Informatik I

14. Weitere Python-Grundlagen

Jan-Georg Smaus

Albert-Ludwigs-Universität Freiburg

11. Januar 2011

Informatik I

11. Januar 2011 — 14. Weitere Python-Grundlagen

14.1 Typen

14.2 Einige Typen

14.3 Testen

14.4 Verzweigungen

14.1 Typen

- Dynamische Typisierung
- Signaturen

Dynamische Typisierung

Python ist dynamisch getypt: erst zur Laufzeit wird überprüft, ob die Typen der Argumente einer aufgerufenen Funktion oder eines Operators korrekt sind.

Dynamische Typisierung

Beispiel

type_error.py

```
def func(x):
    print("spam" / 5)
    return(x)
```

Python-Interpreter

```
>>> from type_error import *
>>> func(3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "type_error.py", line 2, in func
    print("spam" / 5)
TypeError: unsupported operand type(s) for /:
  'str' and 'int'
```

Vergleich zu Scheme

Wie verhält sich in dieser Hinsicht Scheme?

Scheme ist ebenfalls dynamisch getypt.

- Betrachten wir erst die Situation ohne jegliche Signaturen im Programm. Wenn etwa eine Prozedur den Teilausdruck

  ```
  (+ "banana" 1)
  ```

  enthält, so gibt es erst zur Laufzeit einen Typfehler.
  Beachte: hier wird ein primitiver Operator verwendet.
- Zusätzlich erlaubt Scheme Signaturen und erkennt u.U. Signaturverletzungen.
- Die Details sind unübersichtlich. Es ist ohne weiteres möglich, dass eine Prozedur, die gemäß ihrer Signatur verwendet wird, einen Typfehler hervorruft, da die Erkennung von Signaturverletzungen in DrRacket unvollkommen ist. Siehe `dynamic_typing.rkt`.

Keine Signaturen in Python

- In Scheme kann man mit der Syntax
  (: ⟨*name*⟩ ⟨*signature*⟩)
  die Signatur eines Namens (Prozedur oder einfacher Ausdruck) deklarieren.
  Dies ist aber noch nicht lange so, siehe [KS07].
- In Python ist dies nicht möglich: Weder für Variablen noch für Funktionen wird der Typ/die Signatur deklariert. Anders gesagt: Es gibt keine Typeklarationen.
- Es gibt Programmiersprachen, z.B. C, in denen für jede Variable und jede Funktion der Typ / die Signatur deklariert werden muss.

Sorte/Typ vs. Signatur

Zwei Programmiersprachen im Gepäck zu haben, kann verwirrend sein ...

- Die Begriffe Typ und Sorte sind im Wesentlichen austauschbar und von der "Community" abhängig. Bei Scheme sprechen wir von Sorten, bei Python von Typen.
- Die Begriffe Sorte/Typ und Signatur sind nicht ganz scharf gegeneinander abgegrenzt. Der Begriff Signatur kann auf zwei Aspekte abzielen:
  - Die Sorte eines Namens wird ausdrücklich deklariert, etwa in (Scheme!)

    ```
    (: l (liste %a))
    ```
  - Es handelt sich um eine Funktion, also etwas, das Eingabe- und Ausgabetypen hat, im Gegensatz zu einem einfachen Ausdruck, der einen Typen wie `string` oder `integer` hat.

Die erste Verwendung scheidet für Python aus, die zweite bietet sich an.

Syntax für Signaturen

- Wie gesagt: in Python gibt es keine Syntax, um die Signatur einer Funktion zu deklarieren!
- Trotzdem könnte ich Ihnen die Signatur einer Funktion kommunizieren wollen, oder Sie könnten sie in ein Programm als Kommentar schreiben wollen. Hierfür verwenden wir die ad-hoc Syntax

$$\tau_1 \times \ldots \times \tau_n \to \tau$$

14.2 Einige Typen

- Zahlentypen
- `bool`
- `str`

Konversion von Zahlentypen

- Wir erinnern uns: Python kennt die Zahlentypen `int`, `float` und `complex`.
- Zu den Typen `int`, `float` und `complex` gibt es gleichnamige Funktionen, die ihr Argument in den entsprechenden Typ umwandeln. Beispiele:

Python-Interpreter

```
>>> float(7)
7.0
>>> float(7+2j)
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError:  can't convert complex to float
>>> int(7.1)
7
```

Der Typ bool (Wahrheitswerte)

Der Typ bool (nicht: "boolean") enthält zwei Konstanten (Literale):

```
True      ; wahr
False     ; falsch
```

Der Typ bool: Operationen

...mit Signatur bool × bool → bool

`and` `or`

Werden infix geschrieben. `and` bindet stärker als `or`, ansonsten braucht man Klammern:

Python-Interpreter

```
>>> False and False or True
True
>>> False and (False or True)
False
```

Der Typ bool: Operationen II

...mit Signatur bool → bool

`not`

Wird in Präfixsyntax geschrieben:

Python-Interpreter

```
>>> not True
False
>>> not(True)
False
```

Der Typ bool: Operationen III

...mit Signatur real × real → bool

`==` `<` `>` `>=` `<=`

Werden infix geschrieben.

- Beachte den Unterschied zwischen == und =.
- Vergleiche können wie in der Mathematik verkettet werden:

```
0 <= x <= 10
```

ist also äquivalent zu

```
(0 <= x) and (x <= 10).
```

Ausdrücke vom Typ bool

Python-Interpreter

```
>>> y = 1
>>> 5 == 2 * 2 + y and 0 <= y and y < 2
True
```

Wir hatten das gleiche Beispiel in Scheme, siehe den dritten Foliensatz.

bool als Zahl

`True` und `False` entsprechen den Zahlen 1 und 0 (`bool` ist ein "Subtyp" von `int`), und man kann mit ihnen rechnen:

Python-Interpreter

```
>>> int(True)
1
>>> print(True + True)
2
```

Ebenso wie `int`, `float` & Co. kann man `bool` als Funktion verwenden:

- `bool(None)` ist `False`.
- `bool(0)` ist `False` (ebenso für `0.0` oder `0j`).
- Ansonsten gilt `bool(x) == True`, falls `x` eine Zahl ist.

Der Typ str

Den Typen String kennen wir schon aus Scheme. I.a. haben die Konstanten dieses Typs die Form

"$c_1 c_2 \ldots c_n$"

wobei die c_i beliebige Zeichen ("auf der Tastatur") außer " sein dürfen.
Schreibe \" um " in einer Zeichenkette zu verwenden.
Beispiele:

```
"Der Mond ist aufgegangen."
"Harry schrie: \"Expelliarmus!\""
```

Operationen (mit Signatur str × str → bool)

```
==    <    >
```

Es gibt noch viele andere Arten, einen String in Python zu schreiben.

14.3 Testen

Testfälle

- In Scheme haben wir allergrößten Wert auf das Schreiben von Testfällen gelegt.
- Es gibt keine allgemeingültige Art, Testfälle in Python zu definieren. Wir führen jetzt ein paar Konventionen ein . . .

Testfälle in Python

Ein einzelner Test kann mit Hilfe der `assert`-Anweisung realisiert werden. Die allgemeine Form ist

$$\texttt{assert } e,$$

wobei e ein Ausdruck vom Typen `bool` ist. Dem Test

$$\texttt{(check-expect } e_1 \ e_2\texttt{)}$$

in Scheme entspricht dann die Anweisung

$$\texttt{assert } e_1 \texttt{ == } e_2$$

in Python, und dem Test

$$\texttt{(check-within } e_1 \ e_2 \ \varepsilon\texttt{)}$$

in Scheme entspricht die Anweisung

$$\texttt{assert } e_2 - \varepsilon \texttt{ <= } e_1 \texttt{ <= } e_2 + \varepsilon.$$

Testfälle in Python II

Python zählt und protokolliert die fehlgeschlagenen Tests nicht, sondern bricht die Programmausführung nach dem ersten fehlgeschlagenen Test ab. Sei `my_function` die zu testende Funktion. Dann sammeln wir alle Tests, die sich auf `my_function` beziehen, in einer neuen parameterlosen Funktion mit dem Namen `test_my_function`. Die gesammelten Tests zu einer Funktion

```
def euro_to_dollar(euro):
    return euro*1.3
```

könnten beispielsweise wie folgt aussehen:

```
def test_euro_to_dollar():
    assert euro_to_dollar(0) == 0
    assert 1.2999 <= euro_to_dollar(1) <= 1.3001
```

Testfälle in Python III

Um zu überprüfen, ob Ihre Tests erfolgreich sind, muss man die Tests noch aufrufen, etwa mit:

```
test_euro_to_dollar()
```

14.4 Verzweigungen

Verzweigungen in Python

Wir hangeln uns weiter an der Scheme-Präsentation entlang (dritter Foliensatz).

Allerdings geht es hier nur darum, eine andere Syntax zu lernen. Daher können wir schnell vorgehen.

Binäre Verzweigungen

Beispiel: Absolutbetrag

Ein Funktion zur Berechnung des Absolutbetrags einer Zahl sieht folgendermaßen aus:

absolute.py

```
def absolute(x):
    if x >= 0:
        return x
    else:
        return -x
```

Verzicht auf Aufzählungstypen

Für allgemeine Verzweigungen war unser erstes Beispiel folgendes:

Aufgabe: Aggregatzustand von Wasser bestimmen

Eingabe: Wassertemperatur t

Ausgabe: "'solid"', "'liquid"' oder "'gaseous"' je nachdem, ob die Temperatur weniger als 0, zwischen 0 und 100, oder über 100 beträgt.

In Scheme haben wir nun eine Aufzählungssorte `phase` definiert, die die Literale `"solid"`, `"liquid"` und `"gaseous"` enthält.

In Python verzichten wir darauf, einen Aufzählungstypen zu definieren! Es gibt keine einfache und allgemein verbreitete Unterstützung für Aufzählungstypen in Python.
D.h. die Ausgabe hat einfach den Typen `str`.

Allgemeine Verzweigungen: elif

water_phase.py

```
def water_phase(t):
    if t < 0:
        return "solid"
    elif 0 <= t <= 100:
        return "liquid"
    elif t > 100:
        return "gaseous"
```

Kombination von elif und else

water_phase.py

```
def water_phase(t):
    if t < 0:
        return "solid"
    elif 0 <= t <= 100:
        return "liquid"
    else:
        return "gaseous"
```

Zusammenfassung

Heute eine ziemlich bunte Mischung:

- Python ist dynamisch getypt
- "Signatur" hat leicht verschiedene Bedeutungen
- Zahlen, Wahrheitswerte, Strings
- Testfälle
- Verwzeigungen: `if`, `else`, `elif`.

Literatur


Herbert Klaeren and Michael Sperber.
Die Macht der Abstraktion.
Teubner Verlag, 2007.