2 Magic Methods

Es gibt in Python zahlreiche spezielle Methoden, welche einer Klasse besondere Fähigkeiten geben. Die Namen dieser Methoden beginnen und enden immer mit zwei Unterstrichen. Wir haben bereits die Methode `__init__` kennengelernt, d.h. den Konstruktor.

Magisch dabei ist, dass die Funktionen nicht direkt aufgerufen wird sondern bei Bedarf implizit aufgerufen wird. Im Falle des Konstruktors wird `__init__` automatisch beim Erzeugen des Objektes aufgerufen.

Wir werden nun die wichtigsten Magischen Methoden und Attribute kennenlernen. Eine vollständige Liste gibt es in der Python Dokumentation unter:
https://docs.python.org/3/reference/datamodel.html#special-method-names

2.1 Typenumwandlung

2.1.1 __str__

Wenn die Klasse in eine Zeichenkette umgewandelt werden soll, wird die Methode `__str__` implementiert. Diese gibt eine Zeichenkette zurück. Ist es möglich eine Zeichenkette zu generieren so kann die instanz direkt mit der `print` Funktion ausgegeben werden.

Beispiel:

```
class Point:
   def __init__(self,x,y):
      self.x = x
      self.y = y

   def __str__(self):
      return "POINT (" + str(self.x) + "  " + str(self.y) + ")"


P = Point(10,20)
print(P)

S = str(P)
print(S)
```

2.1.2 __bytes__, __bool__, und __complex__

Gewisse Klassen können in eine Byte-Repräsentation, in ein bool und in eine komplexe Zahl umgewandelt werden. Diese Methoden werden jedoch eher selten benötigt.

2.1.3 __int__ und __float__

Klassen welche zum Beispiel eine Zahl repräsentieren können mit den speziellen Methodennamen `__int__` und `__float__` in eine ganze Zahl oder in eine Gleitkommazahl umgewandelt werden. Auch diese Methoden benötigen wir auch eher selten.

2.2 Operatoren überladen

2.2.1 Vergleichsoperatoren

Wir definieren eine einfachere Form der Klasse "Temperatur" welche nur die Temperatur in Celsius speichert. Diese sehr einfache Klasse sieht folgendermassen aus:

```
class Temperature:
   def __init__(self, c=0):
      self.celsius = c
```

Nun können wir mehrere Instanzen dieser Klasse anlegen, beispielsweise eine Zeitreihe verschiedener Temperaturen T0, T1, T2, gemessen am Tag X zu bestimmten Zeiten, an einem Tag wurde folgendes gemessen:

```
T0 = Temperature(15)
T1 = Temperature(18)
T2 = Temperature(17)
```

Die Fragen lauten nun:
Ist T1 grösser als T0 ?
Ist T2 grösser als T1 ?

Der entsprechende Python Code ist:

```
if T1.celsius > T0.celsius:
   print("T1 ist grösser als T0")
if T2.celsius > T1.celsius:
   print("T2 ist grösser als T1")
```

Dieser Code ist jedoch nicht intuitiv, da bei der Anwendung plötzlich das Attribut "celsius" ins Spiel kommt. Schöner wäre es doch, wenn folgendes geschrieben werden könnte:

```
if T1 > T0:
   print("T1 ist grösser als T0")
if T2 > T1:
   print("T2 ist grösser als T1")
```

Python bietet die Möglichkeit Operatoren zu defininieren. Dies geschieht über Magische Mathoden. Im oberen Besipiel muss der "grösser als" Operator definiert werden, was mit der Magischen Funktion `__gt__` (gt = greater than) realisiert werden kann.

Die Magische Funktion hat als erster Parameter eine Referenz auf sich selbst ("self") und als zweiter Parameter eine Referenz auf die zu Vergleichende Instanz ("other"). Als Rückgabewert wird ein Boolean erwartet, welcher True ist, falls "self" > "other" ist.

```
class Temperature:
   def __init__(self, c=0):
      self.celsius = c
   def __gt__(self, other):
      return self.celsius > other.celsius
```

Für andere Vergleichsoperationen ist das Prinzip identisch. In der Klasse sollten immer alle Vergleichsoperationen implementiert werden. In der folgenden Tabelle werden sämtliche Vergleichsoperatoren aufgelistet, welche in einer Python Klasse als Magische Methode implementiert werden können.

Operator	Methode	Beschreibung
<	`__lt__(self, other)`	less than (kleiner als)
<=	`__le__(self, other)`	less or equal (kleiner oder gleich)
==	`__eq__(self, other)`	equal (gleich)
!=	`__ne__(self, other)`	not equal (ungleich)
>	`__gt__(self, other)`	greater than (grösser als)
>=	`__ge__(self, other)`	greater or equal (grösser oder gleich)

2.2.2 Binäre arithmetische Operatoren

Ein binärer arithmetischer Operator verarbeitet zwei Operanden. Ein Beispiel wäre die Addition:

```
a + b
```

In Python können diese auch mit Magischen Methoden implementiert werden. Dabei ist - ähnlich wie den Vergleichsoperatoren - der erste Operand "self" und der zweite Operand "other". Der Rückgabewert ist immer eine neue Instanz welche das Resultat der binären Operation enthält, also in unserem Beispiel die Summe a+b, oder allgemeiner:

Linker-Operand *Operator* Rechter-Operand

Als Beispiel erstellen wir eine Klasse Vector2 welche einen zweidimensionalen Vektor repräsentiert. Die Klasse soll die Attribute x und y besitzen:

```python
class Vector2:
   def __init__(self, x, y):
      self.x = x
      self.y = y
   def __add__(self, other):
      return Vector2(self.x + other.x, self.y + other.y)
```

Die Magische Methode `__add__` bildet die Summe der beiden Operanden. Mit dieser Klasse kann nun folgendes implementiert werden:

```python
v0 = Vector2(0,3)
v1 = Vector2(2,3)

v2 = v0 + v1

print(v2.x, v2.y)
```

In der nachfolgenden Tabelle sind sämtliche binären arithmetischen Operatoren und die dazugehörige Magischen Methoden aufgelistet.

Operator	Magische Methode
+	`__add__(self, other)`
-	`__sub__(self, other)`
*	`__mul__(self, other)`
/	`__truediv__(self, other)`
//	`__floordiv__(self, other)`
divmod	`__divmod__(self, other)`
**	`__pow__(self, other)`
%	`__mod__(self, other)`
>>	`__lshift__(self, other)`
<<	`__rshift__(self, other)`
&	`__and__(self, other)`
|	`__or__(self, other)`
^	`__xor__(self, other)`

Nun gibt es noch eine Besonderheit bei binären arithmetischen Operatoren. Wir müssen beachten, dass es auch Operationen mit unterschiedlichen Datentypen gibt, wie beispielsweise `2*"Hello"`. Existiert beim ersten Datentypen keine magische Funktion für den Operator, wird beim zweiten Datentypen gesucht. In folgender Liste werden diese aufgeführt.

Operator	Magische Methode
+	`__radd__(self, other)`
-	`__rsub__(self, other)`
*	`__rmul__(self, other)`
/	`__rtruediv__(self, other)`
//	`__rfloordiv__(self, other)`
divmod	`__rdivmod__(self, other)`
**	`__rpow__(self, other)`
%	`__rmod__(self, other)`
>>	`__rlshift__(self, other)`
<<	`__rrshift__(self, other)`
&	`__rand__(self, other)`
|	`__ror__(self, other)`
^	`__rxor__(self, other)`

2.2.3 Erweiterte Zuweisungen

Die erweiterten Zuweisungen (Beispiel: +=) können auch überladen werden. Standardmässig verwendet Python für solche zuweisungen den Operator selbst, so dass a+=5 intern wie a = a + 5 ausgeführt wird. Diese Vorgehensweise hat für gewisse Datentypen, wie zum Beuspiel Listen den Nachteil, dass jedesmal eine neue Liste erzeugt werden muss. Aus Gründen der Optimierung macht es mehr Sinn den += Operator zu benutzen und diesen gezielt so anzupassen, dass das die Effizienz gesteigert wird.
In der Nachfolgenden Tabelle sind alle Operatoren und dessen Magische Methoden aufgelistet.

Operator	Magische Methode
+=	`__iadd__(self, other)`
-=	`__isub__(self, other)`
*=	`__imul__(self, other)`
/=	`__itruediv__(self, other)`
//=	`__ifloordiv__(self, other)`
**=	`__ipow__(self, other)`
%=	`__imod__(self, other)`
>>=	`__ilshift__(self, other)`
<<=	`__ishift__(self, other)`
&=	`__iand__(self, other)`
|=	`__ior__(self, other)`
^=	`__ixor__(self, other)`

Die Methoden müssen alle die Instanz self zurückgeben.

2.2.4 Unäre Operatoren

Zu den unären Operatoren gehören vor allem die Vorzeichen + und -. Auch die eigebaute Funktion `abs` (Absolutwert) und der Operator ~ (Komplement) gehören dazu. Auch diese können in einer Python Klasse überladen werden, die Methoden sind in der folgenden Tabelle aufgelistet:

Operator	Magische Methode
+	`__pos__(self)`
-	`__neg__(self)`
abs	`__abs__(self)`
~	`__invert__(self)`

Die Methoden geben in der Regel eine neue Instanz mit dem veränderten Wert zurück.

2.2.5 Container Methoden

Auch Operatoren für Container können als Magische Methoden implementiert werden:

Magische Methode	Beschreibung
`__len__(self)`	Liefert die Anzahl Elemente in dem Container zurück (als Integer)
`__getitem__(self, key)`	Liefert das Resultat für den Operator `[]`.
`__setitem__(self, key, value)`	Verändert das Element über den Operatoren `[]`.
`__delitem__(self, key)`	Entfernt das Element des Containers welches dem Schlüssel key zugeordnet ist
`__iter__(self)`	Gibt einen Iterator über die Werte des Containers zurück.
`__contains__(self, item)`	Prüft, ob ein Element in dem Container enthalten ist.