Algorithmik und Programmieren

Brückenkurs Informatik 2015

Benedikt Hupfauf (benedikt.hupfauf@uibk.ac.at)

Inhaltsübersicht

Termine

Der Kurs findet an folgenden Terminen statt:

- Montag 28.09.2015 von 11:15-13:00
- Dienstag 29.09.2015 von 14:15-16:00
- Mittwoch 30.09.2015 von 11:15-13:00 und 14:15-16:00
- Donnerstag 01.10.2015 von 11:15-13:00
- Freitag 02.10.2015 von 11:15-13:00 und 14:15-16:00

Was bietet dieser Kurs?

Sie lernen in diesem Kurs ...

- Einfache Algorithmen oder Probleme zu abstrahieren
- Grundlagen der Programmierung
- Python

Ablauf

Jede Einheit umfasst zwei Teile:

Theorie Konzepte der Programmierung

Praxis Vertiefung der Theorie anhand zahlreicher Beispiele – hier sind Sie gefragt!

Ablauf der Übungen

- Praktische Übungen machen diese Veranstaltung aus
- Am meisten lernen Sie, indem Sie selbst programmieren!
- Arbeiten Sie nach ihrem eigenen Tempo
- Keine Sorge, falls Sie nicht alle Aufgaben schaffen
- In Gruppen sollten alle Mitglieder mithalten können
- Falls Sie bereits fertig sind, versuchen Sie einem Studienkollegen zu helfen – wirklich verstanden haben Sie etwas erst, wenn Sie es auch einem anderen erklären können
- Lernen Sie ihre Studienkollegen kennen

Python

Vorteile von Python:

- Python ist eine Skriptsprache
- Ziel der Sprache sind kurze, einfach zu lesende Programme
- Unterstützt mehrere Paradigmen (u.A. Objektorientierung)
- Plattformunabhängig
- Umfassende Standardbibliothek
- Gute Unterstützung, weit verbreitet (bei Problemen finden Sie im Internet fast immer eine Lösung!)
- Frei und quelloffen

Resourcen

Hilfreiche Resourcen:

- `http://docs.python.org/3/`
- A Byte of Python, kostenlos zum download hier: `http://swaroopch.com/notes/python/`
- man python bzw. help(' ... ') im interkativen Modus
- Bibliothek

Ein Programm starten

Es gibt zwei Möglichkeiten ein Python Programm auszuführen:

- Führen Sie python auf der Kommandozeile aus, um den interaktiven Modus (auch: Interpreter) zu starten. Jede Zeile Code die Sie eingeben wird sofort ausgeführt.
- Sie können ein Programm mittels python <sourcefile> ausführen. Alle Kommandos in <sourcefile> werden dann so ausgeführt, als ob Sie sie Zeile für Zeile im Interpreter eingeben!

Datentypen, Konstanten

Python bietet Datentypen für die gängigsten Anwendungen:

- Strings (Zeichenketten, Text): 'This␣is␣text!', ''That's␣more␣text.''
- Integer (Ganzzahlen): 1, 4294967296, 010, 0xFF, 0b100
- Float (Gleitkommazahlen): 1.35, 1.56e−5 (entspricht $1.56 * 10^{-5}$)
- Complex (Komplexe Zahlen): 2 + 3j
- Bool (Wahr/Falsch): True, False

Variablen

Konstante Werte sind natürlich nicht immer ausreichend, häufig werden Variablen gebraucht. Variablen können beliebige Werte beliebiger Typen zugewiesen werden.

Achtung: Variablen haben zwar keinen statischen Typ, d.h. es muss nicht vorher angegeben werden, welchen Typ sie haben und der Typ kann sich zur Laufzeit ändern. Variablen haben aber sehrwohl einen dynamischen Typ, der bestimmt welche Operationen für diese Variable zulässig sind! Hinweis: >>> ist nicht Teil des Codes, sondern symbolisiert den Interpreter.

```
1 >>> a = 2
2 >>> a = "Test"
3 >>> a + 3
4 Traceback (most recent call last):
5   File "<stdin>", line 1, in <module>
6 TypeError: cannot concatenate 'str' and 'int' objects
```

Variablennamen

Gültige Variablennamen sind eine beliebige Kombination aus folgenden Zeichen: a-z, A-Z, 0-9 und _. Beachten Sie folgende Ausnahmen:

- Variablennamen dürfen nicht mit einer Ziffer beginnen
- Reservierte Schlüsselwörter:
  and, as, assert, break, class, continue, def, del, elif, else, except, exec, finally, for, from, global, if, import, in, is, lambda, not, or, pass, print, raise, return, try, while, with, yield

Variablen sollten am besten einen selbsterklärenden (vorzugsweise englischen) Namen haben.

Ausdrücke

Sie können mit Ausdrücken Variablen Werte zuweisen. Hierfür stellt Python bereits viele Operatoren bereit, beispielsweise $+$, $*$, $<$, $=$, $==$ uvm.. Beachten Sie, dass $=$ einer Variable einen Wert zuweist, während $==$ zwei Werte vergleicht ohne sie zu verändern.

```
>>> x = 2 + 1
>>> x = 2 * x
>>> print(x)
6
>>> x == 5
False
>>> print(x)
6
```

Operatoren Präzedenz

In welcher Reihenfolge werden Operatoren angewandt?

Operator	Bedeutung
or	Boolean OR
and	Boolean AND
not x	Boolean NOT
in, not in	Membership tests
is, is not	Identity tests
<, <=, >, >=, ! =, ==	Comparisons
|	Bitwise OR
^	Bitwise XOR
&	Bitwise AND
<<, >>	Shifts
+, -	Addition and subtraction
*, /, %	Multiplication, Division and Remainder
+x, -x	Positive, Negative
~x	Bitwise NOT
**	Exponentiation

Kontrollfluss – If

Anweisungen werden sequentiell ausgeführt. Ist das nicht genug, bietet Python (wie fast alle Programmiersprachen) einige Befehle um den Kontrollfluss zu steuern.

Die *if*-Anweisung führt einen Code Block nur aus, wenn die Bedingung <condition> erfüllt ist. Optional kann ein **else** Block angegeben, der ausgeführt wird, wenn die Bedingung nicht erfüllt ist.

```
1 if <condition>:
2     <executed when condition true>
3 else:
4     <executed when condition false>
```

Kontrollfluss – If, Beispiel

Abhängig vom Wert der Variable a wird **print** mit verschiedenen Parametern aufgerufen. Das Schlüsselwort **elif** <other condition>: ist eine Kurzschreibweise für das häufig gebrauchte **else**: **if** <other condition>:.

```
>>> if a < 0:
...     print('a is negative')
... elif a > 0:
...     print('a is positive')
... else:
...     print('a must be zero then')
```

Kontrollfluss – Codeblöcke

Im Gegensatz zu den meisten Programmiersprachen ist die Einrückung in Python nicht optional, sondern definiert einen Codeblock. Sie können Code mit <tab> oder mit 2-4 Leerzeichen einrücken, sollten aber bei einer dieser Optionen bleiben.

```
>>> if a < 0:
...     print('a is negative')
...     print('this line and the previous line belong to
    the same codeblock!')
...
>>> print('this is a new code block')
```

Kontrollfluss – While

Mit dem while-Statement wird ein Programmblock wiederholt solange die Bedingung <condition> erfüllt ist. Optional kann ein **else** Block angegeben werden, der einmal ausgeführt wird, sobald die Bedingung nicht mehr erfüllt ist.

```
1 while <condition>:
2       <executed as long as condition true>
3 else:
4       <executed once after condition is false>
```

Kontrollfluss – While, Beispiel

Achten Sie darauf, dass die Bedingung nicht für immer True sein darf, da die Schleife sonst endlos ist. Der Code Block des **else** Statements wird immer genau einmal ausgeführt.

```
>>> i = 0
>>> while i < 3:
...     i = i + 1
...     print(i)
... else:
...     print('end')
...
1
2
3
end
```

Kontrollfluss – Break und Continue

Eine Schleife muss nicht immer bis zum Ende ausgeführt werden. Mit dem Schlüsselwort **break** wir die Schleife sofort beendet. Mit dem Schlüsselwort **continue** wird der *aktuelle Schleifendurchlauf* abgebrochen. Achtung: der **else** Block wird bei **break** nicht ausgeführt!

```
>>> i = 0
>>> while i < 1000:
...     i = i + 1
...     if i == 3: break
...     print(i)
...     if i == 1: continue
...     print(i)
... else:
...     print('end')
...
1
2
2
```

Funktionen

Häufig verwendete Codeteile können in Funktionen ausgelagert werden. Eine Funktion sollte immer genau einen Zweck erfüllen! Funktionen können, müssen aber keinen Rückgabewert haben (Schlüsselwort **return**). Die Parameterliste parameters kann beliebig lang, also auch leer, sein.

```
1 def <function name>(<parameters>):
2     <function code>
```

Funktionen, Beispiel

Die folgenden Minimalbeispiele sollen nur zeigen wie Funktionen definiert werden können. Eine Funktion umfasst typischerweise mehr als nur eine Zeile Code.

```
>>> def mul(x, y):
...     return x * y
...
>>> mul(3,5)
15
```

```
>>> def printABC():
...         print('abcdefghijklmnopqrstuvxyz')
...
>>> printABC()
abcdefghijklmnopqrstuvxyz
```

Funktionen und Module

Python bietet eine große Auswahl an Funktionen, die bereits vorimplementiert sind (etwa `type()`, `help()`, ...). Funktionen die „zusammen gehören" werden in Module ausgelagert, etwa das Modul `math`, in dem Sie zahlreiche Funktionen aus der Mathematik finden. Werden Funktionen aus einem Modul benötigt, muss dieses explizit geladen werden.

```
>>> import math
>>> math.sin(45)
0.8509035245341184
>>> from math import sin
>>> sin(45)
0.8509035245341184
```

Datenstrukturen – Listen

Bisher haben wir uns mit einzelnen Variablen und Konstanten beschäftigt. Datenstrukturen erlauben es große Datenmengen zusammenzufassen, und zu verarbeiten.

Die einfachste Datenstruktur in Python ist die Liste. Eine Liste ist eine geordnete (geordnet heißt nicht zwangsläufig sortiert!), endliche Sammlung von Elementen beliebiger Datentypen.

```
1 >>> list = [2,3,1,2]
2 >>> list
3 [2, 3, 1, 2]
```

Noch einmal Kontrollfluss – For

Mit dem for-Statement wird ein Programmblock für jedes Element einer Sequenz <sequence> (Beispiele: Liste, Tupel, Map, ...) ausgeführt. Der **else** Block wird einmal am Ende der Schleife ausgeführt.

```
1 for <variable> in <sequence>:
2     <executed once for each element in sequence>
3 else:
4     <executed once in the end>
```

Noch einmal Kontrollfluss – For, Beispiel

Bei dieser Schleife wird jedes Element der Liste [1,2,3] einmal ausgegeben. Sie können statt der Variable i auch jeden anderen gültigen Bezeichner verwenden.

```
>>> for i in [1,2,3]:
...     print(i)
... else:
...     print('end')
...
1
2
3
end
```

Datenstrukturen – Listen, Beispiele

Eine Liste kann Elemente verschiedener Datentypen beinhalten:

```
>>> for i in ['a',1.2,5]:
...     print(i)
...
a
1.2
5
```

Einzelne Elemente bzw. Sublisten können wie folgt referenziert werden:

```
>>> list = [1,2,3]
>>> list[2]
3
>>> list[1] = 27 # Hinweis: 0 ist das erste Element!
>>> list[0:2]
[1, 27]
```

Datenstrukturen – Listenfunktionen

Mit append kann ein neues Element an die Liste angehängt werden. Mit + können zwei Listen verkettet werden, * n wiederholt eine Liste n mal. Mit **in** kann abgefragt werden, ob ein Element in dieser Datenstruktur enthalten ist. len(list) gibt die Anzahl der Element in list zurück.

```
>>> list_a = ['a','b','c']
>>> list_b = ['x','y','z']
>>> list_a.append('d')
>>> list_a
['a', 'b', 'c', 'd']
>>> print(list_a + list_b)
['a', 'b', 'c', 'd', 'x', 'y', 'z']
>>> list_b * 3
['x', 'y', 'z', 'x', 'y', 'z', 'x', 'y', 'z']
>>> 1 in [1,2,2]
True
>>> len(['py','th','on'])
3
```

Datenstrukturen – Tupel

Eine Tupel ist ähnlich einer Liste. Sie kennen Tupel bereits aus der Mathematik, etwa um karthesische Koordinaten (x,y) anzugeben. Die meisten Funktionen für Listen können auch auf ein Tupel angewendet werden.

```
>>> tuple = (3, 7)
>>> x, y = tuple
>>> x
3
>>> tuple[1]
7
>>> tuple * 3
(3, 7, 3, 7, 3, 7)
>>> tuple + (17, 6, 90)
(3, 7, 17, 6, 90)
>>> 3 in tuple
True
```

Datenstrukturen – Map

Eine Map (auch: Dictionary) ist eine Sammlung aus Schlüssel/Wert Paaren. Einem eindeutigem Schlüssel wird dabei ein Wert zugewiesen (dieser muss nicht eindeutig sein). Maps haben keine Ordnung.

```
>>> price = {'car': 30000, 'computer': 2000, 'shoes':
    70, 'cheap mobile': 70}
>>> price['shoes']
70
>>> price.values()
[30000, 2000, 70, 70]
>>> price.keys()
['car', 'computer', 'shoes', 'cheap mobile']
>>> len(price)
4
>>> 'car' not in price
False
```

Referenzen – Referenzkopie und echte Kopie

Möchten Sie eine Datenstruktur kopieren, ist eine einfache Zuweisung nicht genug. Im unten angeführten Beispiel kopiert `list_b` nur eine Referenz, d.h. zwei Listen greifen auf den selben Speicherbereich zu. Änderungen in einer der beiden Listen sind in der anderen unmittelbar sichtbar.

```
>>> list_a = [1, 2, 3]
>>> list_b = list_a
>>> list_c = list(list_a)
>>> list_a[1] = 5
>>> list_b
[1, 5, 3]
>>> list_c
[1, 2, 3]
```

Zeichenketten

Strings werden in Python mit einfachen oder doppelten Hochkomma gekennzeichnet. Zwischen beiden Varianten gibt es keinen Unterschied, außer, dass bei einfachen Hochkomma das doppelte Hochkomma im String enhalten sein darf, und umgekehrt. Vieles was bei Listen besprochen wurde kann auch auf Strings angewendet werden (**for**, +, ∗ n, ...).

```
>>> text = 'a' + "b" "c"
>>> for c in text:
...     print c
...
a
b
c
>>> "abc" * 3
'abcabcabc'
>>> "test" in "This is a testtext"
True
```

Zeichenketten

Umfangreiche Funktionen für Strings sind im Modul string zusammengefasst. Mehr dazu in der Python Dokumentation `http://docs.python.org/2/library/string.html`.

```
>>> import string
>>> string.hexdigits
'0123456789abcdefABCDEF'
>>> string.lower("Convert a String to LOWER case!")
'convert a string to lower case!'
>>> string.replace("Replace one string.", "one", "a")
'Replace a string.'
>>> string.split("This is a test   text")
['This', 'is', 'a', 'test', 'text']
```