Metaprogramming

Proseminar Python

Finn Eilmann

Proseminar Python
Arbeitsbereich Wissenschaftliches Rechnen
Fachbereich Informatik
Fakultät für Mathematik, Informatik und Naturwissenschaften
Universität Hamburg

12.07.2022

Gliederung

1. Einleitung
   - Einleitung
2. Decorators
   - Die Rolle von Funktionen in Python
   - Decorators
3. Metaclasses
   - Beginn Metaclasses
   - Verwendung Metaklassen
   - Metaklasse type
   - type()-Konstruktor
   - Metaklasse erstellen & verwenden
4. Zusammenfassung
5. Literatur

Fragen zu Beginn

- Was versteckt sich hinter dem Begriff „Metaprogramming“?
- Was sind „Decorators“?
- Wie nutzt man Decorators?
- Was sind „Metaklassen“?
- Wofür können wir Metaklassen verwenden?
- Was macht die Klasse *type* und wie ist ihr Konstruktor aufgebaut?
- Wie können wir eigene Metaklassen erstellen?

Was ist Metaprogramming?

- ***meta-*** ist griechisch
  bezeichnet in Hierarchie **darüberliegende Ebene**
- ***programming*** ist englisch für *programmieren*
- Wikipedia-Definition: „*Metaprogrammierung ist in der Informatik die* ***Erstellung von Computerprogrammen*** *("Metaprogramme"),* ***die Computerprogramme erzeugen.***“

[1][2]

Funktionen in Python sind **First-Class-Objekte**

- als **First-Class-Objekte** können Funktionen in Python:
  - in **Programmvariablen** gespeichert werden
    - auch in Datenstrukturen
  - als **Parameter** an Funktionen übergeben werden
  - als **Rückgabewert** von Funktionen dienen
  - **zur Laufzeit** eines Programms **erstellt** werden
  - haben eine **eigene Identität**
- Man kann außerdem eine Funktion **in einer anderen** definieren.

[3]

Beispiel: Funktion als Argument weitergeben

```
1  def rufen(text):
2      return text.upper()
3
4  def fluestern(text):
5      return text.lower()
6
7  def gruessen(func):
8      gruss = func("Guten Tag.")
9      print (gruss)
10
11 gruessen(rufen)
12 gruessen(fluestern)
13
14 -> GUTEN TAG.
15 -> guten tag.
```

Beispiel: Funktion von anderer zurückgeben

```
1  def erstelle_addierer(x):
2      def addierer(y):
3          return x + y
4      return addierer
5
6  plus_15 = erstelle_addierer(15)
7
8  print(plus_15(10))
9
10 -> 25
```

Decorators

- ein Decorator ist eine Wrapper-Funktion, die
  - eine **Funktion als** ihren **einzigen Parameter** entgegennimmt
  - und eine **Funktion zurückgibt**.

man schreibt:

```
1 @decoratorFunctionName
2 def functionName():
3     [...]
```

statt

```
1 decoratorFunctionName(functionName)
```

- auch möglich: **mehrere Decorators miteinander verketten**
  - einfach mehrere @'s über einer Funktion
  - Ausführung von oben nach unten

---

[4] [5]

Beispiel: Decorators

```
1  def ablauf(func):
2      def innen(x):
3          print("vor der Funktion")
4          func(x)
5          print("nach der Funktion")
6      return innen
7
8  @ablauf
9  def funktion(x):
10     print(x+5)
11
12 funktion(3)
13
14 -> vor der Funktion
15 -> 8
16 -> nach der Funktion
```

Beispiel: verkettete Decorators

```
1  def ablauf(func):
2      def innen(x):
3          print("vor der Funktion")
4          func(x)
5          print("nach der Funktion")
6      return innen
7
8  def plus_2(func):
9      def innen(x):
10         func(x+2)
11     return innen
12
13 @ablauf
14 @plus_2
15 def funktion(x):
16     print(x+5)                          -> vor der Funktion
17                                         -> 10
18 funktion(3)                             -> nach der Funktion
```

*args, **kwargs

- innere Funktion nimmt (*args, **kwargs) **als Argument/ Parameter** entgegen:

```
1 def rechne(*args, **kwargs):
2     pass
```

- bedeutet, dass entweder
  - Tuple von möglichen Argumenten (*args)
  - oder Dictionary von Keyword Argumenten (**kwargs)

  jeweils mit beliebiger Länge weitergegeben werden kann
- man erhält so einen **generellen Decorator**, der eine Funktion mit beliebiger Anzahl an Argumenten dekorieren kann.

[4] [6]

Beispiel: Decorators mit *args, **kwargs

```
1  def ablauf(func):
2      def innere(*args, **kwargs):
3          print("Die Summe ist: ", end="")
4          func(*args, **kwargs)
5      return innere
6
7  @ablauf
8  def addierer(*num):
9      sum = 0
10     for i in num:
11         sum += i
12     print(sum)
13
14 addierer(5, 7)
15 addierer(3, 5, 2)
16
17 -> Die Summe ist: 12
18 -> Die Summe ist: 10
```

Beginn Metaclasses

- fast alles ist eine Instanz und hat somit einen Typ
- mit der type()-Funktion kann man den Typ einer Instanz herausfinden:

```
1  class Schueler:
2      pass
3  schueler_obj = Schueler()
4
5  print("Typ von schueler_obj ist:", type(schueler_obj))
6
7  -> Typ von schueler_obj ist: <class '__main__.Schueler'>
```

[7] [8] [9]

Beginn Metaclasses

- alle Objekte sind Instanzen von Klassen
- **Klassen** selbst sind **Instanzen von Metaklassen**
- eine Klasse kann nur eine Metaklasse haben
- Metaklassen erlauben, spezielles Verhalten zu einer Klasse hinzuzufügen

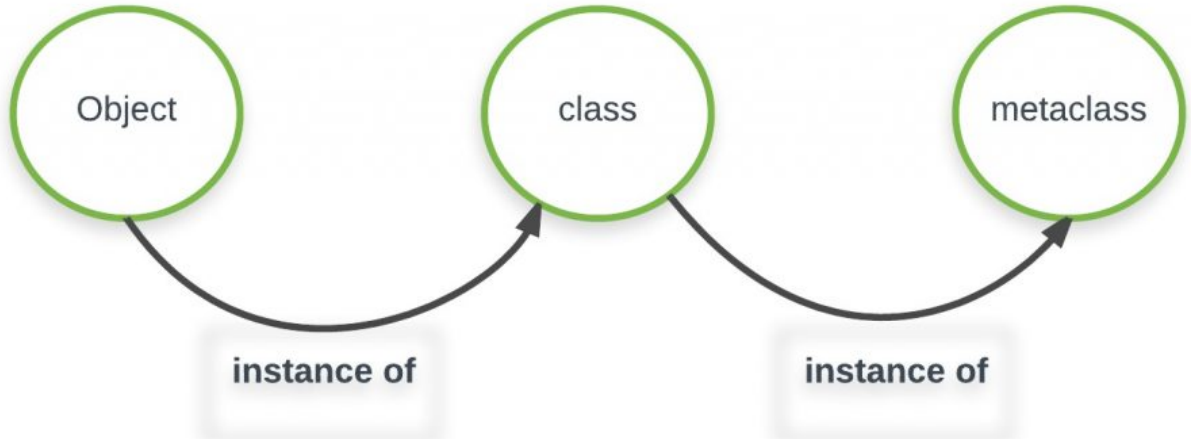

Abbildung: Hierarchie: Objekte, Klassen und Metaklassen in Python [8]

Wann verwendet man Metaklassen?

- Metaklassen haben Auswirkungen auf alle Unterklassen
  $\rightarrow$ in manchen Situationen ist das sehr nützlich
- wenn man Klasse direkt nach ihrer Erstellung verändern will
- prüfen, dass eine Klasse korrekt definiert wurde (sie validieren)
- API Entwicklung
- nützlich, um Frameworks und eigene libraries zu implementieren
  - Fehler beim Importieren von Modulen auslösen

[7] [8]

Die Metaklasse *type*

```
1  class Schueler:
2      pass
3
4  print("Typ von der Schueler Klasse ist:", type(Schueler))
5
6  -> Typ von der Schueler Klasse ist: <class 'type'>
```

- Die Klasse *type* ist die **default Metaklasse** die für die Erstellung von Klassen verantwortlich ist
- jede Klasse ist also eine Instanz vom Typ "*type*"
- wir erstellen eine Klasse mit dem Konstruktor von *type*

[7]

Der *type()*-Konstruktor

```
1 type(object)
2 type(name, bases, dict)
3 type(cls, name, bases, dict)
```

- die Funktion *type()* ist überladen:
  - aufgerufen mit nur **einem Parameter** (einem **Objekt**) gibt sie dessen **Typ** zurück (siehe frühere Beispiele)
  - aufgerufen mit (vier) drei Parametern **erzeugt** sie **eine Klasse**, die **Parameter**:
    - (*cls* immer als erstes Argument bei Klassenmethoden; bindet Methode an die Klasse)
    - Klassenname als String
    - Tuple von base classes
    - Klassen dictionary bestehend aus keys (Methodennamen) und values (Implementation der jeweiligen Methode)

---

[8] [10]

Klassen erstellen mit *type()*

```
1  def methode(self):
2      print("Testmethode")
3
4  class Basis:
5      def geerbt(self):
6          print("geerbte Methode")
7
8  Klasse = type("Testklasse",
9                (Basis,),
10               {"_x": "42", "meine_methode": methode})
```

Klassen erstellen mit *type()*

```
1  def methode(self): print("Testmethode")
2
3  class Basis:
4      def geerbt(self): print("geerbte Methode")
5
6  Klasse = type("Testklasse",
7                (Basis,),
8                {"_x": "42", "meine_methode": methode})
9
10 test_objekt = Klasse()
11 test_objekt.geerbt()
12 test_objekt.meine_methode()
13 print(test_objekt._x)
14
15 -> geerbte Methode
16 -> Testmethode
17 -> 42
```

Eigene Metaklasse erstellen

```
1  class Klasse:
2      pass
3
4  objekt = Klasse()
```

- mit Klasse() wird __call__() von der dazugehörigen Metaklasse (meistens type) aufgerufen
- diese ruft dann auf:
  - __new__()
  - __init__()
- wenn Klasse diese nicht selbst definiert, werden sie **von der jeweiligen Metaklasse geerbt**
- also meistens type.__new__() & type.__init__()

[8] [9]

Eigene Metaklasse erstellen

- Klasse erstellen, die von ***type* erbt** $\Rightarrow$ Metaklasse

```
1 class Metaklasse(type):
```

- darin eine **___new___-Methode erstellen**, die type.___new___() ersetzt, aber auch selbst aufruft:

```
1 def __new__(cls, name, bases, dict):
2     klasse = super().__new__(cls, name, bases, dict)
3     klasse.variable = 42
4     return klasse
```

- einer neuen Klasse unsere **Metaklasse zuordnen**:

```
1 class Unterklasse(metaclass=Metaklasse):
2     pass
```

[8] [9]

Eigene Metaklasse erstellen

insgesamt:

```
1  class Metaklasse(type):
2      def __new__(cls, name, bases, dict):
3          klasse = super().__new__(cls, name, bases, dict)
4          klasse.variable = 42
5          return klasse
6
7  class Unterklasse(metaclass=Metaklasse):
8      pass
9
10 print(Unterklasse.variable)
11
12 -> 42
```

Prüfregeln für neue Klassen

- Prüfregel zur ___new___-Methode hinzufügen:

```
1  class PersonMeta(type):
2      def __new__(cls, name, bases, dict):
3          if "gibAlter" in dict:
4              print("Pflichtmethode vorhanden")
5          else:
6              raise Exception("getAlter fehlt")
7          return super().__new__(cls, name, bases, dict)
8
9  class Person(metaclass=PersonMeta):
10     _alter = 42
11     def gibAlter(self):
12         return _alter
13
14 -> Pflichtmethode vorhanden
```

Klassen mit Metaklassen bearbeiten

```
1  class Metaklasse(type):
2      def __new__(cls, name, bases, dict):
3          new_dict = {}
4          for key, val in dict.items():
5              new_dict["vor_"+key] = val
6          return super().__new__(cls, name, bases, new_dict)
7
8  class Klasse(metaclass=Metaklasse):
9      def methode(self):
10         return "Methode wurde ausgefuehrt"
11
12 objekt = Klasse()
13 print(objekt.vor_methode())
14
15 -> Methode wurde ausgefuehrt
```

Zusammenfassung

- Decorators
  - sind Wrapper-Funktionen mit Funktion als einzigem Parameter und Funktion als Rückgabe
  - Schreibweise mit @name über einer anderen Funktion
  - man kann mehrere Decorators miteinander verketten
  - mit *args, **kwargs als Parameter erhält man generellen Decorator
- Metaklassen
  - Klassen sind Instanzen von sogenannten Metaklassen
  - *type* ist die default Metaklasse; für Erstellung von Klassen verantwortlich
  - mit dem type()-Konstruktor kann man u.a. eigene Klassen erstellen
  - eine neue, eigene Metaklasse muss von type erben, ___new___ implementieren & einer Klasse zugeordnet werden
  - mit Metaklassen kann man Prüfregeln festlegen & Klassen bearbeiten

Literatur I

[1] https://de.wiktionary.org/wiki/meta-.

[10] https://www.geeksforgeeks.org/python-type-function/.

[2] https://de.wikipedia.org/wiki/metaprogrammierung.

[3] https://de.wikipedia.org/wiki/first-class-objekt.

[4] https://www.geeksforgeeks.org/decorators-in-python/.

[5] https://www.geeksforgeeks.org/function-decorators-in-python-set-1-introduction/.

[6] https://www.programiz.com/python-programming/args-and-kwargs.

Literatur II

[7] https://medium.com/fintechexplained/advanced-python-metaprogramming-980da1be0c7d.

[8] https://www.geeksforgeeks.org/metaprogramming-metaclasses-python/.

[9] https://realpython.com/python-metaclasses/.