den gleichen Ergebnis

```
f = filter(lambda n: n % 2 == 0, numbers)
print(list(f))
```

```
[2, 4]
```

```
[n for n in numbers if n % 2 == 0]
```

```
from functools import reduce

r = reduce(lambda c, n: c + n, numbers, 0)
print(r)
```

```
10
```

Teilweise auch `fold()` genannt.

```
c = 0
for n in numbers:
  c = c + n
  # c = (lambda c, n: c + n)(c, n)
```

Intern wir quasi das Folgenden ausgeführt.

Typannotationen

```python
def str_repeat(s:str, n:int, sep:str=None) -> str:
  s = s + sep if sep else s
  return s * n

print(str_repeat("Hallo", 5))
print(str_repeat("Hallo", 2, ", "))
print(str_repeat(5, 5))
```

Typannotationen

```python
def str_repeat(s:str, n:int, sep:str=None) -> str:
  s = s + sep if sep else s
  return s * n

print(str_repeat("Hallo", 5))
print(str_repeat("Hallo", 2, ", "))
print(str_repeat(5, 5))
```

Was passiert jeweils?

Typannotationen

```python
def str_repeat(s:str, n:int, sep:str=None) -> str:
  s = s + sep if sep else s
  return s * n

print(str_repeat("Hallo", 5))
print(str_repeat("Hallo", 2, ", "))
print(str_repeat(5, 5))
```

```
HalloHalloHalloHalloHallo
Hallo, Hallo,
25
```

Was passiert jeweils?

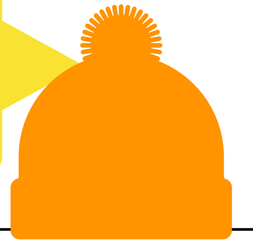
Typannotationen

```python
def str_repeat(s:str, n:int, sep:str=None) -> str:
  s = s + sep if sep else s
  return s * n

print(str_repeat("Hallo", 5))
print(str_repeat("Hallo", 2, ", "))
print(str_repeat(5, 5))
```

```
HalloHalloHalloHalloHallo
Hallo, Hallo,
25
```

Was passiert jeweils?

Englisch „Type Hints“ und mehr als Hinweise sind es auch nicht, Python prüft Parameter nicht und führt die Funktion munter aus!



Typannotationen

```python
def str_repeat(s:str, n:int, sep:str=None) -> str:
  s = s + sep if sep else s
  return s * n

print(str_repeat("Hallo", 5))
print(str_repeat("Hallo", 2, ", "))
print(str_repeat(5, 5))
```

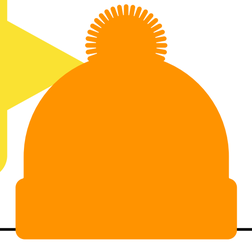
```
HalloHalloHalloHalloHallo
Hallo, Hallo,
25
```

Was passiert jeweils?

```python
from pydantic import validate_arguments

@validate_arguments
def str_repeat(s:str, n:int, sep:str=None):
```

Mit dem externen Paket pydantic ist tatsächlich eine Prüfung der Typen möglich.

Englisch „Type Hints“ und mehr als Hinweise sind es auch nicht, Python prüft Parameter nicht und führt die Funktion munter aus!



Typannotationen

```python
from typing import NoReturn, Union, Optional

def hello(name:Optional[str]="") -> NoReturn:
  print("Hello " + name)

def divide(x:Union[int, float], y:int|float) -> float:
  return x / y
```

Typannotationen

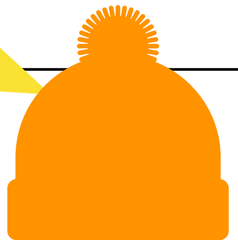
```python
from typing import NoReturn, Union, Optional

def hello(name:Optional[str]="") -> NoReturn:
  print("Hello " + name)

def divide(x:Union[int, float], y:int|float) -> float:
  return x / y
```

```python
from typing import List, Dict, Tuple

Tuple[int, str] # (1, "a")
Tuple[int, ...] # (1, 1), (1, 2, 3)

List[str] # ["a", "b"]
List[Union[str, int]] # [1, "b"], ["a", 2]

Dict[str, str] # {"a" : "A", "b" : "B"}
```

Typannotationen

```python
from typing import NoReturn, Union, Optional

def hello(name:Optional[str]="") -> NoReturn:
  print("Hello " + name)

def divide(x:Union[int, float], y:int|float) -> float:
  return x / y
```

„Type Hints“ sind sehr praktisch für die Dokumentation des Codes!



```python
from typing import List, Dict, Tuple

Tuple[int, str] # (1, "a")
Tuple[int, ...] # (1, 1), (1, 2, 3)

List[str] # ["a", "b"]
List[Union[str, int]] # [1, "b"], ["a", 2]

Dict[str, str] # {"a" : "A", "b" : "B"}
```

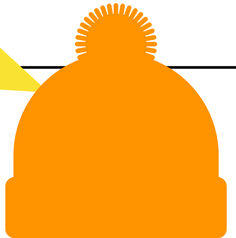
Typannotationen

```python
from typing import NoReturn, Union, Optional

def hello(name:Optional[str]="") -> NoReturn:
  print("Hello " + name)

def divide(x:Union[int, float], y:int|float) -> float:
  return x / y
```

Hier könnte man auch jeweils eine eigene Klasse als Typ nutzen.

```python
from typing import List, Dict, Tuple

Tuple[int, str] # (1, "a")
Tuple[int, ...] # (1, 1), (1, 2, 3)

List[str] # ["a", "b"]
List[Union[str, int]] # [1, "b"], ["a", 2]

Dict[str, str] # {"a" : "A", "b" : "B"}
```

„Type Hints“ sind sehr praktisch für die Dokumentation des Codes!



Funktionsparameter: ** und *

```python
def example(*args, **kwargs):
  print(type(args), args)
  print(type(kwargs), kwargs)


example("A", "B", "C", d="D", e="E")



p = "A", "B", "C"
kp = {"d" : "D", "e" : "E"}
example(*p, **kp)
```

Funktionsparameter: ** und *

```python
def example(*args, **kwargs):
  print(type(args), args)
  print(type(kwargs), kwargs)


example("A", "B", "C", d="D", e="E")
```

```
<class 'tuple'> ('A', 'B', 'C')
<class 'dict'> {'d': 'D', 'e': 'E'}
```

```python
p = "A", "B", "C"
kp = {"d" : "D", "e" : "E"}
example(*p, **kp)
```

Funktionsparameter: ** und *

```python
def example(*args, **kwargs):
  print(type(args), args)
  print(type(kwargs), kwargs)


example("A", "B", "C", d="D", e="E")
```

```
<class 'tuple'> ('A', 'B', 'C')
<class 'dict'> {'d': 'D', 'e': 'E'}
```

```python
p = "A", "B", "C"
kp = {"d" : "D", "e" : "E"}
example(*p, **kp)
```

```
<class 'tuple'> ('A', 'B', 'C')
<class 'dict'> {'d': 'D', 'e': 'E'}
```

Funktionsparameter: ** und *

```python
def example(*args, **kwargs):
  print(type(args), args)
  print(type(kwargs), kwargs)


example("A", "B", "C", d="D", e="E")
```

```
<class 'tuple'> ('A', 'B', 'C')
<class 'dict'> {'d': 'D', 'e': 'E'}
```

```python
p = "A", "B", "C"
kp = {"d" : "D", "e" : "E"}
example(*p, **kp)
```

```
<class 'tuple'> ('A', 'B', 'C')
<class 'dict'> {'d': 'D', 'e': 'E'}
```

Sehr praktisch um Parameter an eine Oberklasse weiterzureichen.

Variablen und Zeiger (Pointer)

- Python nutzt *Call-by-Reference*

Variablen und Zeiger (Pointer)

- Python nutzt *Call-by-Reference*

Call-by-Value: Funktionsaufrufe übertragen den Wert
Call-by-Reference: Funktionsaufrufe übertragen einen Zeiger auf ein Objekt/ den Wert

Variablen und Zeiger (Pointer)

- Python nutzt *Call-by-Reference*
  - Viele Typen sind jedoch unveränderlich (z.B. `str, int, float, tuple`)

Call-by-Value: Funktionsaufrufe übertragen den Wert
Call-by-Reference: Funktionsaufrufe übertragen einen Zeiger auf ein Objekt/ den Wert

Variablen und Zeiger (Pointer)

Call-by-Value: Funktionsaufrufe übertragen den Wert
Call-by-Reference: Funktionsaufrufe übertragen einen Zeiger auf ein Objekt/ den Wert

- Python nutzt *Call-by-Reference*
  - Viele Typen sind jedoch unveränderlich (z.B. `str, int, float, tuple`)
  - Bei „Änderungen“ eines unveränderlichen Typen wird somit immer ein neues Objekt erzeugt

Variablen und Zeiger (Pointer)

Call-by-Value: Funktionsaufrufe übertragen den Wert
Call-by-Reference: Funktionsaufrufe übertragen einen Zeiger auf ein Objekt/ den Wert

- Python nutzt *Call-by-Reference*
  - Viele Typen sind jedoch unveränderlich (z.B. `str, int, float, tuple`)
  - Bei „Änderungen“ eines unveränderlichen Typen wird somit immer ein neues Objekt erzeugt

```
s = "Hallo"
print(id(s))    4517249328
s += "Welt"
print(id(s))    4517249200
```

Variablen und Zeiger (Pointer)

Call-by-Value: Funktionsaufrufe übertragen den Wert
Call-by-Reference: Funktionsaufrufe übertragen einen Zeiger auf ein Objekt/ den Wert

- Python nutzt *Call-by-Reference*
  - Viele Typen sind jedoch unveränderlich (z.B. `str, int, float, tuple`)
  - Bei „Änderungen“ eines unveränderlichen Typen wird somit immer ein neues Objekt erzeugt

```
s = "Hallo"
print(id(s))    4517249328
s += "Welt"
print(id(s))    4517249200
```

```
l = ["Hallo"]
print(id(l))          4516668480
l.append("Welt")
print(id(l))          4516668480
```

Variablen und Zeiger

```python
def change(l):
  for i in range(len(l)//2):
    l[i], l[-(i+1)] = l[-(i+1)], l[i]

l = [1, 2, 3, 4, 5]
change(l)
print(l)
```

Variablen und Zeiger

```python
def change(l):
  for i in range(len(l)//2):
    l[i], l[-(i+1)] = l[-(i+1)], l[i]

l = [1, 2, 3, 4, 5]
change(l)
print(l)
```

```
[5, 4, 3, 2, 1]
```

Variablen und Zeiger

```python
def change(l):
  for i in range(len(l)//2):
    l[i], l[-(i+1)] = l[-(i+1)], l[i]

l = [1, 2, 3, 4, 5]
change(l)
print(l)
```

```
[5, 4, 3, 2, 1]
```

```python
bools = [True] * 5
print(bools)
bools[2] = False
bools[3] = False
print(bools)
```

```python
lists = [[]] * 5
print(lists)
lists[2].append(2)
lists[3].append(3)
print(lists)
```

Variablen und Zeiger

```python
def change(l):
  for i in range(len(l)//2):
    l[i], l[-(i+1)] = l[-(i+1)], l[i]

l = [1, 2, 3, 4, 5]
change(l)
print(l)
```

```
[5, 4, 3, 2, 1]
```

```python
bools = [True] * 5
print(bools)
bools[2] = False
bools[3] = False
print(bools)
```

```
[True, True, True, True, True]
[True, True, False, False, True]
```

```python
lists = [[]] * 5
print(lists)
lists[2].append(2)
lists[3].append(3)
print(lists)
```

Variablen und Zeiger

```python
def change(l):
  for i in range(len(l)//2):
    l[i], l[-(i+1)] = l[-(i+1)], l[i]

l = [1, 2, 3, 4, 5]
change(l)
print(l)
```

```
[5, 4, 3, 2, 1]
```

```python
bools = [True] * 5
print(bools)
bools[2] = False
bools[3] = False
print(bools)
```

```
[True, True, True, True, True]
[True, True, False, False, True]
```

```python
lists = [[]] * 5
print(lists)
lists[2].append(2)
lists[3].append(3)
print(lists)
```

```
[[], [], [], [], []]
[[2, 3], [2, 3], [2, 3], [2, 3], [2, 3]]
```

Variablen und Zeiger

```python
def change(l):
  for i in range(len(l)//2):
    l[i], l[-(i+1)] = l[-(i+1)], l[i]

l = [1, 2, 3, 4, 5]
change(l)
print(l)
```

```
[5, 4, 3, 2, 1]
```

Warum ändern sich alle?!

```python
bools = [True] * 5
print(bools)
bools[2] = False
bools[3] = False
print(bools)
```

```
[True, True, True, True, True]
[True, True, False, False, True]
```

```python
lists = [[]] * 5
print(lists)
lists[2].append(2)
lists[3].append(3)
print(lists)
```

```
[[], [], [], [], []]
[[2, 3], [2, 3], [2, 3], [2, 3], [2, 3]]
```

Variablen und Zeiger

```
def change(l):
  for i in range(len(l)//2):
    l[i], l[-(i+1)] = l[-(i+1)], l[i]

l = [1, 2, 3, 4, 5]
change(l)
print(l)
```

```
[5, 4, 3, 2, 1]
```

Liste wird geändert, ohne zurückgegeben zu werden, denn ein Zeiger wurde übergeben!

Warum ändern sich alle?!

```
bools = [True] * 5
print(bools)
bools[2] = False
bools[3] = False
print(bools)
```

```
[True, True, True, True, True]
[True, True, False, False, True]
```

Tatsächlich kann man auch änderbare Typen in einem unveränderbaren Typen verändern:

```
a = ([], [])
a[0].append(2)
```

```
lists = [[]] * 5
print(lists)
lists[2].append(2)
lists[3].append(3)
print(lists)
```

```
[[], [], [], [], []]
[[2, 3], [2, 3], [2, 3], [2, 3], [2, 3]]
```

Dekoratoren

```
@my_decorator
def example(name):
  print("Hallo " + name)


example("Magnus")
example("Otto")
```

Dekoratoren

```
@my_decorator
def example(name):
  print("Hallo " + name)


example("Magnus")      Hallo Magnus
example("Otto")        Hallo Otto
```

Dekoratoren

```
def my_decorator(some_function):
  print("Funktion laden")

  def my_wrapper(*args, **kwargs):
    print("Vor Ausführung der Funktion")
    v = some_function(*args, **kwargs)
    print("Nach Ausführung der Funktion")
    return v

  return my_wrapper


@my_decorator
def example(name):
  print("Hallo " + name)


example("Magnus")      Hallo Magnus
example("Otto")        Hallo Otto
```

Dekoratoren

```
def my_decorator(some_function):
  print("Funktion laden")

  def my_wrapper(*args, **kwargs):
    print("Vor Ausführung der Funktion")
    v = some_function(*args, **kwargs)
    print("Nach Ausführung der Funktion")
    return v

  return my_wrapper


@my_decorator
def example(name):
  print("Hallo " + name)


example("Magnus")
example("Otto")
```

Dekoratoren

```python
def my_decorator(some_function):
  print("Funktion laden")

  def my_wrapper(*args, **kwargs):
    print("Vor Ausführung der Funktion")
    v = some_function(*args, **kwargs)
    print("Nach Ausführung der Funktion")
    return v

  return my_wrapper

@my_decorator
def example(name):
  print("Hallo " + name)


example("Magnus")
example("Otto")
```

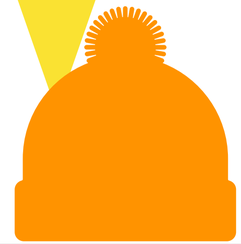
```
Funktion laden

Vor Ausführung der Funktion
Hallo Magnus
Nach Ausführung der Funktion

Vor Ausführung der Funktion
Hallo Otto
Nach Ausführung der Funktion
```

Dekoratoren

```python
def my_decorator(some_function):
  print("Funktion laden")

  def my_wrapper(*args, **kwargs):
    print("Vor Ausführung der Funktion")
    v = some_function(*args, **kwargs)
    print("Nach Ausführung der Funktion")
    return v

  return my_wrapper

@my_decorator
def example(name):
  print("Hallo " + name)


example("Magnus")
example("Otto")
```

```
Funktion laden

Vor Ausführung der Funktion
Hallo Magnus
Nach Ausführung der Funktion

Vor Ausführung der Funktion
Hallo Otto
Nach Ausführung der Funktion
```

Achtung: Beim Erstellen von Dekoratoren gibt es noch ein paar Fallstricke!



Dekoratoren

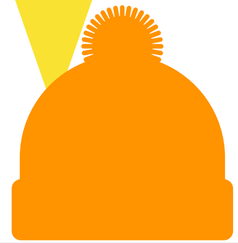
```python
def my_decorator(some_function):
  print("Funktion laden")

  def my_wrapper(*args, **kwargs):
    print("Vor Ausführung der Funktion")
    v = some_function(*args, **kwargs)
    print("Nach Ausführung der Funktion")
    return v

  return my_wrapper

@my_decorator
def example(name):
  print("Hallo " + name)


example("Magnus")
example("Otto")
```

Drei Einstiegspunkte, beim Laden der Funktion durch den Interpreter sowie nach und vor jeder Ausführung.

Die Dekoratoren können auch selbst Parameter haben.

```
Funktion laden

Vor Ausführung der Funktion
Hallo Magnus
Nach Ausführung der Funktion

Vor Ausführung der Funktion
Hallo Otto
Nach Ausführung der Funktion
```

Achtung: Beim Erstellen von Dekoratoren gibt es noch ein paar Fallstricke!



Anwendung Dekoratoren: ABC

- Gibt es abstrakte Klassen in Python?

Anwendung Dekoratoren: ABC

- Gibt es abstrakte Klassen in Python?
- Eigentlich nicht
  - Aber man kann abstrakte Klassen mit Dekoratoren erstellen
  - Bzw. das Paket abc nutzen

Anwendung Dekoratoren: ABC

- Gibt es abstrakte Klassen in Python?
- Eigentlich nicht
  - Aber man kann abstrakte Klassen mit Dekoratoren erstellen
  - Bzw. das Paket abc nutzen

```python
from abc import ABC, abstractmethod

class AbstractExample(ABC):

  def __init__(self, a):
    self.a = a

  @abstractmethod
  def abstract(self, a):
    pass

ae = AbstractExample()
```

Anwendung Dekoratoren: ABC

- Gibt es abstrakte Klassen in Python?
- Eigentlich nicht
  - Aber man kann abstrakte Klassen mit Dekoratoren erstellen
  - Bzw. das Paket abc nutzen

```
from abc import ABC, abstractmethod

class AbstractExample(ABC):

  def __init__(self, a):
    self.a = a

  @abstractmethod
  def abstract(self, a):
    pass

ae = AbstractExample()
```

```
Traceback (most recent call last):
  File "name.py", line 13, in <module>
    ae = AbstractExample()
  TypeError: Can't instantiate abstract
  class AbstractExample with abstract
  method abstract
```

Zusammenfassung

- Magic Methods
- Generatoren
- Erweiterte Schleifen
- Fehlerbehandlung
- Kontextmanager
- Lambda-Funktionen
- Typannotationen
- Variablen und Zeiger
- Dekoratoren

Zusammenfassung

- Magic Methods
- Generatoren
- Erweiterte Schleifen
- Fehlerbehandlung
- Kontextmanager
- Lambda-Funktionen
- Typannotationen
- Variablen und Zeiger
- Dekoratoren

Inhaltsübersicht

1. Programmiersprache Python
   a) *Einführung, Erste Schritte*
   b) Grundlagen
   c) Fortgeschritten
2. Auszeichnungssprachen
   a) LaTeX, Markdown
3. Benutzeroberflächen und Entwicklungsumgebungen
   a) Jupyter Notebooks lokal und in der Cloud (Google Colab)
4. Versionsverwaltung
   a) Git, GitHub
5. Wissenschaftliches Rechnen
   a) NumPy, SciPy
6. Datenverarbeitung und -visualisierung
   a) Pandas, matplotlib, NLTK
7. Machine Learning (scikit-learn)
   a) Grundlegende Ansätze (Datensätze, Auswertung)
   b) Einfache Verfahren (Clustering, …)
8. DeepLearning
   a) TensorFlow, PyTorch, HuggingFace Transformers