Fachbereich Wirtschaftsinformatik
Köllner Chaussee 11, 25337 Elmshorn

Wahlpflichtfach Moderne Softwaretechnologien
6. Semester, Jahrgang 2002
Prof. Dr. Ing. Johannes Brauer

Python
Typen und Objekte

Jochen Schulz
Matrikelnr.: 1882
mailto:jrschulz@well-adjusted.de

23. April 2006

Diese Arbeit verschafft einen Einblick in das Typsystem von Python, einer objektorientierten, dynamisch und streng typisierten Programmiersprache.

Gegenstand der Betrachtung sind die Features der sogenannten *New-Style Classes*, die die Flexibilität der Sprache erheblich erweitern und gleichzeitig das Sprachdesign konsistenter machen. Behandelt werden die Themen Operator Overloading, Metaklassen und Deskriptoren. Zur Veranschaulichung enthält jedes Thema nachvollziehbare und kommentierte Codebeispiele.

Inhaltsverzeichnis

1 Einführung **5**
1.1 Hintergrund 5
1.2 Philosophie 5
1.3 Syntax 6

2 Eingebaute Typen **7**
2.1 Zahlen 8
2.2 Sequenzen 8
2.3 Mappings 9
2.4 Weitere Typen 9

3 Typen und Objekte **10**
3.1 Definition Objekt 10
3.1.1 Identität 11
3.1.2 Attribute 11
3.1.3 Namen 11
3.1.4 Beziehungen 11
3.2 Typen und Klassen 12
3.3 Classic und New-Style Classes 13
3.4 Metaklassen 13

4 Attribute und Methoden **16**
4.1 User-provided vs. Python-provided Attributes 16
4.2 Einfacher Attributzugriff 16
4.2.1 Setzen von Attributen 16
4.2.2 Lesen von Attributen 18
4.2.3 Funktionen und Methoden 18
4.3 Deskriptoren 19
4.3.1 Data- und Non-Data-Descriptors 20
4.3.2 Built-Ins 21
4.3.3 Decoratorsyntax 22
4.4 Zusammenfassung Attributzugriff 22
4.4.1 Lesender Zugriff 23
4.4.2 Schreibender Zugriff 23

5 Bewertung **23**

Literatur **25**

Listings

1 Syntaxbeispiel . . . 7
2 Interaktive Pythonshell . . . 7
3 Methodenaufrufe für arithmetische Operationen . . . 8
4 Verwendung von Dictionaries . . . 10
5 Objektbeziehungen . . . 12
6 Eine eigene Metaklasse . . . 15
7 Benutzung von Metaklassen . . . 15
8 Klassen- und Instanzattribute . . . 17
9 Attributzugriff . . . 18
10 Funktionen und Methoden . . . 19
11 Das Deskriptorprotokoll . . . 20
12 Non-Data Descriptors . . . 21
13 Die Decoratorsyntax . . . 22

Tabellenverzeichnis

1 Zahlentypen . . . 8
2 Sequenztypen (Auswahl) . . . 9

Abbildungsverzeichnis

1 Metaklassen, Klassen und Instanzen . . . 14

1 Einführung

1.1 Hintergrund

Python wurde ursprünglich allein von Guido van Rossum am Centrum voor Wiskunde en Informatica in den Niederlanden entwickelt. Zuvor arbeitete Van Rossum mehrere Jahre an ABC, einer Programmiersprache, die hauptsächlich für Lehrzwecke entwickelt wurde. Noch heute sind Ähnlichkeiten zwischen Python und ABC erkennbar, etwa die syntaktische Signifikanz von Whitespace und die Möglichkeit der interaktiven Nutzung. Van Rossum hat jedoch auch ganz gezielt Fehler bzw. Beschränkungen von ABC vermieden.[1] So konnte Python im Gegensatz zu ABC von Anfang an mit Dateien umgehen und war aufgrund des Modulkonzepts schon immer sehr einfach erweiterbar.

Obwohl Python zu Beginn mit dem bescheidenen Ziel „to bridge the gap between the shell and C“[2] entwickelt wurde, zeigte sich mit der Zeit die Vielseitigkeit der Sprache und immer größere Programme entstanden in den unterschiedlichsten Anwendungsdomänen.[3]

Seit dem Jahr 2001 findet die Arbeit an Python unter dem Dach der *Python Software Foundation*[4] statt. Sie bildet den rechtlichen Rahmen, in dessen Namen Konferenzen veranstaltet sowie Spenden verwaltet werden und sie hält die Eigentumsrechte an Python. Gleichwohl ist Python weiterhin für jedermann verfügbar. Pythons Lizenz[5] erlaubt das freie Kopieren, Verändern und Verteilen von verändertem Code. Sie ist von der Open Source Initiative zertifiziert und GPL-kompatibel,[6] enthält allerdings keine Copyleft-Klausel, also einen Zwang, Veränderungen oder Ergänzungen ebenfalls samt Sourcecode auszuliefern. Das macht Python vor allem unter Anhängern freier Software, aber auch in kommerziellen Umgebungen ausgesprochen beliebt.

1.2 Philosophie

Python ist eine objektorientierte, interaktive, interpretierte Programmiersprache für Windows, Unix, Macintosh und eine Reihe weiterer Betriebssysteme. Es gilt das Smalltalk-Motto „Everything is an object.“ Das heißt, auch Funktionen, Klassen und Module sind Objekte. Die Definition dessen, was ein Objekt ist, ist in Python allerdings relativ locker: ein Objekt muss keine Attribute oder Methoden haben. Was alle Objekte eint, ist die Fähigkeit, einer Variablen zugewiesen, oder als Parameter einer Funktion übergeben zu werden.[7] Funktionen höherer Ordnung und Metaprogrammierung (auch Introspection oder Reflection genannt) sind möglich und in vielen Anwendungen anzutreffen.

Variablen haben in Python keinen Typ. Sie sind bloß *Namen* für Objekte. Sie brauchen deswegen nicht deklariert zu werden und ihnen können auch nacheinander Objekte

[1] (Venners 2003a, Python is Born)
[2] (Venners 2003b, Python's Original Design Goals)
[3] Siehe (PSF 2006d) für eine unvollständige Liste von „Success Stories“.
[4] Online unter `http://python.org/psf/`.
[5] PSF (2006a)
[6] (FSF 2006, GPL-Compatible Free Software Licenses)
[7] Nach (Pilgrim 2004, Kapitel 2.4.2.) – diese vage Definition soll vorerst genügen, das Thema wird in Abschnitt 3 ausführlich erörtert.

verschiedener Typen zugewiesen werden. Ein Objekt dagegen ist während seiner gesamten Lebensdauer an einen bestimmten Typ gebunden. Allerdings kann ein bereits instanziertes Objekt zur Laufzeit neue Methoden und Attribute erhalten oder alte verlieren.

Die Konsequenz daraus ist das sogenannte *Duck Typing*: „If it looks like a duck and quacks like a duck, it must be a duck.“[8] Wer also einem Objekt eine Nachricht schickt, braucht sich nicht um seinen Typ kümmern, denn dieser würde keine verlässlichen Informationen über die verstandenen Nachrichten liefern. Stattdessen muss der Programmierer im Zweifel sicherstellen, dass der Empfänger die Nachricht auch versteht. Solche Zweifel sind aber nicht die Regel. Der Aufwand, in diesen Fällen von Hand zu prüfen ist im Allgemeinen geringer, als den Typ aller Objekte ausdrücklich zu deklarieren. „It's easier to ask for forgiveness than to seek permission“ ist ein weiteres häufig gebrauchtes Idiom in Python. Das bedeutet, der Programmierer sendet einem Objekt einfach eine Nachricht und fängt die passende Exception für den Fall, dass das Objekt diese Nachricht nicht versteht. Selbst Syntaxfehler können auf diese Weise abgefangen werden. Wem dieser Stil nicht zusagt, der kann stattdessen auch sehr einfach die Namen aller Attribute eines gegebenen Objekts erfragen und vor dem Zugriff überprüfen, ob das Gewünschte darunter ist.

1.3 Syntax

Pythons Syntax ist denkbar einfach gehalten. Mit den Worten des Autors Guido van Rossum: „Python code looks like executable pseudo code.“[9] Ein Grund dafür ist die syntaktische Bedeutung von Whitespace, also Zeilenumbrüchen, Tabulatoren und Leerzeichen. Zeilenumbrüche markieren das Ende einer Anweisung, sofern sie nicht mit einem Backslash maskiert sind. Mit Tabulatoren oder Leerzeichen gleichmäßig eingerückte Zeilen kennzeichnen zusammenhängende Codeblöcke, wobei Leerzeichen vorzuziehen sind.[10] Groß- und Kleinschreibung wird unterschieden. Alle Schlüsselwörter[11] werden in Kleinbuchstaben geschrieben. Ein kleines Beispiel findet sich in Listing 1 auf der nächsten Seite (Schlüsselwörter hier und im Folgenden in **Fettdruck**).

Der Pythoninterpreter kann nicht nur gespeicherte Skripte ausführen, sondern auch interaktiv benutzt werden. Er stellt dem Benutzer eine Art Shell zur Verfügung, die das Ergebnis jedes abgeschlossenen Ausdrucks auf dem Bildschirm ausgibt. Wir greifen das vorige Beispiel in Listing 2 auf der nächsten Seite noch einmal auf.

Das Schlüsselwort **`def`** (Zeile 4) leitet eine Funktionsdefinition ein. Die drei Punkte am Beginn der nächsten Zeile druckt der Interpreter zur Erinnerung daran, dass die Funktionsdefinition noch nicht abgeschlossen ist. Er wird erst mit Zeile 13 beendet, weil dort keine um vier Leerschritte eingerückte Zeile steht. In diesem Fall wird das Ergebnis des Ausdrucks nicht ausgegeben. Das liegt daran, dass die Deklaration einer Funktion (eventuell im Gegensatz zu ihrem Aufruf) immer den Rückgabewert `None` hat. Der Interpreter druckt diesen Wert im interaktiven Modus aber nicht. In den folgenden

[8](PSF 2006e, Appendix D (Glossary))
[9](van Rossum 1998, Python's Strenghts)
[10](van Rossum 2001, Code lay-out)
[11](PSF 2006c, Kapitel 2.3.1)

```
1  odds = []                     # leere Listen erzeugen
2  evens = []                    #
3  for i in range(0, 10):
4      if i % 2 == 0:
5          evens.append(i)
6      else:
7          odds.append(i)
8  print odds                    # odds  = [1, 3, 5, 7, 9]
9  print evens                   # evens = [0, 2, 4, 6, 8]
```

Listing 1: Syntaxbeispiel

```
1  Python 2.4.3 (#2, Mar 30 2006, 21:52:26)
2  [GCC 4.0.3 (Debian 4.0.3-1)] on linux2
3  Type "help", "copyright", "credits" or "license" for more
      information.
4  >>> def split(max):
5  ...     odds = []
6  ...     evens = []
7  ...     for i in range(0, max):
8  ...         if i % 2 == 0:
9  ...             evens.append(i)
10 ...         else:
11 ...             odds.append(i)
12 ...     return (evens, odds)
13 ...
14 >>> split(10)
15 ([0, 2, 4, 6, 8], [1, 3, 5, 7, 9])
```

Listing 2: Interaktive Pythonshell

Beispielen werden sowohl reine Codelistings als auch interaktive Sitzungen dargestellt, je nachdem, wie es gerade nützlich erscheint.

Klassendefinitionen sehen ähnlich aus wie Definitionen von Funktionen. Statt des Schlüsselwortes **def** wird hier **class** verwendet. In Klammern hinter dem Namen stehen die Oberklassen. Mehr dazu in Abschnitt 3. Eine vollständige, formale Darstellung der Syntax ist in (PSF 2006c) zu finden.

2 Eingebaute Typen

Python stellt eine Reihe mächtiger eingebauter Typen zur Verfügung. Objekte dieser Typen verhalten sich generell wie alle anderen Objekte auch. Jedoch existieren für die meisten dieser Typen Literalschreibweisen und ebenso existiert oft eine spezielle Syntax für gängige Operationen auf diesen Objekten (zum Beispiel für die Iteration über eine Liste). Doch auch hier greift das Prinzip des Duck Typing, denn viele Syntaxelemente sind nur Verschleierungen von Methodenaufrufen auf einem Objekt. Daher ist es möglich,

Typ	Werte	Literal
`<type 'bool'>`	Wahrheitswerte	`True False`
`<type 'int'>`	Ganze Zahlen	`1 23 1024`
`<type 'long'>`	beliebig große, ganze Zahlen	`29846983214L`
`<type 'float'>`	Fließkommazahlen	`27.2`
`<type 'complex'>`	Komplexe Zahlen	`3.5+10.0j`

Tabelle 1: Zahlentypen

```
>>> a = 5
>>> b = 7.0
>>> print type(a), type(b)
<type 'int'> <type 'float'>
>>> a + b
12.0
>>> type(a + b)
<type 'float'>
>>> a.__add__(b)
12.0
```

Listing 3: Methodenaufrufe für arithmetische Operationen

das Verhalten von Instanzen eigener Klassen bei der Verwendung der Standardsyntax zu definieren (sogenanntes *Operator Overloading*).

2.1 Zahlen

Python kennt von Haus aus verschiedene Arten von Zahlen. Eine Übersicht befindet sich in Tabelle 1. Der Boolesche Typ ist in Python eine Unterklasse der Integers, daher taucht er in dieser Liste ebenfalls auf.

Eine Besonderheit der Zahlentypen ist, dass ihre Instanzen unveränderbar (*immutable*) sind. Alle arithmetischen Operationen erzeugen ein neues Objekt. Dessen Typ wird möglichst „eng" ausgewählt, das heißt eine Addition von zwei Integers liefert einen neuen Integer, eine Addition von einem Integer und einer Fließkommazahl liefert eine eine Fließkommazahl (auch dann, wenn sie ganzzahlig „aussieht"). Eine vollständige Erklärung der zugrunde liegenden Regeln und der unterstützten Operationen befindet sich in (PSF 2006b, Kapitel 2.3.4).

Python benutzt die gängige Infixnotation für die meisten arithmetischen Operationen. Tatsächlich ruft der Interpreter aber wie erwähnt eine bestimmte Methode des Objekts auf. Zur Verdeutlichung ein Beispiel in Listing 3.

2.2 Sequenzen

Python kennt einige vordefinierte Typen von Objektsequenzen. Die einfachsten sind *Listen* und *Tupel*. Beide können beliebig viele Objekte unterschiedlicher Typen aufnehmen und

Typ	Werte	Literal
<type 'tuple'>	Statische Liste	(23, 'abc', obj)
<type 'list'>	Veränderbare Liste	[23, 'abc', obj]
<type 'str'>	Zeichenkette (statisch)	"abc" 'abc' """abc"""

Tabelle 2: Sequenztypen (Auswahl)

adressieren diese per Index, angefangen bei Null.[12] Sie unterscheiden sich dadurch, dass Tupel immutable sind, während Listen veränderbar sind. Zeichenketten sind ebenfalls immutable.

Die Literaldarstellungen der Sequenztypen sind in Tabelle 2 dargestellt. Für Zeichenketten gibt es drei verschiedene Darstellungen, wobei zwischen den Schreibweisen mit einfachen und doppelten Anführungszeichen nicht unterschieden wird. Sie erlauben nur, das jeweils andere Anführungszeichen in der Zeichenkette selbst zu benutzen. Die Variante mit drei doppelten Anführungszeichen erlaubt zudem Zeilenumbrüche innerhalb der Zeichenkette.

Operationen auf den Sequenztypen sind detailliert in (PSF 2006b, Kapitel 2.3.6) beschrieben und können selbstverständlich von eigenen Typen (sofern semantisch sinnvoll) nachgeahmt werden.

2.3 Mappings

Zur Zeit gibt es nur einen einzigen eingebauten Mappingtyp in Python – das Dictionary. Dictionaries verhalten sich wie Listen, nur dass die enthaltenen Objekte (die *Werte*) nicht über einen Index sondern über ein eindeutiges Objekt (den sog. *Schlüssel*) angesprochen werden. Ein weiterer Unterschied ist, dass die Elemente in einem Dictionary ungeordnet vorliegen. Da Python ein Hashingverfahren verwendet, das eine (nahezu) konstante Zugriffszeit garantiert, müssen alle Schlüssel immutable sein. Die Werte können beliebige Objekte sein. Ein Beispiel für die Literalschreibweise und Verwendung von Dictionaries befindet sich in Listing 4 auf der nächsten Seite. Auch hier gilt wieder, dass der Aufruf einer Methode durch eine besondere Syntax verborgen wird. In diesem Fall ist es der Zugriff auf einen bestimmten Wert mittels eines Schlüssels (`[key]`), der den Aufruf der Methode `__getitem__(self, key)` des Dictionaries auslöst.

2.4 Weitere Typen

Wie schon angedeutet, gibt es natürlich auch Typen, die Module, Klassen, Methoden, Funktionen und Codeblöcke beschreiben. Der intensive Umgang mit Objekten dieser Typen ist recht selten. Man sollte aber im Hinterkopf behalten, dass auch solche Objekte

[12] Negative Indizes anzugeben ist ebenfalls möglich. Der Index `-1` zeigt auf das letzte Objekt der Sequenz, `-2` auf das vorletzte und so weiter. Der angegebene Index wird allerdings *nicht* modulo der Sequenzlänge benutzt, um Zugriffe auf nicht-existierende Elemente zu verhindern. Ein ungültiger Index erzeugt einen `IndexError`.

```
1 >>> d = { 'help'          : 'Hilfe',
2 ...       'attention'     : 'Achtung',
3 ...       'mobile phone'  : 'Handy' }
4 >>> type(d)
5 <type 'dict'>
6 >>> d['attention']
7 'Achtung'
8 >>> d.__getitem__('attention')
9 'Achtung'
```

Listing 4: Verwendung von Dictionaries

herumgereicht werden können. Besonders in eventbasierten Systemen (etwa im GUI-Bereich) werden häufig Funktionen als sogenannte *Callbacks* benutzt. Das bedeutet praktisch, dass Funktionen der entsprechenden APIs andere Funktionen als Parameter erwarten. Die übergebene Funktion wird dann nach dem Eintreten eines bestimmten Ereignisses aufgerufen. Die Funktion behält dabei ihren ursprünglichen Kontext, kann also auf Objekte zugreifen, die in der Umgebung (genauer: im *Namespace*) des Aufrufers gar nicht bekannt sind (sog. *Closure*).

Erwähnenswert sind außerdem Dateiobjekte. Sie werden meist durch Aufruf der eingebauten Funktion `open` mit dem Dateinamen als Parameter erzeugt. Was diesen Typ so bedeutend macht, sind die zahlreichen *file-like objects* in Python. Sie sind ein gutes Beispiel dafür, wie in Python lose Verträge über das Verhalten eines Objekts geschlossen werden. Anstatt genau vorzuschreiben, dass ein zu behandelndes Objekt vom Typ `file` sein muss, verlangen Funktionen oder Methoden ein file-like object als Parameter. Es soll also Methoden wie `read`, `write`, `flush` und `close` implementieren.[13] Dass nicht nur Klassen für einfache Textdateien solche Methoden sinnvoll bereitstellen können, kann man sich schnell denken. So gibt es file-like objects für komprimierte und XML-Dateien, Netzwerksockets, einfache Zeichenketten und einiges mehr. Der Benutzer dieser Objekte muss aber gar nicht wissen, womit er es genau zu tun hat, so lange er alle erwarteten Methoden vorfindet. Das kann auch nur eine Teilmenge aller Methoden des Typs `file` sein. Es gibt allerdings keine Möglichkeit anzuzeigen, dass Objekte eines bestimmten Typs ein bestimmtes Interface (man spricht in diesem Zusammenhang auch häufig von *Protokollen*) bereitstellt. Eventuell wird dieses Problem in Python 3.0 angegangen.[14]

3 Typen und Objekte

3.1 Definition Objekt

Ein Objekt in Python zeichnet sich durch vier Eigenschaften[15] aus:

- Identität

[13] (PSF 2006b, Kapitel 2.3.9)
[14] Siehe (van Rossum 2006) für ein diesbezügliches Gedankenspiel.
[15] Entnommen aus (Chaturvedi 2005b, Abschnitt 1)

- Attribute
- Namen und
- Beziehungen zu anderen Objekten.

3.1.1 Identität

Die Identität wird in der aktuellen Implementation durch die Speicheradresse des Objekts repräsentiert. Zu einem gegebenen Zeitpunkt ist die Identität aller existierenden Objekte unterscheidbar. Verschiedene Objekte, deren Lebenszeiträume sich nicht überschneiden, können dennoch die gleiche Identität im Sinne der Implementation besitzen.

3.1.2 Attribute

Die Attribute eines Objekts sind Name-Wert-Paare. Auf den Wert eines Attributs `attr` eines Objekts mit dem Namen `obj` wird mit dem Punktoperator zugegriffen: `obj.attr`. Der Wert eines Attributs kann beliebigen Typs sein. Einem Attributnamen können nacheinander Werte verschiedener Typen zugewiesen werden, ohne eine Warnung oder einen Fehler zu erzeugen. Existiert bei einer Zuweisung noch kein Attribut mit dem gegebenen Namen, wird es kommentarlos erzeugt. Der Abschnitt 4 auf Seite 16 widmet sich den Details.

3.1.3 Namen

Der Name ist die Zeichenkette, unter dem das Objekt im Programm angesprochen werden kann. In anderen Programmiersprachen spräche man vom Variablennamen. Objekte haben oft mehrere Namen, eventuell aber auch gar keinen. Zum Beispiel kann eine Liste benannt sein, die enthaltenen Objekte haben aber keinen eigenen direkt sichtbaren Namen und existieren nur als deren Inhalt.

3.1.4 Beziehungen

Python kennt zwei Arten der Beziehung von Objekten untereinander: Vererbung und Instanzierung. Beide haben die in objektorientierten Sprachen übliche Semantik. In Listing 5 auf der nächsten Seite sehen wir ein Beispiel. Zu Beginn stehen zwei Klassendefinitionen: `Bird` und deren Unterklasse `Duck`. In der Methode `__init__` werden zwei Instanzattribute definiert. Diese Methode wird Python bei jeder Instanzierung einer `Duck` aufrufen. Der Parameter mit dem Namen `self` ist immer die neue Instanz selbst. Beide Klassen implementieren die Methode `talk`. Das Objekt `donald` ist eine Instanz der Klasse `Duck`. Da `Duck` von `Bird` erbt, ist `donald` gleichzeitig eine Instanz von `Bird`. Die Attribute von `Duck` überdecken diejenigen von `Bird`. Attribute, die in `Duck` nicht vorhanden sind, werden von `Bird` geerbt.

```
>>> class Bird(object):
...     def __init__(self):
...             self.num_wings = 2
...             self.num_legs = 2
...
...     def talk(self):
...             return "Chilp!"
...
>>> class Duck(Bird):
...     def talk(self):
...             return "Quack!"
...
>>> issubclass(Duck, Bird)
True
>>> donald = Duck()
>>> isinstance(donald, Duck)
True
>>> isinstance(donald, Bird)
True
>>> donald.talk()
'Quack!'
>>> donald.num_wings
2
```

Listing 5: Objektbeziehungen

Zur Benutzung der eingebauten Funktionen[16] `issubclass` und `isinstance` ist nebenbei noch zu bemerken, dass hier offenbar wird, dass Klassen *first class objects* sind und deswegen als Parameter einer Funktion übergeben werden dürfen. Was Klassen von anderen Objekten unterscheidet, ist Gegenstand des Abschnitts 3.4.

3.2 Typen und Klassen

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. Als Workaround existieren noch heute Wrapperklassen wie `UserDict`, die ein normales `dict` – also einen Typ – kapseln, aber echte Klassen sind, von denen sich auch ableiten läßt.

Erst seit Python 2.2 (aktuell: 2.4.3) verhalten sich die eingebauten Typen größtenteils wie selbst definierte Klassen.[17] Sie können aber weiterhin – im Gegensatz zu eigenen Klassen – nicht verändert werden. Operationen, die einem eingebauten Typ oder einer

[16] Für eine komplette Liste aller eingebauten Funktionen siehe (PSF 2006b, Abschnitt 2.1, Built-In Functions).

[17] Siehe (van Rossum 2002) für die vollständige Dokumentation der Vereinheitlichung von Typen und Klassen.

Instanz eines Typs Attribute zuweisen, erzeugen einen `TypeError`. Das ist heute aber vermeidbar, weil direkt von (fast allen) eingebauten Typen abgeleitet werden kann.

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.

3.3 Classic und New-Style Classes

Weiterhin wird seit Version 2.2 zwischen *Classic Classes* und *New-Style Classes* unterschieden. New-Style Classes sind all diejenigen Klassen, die von `object` ableiten. Dazu zählen ab dieser Pythonversion alle eingebauten Typen und natürlich all diejenigen benutzerdefinierten Klassen, die in ihrer Definition `object` bzw. eine ihrer Unterklassen als Oberklasse angegeben haben. Klassen ohne `object` in ihrer Vererbungshierarchie heißen Classic Classes. New-Style Classes unterstützen einige neue Features, darunter *Managed Attributes* und `super`.

Managed Attributes (auch *Properties* genannt) sind Objektattribute, die vom Benutzer des Objekts wie ein normales Attribut angesprochen werden, innerhalb des Objekts aber beliebige Methodenaufrufe auslösen. Sie ersetzen also ungefähr die in anderen Sprachen üblichen Attribute mit Getter- und Settermethoden. Abschnitt 4.3.2 auf Seite 21 erläutert die genaue Funktionsweise von Properties.

Die Funktion `super`[18] erleichtert den Aufruf von Methoden einer Oberklasse. Sie soll insbesondere bei kreisförmigen Vererbungshierarchien[19] eine Hilfe darstellen. Weitere Informationen hierzu befinden sich im bereits erwähnten (van Rossum 2002).

Zukünftig ist damit zu rechnen, dass Classic Classes vollständig aus der Sprache verschwinden werden.[20] Der weitere Teil dieser Arbeit beschäftigt sich daher ausschließlich mit New-Style Classes.

3.4 Metaklassen

Klassen sind Objekte und wie alle Objekte haben sie einen Typ. Klassen sind also selbst Instanzen einer Klasse, genauer gesagt einer sogenannten *Metaklasse*. Metaklassen unterscheiden sich nur dadurch von normalen Klassen, dass sie die Klasse `type` in ihrer Vererbungshierarchie haben. Die normale Syntax zur Definition einer Klasse erzeugt immer eine Instanz von `type`.

Besonders bemerkenswert sind die Beziehungen zwischen `type` und `object`, der Wurzel der Vererbungshierarchie in Python. Da `object` ein Typ ist, ist es eine Instanz von `type`. Als New-Style Class muss `type` gleichzeitig von `object` erben. Zudem ist `type` als Metaklasse eine Instanz *von sich selbst*! Abbildung 1 auf der nächsten Seite veranschaulicht

[18]Genau genommen sind `property` und `super` Typen, deren Konstruktor aufgerufen wird.
[19]Solche Hierarchien können in Python auftreten, da Mehrfachvererbung erlaubt ist.
[20](Kuchling u. Cannon 2004, Core Language)

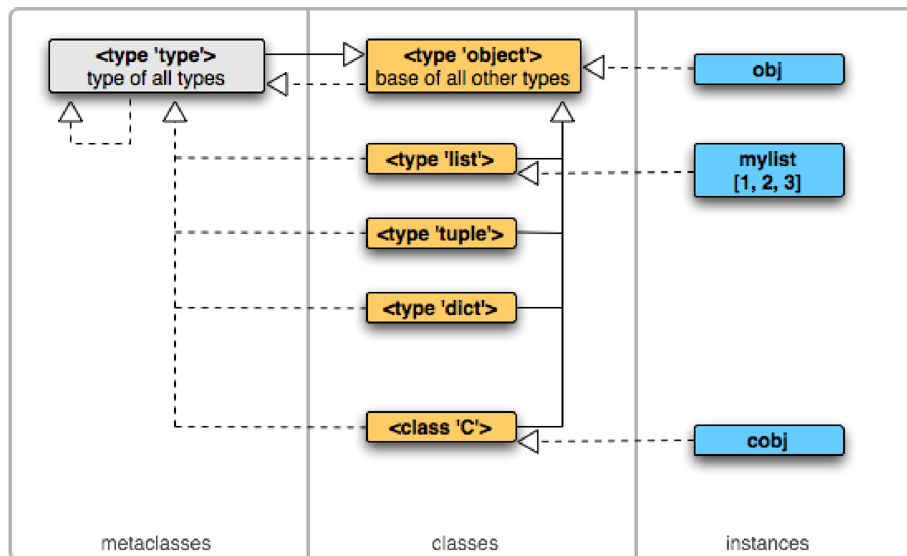

Abbildung 1: Metaklassen, Klassen und Instanzen
Quelle: Chaturvedi (2005b)

diese Beziehungen. Durchgezogene Linien bezeichnen Vererbungsbeziehungen, gestrichelte Linien bezeichnen Instanzierung.

Ein Beispiel

Metaklassen können mit wenig Aufwand selbst programmiert und benutzt werden. Betrachten wir als Beispiel Listing 6.[21] Hier wird eine Klasse autoprop definiert, die von type erbt und damit eine Metaklasse ist. Zweck dieser Klasse ist es, neue Klassen zu erzeugen, die automatisch Properties (s. Abschnitt 4.3.2 auf Seite 21) aus Methoden mit einem Namen der Form _get_<attribut> bzw. _set_<attribut> generieren.

In dieser Klasse wird nur eine einzige Methode definiert: __init__. Diese Methode wird bei der Instanzierung jedes Objekts aufgerufen und wird meist dazu benutzt, Objektattribute auf sinnvolle Initialwerte zu setzen. In diesem Fall wird die __init__-Methode genau dann ausgeführt, wenn der Interpreter auf eine Klassendefinition trifft, die autoprop als Metaklasse angibt. Die Parameter dieser Methode sind die gleichen, wie die Parameter der __init__-Methode der Oberklasse type:

- cls ist die gerade erzeugte Instanz der aktuell definierten Metaklasse,[22]
- name der Name der neu zu generierenden Klasse,

[21] Beispiel entnommen aus (van Rossum 2002, Metaclass examples)
[22] cls entspricht hier self in gewöhnlichen Klassen, siehe Abschnitt 4.2.3 auf Seite 18.

```
class autoprop(type):
    def __init__(cls, name, bases, dict):
        super(autoprop, cls).__init__(name, bases, dict)
        props = []
        for name in dict.keys():
            if name.startswith("_get_") or name.startswith("_set_"):
                props.append(name[5:])
        for name in props:
            fget = getattr(cls, "_get_%s" % name, None)
            fset = getattr(cls, "_set_%s" % name, None)
            setattr(cls, name, property(fget, fset))
            print "Created property %s" % name
```

Listing 6: Eine eigene Metaklasse

```
>>> class AbsX:
...     __metaclass__ = autoprop
...     def _get_x(self): return abs(self.__x)
...     def _set_x(self, x): self.__x = x
...
>>> a = AbsX()
Created property x
>>> a.x = -42
>>> print a.x
42
>>> print type(a)
<class '__main__.AbsX'>
>>> print type(type(a))
<class '__main__.autoprop'>
```

Listing 7: Benutzung von Metaklassen

- `bases` ist eine Liste der Oberklassen und
- `dict` ist ein Dictionary mit allen Attributen, die diese Klasse enthalten soll.

Zunächst wird `super` benutzt, um die `__init__`-Methode der Oberklasse (in diesem Fall also `type`) aufzurufen. Danach wird eine leere Liste mit dem Namen `props` erzeugt. Die erste Schleife (ab Zeile 5) sucht in der neuen Instanz von `autoprop` nach Attributnamen, die dem oben genannten Muster entsprechen. Diese Namen werden abzüglich ihres festen Präfix an die Liste `props` angehängt (also bspw. nur noch `<attribut>` statt `_get_<attribut>`).

Für jeden dieser Namen wird in der zweiten Schleife (ab Zeile 8) versucht, in der neuen Instanz der Metaklasse ein entsprechendes `_get_`- bzw. `_set_`-Attribut zu finden. Gelingt dies nicht, gibt der Aufruf von `getattr` `None` zurück. Schließlich wird in der neuen Klasse mittels `setattr` ein Attribut gesetzt, das von der Methode `property` erzeugt wird. Falls eine der Variablen `fget` und `fset` `None` ist, ist das entsprechende

Objektattribut nur schreib- bzw. lesbar.

In Listing 7 auf der vorherigen Seite sieht man eine mögliche Anwendung der Metaklasse `autoprop`.[23] Die Klasse `AbsX` soll einen Integerwert speichern können (genauer gesagt jedes Objekt, das die Methode `__abs__` implementiert). *Der Betrag* dieses Wertes soll als Objektattribut `x` erreichbar sein, während gleichzeitig der tatsächlich gesetzte Wert gespeichert wird. Eine elegante Möglichkeit dies zu erreichen, ist die Benutzung von Properties, wie sie von `autoprop` bereitgestellt werden. `AbsX` definiert also `_get_x` und `_set_x` als Getter und Setter für das zu erzeugende Attribut `x` und deklariert in Zeile 2 `autoprop` als ihre Metaklasse. Damit ist ein Zugriff auf das Attribut `x` möglich, obwohl weder ein solches Attribut gesetzt, noch die Funktion `property` explizit benutzt wurde. Wird nun diesem Attribut eine negative Zahl zugewiesen, bekommen wir beim Auslesen dieses Wertes dessen Betrag zurückgeliefert.

4 Attribute und Methoden

4.1 User-provided vs. Python-provided Attributes

Wie schon in Abschnitt 3.1 vorweggenommen, haben alle Objekte Attribute, die durch den Punktoperator referenziert werden können. Man unterscheidet zwischen benutzerdefinierten Attributen (*user-provided attributes*) und denjenigen, die der Pythoninterpreter automatisch für jedes Objekt bereitstellt (*python-provided attributes*).[24] Unter letzteren ist auch ein Attribut namens `__dict__`. Wie der Name suggeriert, ist es ein Dictionary.[25] Darin werden alle benutzerdefinierten Attribute des entsprechenden Objekts gespeichert. Der Schlüssel ist immer der Name des Attributs und der zugeordnete Wert ein beliebiges Objekt.

Der Rest dieses Abschnitts behandelt ausschließlich benutzerdefinierte Attribute.

4.2 Einfacher Attributzugriff

Der lesende und schreibende Zugriff auf Objektattribute mit Hilfe des Punktoperators ist nur eine spezielle Syntax für den Aufruf der Methoden `__getattribute_` und `__setattr__`,[26] die alle Objekte von `object` bzw. `type` erben. Betrachten wir zunächst das Verhalten dieser Methoden.

4.2.1 Setzen von Attributen

Alle Zuweisungen und Funktionsdefinitionen auf der obersten Ebene innerhalb einer Klassendefinition erzeugen neue Attribute *der Klasse*. Listing 8 auf der nächsten Seite zeigt ein Beispiel und demonstriert in Zeile 8, dass `__init__`, `f` und `classattr` tatsächlich im

[23] Beispiel angelehnt an (van Rossum 2002, Metaclass examples).
[24] Eine vollständige Liste dieser Attribute findet sich in (PSF 2006b, Kapitel 2.3.11).
[25] Bei Klassenobjekten ist `__dict__` vom Typ `dictproxy`, läßt sich aber leicht in ein `dict` umwandeln, wie im nächsten Beispiel zu sehen ist.
[26] Der Grund für die Asymmetrie in der Benennung der beiden Methoden wird in (PSF 2006c, Kapitel 3.3.2 ff.) beschrieben.

```
>>> class C(object):
...     def __init__(self, x, y):
...             self.x, self.y = x, y
...     def f(self):
...             pass
...     classattr = 42
...
>>> dict(C.__dict__)
{'classattr': 42, 'f': <function f at 0xb7d4756c>, '__init__':
   <function __init__ at 0xb7d47534> }
>>>
>>> obj = C("foo", "bar")
>>> obj.__dict__
{'y': 'bar', 'x': 'foo'}
>>>
>>> obj.z = "baz"
>>> obj.__dict__
{'y': 'bar', 'x': 'foo', 'z': 'baz'}
>>>
>>> del obj.x
>>> obj.__dict__
{'y': 'bar', 'z': 'baz'}
```

Listing 8: Klassen- und Instanzattribute

`__dict__` der Klasse `C` gespeichert werden.[27] Es sei erwähnt, dass Klassen die Methoden `__setattr__` und `__getattribute__` von `type` erben, nicht von `object`. Aus diesem Grund weicht ihr Verhalten von normalen Instanzen ab.

Anders erzeugt man Attribute von Instanzen, deren Klasse nicht von `type` ableitet. Betrachten wir Listing 8 erneut. Die beiden Parameter `"foo"` und `"bar"`, die dem Konstruktor in Zeile 11 übergeben wurden, werden in der Methode `__init__` an die Parameter mit den Namen `x` und `y` gebunden. Der erste Parameter `self` ist wie immer an die Instanz gebunden, die die Nachricht `__init__` empfangen hat. Die Mehrfachzuweisung in Zeile 3 erzeugt also zwei neue benutzerdefinierte Attribute `x` und `y`, die man – wie erwartet – in `obj.__dict__` findet (Zeile 12). Es gibt also keine andere Möglichkeit, als Objektattribute erst *nach* der Instanzierung des Objekts zu erzeugen. Möchte man sicherstellen, dass die Instanz jeder Klasse initial bestimmte Attribute hat, erzeugt man sie gewöhnlich in der Methode `__init__`.

Es dürfte nun nicht mehr überraschen, dass jedem Objekt, das nicht immutable ist, zur Laufzeit beliebige Attribute zugewiesen (siehe Zeile 15) und natürlich auch gelöscht werden können, siehe Zeile 19. Das Löschen von Attributen mit **`del`** ist übrigens auch durch eine Objektmethode realisiert und muss sich dementsprechend nicht genauso verhalten wie hier demonstriert.

[27] Aus Gründen der Übersichtlichkeit wurden einige Attribute weggelassen, die Klassen normalerweise ebenfalls haben.

```
22 >>> obj.classattr
23 42
24 >>> C.classattr is obj.classattr
25 True
26 >>> obj.classattr = 23
27 >>> obj.__dict__
28 {'y': 'bar', 'z': 'baz', 'classattr': 23}
29 >>> C.classattr is obj.classattr
30 False
```

Listing 9: Attributzugriff

4.2.2 Lesen von Attributen

Beim lesenden Zugriff auf ein Objektattribut hört die Suche nach einem entsprechend benannten Attribut nicht im `__dict__` des Objekts auf. Listing 9 führt das vorige Beispiel fort. Wie zu sehen ist, sind Attribute einer Klasse auch über deren Instanzen referenzierbar. Wird das angeforderte Attribut nicht im `__dict__` des Objekts gefunden, wird in seiner Klasse danach gesucht. Bleibt dies ebenfalls erfolglos, werden rekursiv deren Oberklassen durchsucht.[28]

Dieser Vorgang findet allerdings nur beim lesenden Zugriff statt. Beim Setzen eines Attributs wird immer direkt in das `__dict__` des angesprochenen Objekts geschrieben, wie in Zeile 26 zu sehen ist. Danach überdeckt das Attribut der Instanz das gleichnamige Attribut seiner Klasse. So ungewohnt dieser Effekt zuerst erscheinen mag, läßt er sich doch geschickt nutzen. Zum Beispiel können in einer Klasse Attribute mit Defaultwerten bestückt werden, die von allen ihren Instanzen geteilt werden, bis der Wert dieser Attribute in einzelnen Instanzen neu gesetzt wurde. Der Unterschied zum Setzen der Attribute in der `__init__`-Methode ist, dass das Klassenattribut nur *einmal* erzeugt wird. Alle Instanzen teilen sich in diesem Fall dasselbe Objekt.

4.2.3 Funktionen und Methoden

Das oben beschriebene Verhalten beim Zugriff auf Klassenattribute über eine Instanz der Klasse hilft auch beim Verständnis der Entstehung von *Methoden*. Denn bisher haben wir nur gesehen, wie *Funktionen* innerhalb einer Klasse definiert werden können (siehe Listing 8, Zeile 8 auf Seite 17).

Funktionen und Methoden sind in Python Instanzen unterschiedlicher Klassen. Eine spezielle Syntax gibt es nur zur Definition von Funktionen (Schlüsselwort **`def`**). Methoden werden vom Interpreter automatisch für alle Funktionen innerhalb einer Klasse erzeugt. Sie agieren als Wrapper für Funktionen. Alle Methoden haben drei in diesem Zusammenhang interessante Attribute:[29]

[28] Die genaue Reihenfolge beim Durchsuchen von Oberklassen bei Mehrfachvererbung ist sehr detailliert in (Simionato 2003) beschrieben.
[29] Siehe (PSF 2006c, Kapitel 3.2, User-defined Methods)

```
31 >>> obj.f.im_func  is dict(C.__dict__)['f']
32 True
33 >>> obj.f.im_class is C
34 True
35 >>> obj.f.im_self  is obj
36 True
37 >>> obj.f
38 <bound method C.f of <__main__.C object at 0xb7d6b8ac>>
39 >>> C.f.im_self is None
40 True
41 >>> C.f
42 <unbound method C.f>
```

Listing 10: Funktionen und Methoden

- `im_func` ist das Funktionsobjekt, das von der Methode gekapselt wird,
- `im_class` ist die Klasse, in der die gekapselte Funktion definiert wurde,
- `im_self` ist `None` oder die Instanz, auf der die Methode aufgerufen wird.

Listing 10 veranschaulicht diese Zusammenhänge. Zu beachten ist der Unterschied zwischen *bound* und *unbound methods*. Die Methoden `C.f` und `obj.f` sind verschiedene Objekte. Bei ungebundenen Methoden ist das Attribut `im_self` gleich `None`. Gebundene Methoden „kennen" dagegen ihren Aufrufer. Diese Unterscheidung ist wichtig, weil `im_self` genau das Attribut ist, das die Methode beim Aufruf ihrer gekapselten Funktion `im_func` als ersten Parameter (gewöhnlich mit dem Namen `self`) übergibt. Dabei findet sogar eine Typüberprüfung statt. Ist `im_self` keine Instanz von `im_class`, wird ein `TypeError` erzeugt. Ungebundene Methoden können trotzdem erfolgreich aufgerufen werden: der Aufruf `C.f(obj)` ist in unserem Beispiel äquivalent zu `obj.f()`.[30] Diese Notation wird hauptsächlich dann benutzt, wenn innerhalb einer Klassendefinition überladene Methoden einer Oberklasse aufgerufen werden sollen.

4.3 Deskriptoren

Es gibt viele Situationen, in denen man Einfluss auf lesenden und schreibenden Zugriff auf Objektattribute nehmen möchte. In Java würde man hierfür typischerweise Getter- und Setter-Methoden implementieren, die private Attribute manipulieren. In Ruby wird syntaktisch erst gar kein Unterschied zwischen Attributen und Methoden gemacht. Für den Aufrufer ist damit gar nicht klar, ob er das Ergebnis einer Berechnung oder ein (nicht im objektorientierten Sinn) „statisches" Objekt zurückgeliefert bekommt.

Der offensichtlichste Weg, in Python Einfluss auf Attributzugriffe zu nehmen, wäre die Methoden `__getattribute__` und `__setattr__` zu überladen, die den lesenden beziehungsweise schreibenden Attributzugriff implementieren. Das zieht aber einige Nachteile nach sich. Zum Einen werden die entsprechenden Methoden für *alle* Attribute eines

[30] Vgl. (PSF 2006e, Kapitel 9.3.4)

```
1 class Desc(object):
2
3     def __get__(self, obj, typ=None): pass
4     def __set__(self, obj, val): pass
5     def __delete__(self, obj): pass
```

Listing 11: Das Deskriptorprotokoll

Objekts aufgerufen. Das kostet Performance und macht den Code leicht unübersichtlich. Zum Anderen ist – gerade im Umgang mit `__getattribute__` – besondere Vorsicht von Nöten, um endlose Rekursion zu vermeiden. Mit Python 2.2 wurde eine neue Lösung für dieses Problem eingeführt: Deskriptoren.

Rein technisch betrachtet sind Deskriptoren alle Objekte, die das sogenannte Deskriptorprotokoll implementieren, das aus den Methoden `__get__`, `__set__` und `__delete__` besteht. Die letzten beiden Methoden sind optional.[31] Wird beim Attributzugriff ein entsprechend benannter Deskriptor im `__dict__` der *Klasse* des Objekts gefunden, wird nicht der Deskriptor selbst zurückgegeben, sondern dessen der Zugriffsart entsprechende Methode aufgerufen. Das `__dict__` der Instanz wird hierbei übergangen! Beim lesenden Zugriff wird `__get__` aufgerufen, beim Schreiben `__set__` und beim Löschen des Attributs `__delete__`. Die genaue Signatur der genannten Methoden ist Listing 11 zu sehen.[32] Der Parameter `obj` ist das jeweilige Objekt, dessen Attribut abgefragt wird und `typ` dessen Typ.

4.3.1 Data- und Non-Data-Descriptors

Unterschieden werden zwei Arten von Deskriptoren: *Data Descriptors* und *Non-Data Descriptors*. Data Descriptors werden all diejenigen Deskriptoren genannt, die sowohl die Methode `__get__` als auch `__set__` implementieren. Deskriptoren, die nur `__get__` implementieren, heißen Non-Data Descriptors. Die Methode `__delete__` spielt für diese Unterscheidung keine Rolle.

Das Besondere an Non-Data Descriptors ist, dass sie von regulären Objektattributen überdeckt werden können. Betrachten wir Listing 12 auf der nächsten Seite als Beispiel.[33] Die Klasse `Descr` implementiert ausschließlich die Methode `__get__`. Damit sind dessen Instanzen Non-Data Descriptors. Die Klasse `C` enthält ein Attribut von eben diesem Typ, wie in Zeile 8 zu sehen ist. Und wie erwartet, können wir in Zeile 11 erkennen, dass dieses Klassenattribut nicht nur über die Instanz `obj` erreichbar ist, sondern auch das Ergebnis der Methode `__get__` statt des Deskriptors zurückgegeben wird. Soweit also keine Überraschungen.

Setzen wir aber das Attribut `d` von `obj` neu wie in Zeile 13, überdeckt anschließend der neue Wert im `__dict__` von `obj` den Deskriptor in `C`. Dieser Vorgang ist einfach rückgängig zu machen, indem das Attribut `d` wieder aus `obj` gelöscht wird (Zeile 16).

[31] Vgl. (Chaturvedi 2005a, Creating Descriptors)
[32] Ebd.
[33] Beispiel angelehnt an (Chaturvedi 2005a, Beispiel 1.5).

```
>>> class Descr(object):
...     def __get__(self, obj, typ=None):
...         return 1
...
>>> class C(object):
...     d = Descr()
...
>>> C.__dict__['d']
<__main__.Descr object at 0xb7cf83ec>
>>> obj = C()
>>> obj.d
1
>>> obj.d = "a value"
>>> obj.d
'a value'
>>> del obj.d
>>> obj.d
1
```

Listing 12: Non-Data Descriptors

4.3.2 Built-Ins

Seit Deskriptoren verfügbar sind, macht Python intern exzessiven Gebrauch davon. Prominentestes Beispiel hierfür sind Funktionen. Funktionen sind Non-Data Descriptors. Zum Einen macht dies möglich, dass der Zugriff auf eine Funktion eine bound beziehungsweise eine unbound method zurückliefert, je nachdem, ob der Zugriff auf die Funktion über eine Instanz oder deren Klasse geschieht. Zum Anderen können Methoden auch für einzelne Instanzen überschrieben werden, denn das `__dict__` der Instanz hat Vorrang über Non-Data Descriptors in der Klasse.

Weitere eingebaute Deskriptoren sind `property`, `classmethod` und `staticmethod`. Properties wurden bereits in Abschnitt 3.3 angesprochen und in Listing 6 benutzt. Ihre Funktionsweise ist äußerst simpel: der Konstruktor nimmt bis zu drei Funktionen für das Lesen, Setzen und Löschen eines Attributs entgegen, die im entsprechenden Fall aufgerufen werden. Im Ergebnis verhalten sich Properties also wie Attribute mit Getter und Setter in anderen Programmiersprachen, nur dass deren Benutzung völlig transparent ist. Ein reguläres Attribut kann auch im Nachhinein durch ein Objekt vom Typ `property` ersetzt werden, ohne dass sich die Schnittstelle ändert.

Die Klassen `classmethod` und `staticmethod` erzeugen alternative Wrapper für Funktionen innerhalb von Klassen. Ihre Instanzen verhalten sich ähnlich wie Methoden, nur dass im Fall von `classmethod` der erste übergebene Parameter nicht die Instanz des aufgerufenen Objekts ist, sondern dessen Klasse, und im Fall von `staticmethod` überhaupt kein künstlicher erster Parameter eingefügt wird. Objekte von diesem Typ verhalten sich also wie normale Funktionen, leben aber im Namespace einer Klasse.

```
class C(object):
    def f(cls): pass
    f = classmethod(f)

    def g(): pass
    g = staticmethod(g)

class C(object):
    @classmethod
    def f(cls): pass

    @staticmethod
    def g(): pass
```

Listing 13: Die Decoratorsyntax

4.3.3 Decoratorsyntax

Seit Python 2.4 existiert eine spezielle Syntax für die Benutzung von Wrappern für Funktionen innerhalb einer Klasse.[34] Sie wird meist im Zusammenhang mit den eben erwähnten Deskriptoren benutzt, ist aber für beliebige Transformationen von Funktionen anwendbar. Listing 13 zeigt die ursprüngliche und die neue Schreibweise, die im Ergebnis äquivalent sind.

Die erste Klassendefinition benutzt die herkömmliche Schreibweise, die schon in Python 2.2 erlaubt war. Es werden zwei Funktionen `f` und `g` definiert, die anschließend mit dem Rückgabewert der Wrapper überschrieben werden. Problematisch an dieser Schreibweise ist, dass recht leicht zu übersehen ist, dass die bereits definierten Funktionen später wieder überschrieben werden. Insbesondere bei längeren Funktionen ist der Zusammenhang der beiden Ausdrücke nicht immer leicht zu erkennen. Die neue Decoratorsyntax verschiebt die Anweisung zur Transformation direkt vor den Rumpf der betroffenen Funktion und behebt so diesen Mangel. Direkt notwendig ist sie aber nicht.

Die neue Syntax ist dennoch sehr nützlich, denn sie betont eine äußerst mächtige Technik. Das dekorieren von Funktionen kann für viele verschiedene Zwecke eingesetzt werden, zum Beispiel für das Prüfen von Vor- und Nachbedingungen einer Funktion.

4.4 Zusammenfassung Attributzugriff

Um Übersicht in die lange Geschichte des Attributzugriffs in Verbindung mit Deskriptoren zu bringen, folgt hier noch einmal eine komprimierte, vollständige Auflistung der Schritte, die Python beim Zugriff auf ein Objektattribut ausführt:[35]

[34] Erstmals wurde diese Syntax in (Smith u. a. 2005) als das Ergebnis einer langen Diskussion beschrieben.
[35] Nach (Chaturvedi 2005a, Attribute Search Summary).

4.4.1 Lesender Zugriff

1. Wenn das gesuchte Attribut python-provided ist, gib es zurück.
2. Durchsuche die Klasse des Objekts und deren Oberklassen nach einem Data Descriptor. Wenn einer gefunden wird, rufe seine `__get__`-Methode auf und gib dessen Ergebnis zurück.
3. Suche im Objekt nach dem Attribut und gib es zurück. Wenn das Objekt eine Instanz von `type` ist, werden auch seine Oberklassen durchsucht. Ist dies der Fall, und das gefundene Attribut ist ein Deskriptor, wird das Ergebnis seiner `__get__`-Methode zurückgegeben.
4. Suche in der Klasse des Objekts und deren Oberklassen nach dem Attribut und gib es zurück. Ist es ein Deskriptor, gib das Ergebnis seiner `__get__`-Methode zurück.
5. Wurde das Attribut nicht gefunden, erzeuge einen `AttributeError`.

4.4.2 Schreibender Zugriff

1. Durchsuche die Klasse des Objekts und deren Oberklassen nach einem Data Descriptor. Wird einer gefunden, rufe seine `__set__`-Methode mit den entsprechenden Parametern auf.
2. Wurde kein Data Descriptor gefunden, erzeuge ein neues Attribut im Objekt.

Das Setzen von Attributen, die vom Interpreter bereitgestellt werden, verhält sich je nach Attribut unterschiedlich. Einige sind manipulierbar, andere nicht. Das Löschen von Attributen verhält sich ähnlich, wie das Schreiben.

Es muss erwähnt werden, dass sich die komplexe Prozedur des lesenden Attributzugriffs negativ auf die Performance von Pythonprogrammen auswirken kann. Aus diesem Grund und zugunsten besserer Lesbarkeit findet man in Pythoncode relativ selten tief geschachtelte Objekt- und Vererbungshierarchien. Gelegentlich wird auch ein sehr häufig gebrauchtes Attribut an einen lokalen Namen gebunden und weiterverwendet, anstatt bei jedem Zugriff die komplette Zuordnungskette zu durchwandern. Aufgrund der Referenzsemantik ist das problemlos möglich.

5 Bewertung

Die dargestellten Sprachfeatures machen Python zu einer äußerst flexiblen Programmiersprache, die das objektorienterte Modell konsequent umsetzt. Selbst Weiterentwicklungen wie aspektorienterte Programmierung können teilweise mit Hilfe von Metaklassen umgesetzt werden.

Python hat außerdem in seiner langen Entwicklungszeit einen hohen Reifegrad erworben. Neue Features werden behutsam und mit Bedacht auf Abwärtskompatibilität in die Sprache eingeführt. Gleichzeitig hat eine sehr aktive Entwicklergemeinde – auch außerhalb

des Kernteams – Python zu jeder Zeit um aktuell benötigte Funktionalitäten erweitert, so dass die Standarddistribution mit ihrer Vielzahl an Modulen heute kaum noch Wünsche offen läßt und keinen Vergleich mit kommerziellen Bibliotheken zu scheuen braucht. Eigene Entwicklungen sind weiterhin kein Problem, da alle benötigten Schnittstellen und Vorgänge offen dokumentiert sind.

Neben dem fehlenden Marketing, das Pythons Sichtbarkeit unter Konkurrenten wie Java und C# einschränkt, liegt eine große Schwäche aber auch gerade in der dynamischen Typisierung. Besonders im Unternehmensumfeld werden sogenannte *Skriptsprachen*[36] häufig belächelt und als untauglich für ernsthafte Projekte betrachtet. Eventuell wird diese Sichtweise in Zukunft weniger Anhänger finden, wenn Unittesting (wofür es in Python Unterstützung gibt) weitere Verbreitung findet. Zumindest nach Erfahrung vieler Pythonentwickler werden gerade diejenigen Fehler, die der Compiler einer statisch typisierten Sprache findet, auch bei Unittests gefunden. Und deren Nützlichkeit wird auch für statisch typisierte Sprachen kaum angezweifelt. Insofern stellt sich die Frage, ob die Deklaration aller Typen zur Zufriedenstellung des Compilers in Kombination mit Unittesting nicht doppelte Arbeit ist, die gleichzeitig die Flexibilität und Ausdrucksfähigkeit des Programmierers einschränkt.

[36] Der Begriff der Skriptsprache selbst ist kaum einheitlich definiert und viele der mit diesem Begriff verbundenen Vorurteile treffen auf Python nicht zu. So gehört Python zwar zu den dynamisch, aber gleichzeitig zu den streng typisierten Sprachen.

Literatur

Hinweis: Alle aufgeführten Quellen liegen der Arbeit in der zitierten Version auf CD-ROM bei und können bei Bedarf vom Autor angefordert werden.

Chaturvedi 2005a
CHATURVEDI, Shalabh: *Python Attributes and Methods*. Version: 1.19, Nov. 2005. http://www.cafepy.com/articles/python_attributes_and_methods/. – Online–Ressource, Abruf: 19. Apr. 2006

Chaturvedi 2005b
CHATURVEDI, Shalabh: *Python Types and Objects*. Version: 1.12, Nov. 2005. http://www.cafepy.com/articles/python_types_and_objects/. – Online–Ressource, Abruf: 19. Apr. 2006

FSF 2006
Free Software Foundation: *Various Licenses and Comments about Them*. 28. März 2006. http://www.fsf.org/licensing/licenses/index_html

Kuchling u. Cannon 2004
KUCHLING, A.M. ; CANNON, Brett: *PEP 3100: Miscalleneous Python 3.0 Plans*. Version: 6. Apr. 2004. http://www.python.org/peps/pep-3100.html. – Online–Ressource, Abruf: 9. Apr. 2006

Pilgrim 2004
PILGRIM, Mark: *Dive Into Python*. 2. Auflage. Apress, 2004. – Online verfügbar unter http://diveintopython.org/

PSF 2006a
Python Software Foundation: *Python 2.4 license*. Release 2.4.3. 29. März. 2006. http://www.python.org/download/releases/2.4.3/license/

PSF 2006b
Python Software Foundation: *Python Library Reference*. Release 2.4.3. 29. März. 2006. http://www.python.org/doc/2.4.3/lib/lib.html

PSF 2006c
Python Software Foundation: *Python Reference Manual*. Release 2.4.3. 29. März. 2006. http://www.python.org/doc/2.4.3/ref/ref.html

PSF 2006d
Python Software Foundation: *Python Success Stories*. 2006. http://www.python.org/about/success/

PSF 2006e
Python Software Foundation: *Python Tutorial*. Release 2.4.3. 29. März. 2006. http://www.python.org/doc/2.4.3/tut/tut.html

van Rossum 1998
Rossum, Guido van: *Glue it all together with Python*. Version: 6-8. Jan. 1998. http://www.python.org/doc/essays/omg-darpa-mcc-position.html. – Online–Ressource, Abruf: 19. Apr. 2006

van Rossum 2001
Rossum, Guido van: *PEP 8: Style Guide for Python Code*. Version: 5. Jul. 2001. http://www.python.org/peps/pep-0008.html. – Online–Ressource, Abruf: 19. Apr. 2006

van Rossum 2002
Rossum, Guido van: *Unifying Types and Classes*. Version: 4. Apr. 2002. http://www.python.org/2.2.3/descrintro.html. – Online–Ressource, Abruf: 19. Apr. 2006

van Rossum 2006
Rossum, Guido van: *Python 3000 - Adaptation or Generic Functions?* Version: 5. Apr. 2006. http://www.artima.com/weblogs/viewpost.jsp?thread=155123. – Online–Ressource, Abruf: 19. Apr. 2006

Simionato 2003
Simionato, Michele: *The Python 2.3 Method Resolution Order*. Version: 1.4, 17. Nov. 2003. http://www.python.org/2.3/mro.html. – Online–Ressource, Abruf: 19. Apr. 2006

Smith u. a. 2005
Smith, Kevin D. ; Jewett, Jim ; Montanaro, Skip ; Baxter, Anthony: *PEP 318: Decorators for Functions and Methods*. Version: 1965, 29. Jan. 2005. http://www.python.org/peps/pep-0318.html. – Online–Ressource, Abruf: 19. Apr. 2006

Venners 2003a
Venners, Bill: *The Making of Python*. Version: 13. Jan 2003. http://www.artima.com/intv/pythonP.html. – Online–Ressource, Abruf: 19. Apr. 2006

Venners 2003b
Venners, Bill: *Python's Design Goals*. Version: 20. Jan 2003. http://www.artima.com/intv/pyscaleP.html. – Online–Ressource, Abruf: 19. Apr. 2006

Erklärung der Selbständigkeit

Ich erkläre hiermit, dass ich die vorliegende Arbeit ohne fremde Hilfe selbstständig verfasst und nur die angegebenen Quellen und Hilfsmittel verwendet habe. Wörtlich oder sinngemäß aus anderen Werken entnommene Stellen habe ich unter Angabe der jeweiligen Quellen kenntlich gemacht.

Jochen Schulz