
Flask Documentation

Release 1.0

Christoph Piechula, Christopher Pahl

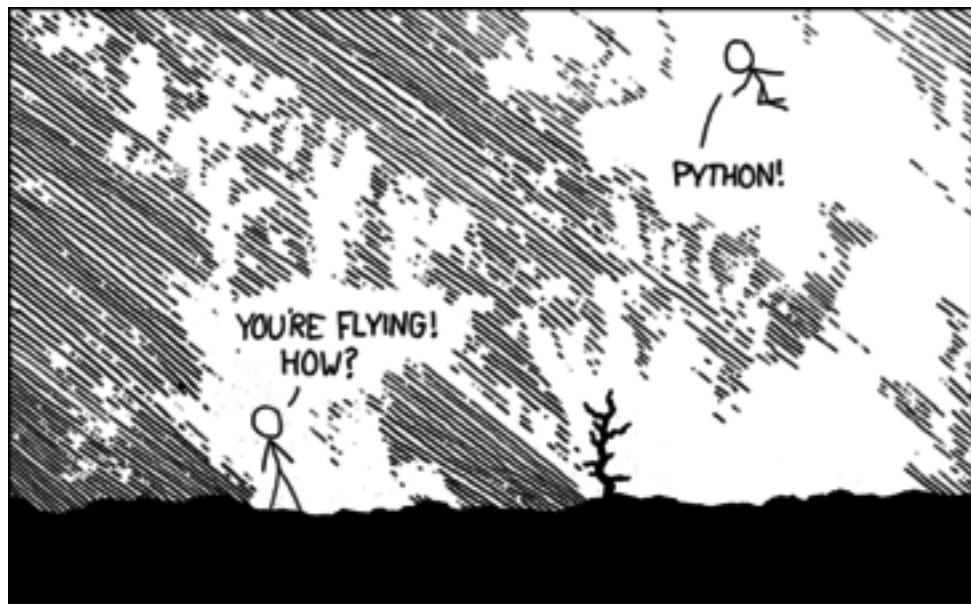
December 10, 2012

CONTENTS

1 Ein Hochgeschwindigkeitskurs durch Python	3
1.1 Vorwort	3
1.2 Wozu überhaupt?	4
1.3 History!	4
1.4 Wiederverwendung	5
1.5 Die Shell (REPL)	5
1.6 Datentypen?	5
1.7 Literale	6
1.8 Listen	6
1.9 Tupel	7
1.10 Dictionaries	7
1.11 Bedingungen	7
1.12 Schleifen	8
1.13 Funktionen #1	8
1.14 Funktionen #2	8
1.15 Exceptions	9
1.16 Hilfe? (...Don't Panic!)	9
1.17 Klassen #1	10
1.18 Klassen #2	10
1.19 Duck Typing	10
1.20 Module #1	11
1.21 Module #2	11
1.22 Übungen	12
1.23 Diese Folie soll Spicken verhindern	12
1.24 EinMalEins - Lösung	13
1.25 Diese Folie auch	14
1.26 SortedList - Lösung	14
1.27 Operatorüberladung	15
1.28 λ!	15
1.29 Multiple Inheritance	16
1.30 Higher Order Functions (aka Closures)	16
1.31 Dekoratoren	16
1.32 List Comprehensions	17
1.33 Generatoren	17
1.34 with - Context Management	17
1.35 Die Philosophie	18
1.36 It's short!	18
1.37 Unit-Testing	18
1.38 Python2 vs. Python3	19

1.39	Interpreter / Compiler	19
1.40	Fragen?	20
2	Eine Einführung in	21
2.1	Um was geht's?	21
2.2	Architektur	22
2.3	Warum nicht Django?	22
2.4	Weitere Features	23
2.5	Hello World!	23
2.6	Und nun... Schlangen!	24
2.7	View Functions	24
2.8	Routing & Troubleshooting #1	25
2.9	Routing & Troubleshooting #2	25
2.10	Routing & Troubleshooting #3	25
2.11	Templates & How to render them	26
2.12	Templates #2	26
2.13	Templates #3	26
2.14	Request Object	27
2.15	Session Object #1	27
2.16	Session Object #2	28
2.17	URL Parameter	28
2.18	Debugging Inside #1	28
2.19	Debugging Inside #2	29
2.20	Server Inside *	29
2.21	Deployment Options	29
2.22	Ausblick	30
2.23	moosr - Ein Beispielprojekt	30
2.24	Practice!	31

EIN HOCHGESCHWINDIGKEITSKURS DURCH PYTHON



1.1 Vorwort

- Eine Stunde ist wenig Zeit um eine ganze, umfangreiche Sprache zu vermitteln, geschweige denn mehr als Grundlagen zu vermitteln.
- Deshalb soll der Vortrag eher dazu dienen euren Appetit zu wecken, und euch ein gewisses Gefühl für die Sprache geben. Wenn jemand später Lust verspürt mit Python zu spielen haben wir unser Ziel erreicht.
- Hier sollen nur Grundbegriffe vermittelt und ein Ausblick gegeben werden. Speziell auch ein paar von Python's coolen Features.
- Falls jemand Fragen hat: einfach reinbrüllen.
- Wer dazu Lust verspürt kann auch die Beispiele selbst ausprobieren, siehe nächste Folie.
- Kurzbeschreibung von Python:

Python is a general-purpose, interpreted, (optionally-), garbage-collected, typeless high-level programming language whose design philosophy emphasizes code readability.

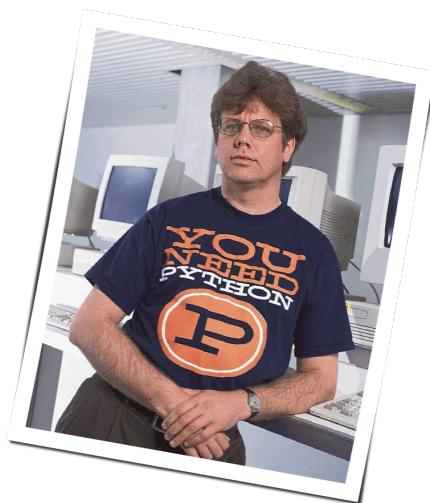
- **Paradigmen:** Object-Oriented, Imperative, Funktional, Prozedural, Reflektiv

1.2 Wozu überhaupt?

- Einsatzzwecke:
 - ... Wissenschaftlichen Bereich (numpy, SciPy bei NASA)
 - ... Webprogrammierung (Flask, Zope, Django)
 - ... Systemprogrammierung (**Admintasks, Networking**)
 - ... Prototyping (**3-10x** Entwicklungsgeschwindigkeit)
 - ... als „Glue“ (Compiler für C++, Java verfügbar)
 - ... integrierte „Scriptssprache“ (Blender3D, viele Spiele)
 - ... Educationbereich (Primäre Sprache auf dem Raspberry)
 - ... öfter als man denkt.
- Andere Gründe:
 - Einfach erlernbar (Ehrlich!)
 - Meist schnell genug.
 - Plattformunabhängig.
 - ...

1.3 History!

- Entstanden 1991 (älter als Java, mit 1995)



- (Ursprüngliche) Designziele:
 - Als Ergänzung zu C++, Ersetzung von Shellscripts.
 - Code der sich wie einfaches Englisch liest.
 - Kurze Entwicklungszeiträume.

1.4 Wiederverwendung

- Guido's Time Machine
 - Batteries included
- sehr wenig Boilerplate Code
- PEP8

Vermeidet:



1.5 Die Shell (REPL)

- Python kann interaktiv ausprobiert werden.
- Mitgeliefert gibt es die `python` Shell
- Auch „REPL“ (Read-Eval-Print-Loop) genannt:

```
$ python

>>> a = 1 + 1
2
>>> print('1 + 1 =', a)
1 + 1 = 2
```

- Leider: Etwas unbequem, da keine Syntaxhighlighting / Autocomplete.
- Wir empfehlen/verwenden daher **bpython**.
- Unheimlich praktisch um sich mit der Sprache vertraut zu machen.

1.6 Datentypen?

Gibts nicht. Waren zu teuer.

- Python ist eine dynamisch typisierte Sprache.
- Man verlässt sich nicht auf den Typen einer Variable, sondern auf dessen Verhalten.
- Bietet eine Klasse `Liste` die gleichen Funktionen wie `Array` so lassen sie sich gleich verwenden.
 - Ganz ohne Interfaces wie `Iterable` oder `List`.
- Dieses Prinzip nennt sich Duck Typing.
- Zur Laufzeit lassen sich `Array` und `Liste` mittels `type()` auseinanderhalten.
 - Aber für gewöhnlich braucht man das nicht, da sich viele Klassen ähnlich verhalten:

```
>>> a, b = [1, 2, 3], (4, 5, 5)
>>> print(a.count(1), b.count(5))
1 2
```

1.7 Literale

Strings (immer Immutable):

```
>>> 'Hello World' # Ein Stringliteral
>>> "Hello World" # Dasselbe (Anders als in Ruby!)
>>> '''Geht auch # Das ist kein Kommentar.
... über mehrere
... Zeilen'''
```

Numbers (auch Immutable):

```
>>> a, b = 42, 42.21 # Ganzzahlen, Floats + Zuweisung.
>>> a, b = 0o777, 0xDEADBEEF # Hex/Oktal-zahlen
```

Zuweisungen:

```
>>> a, b = b, a      # Swap a, b
```

1.8 Listen

Listen werden wie Arrays in anderen Sprachen genutzt:

```
>>> pointless_list = [7, 'Apple', []]
```

Zugriff auf Elemente und „Slicing“ (wie `subList()`):

Im Allgemeinen: `liste[start{:end{:step}}]`

```
>>> pointless_list[0] = 42
>>> pointless_list[0]
42
>>> pointless_list[0:2]
[42, 'Apple']
>>> pointless_list[:-1]
[42, 'Apple']
>>> pointless_list[0:3:2]
[42, []]
```

(In Etwa) Java-Äquivalent: `java.util.ArrayList`

1.9 Tupel

- Tupel sind wie Listen, nur mit runden Klammern + Immutable

```
>>> pointless_tuple = (1, 2, 3)
>>> pointless_tuple = 1, 2, 3
>>> pointless_tuple[0] = 2 # Nope, TypeError.
```

- Tupel werden immer dann verwendet wenn man Dinge in einer bestimmten Reihenfolge packen muss.

 - Beispielsweise einen Vertex mit 3 Koordinaten: `(1, 0, 42)`

- Ein Tupel mit einem Element wird mit folgender Syntax deklariert:

```
>>> one_elem_tuple = (1,) # Sieht seltsam aus
>>> one_elem_tuple = tuple([1]) # Alternative
```

- Tuple Zuweisung (**wichtig!**):

```
>>> a, b = (42, 21)
```

1.10 Dictionaries

```
>>> pointless_dict = {
...     'Apple': ['juicy', 'red', 'healthy'],
...     'Orange': ['juicy', 'not red'],
...     'Watermelon': 42
}
```

```
>>> pointless_dict['Apple']
['juicy', 'red', 'healthy']
>>> pointless_dict['Peach']
<KeyError>
>>> pointless_dict['Peach'] = 'A hairy fruit'
>>> pointless_dict['Peach']
'A hairy fruit'
>>> del pointless_dict['Peach']
```

- Java-Äquivalent: `java.util.HashMap`
- Dictionaries werden in Python ständig eingesetzt.

1.11 Bedingungen

```
# Beachte Einrückung statt {}!
if 'cow' == 'dog':
    pass
elif 1 == 2:
    pass
else: pass
```

Bedingte Zuweisung:

```
>>> a = 21 if not truth else 42 # a = (truth) ? 21:42;
42
```

Unwahrheitswerte (unvollständig):

```
0, 0.0, False, None, '', [], {}, set()
```

Sonst gilt für gewöhnlich alles als True.

1.12 Schleifen

```
# 1,3,5,7,9                      # Ungeraden Zahlen von 1-10
for i in range(1,10,2):#
    print(i)                      # 1  = Start (optional)
                                    # 10 = End
                                    # 2  = Step (optional)

for idx, chr in enumerate('Hello'):
    print(idx, chr)              # In C-Ähnlichen Sprachen:
    if chr == 'l':               # char * s = "Hello"
        break                     # for(int i=1; i<10; i+=2) {
    else:                        #     printf("s[%d]=%d\n",s[i],i)
        continue                  # }
```

- → `range()` und `enumerate()` geben Iteratoren zurück.

```
while metal is True:      # while(<expression>) {
    do_something          #     do_something;
                           # }
```

1.13 Funktionen #1

```
>>> # Definition
>>> def hello():
...     print('Hello')
...
>>> # Redefinition
>>> def hello():
...     return 'Hello'
...
>>> print(hello())

>>> # Parametrisierte Funktionen
>>> def doublegreet(message):
...     return message * 2
...
>>> print(doublegreet('Hello'))
HelloHello
>>> print(doublegreet(message='Hello'))
HelloHello
```

1.14 Funktionen #2

- `*args` - Variable Argumentlisten

```
def print_bracketed(*args):
    for i in args: print('[%d]' % i)

print_bracketed(1, 2, 3) # Prints: [1] [2] [3]
```

- `**kwargs` - Variable KeyWord Parameter

```
def print_params(**kwargs):
    for key, value in kwargs.items():
        print(key, '=>', value)

print_params(name='Paul', job='Hauskatze')
```

- Alle möglichen Mischformen möglich.
- `kwargs` muss als letztes stehen.
- `args` mindestens als vorletztes.

1.15 Exceptions

Fangen:

```
try:
    a = b
except NameError:
    print('Du hast vergessen b zu definieren.')
finally:
    print('Wird immer ausgeführt.')
```

Werfen:

```
raise AttributeError('Keine Kuscheldecke gefunden.')
```

Eigene Wurfgeschosse erstellen:

```
class OnSuccessError(Exception):
    pass
```

1.16 Hilfe? (...Don't Panic!)

- Python setzt auf Selbstdokumentation, sprich auslesbare Kommentare:

```
def make_money(papier, tinte, schein):
    """
    Erzeugt Geld aus Papier und Tinte.

    :papier: Eine Instanz der Klasse Papier
    :tinte: Die Helligkeit der Tinte von 0-100.
    :returns: Eine neue Schein Instanz
    """
    return Schein(papier, tinte)
print(make_money.__doc__)
```

- RestructuredText ist dabei das gängige Dokumentationsformat.
 - Diese Folien sind zum Beispiel darin verfasst.
- Die offizielle Referenz/Tutorial: <http://python.org/doc/>
- Auch nützlich: die `dir()` Funktion, zum Auflisten von Membern.

1.17 Klassen #1

Überraschung: Es gibt keine `private` / `protected` Variablen:

```
class Mom:
    def __init__(self, name):
        self.name = name

    def call_me_please(self):
        print('<Mom>:', self.name)

class Son(Mom):
    def __init__(self, name):
        Mom.__init__(self, name + "'s Son")

    def call_me_please(self):
        Mom.call_me_please(self)
        print('<Son>:', self.name)

son = Son('Peter')
son.call_me_please() # same as: Son.call_me_please(son)
```

1.18 Klassen #2

Properties machen das Ersetzen von Attributen mit Gettern/Settern einfach, **ohne** dabei die Schnittstelle seiner Klasse zu ändern:

```
class Coffee:
    def __init__(self, vol=1):
        self._vol = vol

    def set_vol(self, new_vol): self._vol = new_vol * 3

    def get_vol(self): return self._vol

    vol = property(get_vol, set_vol)

>>> mocka = Coffee()
>>> mocka.vol = 3      # Setter Aufruf
>>> print(mocka.vol)  # Getter Aufruf
9
```

1.19 Duck Typing

„When I see a bird that walks like a duck and swims like a duck and quacks like a duck, I call that bird a duck.“
– James Whitcomb Riley

```
class Bird(object):
    def peep(self): print('Peep?')

class Duck(object):
    def quak(self): print('Quak!')

for duck in [Duck(), Bird(), dict()]:
    if hasattr(duck, 'quak'):
        duck.quak()
    else:
        print('Sieht nicht aus wie ne Ente:', duck)
```

1.20 Module #1

Beispiel-Layout:

```
app                  | Import Beispiel:
|
|-- effects          |
|   |-- __init__.py  | # In app/logic/run.py
|   |-- sinus.py     | >>> import app.sound.decode
|   |-- warp.py      | ...
|
|-- logic             | # Use the Force:
|   |-- __init__.py  | >>> app.sound.decode.some_func()
|   |-- run.py        |
```

```
|           |
-- __main__.py    | # Alternativ:
-- __init__.py     | >>> import app.sound.decode as d
|                 | >>> d.some_func()
-- sound          |
  -- decode.py   |
  -- __init__.py  |

```

1.21 Module #2

Anderer Formen von `import`:

```
>>> from app.sound.decode import some_func, some_var
>>> some_func(some_var)
```

Unqualifizierter Import (**Don't do it**):

```
>>> # Bitte nicht tun da Namenskonflikte möglich:
>>> from app.sound.decode import *
>>> some_func(some_var)
```

Lange Modulnamen können abkürzt werden:

```
>>> import app.sound.decode as asd
>>> asd.some_func(asd.some_var)
```

1.22 Übungen

EinMalEins: Schreibe ein Programm dass das 1x1 ausgibt (Formatierung egal):

```
1x1 = 1, 1x2 = 2, ...
2x1 = 2, 2x2 = 4, ...
```

SortedList:

Implementiere eine Collection die sich wie eine Liste verhält, nur dass `append()` Elemente sortiert hinzufügt.

- Die Oberklasse sollte `list` sein.
- Methoden der Oberklasse können mit `list.obermethode(self, argumente)` ange- sprochen werden.
- Nützliche Funktionen: `list.insert(idx, obj)`, `list.sort()`, `enumerate(iterable)`.

1.23 Diese Folie soll Spicken verhindern



Siehe auch: <http://codingbat.com/python> wer mehr Üben will

1.24 EinMalEins - Lösung

Die einfache, klare Lösung:

```
>>> for x in range(1,11):
...     for y in range(1,11):
...         print('%dx%d = %d' % (x, y, x * y))
```

Die Elegante und das Biest:

```
>>> from itertools import product
>>> ten = range(1,11)
>>> for x,y in product(ten, ten):
...     print('%dx%d = %d' % (x, y, x * y))
```

```
>>> from itertools import product
>>> ten = range(1,11)
>>> ['%dx%d=%d' % (x,y,x*y) for x,y in product(ten,ten)]
```

1.25 Diese Folie auch



Für harte Männer: <http://learnpythononthehardway.org/book/> (Empfehlung!)

1.26 SortedList - Lösung

```
class SortedList(list):
    def __init__(self, iterable=[]):
        iterable.sort()
        list.__init__(self, iterable)

    def append(self, obj):
        'Append obj sorted to list'
        for i, elem in enumerate(self):
            if elem >= obj:
                self.insert(i, obj)
                break
        else:
            list.append(self, obj)

sl = SortedList([3,4,8,9])
sl.append(5)
sl.append(0)
sl.append(42)
print(sl)
```

1.27 Operatorüberladung

Auf Wunsch von Thomas:

```
class SimpleVec(object):
    def __init__(self, *coord):
        self._coord = coord

    def __iter__(self):
        return iter(self._coord)

    def __add__(self, rhs):
        self._coord = tuple(map(lambda x, y: x + y,
                               self._coord, rhs))
        return self

    def __repr__(self):
        return repr(self._coord)

__contains__, __eq__, __getitem__, __len__, __getattr__
```

1.28 λ !

Lambdas sind auch nur Funktionen:

```
fac = lambda x: 1 if x == 0 else x * fac(x-1)
fac(23) # 25852016738884976640000
```

Vergleiche:

```
public long fac(long n) {
    if (n == 0) return 1;
    else         return fac(n - 1) * n;
}

fac(23); // 8128291617894825984 huh?
```

Python switcht bei Integer Overflows intern auf eine BigInteger Repräsentation. Das ist zwar weniger performant als good ol' Java, aber einfacher bequemer.

1.29 Multiple Inheritance

Auf Wunsch von Herrn Schaible:

```
class Base(A, B, C):
    pass
```

Methodenauflösung nach ...

- ... Depth First.
- ... links nach rechts.
- ... Rekursiv.
- ... immer eine Instanz.
- ... Erst A rekursiv, dann B rekursiv, dann C.

Siehe auch Tafelbild.

1.30 Higher Order Functions (aka Closures)

- In Python können Funktionen Funktionen zurückgeben.
- Da Funktionen auch nur Objekte sind können "spezialisierte" Funktionen auch zur Laufzeit instanziert werden.

Beispiel: Eine Funktion die einen speziellen Greeter zurückgibt.

```
def greeting_generator(name):
    def greeter():
        print('Hello', name + '!')
    return greeter

>>> f = greeting_generator('Python')
>>> f()
Hello Python!
```

Eine Art Factory Pattern für Funktionen.

1.31 Dekoratoren

Funktionen/Klassen können “dekoriert” werden, *ähnlich* dem aus Java bekannten Decorator-Pattern. Nur weitaus einfacher zu nutzen:

```
def bold(fn):
    def wrapped():
        return '<b>' + fn() + '</b>'
    return wrapped

def italic(fn):
    def wrapped():
        return '<i>' + fn() + '</i>'
    return wrapped

@bold
@italic
def hello():
    return 'Hello World'

>>> hello() # Im Hintergrund: bold(italic(hello))()
'<b><i>Hello World</i></b>'
```

1.32 List Comprehensions

Wie kann man alle y in einem Intervall für eine bestimmte Funktion berechnen?

```
[f(x) for x in interval] # f(x) für x in interval
```

Beispiel: Die Funktion $2^{**}x$ im Definitionsbereich 0-9:

```
>>> [2**x for x in range(10)]
[1, 2, 4, 8, 16, 32, 64, 128, 256, 512]
```

Oft nutzt man Comprehensions auch für das Filtern von Listen.

Beispiel: Wie oben, aber nur alle ungeraden Exponenten, und als String formattiert:

```
>>> ['f(%d)=%d' % (x, 2**x) for x in range(10) if x%2]
['f(1)=2', 'f(3)=8', 'f(5)=32', 'f(7)=128', 'f(9)=512']
```

1.33 Generatoren

`yield` macht eine Funktion zum Generator:

```
# Ein erbärmlicher Random Generator
def random42(max_num):
    for i in range(max_num):
        yield 42 ** i

# Printe 10 „Zufallszahlen“
for i in random42(10):
    print(i)
```

Generator Expressions nutzen die von LH bekannten Syntax, erzeugen die Werte aber erst beim Iterieren:

```
# Zeige alle Quadratzahlen mit ungerader Wurzel
odd_quads = (x**2 for x in range(10) if x % 2)
for i in odd_quads:
    print(i)
```

1.34 with - Context Management

```
try:
    f = open('file.txt', 'w')
    f.write('hello world')
finally:
    f.close()
```

Python way (Strichwort **RAII** in C++):

```
with open('file.txt', 'w') as f:
    f.write('hello world')
```

Es lassen sich eigene Funktionen/Klassen definieren die das with Statement nutzen. Als Beispiel könnte man eine Mutex-Klasse implementieren:

```
with locked(some_mutex):
    do_something_while_locked()
```

1.35 Die Philosophie

Pragma statt Dogma! Es gibt keinen „goldenen Hammer“.

Zen of Python: In der Python-Shell abrufbar als: `import this`

Explizit ist besser als Implizit Siehe beispielsweise explitzes `self` statt implizites `this`.

Batteries included Große Standardbibliothek mit vielen Funktionen.

Man liest Code öfters als man ihn schreibt. Und man sollte ihn nicht widerwillig lesen müssen.

Programmieren sollte Spass machen. Gegen Compiler/Sprache/Konfiguration kämpfen macht wenig Spaß.

1.36 It's short!

```
#!/usr/bin/env python
# encoding: utf-8
import sys, pprint, os, hashlib

def find_dups(path):
    hashes, dups = {}, {}
    for path, dirs, files in os.walk(path):
        abspathes = (os.path.join(path, n) for n in files)
        for fpath in filter(os.path.isfile, abspathes):
            with open(fpath, 'r') as f:
                md5 = hashlib.md5(f.read()).hexdigest()
```

```

if hashes.setdefault(md5, fpath) is not fpath:
    at = dups.setdefault(md5, [hashes[md5]])
    at.append(fpath)
return dups

if __name__ == '__main__':
    pprint.pprint(find_dups(sys.argv[1]))

```

1.37 Unit-Testing

Das Testframework ist mit dem Modul `unittest` in die Sprache eingebaut:

```

import unittest

def greeter(name): return 'Hello ' + name + '!'

class TestGreeter(unittest.TestCase):
    def setUp(self):
        self.test_name = 'Workshop'

    def test_greeter(self):
        self.assertEqual(greeter(self._test_name),
                        'Hello ' + self.test_name + '!')

    def tearDown(self):
        self.test_name = ''

if __name__ == '__main__':
    unittest.main()

```

1.38 Python2 vs. Python3

Auf Python3 Seite:

- Einfache Unicodeunterstützung.
- Alles leitet von `object` ab.
- Syntaxänderungen und Änderungen an der C-API.
- Leider **inkompatibel** zu Python2.
 - Viele Features aber backported.
- Manche externe Libraries **noch** Python2 basiert (Flask)!
 - Python2 wird allerdings noch lange supported.
- Oft noch Python 2 per Default installiert (Debian, Mac OS X).
- Für die Übungen / Flask wird Python2 verwendet!
- (Fast) Alle Beispiele liefen in beiden Versionen.

1.39 Interpreter / Compiler

Es gibt eine Reihe verschiedener Intepreter / Compiler für Python:

- CPython - Der *Referenz* Interpreter.
 - Jython - Ein ByteCode Compiler für die JVM. *
 - IronPython - Die .Net Variante von Jython.
 - Cython - Übersetzt Python zu C-Code. *
 - PyPy - Ein Interpreter/JIT Compiler in Python.
 - Stackless Python - Interpreter; Verbesserter Threadingsupport.
-

* Es handelt sich um Spracherweiterungen.

1.40 Fragen?

Beispielsfrage #1: Mit was wurde die Präsentation gemacht?

Blut, Spucke, Python und HTML.

Genau genommen mit `python-impress` gerendert: <http://www.github.com/gawell/impress>

Beispielsfrage #2: Machen wir 5 Minuten Pause?

Ja.

Beispielfrage #3: Wo gibts die Folien und den Rest des Workshops?

Folien/Übungen sind verfügbar unter: <http://www.github.com/studentkittens/flascat>

EINE EINFÜHRUNG IN



Flask

web development,
one drop at a time

<http://flask.pocoo.org/>

Präsentation angelehnt an den [Flask Quickstart](#)

2.1 Um was geht's?

- Flask ist ein **Microwebframework**.
 - **Fokus:** Erweiterbarkeit + gute Dokumentation.
 - Abhängigkeiten:
 - [Werkzeug](#): WSGI Middleware.
 - [Jinja2](#): Eine Template Engine.
 - BSD Lizenz → Kommerzielle Projekte möglich.
-

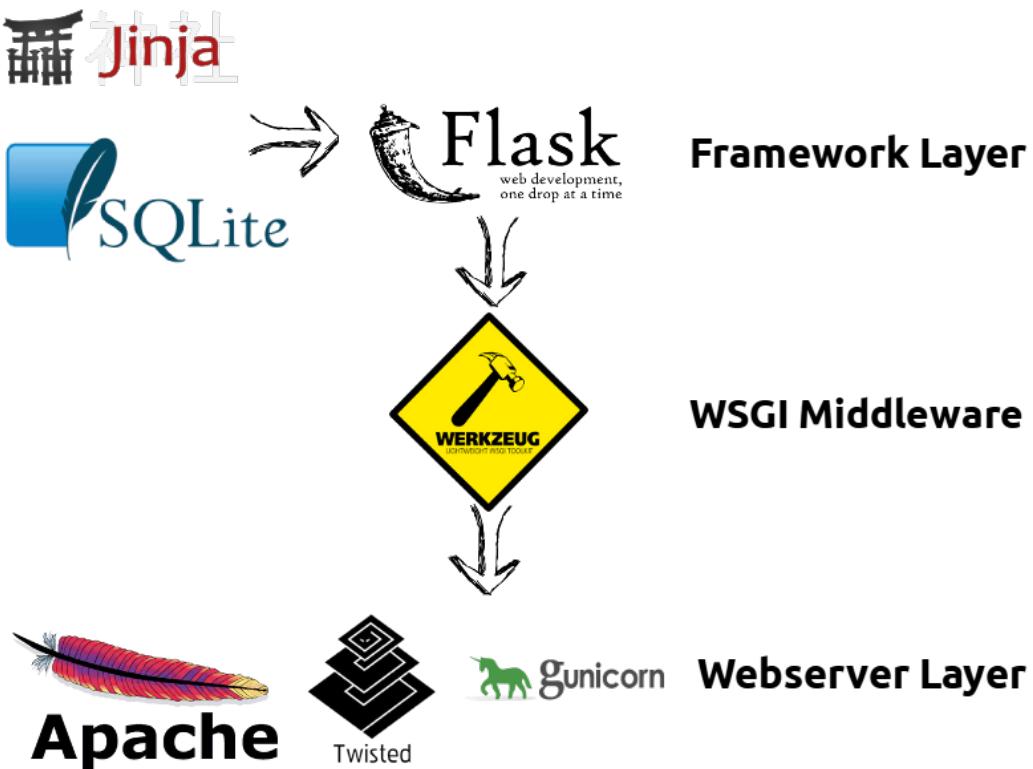
Warum denn “micro”?

*Flask aims to keep the core simple but extensible.
Flask won't make many decisions for you, such as what database to use.*

Anderer Grund:

800 LOC Code, 1500 LOC Testcases, 200 A4 Seiten Dokumentation.

2.2 Architektur



2.3 Warum nicht Django?

Im Vergleich zu Django:

- Beschränkung auf Kernfunktionalität.
- don't reinvent the wheel.
- Modular, Erweiterung durch Plugins.
- zB.: Datenbank und Templateengine austauschbar.
- Sehr viel kleiner / handlicher.
- Skalierbarkeit.

Anmerkung:

- Flask läuft momentan nur mit Python 2.x.
- Python 3.x Port ist in Arbeit.
- Django läuft bereits auf Python 3.x.

- Admin Funktionalität.

2.4 Weitere Features

- Handhabung von Authentifizierung, Cookies, Sessions
- Konfigurierbares Caching
- Internationalisierung
- Abstraktionsschicht für Datenbanken, die dynamisch SQL erzeugt
 - ORM (Object-Relational-Mapper) als Plugin.
- Kompatibilität zu vielen Datenbankmanagementsystemen.
- Kein Zwang bestimmte Software zu nutzen, → Wiederverwendung anderer Frameworks!

2.5 Hello World!

Die folgende Anwendung wird auf localhost:5000 horchen und bei einem GET einem unformatierten **Hallo Welt** ausgeben.

```
# Importiere die Flask Libraries,
# und instanziere eine Flask-Anwendung.
from flask import Flask
app = Flask('MyFirstFlaskApp')

# hello() soll für ein Zugriff auf
# die root-url aufgerufen werden.
@app.route("/")
def hello():
    return "Hallo Welt"

# Falls das Skript direkt ausgeführt wird,
# so lasse die Anwendung laufen.
if __name__ == "__main__":
    app.run(debug=True)
```

2.6 Und nun... Schlangen!



Ein Hochgeschwindigkeitskurs durch Python

2.7 View Functions

Gute Tiere anzeigen:

```
@app.route('/')
def show_good_ones():
    # Fake-Daten aus der Datenbank
    db = ['turtle', 'owl',     'dog',
          'kitteh', 'koala',   'moose']

    # Tue etwas mit den Daten
    good = [y for x, y in enumerate(db) if x % 2 != 0]
```

```
# Visualisiere sie (hier einfach "raw")
return str(good)

if __name__ == '__main__':
    app.run(debug=True)
```

- *View Funktionen* dienen zum Visualisieren von Daten

2.8 Routing & Troubleshooting #1

Routing:

```
def compose_hello(name):
    return '<h1><b>Hello ' + name + '!</b></h1>'

@app.route('/hello')
def hello():
    return compose_hello('Workshop'), 200
```

Gerendertes HTML im Browser (localhost:5000):

```
<h1><b>Hello Workshop</b></h1>
```

2.9 Routing & Troubleshooting #2

Redirects:

- http://www.domain.de/newest_article → <http://www.domain.de/article/month/week/day/blah.html>
 - Realisierbar mit `redirect(url)`
- ```
from flask import redirect
@app.route('/redirect_to_google')
def hello():
 return redirect('http://www.google.de')
```
- Würde bei einem GET von `localhost:5000/redirect_to_google` `www.google.de` mittels eines HTTP Redirects aufrufen.
  - Lässt man das Protokoll (`http://`) weg, so wird innerhalb der Seitengrenzen redirected, also zu `localhost:5000/www.google.de`.

## 2.10 Routing & Troubleshooting #3

### HTTP Verben:

- GET, POST, PUT, HEAD, OPTIONS

### URL Building:

- Vermeidung von hardgecodeten URLs im Programm:

```
url_for('a_name_of_a_view_function')
```

#### Statische Komponenten:

- ... werden in einem static/ folder abgelegt (CSS, Bilder).
- Templates gehen per default nach templates/.

```
url_for('static/', filename='cover.png')
```

## 2.11 Templates & How to render them

### Templates

- Mit render\_template('hello.html') wird über Jinja2 die Seite hi.html gerendert

```
@app.route('/greet/<name>')
def hello(name):
 return render_template('hi.html', you=name)

<!-- hi.html -->
<html>
 <body>
 <h1>Hello {{ you }}!</h1>
 </body>
</html></pre>
```

## 2.12 Templates #2

```
<!-- userlist.html -->
<!doctype html>
<title>Userlist</title>

 {% for user in users %}
 {% if user != 'admin' %}
 {{ user }}
 {% endif %}
 {% endfor %}

```

#### Rendern des Templates aus einer View Funktion:

```
users = ['admin', 'sam', 'phil']
return render_template('userlist.html', users=users)
```

## 2.13 Templates #3

#### parent.html:

```
<html>
 <title>Flaskr</title>
 <h1>Flaskr Headline</h1>
```

```
<body>
 { % block body %}{% endblock %}
</body>
</html>
```

**child.html:**

```
{% extends "parent.html" %}
{% block body %}
 <p>Hello I am a child</p>
{% endblock %}
```

- Nützlich zur Realisierung verschachtelter Layouts.

## 2.14 Request Object

Das **Request Object** dient u.a. dazu POST Daten auszulesen.

```
@app.route('/login', methods=['POST', 'GET'])
def login():
 if request.method == 'POST':
 return '' + request.form['text'] + ''
 else:
 return '''<form action="" method="post">
 <p><input type=text name=text></p>
 <p><input type=submit value=Do></p>
 </form>'''
```

**Anmerkung:**

- Über `request.method` (String) wird die HTTP Methode geprüft
- Über `request.form` (Dictionary) können Formulare ausgelesen werden

## 2.15 Session Object #1

**Codeblock um Login zu realisieren:**

```
from flask import Flask, session, redirect
from flask import escape, request, url_for

app = Flask(__name__)

@app.route('/')
def index():
 if 'user' in session:
 return 'You are: ' + escape(session['user'])
 return 'You are not logged in!'

@app.route('/login', methods=['GET', 'POST'])
def login():
 if request.method == 'POST':
 session['user'] = request.form['user']
 return redirect(url_for('index'))
 return render_template('login.html')
```

## 2.16 Session Object #2

**Logout:**

```
@app.route('/logout')
def logout():
 session.pop('username', None)
 return redirect(url_for('index'))
```

**Secret Key:**

```
app.secret_key = '68b329da9893e34099c7d8ad5cb9c940'
```

**HTML Formular (login.html):**

```
<form action="" method="post">
 <p><input type=text name=user></p>
 <p><input type=submit value=Login></p>
</form>
```

## 2.17 URL Parameter

- Im Web/RESTful APIs sieht man oft sowas wie:

```
http://www.lastfm.com/api/?method=artist.getSimilar&apikey=xyz
```

- → Einfacher Weg um optionale Parameter zu realisieren (wie `**kwargs`)
- Flask legt diese Parameter als Dictionary im Request Objekt ab:

```
@app.route('/api')
def api_root():
 method = request.args.get('method')
 apikey = request.args.get('apikey')

 if apikey == 'xyz':
 xml_data = make_xml_response(method, apikey)
 return Response(xml_data, mimetype='text/xml')
 else:
 return 'Access Denied', 404
```

## 2.18 Debugging Inside #1

**Live debugging flask applications**

```
app.debug = True
app.run()
```

- wird eine Flask Applikation mit `debug = True` gestartet, so wird im Browser bei Fehlern der Traceback geprintet. Dieser ist interaktiv, es können Variablen ausgelesen und Kommandos interaktiv abgesetzt werden.

- Da man willkürlich Code im Browser eintippen kann empfiehlt sich dieser Switch nicht sonderlich auf Produktivsystemen.
- Aber das war euch ja klar.

## 2.19 Debugging Inside #2

Let's demo!

```
from flask import Flask
app = Flask(__name__)

@app.route('/<name>')
def hello(name):
 answer = 42
 if name == 'lybrial':
 raise Exception('Clitical Error.')
 else:
 return "Hello {0},\nthe answer is {1}!".format(name, answer)

if __name__ == '__main__':
 app.run(debug=True)
```

## 2.20 Server Inside \*

\*kind of

- Flask startet beim Starten der Applikation einen Server der Standardmäßig auf localhost:5000 horcht.
- Server Parameter änderbar:

```
if __name__ == '__main__':
 app.run(debug=True,
 host='0.0.0.0',
 port=4242)
```

- debug aktiviert den live debugger über den Browser
- host definiert die IP-Adresse auf der gelauscht werden soll
- port definiert den Port auf dem gelauscht werden soll

## 2.21 Deployment Options

### Do it yourself - Deploying Flask

- mod\_wsgi (Apache)
- Standalone WSGI Containers (Gunicorn Python WSGI HTTP Server)
- uWSGI
- FastCGI

- CGI

### Deploying Flask on Business Enterprise Platforms

- Flask on Heroku
- Deploying WSGI on dotCloud
- Flask on Webfaction
- Google App Engine

## 2.22 Ausblick

**Also known as:** Wozu wir keine Zeit mehr hatten:

- Datenbankintegration. (Das g Objekt)
- File Uploads (`request.files`)
- Logging (`app.logger.warning(msg)`)
- Message Flashing (`flash`)
- Blueprints (verschiedene Seiten für Admin/User zB.)
- Extensions, zB:
  - Flask-Admin, Flask-OpenID, Flask-SQLAlchemy
  - Frozen-Flask, Flask-OAuth, Flask-XML-RPC ...
- Caching
- ...

Wir empfehlen auch das ausführliche Tutorial auf der Flaskseite:

<http://flask.pocoo.org/docs/tutorial/>

## 2.23 moosr - Ein Beispielprojekt



## 2.24 Practice!

- Bitte die VM starten.
- Auf dem Desktop findet ihr eine `Excercise.pdf`.
- Im Homedirectory findet ihr unter `flascat/practice` die Dateien.
- Unter Chromium ist `localhost:5000` die Startseite.
- Der Vortag ist auch auf dem Desktop verlinkt.
- Zusätzlich findet ihr den Flask Userguide dort.

```
$ cd ~/flascat/practice
$ gedit app.py # Übungsdatei öffnen
$ python app.py # Flask Server starten
$ python test.py # Eure Bemühungen testen
```

---

### Zeitaufteilung:

#	Min.	Inhalt		#	Min.	Inhalt
0	25	Python Basic		0	15	Flask Extended
25	15	Übungen		15	30	Vorstellung von moosr
40	20	Python #2		45	5	Pause
60	5	Pause		50	Rest	Stundenplan-Übung
65	20	Flask Basics				
85	5	Fragerunde				

---

Spätestens für Tag 2 VirtualBox installieren!