Python

Crashkurs im Modul Praktische Optimierung

Dr. rer. nat. Roman Kalkreuth
Lehrstuhl XI Algorithm Engineering
Fakultät für Informatik - Technische Universität Dortmund

Hintergrund

- ➔ Python ist eine universelle Hochsprache
- ➔ Deckt in seiner Architektur mehrere Paradigmen ab
  - ◆ Objektorientiert, modular, funktional, prozedural …
- ➔ Grundlegende Philosophie -> Förderung eines gut lesbaren und simplen Programmierstils

Hintergrund: *The Zen of Python*

The Zen of Python, by Tim Peters

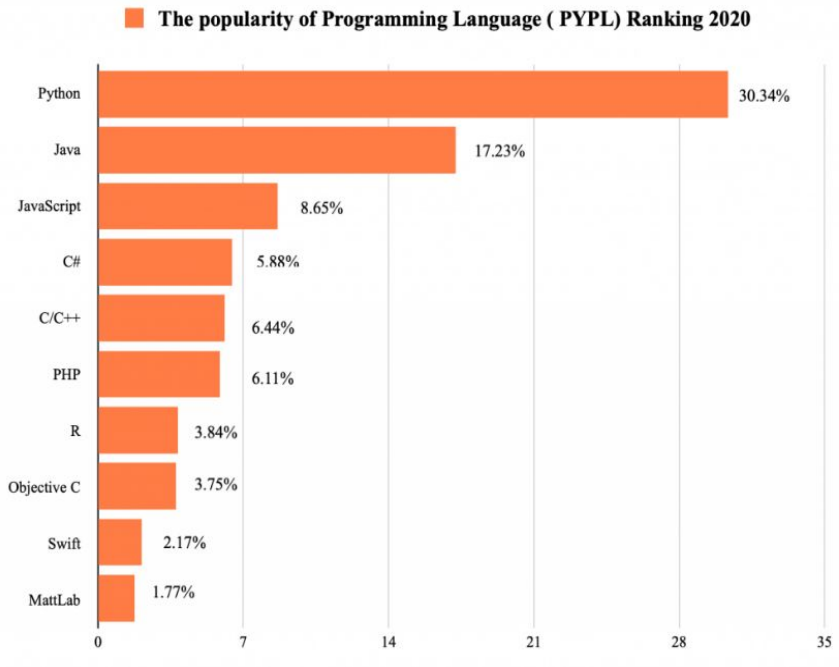
Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.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

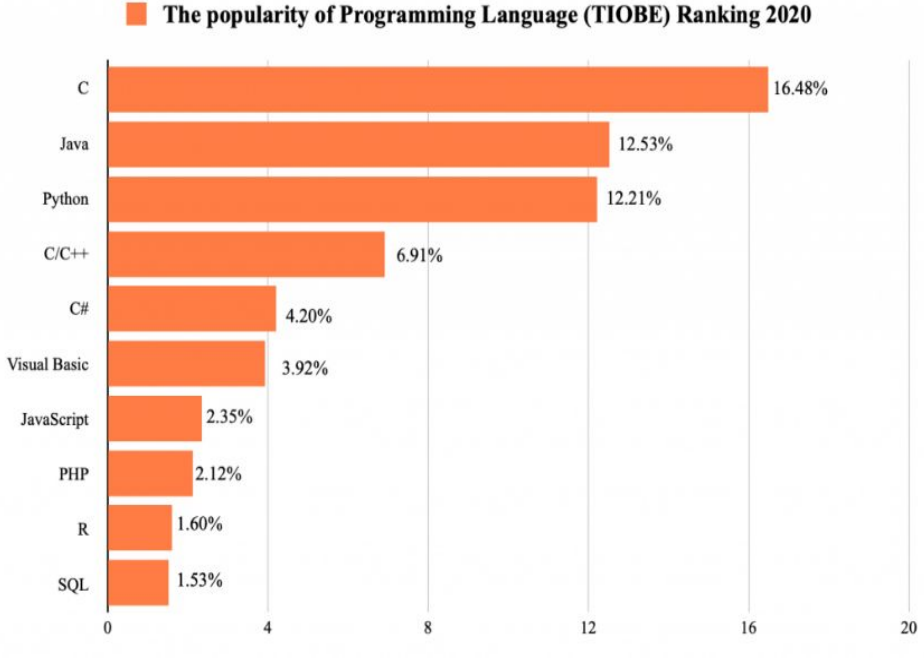
- ➔ Sammlung von 19 Leitsätzen
- ➔ Leitprinzipien für das Design von Python
- ➔ Verfasst von Python Pionier Tim Peter
- ➔ Exkurs: *Zen of Python by Example* (Link: siehe Ressourcen)

Hintergrund

- ➔ Python wurde Anfang der 1990er erstmalig bereitgestellt
  - ◆ Erfinder ist Guido van Rossum
- ➔ Namensgebung inspiriert durch die Komikergruppe Monty Python
- ➔ Aktuelle Version ist Python 3 (3.10)

Hintergrund: *Python is quite popular*





- ➔ Grundvoraussetzung: Installation des Python Interpreters
  - ◆ www.python.org

- ➔ Gängige Entwicklungsumgebungen: **PyCharm**, Eclipse, Visual Studio
  - ◆ https://www.jetbrains.com/pycharm/

- ➔ Python arbeitet im Abhängigkeitsmanagement mit virtuellen Umgebungen
  - ◆ virtualenv

Syntaktisches Grundmodell

- ➔ Strukturierung durch Einrückungen
  - ◆ Verschachtelungen durch Tabulator
- ➔ Zwei Schleifenformen: **for** und **while**
- ➔ Subfunktionen können in Funktionen eingebettet werden
  - ◆ **Innere Funktionen**
- ➔ Verzweigungen: bis Python 3.9 nur **if** - **else if** - **else**
  - ◆ Python 3.10 bietet **match - case**

Schlüsselwörter

False	def	if	raise
None	del	import	return
True	elif	in	try
and	else	is	while
as	except	lambda	with
assert	finally	nonlocal	yield
break	for	not	match
class	from	or	case
continue	global	pass	

Syntaktisches Grundmodell

Codebeispiel aus *derivation.py*

```
import math

def function(X, func):

    """ Calculates function values for the sine and cosinus function "

    # If the selected function is sine, calculate the sine of X

    if func == "sin":

        Y = math.sin(X)

    # if the selected function is cosine,  calculate the cosine of X

    elif func == "cos":

        Y = math.cos(X)

    # Otherwise raise a value error

    else:

        raise ValueError("Unknown function")

    return Y
```

- ➔ Befehle werden nicht mit einem Semikolon abgeschlossen
- ➔ Die Python Syntax vermeidet Klammerungen für Kontrollstrukturen und Funktionsdefinitionen
- ➔ Beispiel hier mit zwei Einrückungs- bzw. Verschachtelungsebenen

Syntaktisches Grundmodell

Codebeispiel aus *derivation.py*

```
import numpy as np

def derive(X, Y):

    """ Calculates the first derivative of X """

    dim = len(X)

    D = np.zeros(dim, dtype=X.dtype)

    for i, (vx, vy) in enumerate(zip(X, Y)):

        if i < dim - 1:

            dx = X[i] - X[i + 1]

            dy = Y[i] - Y[i + 1]

        D[i] = dy / dx

    return D
```

➔ Verschachtelung durch Einrückung wird auch bei Schleifen genutzt

Syntaktisches Grundmodell

Codebeispiel aus *fibonacci.py*

```
def fibonacci(n):
    if n <= 1:
        raise ValueError("Number must be greater than one!")
    fib = []
    def calc_fib(fib,  n):
        l = len(fib)
        i = l - 1
        if l == n:
            return fib
        elif l <= 1:
            fib.append(1)
        else:
            a = fib[i]
            b = fib[i-1]
            fib.append(a + b)
        return calc_fib(fib, n)
    return calc_fib(fib, n)
```

- ➔ Verschachtelung setzt sich bei inneren Funktionen fort
- ➔ Trennung/Aufspaltung der Funktionalität
  - ◆ z.B. Validierung und Rekursion

- ➔ Python unterscheidet, wie andere populäre Hochsprachen, einfache und zusammengesetzte Datentypen

- ➔ Variablen in Python besitzen keinen bestimmten Typ
  - ◆ Handhabung durch den Python Interpreter

- ➔ In Python benötigt man keine Typdeklaration

Text	str
Numerisch	int, float, complex
Sequentiell	list, tuple, range
Mapping	dict
Mengen	set, frozenset
Boolesche Werte	bool
Binär	bytes, bytearray
Sonstige	NoneType

```
>>> type(4)
<type 'int'>
>>> type(2.3)
<type 'float'>
>>> type(2+3j)
<type 'complex'>
>>> type("Hello World")
<type 'str'>
>>> type(False)
<type 'bool'>
```

- ➔ Anzeige des jeweiligen Datentyps über den Befehl type()
- ➔ Ein- und Ausgabe in diesem Beispiel direkt über den Python Interpreter

```
a = 42
print(a)
a = 3.14159265359
print(a)
a = False
print(a)
a = "Hello World!"
print(a)
```

```
42
3.14159265359
False
Hello World!
```

- ➔ Typ einer Variable kann variieren
- ➔ Angabe des Typs auch bei print() nicht erforderlich

- ➔ Das C++ Schlüsselwort **const** gibt es in Python nicht
- ➔ Python hat keine eingebauten *Konstantendeklaration*
- ➔ Konventionell wird in Python eine Variable, welche nur Großbuchstaben enthält, als Konstante *definiert*

```
PI = 3.14
EULER = 2.72
```

```
def hello_world(msg):

def hello_world(msg: str) -> str:
```

➔ Mit Python 3.6 wurde eine Syntax für das Annotieren von Variablen eingeführt
- **type hints**

➔ Allgemein erleichtert die dynamische Typisierung die Programmierung aber kann leicht zu unerwarteten Fehlverhalten führen

➔ **Wichtig:** Der Python-Interpreter ignoriert Typ-Hinweise vollständig

```
# binary
a = 0b1110


# decimal
b = 42


# octal
c = 0o35


# hexadecimal
d = 0x14b
```

➔ Neben Dezimalzahlen und Fließkommazahlen können auch Binär-, Oktal- und Dezimalzahlen deklariert werden

➔ Präfix: 0 + ‘**b’**, ‘**o**’, ‘**x**’

Ressourcen und Literatur

Python Documentation

https://docs.python.org/3/index.html

GitHub Repository

https://github.com/RomanKalkreuth/practical-optimization

Python Code Design

Pep 8 Standard	https://pep8.org/
Zen of Python by Example	https://github.com/hblanks/zen-of-python-by-example

Ressourcen und Literatur

Literatur **in progress**

The Hitchhiker's Guide to Python	*https://docs.python-guide.org/*
Python Data Science Handbook	https://jakevdp.github.io/PythonDataScienceHandbook/