Typen & Typdefinitionen in Python 2.2 – 2.5

Hintergrund diesesTextes ist der Text *Objekte, Typen, Typhierarchien, Instanzen, Klasse*. Der vorliegende Text greift die darin beschriebenen Konzepte auf und verdeutlicht sie über eine Vielzahl von konkreten Python-Anweisungen.
Es empfiehlt sich, dass Sie diese bei der Lektüre gleich in Python umsetzen und auch mit weiteren, eigenen Beispielen ausprobieren.

Teil 1: Typen bzw. Klassen und Objekte

Ausgangspunkt sind die folgenden Objekte, jeweils Objektinstanzen der Typen **`list`**, **`str`** und **`dict`**:

```
>>> Kette = 'Hallo Freunde'
>>> Liste = ['Hallo','Freunde']
>>> Dict = {'Hallo':'Freunde'}
```

Alles ist eine Instanz von **`object`** (ugs.: *alles ist ein Objekt*)

```
>>> isinstance(Kette,object)
True
>>> isinstance(Liste,object)
True
>>> isinstance(Dict,object)
True
```

```
>>> isinstance(list,object)
True
>>> isinstance(dict,object)
True
>>> isinstance(str,object)
True
```

Jede Objektinstanz ist einem Typ zugewiesen

```
>>> type(Kette)
<type 'str'>
>>> type(Liste)
<type 'list'>
>>> type(Dict)
<type 'dict'>
```

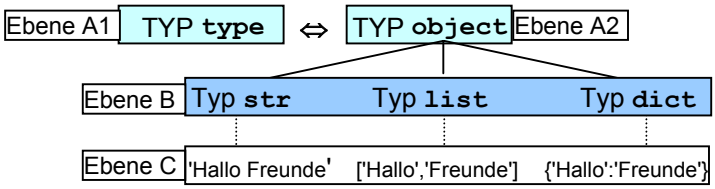
```
>>> type(str)
<type 'type'>
>>> type(list)
<type 'type'>
>>> type(dict)
<type 'type'>
```

Auch **`object`** ist einem Typ zugewiesen:

```
>>> type(object)
<type 'type'>
```

Auch **`type`** ist ein Objekt:

```
>>> isinstance(type,object)
True
```



Objekte auf B sind Subtypen von A2

```
>>> issubclass(str,object)
True
>>> issubclass(list,object)
True
>>> issubclass(dict,object)
True
```

Objekte auf B sind keine Subtypen von A1

```
>>> issubclass(str,type)
False
>>> issubclass(list,type)
False
>>> issubclass(dict,type)
False
```

Objekte auf B sind Instanzen von A2

```
>>> issubclass(str,object)
True
>>> issubclass(list,object)
True
```

```
>>> issubclass(dict,object)
True
```

Objekte auf C sind Instanzen von B

```
>>> isinstance(Kette,str)
True
>>> isinstance(Liste,list)
True
>>> isinstance(Dict,dict)
True
```

Objekte auf C sind keine Instanzen von A1

```
>>> isinstance(Kette,type)
False
>>> isinstance(Liste,type)
False
>>> isinstance(Dict,type)
False
```

Objekte auf C sind keine (Sub)Typen von irgendetwas, sondern reine Instanzen:

```
>>> issubclass(Kette,str)
[...] TypeError: issubclass() arg 1 must be a class
```

Typen sind durch Attribute charakterisiert. Zu diesen gehören auch die typgebundenen Funktionen, die *Methoden* genannt werden. Es gibt in Python auch nicht-typgebundene, vordefinierte Funktionen (die sogenannten *builtin-functions*), die auf Objektinstanzen verschiedener Typen angewendet werden können und immer zur Verfügung stehen:

```
>>> sorted(Kette)
[' ', 'F', 'H', 'a', 'd', 'e', 'e', 'l', 'l', 'n', 'o', 'r', 'u']
>>> sorted(Liste)
['!', 'Freunde', 'Hallo']
```

Methoden dagegen sind **typgebunden**:

```
>>> Kette.split()
['Hallo', 'Freunde']
>>> Liste.split()
[...] AttributeError: 'list' object has no attribute 'split'
>>> Dict.keys()
['Hallo']
>>> Liste.keys()
[...] AttributeError: 'list' object has no attribute 'keys'
>>> Liste.append("!")
>>> Liste
['Hallo', 'Freunde', '!']
>>> Kette.append("!")
[...] AttributeError: 'str' object has no attribute 'append'
```

Manche Methoden, die eine analoge Funktion haben, tragen den gleichen Namen, sind aber dennoch typgebunden (Stichwort ***Namensraum***):

```
>>> Kette.__add__(' wie geht es euch?')
'Hallo Freunde wie geht es euch?'
>>> Liste.__add__(['wie', 'geht', 'es', 'euch','?'])
['Hallo', 'Freunde', 'wie', 'geht', 'es', 'euch', '?']
```

Für manche Methoden wird die Eingabe 'versüßt' (***syntactic sugar***), beispielsweise durch die Verwendung von Operatoren:

```
>>> Kette + ' wie geht es euch?'
'Hallo Freunde wie geht es euch?'
>>> Liste + ['wie', 'geht', 'es', 'euch','?']
['Hallo', 'Freunde', 'wie', 'geht', 'es', 'euch', '?']
```

Bei der Anwendung einer Methode wird die Objektinstanz, auf die die Methode angewendet wird, dieser vorangestellt und durch einen Punkt vom Methodennamen getrennt.

```
<Objektinstanz>.Methodenname(Arg1, Arg2,...Arg_n)
>>> Kette.startswith('H')
True
>>> Liste.count('Hallo')
1
```

Genauso kann man allerdings auch folgenden Ausdruck verwenden:

```
<typ>.Methodenname(<Objektinstanz>,Arg2, Arg3,...Arg_n)
>>> str.startswith(Kette,'H')
True
>>> list.count(Liste,'Hallo')
1
```

Merke:

> Ursprünglich wurde in Python streng zwischen eingebauten Typen und vom Nutzer erstellten (oder aus Modulen importierten) Klassen unterschieden. Der Unterschied äußerte sich unter anderem darin, dass keine Unterklassen von Typen erstellt werden konnten. [...]
> Erst seit Python 2.2 verhalten sich die eingebauten Typen größtenteils wie selbst definierte Klassen. [...] Aufgrund dieser Annäherung von Typen und Klassen werden beide Begriffe im Zusammenhang mit Python inzwischen häufig synonym verwendet. Will man die weiterhin bestehenden Unterschiede zwischen Typen und Klassen betonen, bleibt es jedoch sinnvoll, beide Begriffe zu unterscheiden. Ungeachtet dessen spricht man in Python immer vom *Typ* eines Objekts, wenn man die Klasse meint, deren Instanz sie ist.
> (JOCHEN SCHULZ, well-adjusted.de/~jrschulz/papers/2005-wpk-modswt/python-typen_und_objekte.pdf)

Teil 2: Typen selber definieren

Man kann über die Anweisung **`class`** in Python neue Typen definieren. Dabei unterscheiden wir zwei verschiedene Verfahren:

1. Typen, die aus dem allgemeinsten Typ **`object`** abgeleitet sind
2. Typen, die aus Subtypen von **`object`** abgeleitet sind (z.B. aus **`str`**, **`list`**, **`dict`** usw.)

2.1: Typen, die aus dem allgemeinsten Typ `object` abgeleitet sind

Beispieldefinition: der Typ **`Dog`**

```
class Dog(object):
    legs = 4
    sound = 'barks'
    tail = True
    def __init__(self,name,owner):
        self.name = name
        self.owner = owner
    def favourite(self):
        return "%s likes meat" %self.name
```

Erläuterung:

`class Dog(object):`	Der Typ **`Dog`**. Das Argument zeigt, dass dieser Typ aus dem allgemeinsten Typ **`object`** abgeleitet ist.

`legs = 4`
`sound = 'barks'`
`tail = True`
	Hier sind drei Klassenattribute von **`Dog`** beschrieben.

`def __init__(self,name,owner):`	**`__init__`** (für *initialize*) wird aufgerufen, wenn eine Objektinstanz des neu definierten Typs konstruiert wird. Das Argument **`self`** (*selbst*) drückt aus, dass es sich um eine Objektinstanz des Typs **`Dog`** handelt. Die Argumente **`name`** und **`owner`** sind Instanzattribute, die bei der Initialisierung des Objektes übergeben werden. Objektinstanzen vom Typ **`Dog`** werden also immer mit zwei Argumenten aufgerufen.
`self.name = name` `self.owner = owner`	Hier werden den Attributen **`name`** und **`owner`** der Klasse **`Dog`** die bei der Initialisierung übergebenen Werte zugewiesen.
`def favourite(self):` `return "%s likes meat" %self.name`	Hier ist eine typgebundene Funktion, sprich eine Methode definiert. Diese macht Gebrauch vom Instanzattribut **`name`**.

Anwendung:

`>>> X = Dog('Fido','Tim')`	Hier wird eine Objektinstanz des Typs **`Dog`** initialisiert, die den Namen **`X`** trägt. Die Instanzattribute **`name`** und **`owner`** sind jeweils an die Zeichenketten **`'Fido'`** und **`'Tim'`** gebunden.

`>>> X.name`
`'Fido'`
`>>> X.owner`
`'Tim'`
	Hier werden die Instanzattribute von **`X`** abgefragt

`>>> X.legs`
`4`
`>>> X.sound`
`'barks'`
`>>> X.tail`
`True`
	Hier werden die Klassenattribute von **`X`** abgefragt

`>>> X.favourite()`
`'Fido likes meat'`
`>>> Dog.favourite(X)`
`'Fido likes meat'`
	Hier wird die Methode **`favourite()`** auf **`X`** angewendet. Dieser Ausdruck ist mit dem nachstehenden Ausdruck äquivalent.

Merke: Wir können zwischen Klassen- und Instanzattributen differenzieren. Letzere werden erst bei der Initiaisierung an spezifische Werte gebunden. Die dafür zu verwendende Methode `__init__` wird 'Initialisierungsmethode' genannt.

2.2: Typen, die aus dem aus Subtypen von `object` abgeleitet sind

Beispieldefinition: der Typ **`Boxer`**

```
class Boxer(Dog):
    tail = False
```

Erläuterung:

`class Boxer(Dog):`	Der Typ **`Boxer`**. Das Argument zeigt, dass dieser Typ ein Subtyp von **`Dog`** ist.
`  tail = False`	Hier ist ein Klassenattribut beschrieben. Dieses überschreibt das entsprechende Klassenattribut aus **`Dog`** für Instanzen vom Typ **`Boxer`**

Anwendung:

`>>> Y = Boxer('Bobo','Tim')`	Hier wird eine Objektinstanz des Typs **`Boxer`** initialisiert, die den Namen **`Y`** trägt. Bei der Initialisierung dieser Instanz wird die Initalisierungsmethode des Supertyps **`Dog`** verwendet.

`>>> Y.name`
`'Bobo'`
`>>> Y.owner`
`'Tim'`
	Hier werden die Instanzattribute von **`Y`** abgefragt. Diese wurden bei der Initialisierung an die Zeichenketten **`'Bobo'`** und **`'Tim'`** gebunden.

`>>> Y.legs`
`4`
`>>> Y.tail`
`False`
	`Y` erbt die Klassenattribute aus **`dog`** – allerdings nur dann, wenn diese nicht für Instanzen von **`Boxer`** umdefiniert wurden.

`>>> Y.favourite()` `'Bobo likes meat'`	Hier wird die von **`Dog`** vererbte Methode **`favourite()`** auf **`Y`** angewendet.

Beispieldefinition: der Typ **`Chow_Chow`**

```
class Chow_Chow(Dog):
    fur = 'soft and long'
    def favourite(self):
        return "%s likes cake" %self.name
```

Erläuterung:

`class Chow_Chow(Dog):`	Der Typ **`chow_chow`**. Das Argument zeigt, dass dieser Typ ein Subtyp von **`Dog`** ist.
`  fur = 'soft and long'`	Hier ist ein Klassenattribut beschrieben.
`def favourite(self):` `  return "%s likes cake" %self.name`	Hier ist eine typgebundene Funktion, sprich eine Methode definiert. Diese überschreibt die gleichnamige Methode aus der Typdefinition von **`Dog`** für Instanzen von **`Chow_Chow`**

Anwendung:

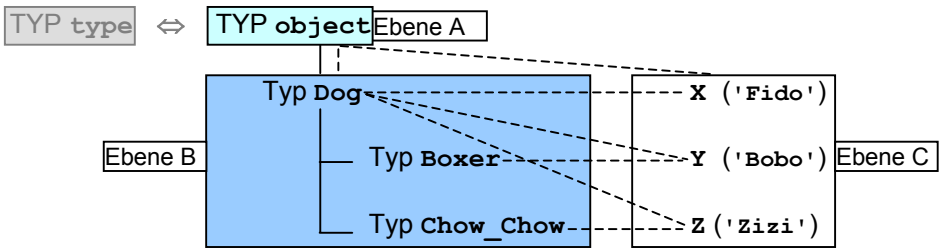
`>>> Z = Chow_Chow('Zizi','Paris')`	Hier wird eine Objektinstanz des Typs **`Chow_Chow`** initialisiert, die den Namen **`Z`** trägt. Bei der Initialisierung dieser Instanz wird die Initalisierungsmethode des Supertyps **`Dog`** verwendet.

`>>> Z.name`
`'Zizi'`
`>>> Y.owner`
`'Paris'`
	Hier werden die Instanzattribute von **`Y`** abgefragt. Diese wurden bei der Initialisierung an die Zeichenketten **`'Zizi'`** und **`'Paris'`** gebunden.

`>>> Z.fur`
`'soft and long'`
`>>> Z.tail`
`True`
`>>> Z.sound`
`'barks'`
	`Z` hat ein eigenes Klassenattribut und erbt die Klassenattribute aus **`Dog`**.

`>>> Z.legs = 3`
`>>> Z.legs`
`3`
`>>> Chow_Chow.legs`
`4`
	Hier wird das Klassenattribut **`legs`** für die konkrete Objektinstanz **`Z`** überschrieben. Das ändert das Attribut **nicht** für den Typ **`Chow_Chow`**.

Wie man in den Beispielen gesehen hat, spielt der Typ **`type`** in der konkreten Anwendung keine Rolle. Wir können die Hierarchie von Seite 1 mit Bezug auf die neu definierten Klassen deshalb wie folgt re-notieren:



Gestrichelte Linien: 'ist Instanz von'
Durchgezogene Linien: 'ist Subtyp von'

Diese Darstellung können wir mit Python belegen. Nachstehend einige Beispiele (bei denen nicht geprüft wird, was im Einzelnen eine Instanz von **`object`** ist – die Antwort ist schließlich 'alles'):

```
>>> isinstance(X,Dog)
True
>>> isinstance(Z,Chow_Chow)
True
>>> isinstance(Z,Dog)
True
>>> isinstance(Z,Boxer)
False
>>> issubclass(Chow_Chow,Dog)
True
>>> isinstance(Chow_Chow,Dog)
False
>>> issubclass(Boxer,object)
True
```