



TECHNISCHE
UNIVERSITÄT
DRESDEN

GRUNDLAGEN PYTHON

Anwendungsorientierter Pythonkurs für Ingenieure

Material von Ingo Keller, Bearbeitung: Christoph Pohl, Carsten Knoll

Dresden, 14.10.2013

Python Code ausführen

- Kommandozeile
 - Start → Ausführen → `cmd`
 - `C:\>python`
- Spyder (im Kurs verwendete Entwicklungsumgebung (IDE))
 - Start → Python(x,y) / Winpython → Spyder
- Interaktiver Modus ("normal" oder IPython)

```
>>> <Anweisung>          >>> type("Hello Word!")  
<Ergebnis>              <type 'str'>
```

- Nützliche Built-in Funktionen
`type`, `dir`, `help`

Numerische Datentypen

- Integer

```
>>> type(1)
<type 'int'>
```

- lange Integer

```
>>> type(1L)
<type 'long'>
```

- Gleitkommazahlen

```
>>> type(1.0)
<type 'float'>
```

- Komplexe Zahlen

```
>>> type(1 + 2j)
<type 'complex'>
```

- Operatoren

Addition	+
Subtraktion	-
Division	/
Integerdivision	//
Multiplikation	*
Exponentieren	**
Modulo	%

- Built-in Funktionen

- `round`, `pow`, etc.
- siehe `dir(__builtins__)`

- Modul `math`

- siehe `help(math)`

NoneType und Boolesche Werte

- **None**
 - Universeller False-Wert

```
>>> type(None)
<type 'NoneType'>
```

- Boolesche Werte

- **True** und **False**

```
>>> type(True)
<type 'bool'>
```

Datentyp	False-Wert
NoneType	None
int, long	0
float	0.0
complex	0 + 0j
str	""
list	[]
tuple	()
dict	{}
set, frozenset	set(), frozenset()

Operationen

Operation

Abkürzung

<code>x = x + y</code>	<code>x += y</code>
<code>x = x - y</code>	<code>x -= y</code>
<code>x = x * y</code>	<code>x *= y</code>
<code>x = x / y</code>	<code>x /= y</code>
<code>x = x % y</code>	<code>x %= y</code>
<code>x = +x</code>	
<code>x = -x</code>	
<code>x = x ** y</code>	<code>x **= y</code>
<code>x = x // y</code>	<code>x //= y</code>

Vergleichsoperation

<code>x == y</code>
<code>x != y</code>
<code>x < y</code>
<code>x <= y</code>
<code>x > y</code>
<code>x >= y</code>

Strings

```
str1 = "abc"
str2 = 'xyzabcefg'hi'
str3 = """
    abc
    """
str4 = ("abc"
        "def")
>>> str2[0] # 0 ist erster Index
'x'
>>> str2[1:4]
'yz'
>>> str2[-3:]
'ghi'
```

Escape-Sequenz	Erklärung
\a	erzeugt Signalton
\b	Backspace
\f	Seitenvorschub
\n	Zeilenumbruch
\r	Carriage Return
\t	horizontaler Tab
\v	vertikaler Tab
\"	Escaping "
\'	Escaping '
\\	Escaping \

Formatierung mit Strings

- Allgemeine Syntax

```
"...%n...%m..." % (Wert1, Wert2)
```

- Beispiele

```
>>> a = 'H'  
>>> b = 'ello World'  
>>> "%c%s" % (a, b)  
'Hello World'
```

- Erweiterung

```
>>> '%10.2f' % 3.1415  
'      3.14'
```

- Wichtige Methoden

- `endswith`, `index`, `replace`, `split`, `startswith`, ...

Format	Erklärung
d, i	Integer mit Vorzeichen
f	Float (Dezimaldarstellung)
g, G	Float (wiss. mit Exponent)
u	Integer ohne Vorzeichen
x	Hexadezimalzahl
o	Oktalzahl
e, E	Float (Exponentendarstellung)
c	einzelnes Zeichen
s, r	Zeichenkette

Listen

- Syntax
`[Wert_1, ..., Wert_n]`
- kann verändert werden
- kann Elemente beliebigen Typs enthalten
- kann sortiert werden
- Wichtige Methoden
`append, count, index, insert, pop, remove, reverse, sort`
- `sort` und `reverse` arbeiten „in place“ (Rückgabewert: `None`)

- Beispiele

```
>>> m = [7, 8, 9]
>>> n = ['a', 'z', 1, False]
>>> m.append('x')
>>> m[0]
7
>>> m[-1]
'x'
>>> m[:] # Anf. bis Ende
[7, 8, 9, 'x']
>>> m.pop(0)
7
>>> m.reverse()
>>> print(m)
['x', 9, 8]
```

- Syntax
(Wert_1, ..., Wert_n)
- kann **nicht** verändert werden
- → Zugriff schneller als auf Listen
- kann Elemente beliebigen Typs enthalten
- Wichtige Methoden
index

- Beispiele

```
>>> t = (7,8,9)
>>> t[0]
7
>>> t[-1]
9
>>> t[:] # Anf. bis Ende
(7,8,9)
>>> z = ('a', 'z', 1, False)
>>> t.index(8)
1
>>> z.index('a')
0
```

Sequentielle Datentypen

Operation	Erklärung
<code>s in x</code>	prüft, ob s Element von x ist
<code>s not in x</code>	prüft, ob s kein Element von x ist
<code>x + y</code>	Verkettung von x und y
<code>x * n</code>	Verkettung, sodass n Kopien von x existieren
<code>x[n]</code>	gibt das n-te Element von x zurück
<code>x[n:m]</code>	gibt die Teilsequenz von n bis m zurück
<code>x[n:m:k]</code>	gibt die Teilsequenz von n bis m zurück, bei der nur jedes k-te Element berücksichtigt wurde
<code>len(x)</code>	gibt die Anzahl von Elementen zurück
<code>min(x)</code>	gibt das kleinste Element zurück
<code>max(x)</code>	gibt das größte Element zurück

Assoziative Arrays (Dictionaries)

- Syntax

```
{ Key_1: Value_1,  
  Key_2: Value_2,  
  ... }
```

- Schlüssel-Wert-Paare
- Schlüssel müssen unveränderlich sein
- Zugriff auf Werte via
 - `d.get(key, default)`
 - `d[key]`
- Wichtige Methoden:
 - `keys`, `values`, `items`

- Beispiele

```
>>> d = { "Sachsen" : "Dresden",  
...      "Brandenburg" : "Potsdam"}  
>>> e = {1:'a', 2:'b', 3:'c'}  
>>> e[1]  
'a'  
>>> d.get("Sachsen")  
'Dresden'  
# kein Eintrag -> keine Ausgabe  
>>> d.get("Bayern") # -> None  
>>> d["Blubb"]  
KeyError: 'Blubb'  
>>> type(d)  
<type 'dict'>
```

Sets

- Syntax

```
set([Element_1, ..., Element_n])
```
- kann jedes Element nur einmal enthalten
- kann nicht sortiert werden
- kann verändert werden
- `frozenset` ist unveränderlich

- Beispiele

```
>>> engineers = set(['John', 'Jane',  
... 'Jack', 'Janice'])  
>>> programmers = set(['Jack', 'Sam',  
... 'Susan', 'Janice'])  
>>> managers = set(['Jane', 'Jack',  
... 'Susan', 'Zack'])  
>>> union = engineers | programmers  
>>> intersect = engineers & managers  
>>> difference = managers - engineers  
>>> engineers.add('Marvin')  
>>> print engineers  
set(['Jane', 'Marvin',  
'Janice', 'John', 'Jack'])
```

if-elif-else Verzweigungen

- Syntax

```
if <Bedingung>:  
    ...  
elif <Bedingung>:  
    ...  
else:  
    ...
```

- Abkürzung

```
y = (1 if x == 'a' else 2)
```

- Beispiele

```
>>> x = 1  
>>> if x == 1:  
...     print("x=1")  
...  
x=1  
>>> x = 4  
>>> if x == 1:  
...     print("x=1")  
... elif x == 3:  
...     print("x=3")  
... else:  
...     print("x !=1 und x != 3")  
x !=1 und x != 3
```

for-Schleife

- Syntax:

```
for <Variable> in <object>:  
    ...
```

- range Funktion

```
range(start, stop, step)
```

```
>>> range(1, 10, 2)  
[1, 3, 5, 7, 9]
```

- Beispiele:

```
>>> x = ['a', 'b', 'c']  
>>> count = 0  
>>> for a in x:  
...     count += 1  
...     print a  
a  
b  
c  
>>> print(count)  
3
```

while-Schleife

- Syntax

```
while <Bedingung>:  
    ...
```

- **break**

bricht komplette Schleife ab

```
while <Bedingung1>:  
    if <Bedingung2>:  
        break
```

- **continue**

beginnt nächsten Schleifendurchlauf

```
while <Bedingung1>:  
    if <Bedingung2>:  
        continue
```

- Beispiele

```
>>> x = 4  
>>> while x > 1:  
...     print(x)  
...     x -= 1  
...     print("x = 1")  
4  
3  
2  
x = 1
```

Funktionen

- Syntax

```
def funcName(Param_1, ..., Param_n):  
    ...  
    return <Resultat>
```

- optionale Parameter

```
def test(param = 'Hallo'):  
    print(param)
```

- Mehr zu Funktionen: → Kurs 02

- Beispiele

```
>>> def printSum(a, b):  
...     print(a+b)  
>>> printSum(1, 2)  
3  
>>> def printMult(a, b, c, d=0):  
...     print((a*b*c)+d)  
>>> printMult(2, 4, 3)  
24  
>>> printMult(a=3, c=4, b=3, d=3)  
39
```

Syntax

- Anweisungsblöcke durch Einrückung definiert (an Stelle von z.B. { ... })
 - Leerzeichen anstatt Tabulatoren nutzen!
 - ggf. eigenen Texteditor entsprechend einstellen
(in Spyder: **TAB** zum Einrücken, **SHIFT+ TAB** zum Ausrücken)
- Kommentare:
- Empfohlene maximale Zeilenlänge: 80 Zeichen
- wenn eine Anweisung mehr benötigt:
 - mit \ lässt sich die Anweisung auf der nächsten Zeile fortsetzen
 - bei Klammern erfolgt dies automatisch bis zur schließenden Klammer
 - über Aufteilung auf mehrere Anweisungen nachdenken

„Python-Spezialitäten“

- Indizierung beginnt immer bei 0
- Entpacken sequentieller Datentypen:

```
>>> x, y, z = range(3) # range(3) -> [0, 1, 2]
>>> y
1
```
- ∃ Umfangreiche Standardbibliothek („batteries included“)
 - <http://docs.python.org/2/library/>
 - bevor man selber etwas programmiert, nachschauen ob Funktionalität nicht bereits in Standardbibliothek vorhanden ist„Don't reinvent the wheel!“
- Python 3.x ist nicht vollständig abwärtskompatibel
- Kurs verwendet noch Python 2.7 (noch sehr weit verbreitet)

- Erweiterte interaktive Python Konsole
- sehr gut als „Taschenrechner“ oder um schnell mal was auszuprobieren
- auch sehr gut zum Erforschen unbekannter Module (meist schneller, als richtige Doku zu lesen)
- History inklusive Suche; bleibt über mehrere Sessions erhalten
- Formatierung und farbige Hervorhebung
- kann in eigene Programme eingebunden werden:

```
from IPython import embed
... # eigener Code
embed() # IPython starten
```

IPython (II)

- Auto-Vervollständigungen mit **TAB**
- Speichern von Ein- und Ausgabe (`_`, `__`, `---`)
- Objektinformationen mit `?`; Quellcode mit `??`
- „Magische Kommandos:“
 - Beispiele: `%macro`, `%bookmark`, `%store ...`
 - Hilfe: `%magic`
- Ausführen von Systembefehlen
 - Beispiel: `!cp $fname1 $fname2`
 - `cp`, `rm`, `ls`, ... schon eingebaut

IPython (III)

- Ausführen des Skripts in der aktiven IPython-Konsole
 - Die run-configuration einstellen, dass Spyder das aktuelle Skript in einer bereits bestehenden Konsole ausführt. Ggf. muss dieser Konsolenreiter neu erstellt werden.
 - Im Skript aus IPython das Untermodul `embed` importieren und an gewünschter Stelle ausführen

```
from IPython import embed
[...code...]
embed()
```

- Im Windowsexplorer die zusammen mit PythonXY installierte **enhanced console** ausführen, IPython starten und manuell das Skript mit `%run <filename.py>` ausführen.

Links (auch im OPAL-Wiki)

- Das offizielle englische Tutorial
- Detaillierter Einstiegskurs (verschiedene Sprachen)
- www.learnpython.org - Interaktives Tutorial

- Python: Programmieren macht Spaß, Crash Kurs (I) (auf deutsch)
- Crash Kurs (II)
- Umfangreicher, gut strukturierter Kurs