Einführung in die Programmierung mit Python

Kurze, aber umfassende Einführung in die Programmiersprache Python

Dominik George

OpenRheinRuhr 2013, Oberhausen

9. November 2013

- Dominik George (Nik, Natureshadow)
- 23 Jahre alt
- Student (Anglistik. Informatik, Elektrotechnik)
- Informatik-Projektlehrer an der Sekundarschule
- Organisation des Kinder- und Jugendprogramms der FrOSCon
- Routinemäßiger Konferenz-Besucher und -Mitmacher

Inhalt

Was ist Python?

- Eine Programmiersprache
- Frei und offen
- Einfach zu lernen
- Lesbar
- Erweiterbar
- Umfassend

Was ist Python?

- Eine Programmiersprache
- Frei und offen
- Einfach zu lernen
- Lesbar
- Erweiterbar
- Umfassend

Was ist Python?

- Eine Programmiersprache
- Frei und offen
- Einfach zu lernen
- Lesbar
- Erweiterbar
- Umfassend

Was ist Python?

- Eine Programmiersprache
- Frei und offen
- Einfach zu lernen
- Lesbar
- Erweiterbar
- Umfassend

Was ist Python?

- Eine Programmiersprache
- Frei und offen
- Einfach zu lernen
- Lesbar
- Erweiterbar
- Umfassend

Was ist Python?

- Eine Programmiersprache
- Frei und offen
- Einfach zu lernen
- Lesbar
- Erweiterbar
- Umfassend

Ideen, die Python abheben

- Code soll schön sein
- Einfaches soll einfach sein
- Lesbarkeit zählt
- Regeln sollen einhaltbar sein
- Der richtige Weg soll offensichtlich sein
- Holländer sind merkwürdig ;-)

Ideen, die Python abheben

- Code soll schön sein
- Einfaches soll einfach sein
- Lesbarkeit zählt
- Regeln sollen einhaltbar sein
- Der richtige Weg soll offensichtlich sein
- Holländer sind merkwürdig ;-)

Ideen, die Python abheben

- Code soll schön sein
- Einfaches soll einfach sein
- Lesbarkeit zählt
- Regeln sollen einhaltbar sein
- Der richtige Weg soll offensichtlich sein
- Holländer sind merkwürdig ;-)

Ideen, die Python abheben

- Code soll schön sein
- Einfaches soll einfach sein
- Lesbarkeit zählt
- Regeln sollen einhaltbar sein
- Der richtige Weg soll offensichtlich sein
- Holländer sind merkwürdig ;-)

Ideen, die Python abheben

- Code soll schön sein
- Einfaches soll einfach sein
- Lesbarkeit zählt
- Regeln sollen einhaltbar sein
- Der richtige Weg soll offensichtlich sein
- Holländer sind merkwürdig ;-)

Ideen, die Python abheben

- Code soll schön sein
- Einfaches soll einfach sein
- Lesbarkeit zählt
- Regeln sollen einhaltbar sein
- Der richtige Weg soll offensichtlich sein
- Holländer sind merkwürdig ;-)

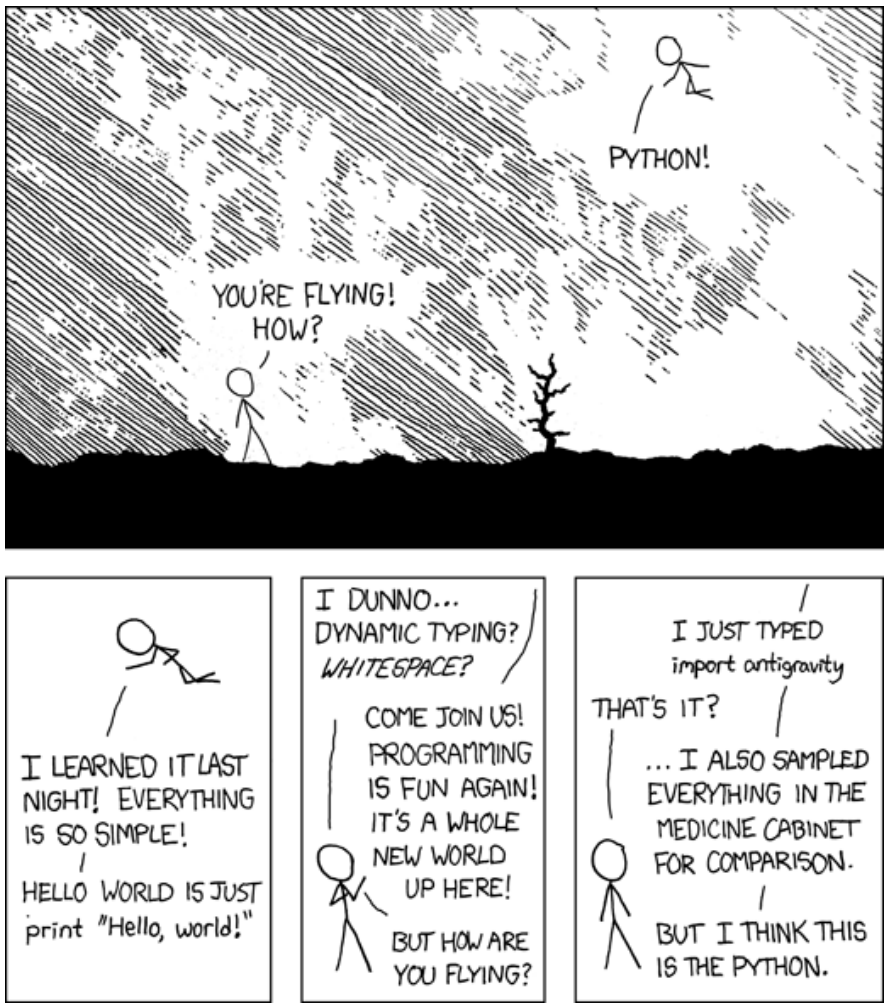
The Zen of Python, by Tim Peters

- Beautiful is better than ugly.
- Explicit is better than implicit.
- Simple is better than complex.
- Complex is better than complicated.
- Flat is better than nested.
- Sparse is better than dense.
- Readability counts.
- Special cases aren't special enough to break the rules.
- Although practicality beats purity.
- Errors should never pass silently.

The Zen of Python, by Tim Peters (Teil 2)

- Unless explicitly silenced.
- In the face of ambiguity, refuse the temptation to guess.
- There should be one– and preferably only one –obvious way to do it.
- Although that way may not be obvious at first unless you're Dutch.
- Now is better than never.
- Although never is often better than *right* now.
- If the implementation is hard to explain, it's a bad idea.
- If the implementation is easy to explain, it may be a good idea.
- Namespaces are one honking great idea – let's do more of those!

Python macht Spaß



Code soll schön sein

```
for i in range(0, 10):
    print(i)
```

vs.

```
int i;
for (i = 0; i < 10; i++) printf("%d\n", i);
```

Einfaches soll einfach sein

```
print("Hallo Welt!")
```

vs.

```
public class HalloProgramm {
 public static void main(String[] args) {
  System.out.println("Hallo Welt!");
  System.exit(0);
 }
}
```

Lesbarkeit zählt

```
for quant in range(99, 0, -1):
    if quant > 1:
        print(quant, "bottles of beer on the wall,", quant, ↩
            "bottles of beer.")
        suffix = str(quant - 1) + " bottle" + ("s" if quant ↩
            > 2 else "") + " of beer on the wall."
    elif quant == 1:
        print("1 bottle of beer on the wall, 1 bottle of ↩
            beer.")
        suffix = "no more beer on the wall!"
    print("Take one down, pass it around,", suffix)
```

vs.

```
sub b{$n=99-@_-$_||No;"$n bottle"."s"x!!--$n." of beer"};↩
    $w=" on the wall";
die map{b."$w,\n".b.",\nTake one down, pass it around,\n".↩
    b(0)."$w.\n\n"}0..98
```

Holländer sind merkwürdig ;-)

```
>>> a = "abd"
>>> a[2]
'd'
>>> a[2] = "c"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

Wait... what ;)?

Was wir für Python brauchen

- Einen Texteditor
  - Möglichst etwas brauchbarer als Microsoft® Notepad ;-)
  - Empfehlung für die Linuxer/BSDler: jupp
    - Schnell
    - Viele Features
    - Einfach zu bedienen
    - Gute Online-Hilfe (Strg+J)
    - Syntax-Highlighting
- Python-Interpreter
  - Python 2.7 oder 3.3
  - Interaktiv: bpython oder IDLE

Was wir für Python brauchen

- Einen Texteditor
  - Möglichst etwas brauchbarer als Microsoft® Notepad ;-)
  - Empfehlung für die Linuxer/BSDler: jupp
    - Schnell
    - Viele Features
    - Einfach zu bedienen
    - Gute Online-Hilfe (Strg+J)
    - Syntax-Highlighting
- Python-Interpreter
  - Python 2.7 oder 3.3
  - Interaktiv: bpython oder IDLE

Was wir für Python brauchen

- Einen Texteditor
  - Möglichst etwas brauchbarer als Microsoft® Notepad ;-)
  - Empfehlung für die Linuxer/BSDler: jupp
    - Schnell
    - Viele Features
    - Einfach zu bedienen
    - Gute Online-Hilfe (Strg+J)
    - Syntax-Highlighting
- Python-Interpreter
  - Python 2.7 oder 3.3
  - Interaktiv: bpython oder IDLE

Was wir für Python brauchen

- Einen Texteditor
  - Möglichst etwas brauchbarer als Microsoft® Notepad ;-)
  - Empfehlung für die Linuxer/BSDler: jupp
    - Schnell
    - Viele Features
    - Einfach zu bedienen
    - Gute Online-Hilfe (Strg+J)
    - Syntax-Highlighting
- Python-Interpreter
  - Python 2.7 oder 3.3
  - Interaktiv: bpython oder IDLE

Was wir für Python brauchen

- Einen Texteditor
  - Möglichst etwas brauchbarer als Microsoft® Notepad ;-)
  - Empfehlung für die Linuxer/BSDler: jupp
    - Schnell
    - Viele Features
    - Einfach zu bedienen
    - Gute Online-Hilfe (Strg+J)
    - Syntax-Highlighting
- Python-Interpreter
  - Python 2.7 oder 3.3
  - Interaktiv: bpython oder IDLE

Was wir für Python brauchen

- Einen Texteditor
  - Möglichst etwas brauchbarer als Microsoft® Notepad ;-)
  - Empfehlung für die Linuxer/BSDler: jupp
    - Schnell
    - Viele Features
    - Einfach zu bedienen
    - Gute Online-Hilfe (Strg+J)
    - Syntax-Highlighting
- Python-Interpreter
  - Python 2.7 oder 3.3
  - Interaktiv: bpython oder IDLE

Was wir für Python brauchen

- Einen Texteditor
  - Möglichst etwas brauchbarer als Microsoft® Notepad ;-)
  - Empfehlung für die Linuxer/BSDler: jupp
    - Schnell
    - Viele Features
    - Einfach zu bedienen
    - Gute Online-Hilfe (Strg+J)
    - Syntax-Highlighting
- Python-Interpreter
  - Python 2.7 oder 3.3
  - Interaktiv: bpython oder IDLE

Was wir für Python brauchen

- Einen Texteditor
  - Möglichst etwas brauchbarer als Microsoft® Notepad ;-)
  - Empfehlung für die Linuxer/BSDler: jupp
    - Schnell
    - Viele Features
    - Einfach zu bedienen
    - Gute Online-Hilfe (Strg+J)
    - Syntax-Highlighting
- Python-Interpreter
  - Python 2.7 oder 3.3
  - Interaktiv: bpython oder IDLE

Was wir für Python brauchen

- Einen Texteditor
  - Möglichst etwas brauchbarer als Microsoft® Notepad ;-)
  - Empfehlung für die Linuxer/BSDler: jupp
    - Schnell
    - Viele Features
    - Einfach zu bedienen
    - Gute Online-Hilfe (Strg+J)
    - Syntax-Highlighting
- Python-Interpreter
  - Python 2.7 oder 3.3
  - Interaktiv: bpython oder IDLE

Was wir für Python brauchen

- Einen Texteditor
  - Möglichst etwas brauchbarer als Microsoft® Notepad ;-)
  - Empfehlung für die Linuxer/BSDler: jupp
    - Schnell
    - Viele Features
    - Einfach zu bedienen
    - Gute Online-Hilfe (Strg+J)
    - Syntax-Highlighting
- Python-Interpreter
  - Python 2.7 oder 3.3
  - Interaktiv: bpython oder IDLE

Was wir für Python brauchen

- Einen Texteditor
  - Möglichst etwas brauchbarer als Microsoft® Notepad ;-)
  - Empfehlung für die Linuxer/BSDler: jupp
    - Schnell
    - Viele Features
    - Einfach zu bedienen
    - Gute Online-Hilfe (Strg+J)
    - Syntax-Highlighting
- Python-Interpreter
  - Python 2.7 oder 3.3
  - Interaktiv: bpython oder IDLE

Hallo Welt!

Ein einfaches, klassisches Hallo, Welt!-Programm in Python:

```
#!/usr/bin/env python
# ~*~ coding: utf-8 ~*~

print("Hallo, Welt!")
exit(0)
```

Funktionsaufrufe

Funktionsaufrufe haben in Python immer - grundlegend - dieselbe Struktur:

```
funktionsname(argument1, argument2, argument3, ...)
```

Statements (Anweisungen) und Blöcke

In Python wird meistens ein Statement pro Zeile geschrieben.

```
# Richtig
print("Hallo")
print("Welt")

# Meistens falsch
print("Hallo"); print("Welt")
```

Blöcke werden durch Einrückung markiert:

```
def eggs():
    spam()
    bacon()
```

Wichtig: Immer die gleiche Einrückung verwenden.
Empfehlung: 4 Leerzeichen (macht jupp automatisch richtig)

Variablen und Datentypen

- Variablen sind nicht typisiert ...
- ... zeigen aber auf Daten mit einem bestimmten Datentyp!
- Datentypen (unvollständig):
  - str, bool, int, float, complex, list, tuple, dict
- Komplexere Typen (später mehr):
  - Objekte, Iteratoren, ...
- In Python ist alles ßuweisbar"(später mehr)

Variablen und Datentypen

- Variablen sind nicht typisiert ...
- ... zeigen aber auf Daten mit einem bestimmten Datentyp!
- Datentypen (unvollständig):
  - str, bool, int, float, complex, list, tuple, dict
- Komplexere Typen (später mehr):
  - Objekte, Iteratoren, ...
- In Python ist alles ßuweisbar"(später mehr)

Variablen und Datentypen

- Variablen sind nicht typisiert ...
- ... zeigen aber auf Daten mit einem bestimmten Datentyp!
- Datentypen (unvollständig):
  - str, bool, int, float, complex, list, tuple, dict
- Komplexere Typen (später mehr):
  - Objekte, Iteratoren, ...
- In Python ist alles ßuweisbar" (später mehr)

Variablen und Datentypen

- Variablen sind nicht typisiert ...
- ... zeigen aber auf Daten mit einem bestimmten Datentyp!
- Datentypen (unvollständig):
  - str, bool, int, float, complex, list, tuple, dict
- Komplexere Typen (später mehr):
  - Objekte, Iteratoren, ...
- In Python ist alles ßuweisbar"(später mehr)

Variablen und Datentypen

- Variablen sind nicht typisiert ...
- ... zeigen aber auf Daten mit einem bestimmten Datentyp!
- Datentypen (unvollständig):
  - str, bool, int, float, complex, list, tuple, dict
- Komplexere Typen (später mehr):
  - Objekte, Iteratoren, ...
- In Python ist alles ßuweisbar" (später mehr)

Variablen und Datentypen

- Variablen sind nicht typisiert ...
- ... zeigen aber auf Daten mit einem bestimmten Datentyp!
- Datentypen (unvollständig):
  - str, bool, int, float, complex, list, tuple, dict
- Komplexere Typen (später mehr):
  - Objekte, Iteratoren, ...
- In Python ist alles ßuweisbar" (später mehr)

Variablen und Datentypen

- Variablen sind nicht typisiert ...
- ... zeigen aber auf Daten mit einem bestimmten Datentyp!
- Datentypen (unvollständig):
  - str, bool, int, float, complex, list, tuple, dict
- Komplexere Typen (später mehr):
  - Objekte, Iteratoren, ...
- In Python ist alles ßuweisbar"(später mehr)

Operatoren

- Arithmetische Operatoren (Berechnungen):
  - +, -, *, /, %, **, //
- Zuweisungen:
  - =, +=, -=, *=, /=, %=, **=, //=
- Vergleiche:
  - ==, !=, ¿, ¡, ¿=, ¡=
- Listenoperatoren:
  - in, not in
- Logische Operatoren:
  - and, or, not

Operatoren

- Arithmetische Operatoren (Berechnungen):
  - +, -, *, /, %, **, //
- Zuweisungen:
  - =, +=, -=, *=, /=, %=, **=, //=
- Vergleiche:
  - ==, !=, ¿, ¡, ¿=, ¡=
- Listenoperatoren:
  - in, not in
- Logische Operatoren:
  - and, or, not

Operatoren

- Arithmetische Operatoren (Berechnungen):
  - +, -, *, /, %, **, //
- Zuweisungen:
  - =, +=, -=, *=, /=, %=, **=, //=
- Vergleiche:
  - ==, !=, ¿, ¡, ¿=, ¡=
- Listenoperatoren:
  - in, not in
- Logische Operatoren:
  - and, or, not

Operatoren

- Arithmetische Operatoren (Berechnungen):
  - +, -, *, /, %, **, //
- Zuweisungen:
  - =, +=, -=, *=, /=, %=, **=, //=
- Vergleiche:
  - ==, !=, ¿, ¡, ¿=, ¡=
- Listenoperatoren:
  - in, not in
- Logische Operatoren:
  - and, or, not

Operatoren

- Arithmetische Operatoren (Berechnungen):
  - +, -, *, /, %, **, //
- Zuweisungen:
  - =, +=, -=, *=, /=, %=, **=, //=
- Vergleiche:
  - ==, !=, ¿, ¡, ¿=, ¡=
- Listenoperatoren:
  - in, not in
- Logische Operatoren:
  - and, or, not

Operatoren

- Arithmetische Operatoren (Berechnungen):
  - +, -, *, /, %, **, //
- Zuweisungen:
  - =, +=, -=, *=, /=, %=, **=, //=
- Vergleiche:
  - ==, !=, ¿, ¡, ¿=, ¡=
- Listenoperatoren:
  - in, not in
- Logische Operatoren:
  - and, or, not

Operatoren

- Arithmetische Operatoren (Berechnungen):
  - +, -, *, /, %, **, //
- Zuweisungen:
  - =, +=, -=, *=, /=, %=, **=, //=
- Vergleiche:
  - ==, !=, ¿, ¡, ¿=, ¡=
- Listenoperatoren:
  - in, not in
- Logische Operatoren:
  - and, or, not

Operatoren

- Arithmetische Operatoren (Berechnungen):
  - +, -, *, /, %, **, //
- Zuweisungen:
  - =, +=, -=, *=, /=, %=, **=, //=
- Vergleiche:
  - ==, !=, ¿, ¡, ¿=, ¡=
- Listenoperatoren:
  - in, not in
- Logische Operatoren:
  - and, or, not

Operatoren

- Arithmetische Operatoren (Berechnungen):
  - +, -, *, /, %, **, //
- Zuweisungen:
  - =, +=, -=, *=, /=, %=, **=, //=
- Vergleiche:
  - ==, !=, ¿, ¡, ¿=, ¡=
- Listenoperatoren:
  - in, not in
- Logische Operatoren:
  - and, or, not

Operatoren

- Arithmetische Operatoren (Berechnungen):
  - +, -, *, /, %, **, //
- Zuweisungen:
  - =, +=, -=, *=, /=, %=, **=, //=
- Vergleiche:
  - ==, !=, ¿, ¡, ¿=, ¡=
- Listenoperatoren:
  - in, not in
- Logische Operatoren:
  - and, or, not

Variablen, Datentypen und Operatoren (Beispiele)

```
a = "Hallo Welt!"
b = 5
c = 2.5
d = b / c
e = ["spam", "bacon", "eggs"]
f = {"name": "Monty Python",
     "favourite_food": "spam"}
```

Das Aufzählen endloser Beispiele wäre etwas langweilig, im Zweifelsfall einfach mal ausprobieren!

Einfache Ein- und Ausgabe

Die Funktionen print() und input() ermöglichen einfache Ein- und Ausgabe:

```
print("Hallo Welt!")
print(2)

mein_name = input("Wie heißt du? ")
print("Hallo, " + mein_name)
```

Achtung: In Python 2.x heißt input noch raw_input, pythonischer Hack siehe später!

Schleifen

- Es gibt beide klassischen Schleifen-Typen
  - while-Schleife
    - Code-Block so lange ausführen, bis eine Bedingung nicht mehr erfüllt ist.
    - Kleine pythonische Besonderheit, dazu später mehr ;-).
  - for-Schleife
    - Code-Block für jedes Element einer Menge einmal ausführen.
    - Kleine pythonische Besonderheit mit Iteratoren ...

Schleifen

- Es gibt beide klassischen Schleifen-Typen
  - while-Schleife
    - Code-Block so lange ausführen, bis eine Bedingung nicht mehr erfüllt ist.
    - Kleine pythonische Besonderheit, dazu später mehr ;-).
  - for-Schleife
    - Code-Block für jedes Element einer Menge einmal ausführen.
    - Kleine pythonische Besonderheit mit Iteratoren ...

Schleifen

- Es gibt beide klassischen Schleifen-Typen
  - while-Schleife
    - Code-Block so lange ausführen, bis eine Bedingung nicht mehr erfüllt ist.
    - Kleine pythonische Besonderheit, dazu später mehr ;-).
  - for-Schleife
    - Code-Block für jedes Element einer Menge einmal ausführen.
    - Kleine pythonische Besonderheit mit Iteratoren ...

Schleifen

- Es gibt beide klassischen Schleifen-Typen
  - while-Schleife
    - Code-Block so lange ausführen, bis eine Bedingung nicht mehr erfüllt ist.
    - Kleine pythonische Besonderheit, dazu später mehr ;-).
  - for-Schleife
    - Code-Block für jedes Element einer Menge einmal ausführen.
    - Kleine pythonische Besonderheit mit Iteratoren ...

Schleifen

- Es gibt beide klassischen Schleifen-Typen
  - while-Schleife
    - Code-Block so lange ausführen, bis eine Bedingung nicht mehr erfüllt ist.
    - Kleine pythonische Besonderheit, dazu später mehr ;-).
  - for-Schleife
    - Code-Block für jedes Element einer Menge einmal ausführen.
    - Kleine pythonische Besonderheit mit Iteratoren ...

Schleifen

- Es gibt beide klassischen Schleifen-Typen
  - while-Schleife
    - Code-Block so lange ausführen, bis eine Bedingung nicht mehr erfüllt ist.
    - Kleine pythonische Besonderheit, dazu später mehr ;-).
  - for-Schleife
    - Code-Block für jedes Element einer Menge einmal ausführen.
    - Kleine pythonische Besonderheit mit Iteratoren ...

Schleifen

- Es gibt beide klassischen Schleifen-Typen
  - while-Schleife
    - Code-Block so lange ausführen, bis eine Bedingung nicht mehr erfüllt ist.
    - Kleine pythonische Besonderheit, dazu später mehr ;-).
  - for-Schleife
    - Code-Block für jedes Element einer Menge einmal ausführen.
    - Kleine pythonische Besonderheit mit Iteratoren ...

Schleifen (Beispiele)

```
eingabe = ""
while eingabe != "exit":
    eingabe = input("prompt> ")
```

```
for food in ["spam", "bacon", "eggs"]:
    print("I like %s!" % food)
```

(Habt ihr die pythonische Besonderheit bemerkt ;-)? Später mehr!)

Bedingungen (Beispiele)

Weil Python so schön einfach ist, vertiefen wir direkt einmal das Eingabe-Beispiel von vorhin ;-) ...

```
eingabe = ""
while eingabe != "exit":
    eingabe = input("gimme food> ")
    if eingabe in ["eggs", "bacon", "ham"]:
        print("I like %s!" % eingabe)
    elif eingabe == "spam":
        print("Don't you have Python without the spam?")
    else:
        print("What the spam is %s?" % eingabe)
```

(Haben Sie den Bug bemerkt ;-)?)

Funktionen

Funktionen sind benannte Code-Blöcke mit einer Parameterliste und einem Rückgabewert.

```
def double(a):
    return 2 * a

print(double(5))
```

```
def count_to(n):
    i = 0
    while i <= n:
        print(n)

count_to(10)
```

Listen und Tuples

- Listen sind ein bisschen mächtiger als traditionelle Arrays.
- Listen beinhalten Einträge beliebiger Typen, auch gemischt.
- Listen kann man an beliebigen Stellen beliebig manipulieren.
- Tuples sind unveränderbare Listen.
- Allgemein heißen Äufzählungstypenïterables.

Listen und Tuples

- Listen sind ein bisschen mächtiger als traditionelle Arrays.
- Listen beinhalten Einträge beliebiger Typen, auch gemischt.
- Listen kann man an beliebigen Stellen beliebig manipulieren.
- Tuples sind unveränderbare Listen.
- Allgemein heißen Äufzählungstypenïterables.

Listen und Tuples

- Listen sind ein bisschen mächtiger als traditionelle Arrays.
- Listen beinhalten Einträge beliebiger Typen, auch gemischt.
- Listen kann man an beliebigen Stellen beliebig manipulieren.
- Tuples sind unveränderbare Listen.
- Allgemein heißen Äufzählungstypenïterables.

Listen und Tuples

- Listen sind ein bisschen mächtiger als traditionelle Arrays.
- Listen beinhalten Einträge beliebiger Typen, auch gemischt.
- Listen kann man an beliebigen Stellen beliebig manipulieren.
- Tuples sind unveränderbare Listen.
- Allgemein heißen Äufzählungstypenïterables.

Listen und Tuples

- Listen sind ein bisschen mächtiger als traditionelle Arrays.
- Listen beinhalten Einträge beliebiger Typen, auch gemischt.
- Listen kann man an beliebigen Stellen beliebig manipulieren.
- Tuples sind unveränderbare Listen.
- Allgemein heißen Äufzählungstypenïterables.

Listen (Beispiele)

```
foods = ["spam", "bacon", "eggs"]
foods[0]   # "spam"
foods[2]   # "eggs"
foods[-1]  # "eggs" -- vom Ende!

foods.append("beans")
foods      # ["spam", "bacon", "eggs", "beans"]

foods.index("eggs")   # 2
del foods[1]          # ["spam", "eggs", "beans"]
len(foods)            # 3
```

Strings

Strings können in Python nur mit speziellen String-Methoden verändert werden - Strings sind keine Zeichen-Arrays, wie in C! Aber: Strings sind ïterables d.h., lesend verhalten sie sich genau wie Listen!

```
message = "I like spam!"
message[0]     # "I"
message[-1]    # "!"
message[7:11] # "spam"

message = message.replace("spam", "bacon")
message        # "I like bacon!"
```

Dictionaries

Dictionaries sind prinzipiell Python's Art von assoziativen Arrays / Hashtables.

```
steckbrief = {"vorname": "John", "nachname": "Doe"}
steckbrief["vorname"]      # "John"

steckbrief["alter"] = 42
steckbrief      # {"vorname": "John", "nachname": "Doe", "↩
    alter": 42}

for key in steckbrief:
    print(steckbrief[key])
```

Umwandeln von Typen (Casts)

Manchmal ist es notwendig, ausdrücklich Typen umzuwandeln, auch wenn Python das meistens recht gut macht.

```
food = "Spam"
list(food)      # ["S", "p", "a", "m"]

float(5)        # 5.0
str(42)         # "42" -- ganz wichtig, siehe unten ...

a = 12
print("Ich bin " + str(a) + " Jahre alt!")
```

Klassen und Objekte

Klassen sind SSammlungen"von Variablen (Attributen) und Funktionen (Methoden) unter einem Namen. Von ihnen kann man voneinander unabhängige Kopien (Instanzen) erzeugen.

```
class Baum():
    def __init__(self, groesse):
        self.groesse = groesse

    def wachsen(self):
        self.groesse += 1

mein_baum = Baum(10)
mein_baum.wachsen()
mein_baum.groesse       # 11
```

Der erste Parameter der Methoden (hier self genannt) bekommt immer automatisch eine Referenz auf die Instanz übergeben, in der die Methode aufgerufen wurde.

Einsetzen in Strings

In Python sollten Variablen nicht einfach durch Verketten, sondern durch Platzhalter in Strings eingesetzt werden.

```
name = "John"
alter = 42

print("Mein Name ist %s; ich bin %i Jahre alt!" % (name, ↩
    alter))
```

List comprehensions

List comprehensions sind eine Kurzschreibweise, um schnell Listen nach einem bestimmen Muster erzeugen zu können. Im Prinzip wird eine for-Schleife aufgerufen und mit jedem Wert dieselbe Operation ausgeführt.

```
# Eine Reihe Zahlen, aber als String (direkter Cast)
zahlen = [str(x) for x in range(10)]
zahlen    # ["0", "1", "2", "3", "4", "5", "6", "7", "8", ↩
    "9"]

# Oder die Buchstaben von A-Z aus ihren ASCII-Werten
alpha = [chr(x= for x in range(65, 91)]
```

Iteratoren/Generatoren

Iteratoren sind eine Möglichkeit, eine Menge von Elementen zu generieren, ohne diese komplett vorberechnen zu müssen. Was sinnlos klingt, hat Performance-Gründe ...

- Wenn ich eine Bibliothek entwickle, und dabei eine Menge erzeuge, weiß ich nicht unbedingt, wie viele Elemente davon der Benutzer braucht.
- for-Schleifen laufen "gleichmäßiger jedes Element wird zum Zeitpunkt der Benutzung erzeugt.
- Speicherverbrauch ist minimal - nur ein Element muss im Speicher gehalten werden.
- Prinzip ist einfach: Eine Funktion erzeugt einen Rückgabewert, und die Position im Code wird gespeichert. Beim nächsten Aufruf läuft die Funktion an dieser Stelle weiter.

Iteratoren/Generatoren

Iteratoren sind eine Möglichkeit, eine Menge von Elementen zu generieren, ohne diese komplett vorberechnen zu müssen. Was sinnlos klingt, hat Performance-Gründe ...

- Wenn ich eine Bibliothek entwickle, und dabei eine Menge erzeuge, weiß ich nicht unbedingt, wie viele Elemente davon der Benutzer braucht.
- for-Schleifen laufen "gleichmäßiger jedes Element wird zum Zeitpunkt der Benutzung erzeugt.
- Speicherverbrauch ist minimal - nur ein Element muss im Speicher gehalten werden.
- Prinzip ist einfach: Eine Funktion erzeugt einen Rückgabewert, und die Position im Code wird gespeichert. Beim nächsten Aufruf läuft die Funktion an dieser Stelle weiter.

Iteratoren/Generatoren

Iteratoren sind eine Möglichkeit, eine Menge von Elementen zu generieren, ohne diese komplett vorberechnen zu müssen. Was sinnlos klingt, hat Performance-Gründe ...

- Wenn ich eine Bibliothek entwickle, und dabei eine Menge erzeuge, weiß ich nicht unbedingt, wie viele Elemente davon der Benutzer braucht.
- for-Schleifen laufen "gleichmäßiger jedes Element wird zum Zeitpunkt der Benutzung erzeugt.
- Speicherverbrauch ist minimal - nur ein Element muss im Speicher gehalten werden.
- Prinzip ist einfach: Eine Funktion erzeugt einen Rückgabewert, und die Position im Code wird gespeichert. Beim nächsten Aufruf läuft die Funktion an dieser Stelle weiter.

Iteratoren/Generatoren

Iteratoren sind eine Möglichkeit, eine Menge von Elementen zu generieren, ohne diese komplett vorberechnen zu müssen. Was sinnlos klingt, hat Performance-Gründe ...

- Wenn ich eine Bibliothek entwickle, und dabei eine Menge erzeuge, weiß ich nicht unbedingt, wie viele Elemente davon der Benutzer braucht.
- for-Schleifen laufen "gleichmäßiger jedes Element wird zum Zeitpunkt der Benutzung erzeugt.
- Speicherverbrauch ist minimal - nur ein Element muss im Speicher gehalten werden.
- Prinzip ist einfach: Eine Funktion erzeugt einen Rückgabewert, und die Position im Code wird gespeichert. Beim nächsten Aufruf läuft die Funktion an dieser Stelle weiter.

Iteratoren/Generatoren (Beispiel)

Wir berechnen die Fibonacci-Folge mit einem Generator.

```
def fibonacci():
    a = 1
    b = 1
    while True:
        c = a + b
        a = b
        b = c
        yield c   # Hier wird angehalten!

a = fibonacci()
a.next()          # 2
a.next()          # 3
a.next()          # 5
a.next()          # 8

for f in fibonacci():
    print(f)
```

Alles ist zuweisbar

Auch eine Funktion ist ein zuweisbares Öbjekt", das wir beliebig benennen, zurückgeben, ... können.

```
def give_me_spam():
    def spam():
        print("I don't like spam!")

    return spam

food = spam     # Hier bekommen wir die Fnuktion spam zurückgegeben und speichern sie unter dem Namen food!
food()          # Und dieser Aufruf ist identisch zum Aufruf spam() in give_me_spam()
```

Dynamische Argumentlisten und Keyword-Argumente

Eine Funktion kann eine beliebige, undefinierte Anzahl von Argumenten annehmen.

```
def spam(*args):
    print(args)

spam(1, 2, 3)      # [1, 2, 3]
```

Außerdem kann eine Funktion beliebige Argumente mit Namen annehmen.

```
def eggs(**kwargs):
    print(kwargs)

eggs(good="bacon", bad="spam")      # {"good": "bacon", "bad↩
    ": "spam"}
```

Dekoratoren

Dekoratoren sind eine einfache Möglichkeit, Funktionen zu manipulieren, ohne ihren Code zu verändern. Man kann sie aber auch kreativ missbrauchen ...

- Dekoratoren sind Funktionen, die als Argument die zu dekorierende Funktion übergeben bekommen und dann eine andere Funktion zurückgeben.
- Die zu dekorierende Funktion wird dann durch die zurückgegebene Funktion ersetzt.
- Man kann auch dieselbe Funktion unverändert zurückgeben ... Sinn siehe später ;-).

Dekoratoren

Dekoratoren sind eine einfache Möglichkeit, Funktionen zu manipulieren, ohne ihren Code zu verändern. Man kann sie aber auch kreativ missbrauchen ...

- Dekoratoren sind Funktionen, die als Argument die zu dekorierende Funktion übergeben bekommen und dann eine andere Funktion zurückgeben.
- Die zu dekorierende Funktion wird dann durch die zurückgegebene Funktion ersetzt.
- Man kann auch dieselbe Funktion unverändert zurückgeben ... Sinn siehe später ;-).

Dekoratoren

Dekoratoren sind eine einfache Möglichkeit, Funktionen zu manipulieren, ohne ihren Code zu verändern. Man kann sie aber auch kreativ missbrauchen ...

- Dekoratoren sind Funktionen, die als Argument die zu dekorierende Funktion übergeben bekommen und dann eine andere Funktion zurückgeben.
- Die zu dekorierende Funktion wird dann durch die zurückgegebene Funktion ersetzt.
- Man kann auch dieselbe Funktion unverändert zurückgeben ... Sinn siehe später ;-).

Dekoratoren (Beispiel)

Wir schreiben einen Dekorator, der den Rückgabewert der dekorierten Funktion verdoppelt.

```
def double_me(orig):
    def doubled(*args, **kwargs):
        return 2 * orig(*args, **kwargs)
    return doubled

@double_me      # Ist dasselbe wie ein nachgestelltes sum =↩
    double_me(sum)!
def sum(a, b):
    return a + b

sum(2, 3)      # 10
```

Missbrauch von Dekoratoren

Dekoratoren kann man teilweise kreativ "missbrauchen".

- Einfaches Callback/Plugin-System
  - Dekorator verwaltet eine Liste aller jemals übergebenen Funktionen
  - Dekorator gibt die Funktion unverändert zurück
    - Dadurch erst einmal kein Effekt für die dekorierte Funktion ...
    - ... aber alle dekorierten Funktionen sind gespeichert ("registriert").
  - Mit den gespeicherten Funktionen kann man dann gesammelt etwas tun

Missbrauch von Dekoratoren

Dekoratoren kann man teilweise kreativ "missbrauchen".

- Einfaches Callback/Plugin-System
  - Dekorator verwaltet eine Liste aller jemals übergebenen Funktionen
  - Dekorator gibt die Funktion unverändert zurück
    - Dadurch erst einmal kein Effekt für die dekorierte Funktion ...
    - ... aber alle dekorierten Funktionen sind gespeichert ("registriert").
  - Mit den gespeicherten Funktionen kann man dann gesammelt etwas tun

Missbrauch von Dekoratoren

Dekoratoren kann man teilweise kreativ "missbrauchen".

- Einfaches Callback/Plugin-System
  - Dekorator verwaltet eine Liste aller jemals übergebenen Funktionen
  - Dekorator gibt die Funktion unverändert zurück
    - Dadurch erst einmal kein Effekt für die dekorierte Funktion ...
    - ... aber alle dekorierten Funktionen sind gespeichert ("registriert").
  - Mit den gespeicherten Funktionen kann man dann gesammelt etwas tun

Missbrauch von Dekoratoren

Dekoratoren kann man teilweise kreativ "missbrauchen".

- Einfaches Callback/Plugin-System
  - Dekorator verwaltet eine Liste aller jemals übergebenen Funktionen
  - Dekorator gibt die Funktion unverändert zurück
    - Dadurch erst einmal kein Effekt für die dekorierte Funktion ...
    - ... aber alle dekorierten Funktionen sind gespeichert ("registriert").
  - Mit den gespeicherten Funktionen kann man dann gesammelt etwas tun

Missbrauch von Dekoratoren

Dekoratoren kann man teilweise kreativ "missbrauchen".

- Einfaches Callback/Plugin-System
  - Dekorator verwaltet eine Liste aller jemals übergebenen Funktionen
  - Dekorator gibt die Funktion unverändert zurück
    - Dadurch erst einmal kein Effekt für die dekorierte Funktion ...
    - ... aber alle dekorierten Funktionen sind gespeichert ("registriert").
  - Mit den gespeicherten Funktionen kann man dann gesammelt etwas tun

Missbrauch von Dekoratoren

Dekoratoren kann man teilweise kreativ "missbrauchen".

- Einfaches Callback/Plugin-System
  - Dekorator verwaltet eine Liste aller jemals übergebenen Funktionen
  - Dekorator gibt die Funktion unverändert zurück
    - Dadurch erst einmal kein Effekt für die dekorierte Funktion ...
    - ... aber alle dekorierten Funktionen sind gespeichert ("registriert").
  - Mit den gespeicherten Funktionen kann man dann gesammelt etwas tun

Missbrauch von Dekoratoren (Beispiel)

```
callbacks = []
def callback(func):
    callbacks.append(func)   # Funktion speichern
    return func              # Unverändert zurückgeben

def run_callbacks():
    for func in callbacks:
        func()

@callback
def spam():
    print("Spam is great!")

@callback
def eggs():
    print("Eggs are neat!")

run_callbacks()
```

Pythonische Hacks

Dank der erwähnten Besonderheiten der Sprache und ihrer Dynamik sind Probleme oft durch pythonische Hacks zu lösen. Ein Beispiel, um die Umbenennung der raw_input-Funktion zu input in Python 3 zu reparieren:

```
# Coole Kids schreiben Code für Python 2 und 3 ;-)!
try:
    raw_input           # Das wirft einen Fehler in Python ↩
        3
except:                 # Den Fehler fangen wir ab.
    raw_input = input   # Und zum Glück können wir ja alles↩
         zuweisen!

mein_name = raw_input("Name: ")
```

Die nächsten wichtigen pythonischen Konzepte

Wenn das Interesse an der Sprache gewachsen ist, sollten diese Konzepte unbedingt erforscht werden:

- Module importieren, eigene Module schreiben, Namespaces
- Vererbung
- __repr__(self), __str__(self)
- Exception-Handling (try.. except..
- Exceptions erzeugen und werfen
- with-Statement

Meine Lieblingsbibliotheken

Kurze Liste meiner Lieblingsmodule für bestimmte Zwecke (kann sich täglich ändern ;-)):

- Webanwendungen: Flask (`http://flask.pocoo.org`)
- Spiele: PyGame (`http://www.pygame.org`)

Literaturempfehlungen

Es gibt nicht viele gute Bücher für Einsteiger ...

- Für große und kleine Kinder:
  - Hello World! - Programmieren für Kids und andere Anfänger (Warren D. Sande und Carter Sande), ISBM: 978-3446421448
  - Python kinderleicht!: Einfach programmieren lernen - nicht nur für Kids (Jason Briggs), ISBN: 978-3864900228
  - Dr. Nuts und der teuflische Nussinator: Ein Jump'n'Run-Spiel in 14 Tagen (Eike Tim Jesinghaus, Dominik George), in Arbeit ;-)
- `http://docs.python.org`
- Monty Python: Spam - `https://www.youtube.com/watch?v=anwy2MPT5RE`

Credits / Attribution

- Für meine Begeisterung und Vertiefung der Programmiersprache Python sowie gelegentliche Erziehungshiebe:
  - Eike Tim "Creeparoo" Jesinghaus (12)
- Antigravity-Comic (Folie 8):
  - Randall Munroe, `http://xkcd.com/353/` (CC-BY-NC 2.5)

Vielen Dank für eure Teilnahme!
Für weitere Fragen, Anregungen, etc.:
Website: `http://www.dominik-george.de`
E-Mail / Jabber: nik@naturalnet.de