

Hochschule Furtwangen University
U N F U G

Python
www.python.org

von

□ Dominik Jall


Agenda

- 👉 Einleitung
- 👉 Basiskonzepte
- 👉 Sequenzen, Dictionaries, Mengen
- 👉 Kontrollstrukturen und Iteratoren
- 👉 Funktionen
- 👉 Klassen
- 👉 GUIs mit Tk
- 👉 Ende

Einleitung


Was ist Python?

 Interpretiert

 Portabel

 Objektorientiert

 „Free as in freedom“


 Leicht zu lernen

Wer hat's erfunden?

 Guido van Rossum



 Über den Ursprung von Python schrieb van Rossum 1996:

 Im Dezember 1989, suchte ich nach einem Programmierprojekt, das mich über die Weihnachtswoche beschäftigen würde. Mein Büro würde geschlossen bleiben, aber ich hatte auch zu Hause einen PC und sonst nicht viel zu tun. Ich entschied mich, einen Interpreter für die Scriptsprache zu schreiben, über die ich kürzlich nachdachte. Ich wählte Python als Arbeitstitel für das Projekt, weil ich in einer leicht respektlosen Stimmung (und ein großer Fan des Monty Python's Flying Circus) war.

Monty Python?



© BBC

www.bbc.co.uk/comedy

Die heutige Situation






 Python wird durch die PSF (Python Software Foundation) entwickelt und verwaltet.

 Python unterliegt der GPL





 Aktuelle Version ist 2.4

Python ist toll – na und?

Das Besondere an Python

-  Python-Syntax ermöglicht extrem kurze, aber dennoch mächtige Programme.
-  Indentation gehört zur Grammatik. Blöcke werden nicht durch Klammern markiert.
-  Python kann objektorientiert, imperativ und funktional programmiert werden.
-  Datentypen ergeben sich aus dem Kontext
-  „Tupeling“ von Variablen ($x, y = 1, 2$)

Weitere Merkmale von Python


-  In der Mathematik übliche Schreibweisen sind Bestandteil der Sprache ($a < b < c$)
-  Wenige, aber dafür mächtige Sprachkonstrukte. Python ist „minimalistisch“
-  Alle Objekte besitzen automatisch einen Wahrheitswert. (z.B. alle nicht-leeren Strings und alle Zahlen $\neq 0$ sind true)
-  Rechnen mit Zahlen beliebiger Länge ist integriert.


Basiskonzepte

 Python kann unterschiedlich benutzt werden:

 In einer interaktiven Shell

```
 >>> print „hallo“
```

 Übergabe von Anweisungen an den Interpreter

```
 $ python -c „print 'hallo'“
```

 Übergabe eines kompletten Skripts

```
 $ python hallo.py
```

- alternativ: `#!/usr/bin/python`
`print „hallo“`

Zeilenstrukturen

🦶 Erlaubt ist:

🐍 `summe = 1 + 1`

🦶 Nicht erlaubt ist:

🐍 `summe = 1 +
1`

🐍 `aber: summe = 1 + \
1`

🦶 Aber nicht:

🐍 `summe = 1 + \
1 #Kommentar`




Einrückungen

 Es gibt keine {Blöcke}

 Es gibt keine Semikola oder anderes;

 Blöcke werden über Indents erzeugt:


```
 for I in range(5):  
    print I  
    print „GROG!“
```


print „Das wird nur einmal ausgegeben!“


 Alle Anweisungen desselben Blocks müssen um die gleiche Stellenzahl eingerückt sein.


Objekte, Werte und Adressen

 „Alles ist ein Objekt!“


 Nicht nur Klassen, auch Daten, Funktionen, Programmcode, Ausnahmen und alles andere.


 Jedes Objekt hat eine Identität, einen Wert und einen Typ

 Bsp: $x = 3$
 $y = x$

 Das Objekt mit dem Wert 3 hat jetzt die Namen x und y .


Änderbare und unveränderbare Objekte

 Änderbare Objekte (mutable objects) sind z.B. Listen.

 Unveränderbare (immutable) behalten ihren Wert bis sie zerstört wurden.

 Dazu gehören: u.a. ganze Zahlen, floating Points, Strings und Tupel


 ?? - Was meint er denn damit ??


 ganz einfach:

```
□ a = 'ab'  
  a = 'cd'
```

Objekte(2)


 Bedeutet:

 a ist nicht etwa ein String, der erst den Wert „ab“ und dann „cd“ enthält.

 a ist nur ein Name!


 Das (unveränderliche) Objekt mit dem Wert „ab“ hat erst den Namen a und dann keinen mehr.


 a ist dann der Name für den String „cd“


 Der GarbageCollector entsorgt das Objekt „ab“


Namespaces: lokale und globale Namen

 Zu jedem Block existiert ein lokaler und ein globaler Namensraum.

 Namespaces sind als Dictionairies implementiert.

 Können mit `globals()` und `locals()` abgefragt werden.


```
 def test():  
    x = 1  
    x = 2
```

 Es handelt sich dabei um verschiedene Objekte.


Sequenzen, Dictionaries, Mengen

 Eine Sequenz ist eine Folge mehrerer Objekte.

 In Python: Strings, Tupel und Listen:

 Strings: „abc“ oder 'abc'

 Tupel: (1,2,3) oder („String“,2,“anotherString“,3.5)


 Liste: [1,2,3] oder [„String“,2]


 Im Unterschied zu Tupeln sind Listen veränderbar.


- Tupel und Listen können Objekte beliebigen Typs enthalten, also auch andere Listen und Tupel.

Effektive Operationen auf Sequenzen

 Einige mächtige Operationen:

 `x in s` : True, wenn ein Element mit dem Wert `x` von `s` enthalten ist (Typ egal).


 `n * s` : `n` Kopien der Sequenz werden angehängt.


 `reversed(s)` : Dreht die Sequenz `s` um.

 `sorted(s)` : Sortiert sie Sequenz `s`.

Effektive Operationen auf Sequenzen(2)


Slicing von Sequenzen

 Mit Slicing können bequem Abschnitte von Sequenzen gebildet werden.

```
 s = [1,2,3,4]  
    slice = s[1:3]  
    print slice  
    [2,3]
```


 `s[i:j]` erzeugt einen Slice zwischen `i` und `j-1`

 `s[i:]` erzeugt einen Slice von `i` bis zum Ende von `s`.

 `s[:j]` erzeugt einen Slice von 0 bis `j-1`.

 `s[:]` erzeugt einen kompletten Klon von `s`.

Listen

 Listen können als änderbare Tupel gesehen werden.

 Listen verfügen bereits über alle nötigen „Bordmittel“

- `append(x)` – An die Liste wird `x` angehängt.
- `count(x)` – Anzahl Elemente in der Liste.
- `extend(l)` – An die Liste werden die Elemente der Liste `l` angehängt. (nicht die Liste selbst)
- `insert(x)` – Element `x` einfügen
- `pop()` - Löscht das erste Element.
- `remove(x)` – Löscht erstes Element mit dem Wert `x`.

Aliasieren und Kopieren von Listen

☞ Ein Alias ist ein anderer Name für das gleiche Objekt.


☞ Verändert man das Objekt über einen Namen, wirkt sich dies auch auf alle Aliase aus:

```
a = [1,2,3]
b = a
b[0] = 4
a
[4,2,3]
```




Aliasieren und Kopieren von Listen (2)

 Möchte man eine Liste „richtig“ kopieren, muss man einen Klon anlegen:


```
 a = [1,2,3]
b = a[:]
a[0] = 4
a
[4,2,3]
b
[1,2,3]
```

Listen – Immer zu empfehlen?

 Listen sind nett, dynamisch, leicht zu verstehen und bei 40 Grad waschbar.

 Man könnte sie doch dauern verwenden!

 Nein!

 Listen sind zwar komfortabel, aber das bezahlt man mit hohem Speicherverbrauch und belastet den Interpreter zusätzlich.

 Deshalb: Ausweichen auf Sets (Mengen)!


Mengen

 Es gibt zwei Arten:


 Sets

 FrozenSets (immutable sets)








 In Python sind Mengen mathematisch korrekt:

 Ungeordnete Sammlung von Elementen

 Keine Dubletten

 Mathe-Kram geht: Vereinigung, Durchschnitt, Differenz, etc.

Operationen auf Mengen


-  `iter(s)` – Liefert einen Iterator auf `s`
-  `difference(t)` – Differenzmenge zu `t` (`s - t`)
-  `intersection(t)` – Durchschnitt mit `t` (`s & t`)
-  `issubset(t)` – Teilmenge von `t`? (`s <= t`)
 -  `s < t` – Echte Teilmenge von `t`
-  `issuperset(t)` – Obermenge von `t`? (`s >= t`)
-  `union(t)` – Vereinigungsmenge mit `t` (`s | t`)

Dictionaries


 Auch assoziatives Feld genannt.

 Möglicherweise leere Folge von Wertepaaren in {...}


 z.B. `dict = {„foo“:„bar“, „monty“:„python“}`


 Ermöglicht schnellen Zugriff auf Daten über „Keys“, ähnlich eines Hashes bei Perl oder Hashtables bei Java.


Dictionaries (2)

 Mit Dictionaries kann man z.B. richtige Wörterbücher generieren oder user/passwd Information ordentlich speichern:

 `woerterbuch = {„sun“:“Sonne“, „earth“:“Erde“}`

 Zugriff über Key/Value:
`woerterbuch[„sun“]`
`'Sonne'`


 `passwd = {„Guybrush“:“Threepwood“, „23“:“42“}`

 `passwd[„Guybrush“]`


...

Kontrollstrukturen und Iteratoren


 „Stinknormale“ if-Verzweigungen:

```
 if x != 0:  
    y = 1 / x  
    print „Kehrwert:“, y
```

 Else:

```
 if x != 0:  
    ...  
else  
    print „Geht nicht!“
```

 Elif:

```
 elif x < 0  
    print „Negativ!“
```


Kontrollstrukturen


 Verzweigung mit logischem Operator

 Macht Sinn bei Zuweisungen oder Returns


 In C/C++/Java mit $a = (b < c) ? b : c$


 In Python können `and` / `or` benutzt werden:

 $a = ((b < c) \text{ and } b) \text{ or } c$

 Gesprochen: „Wenn $b < c$ wahr ist, wird der Ausdruck zu b , sonst zu c “.


Iteratoren

 Die üblichen Verdächtigen: for und while

 For:

```
humpen = („Grog!“, „Grog!“, „Grog!“)  
for drink in humpen:  
    print drink
```


 Nach for kommt die Zählvariable, dann eine Sammlung von Objekten (z.B. Sequenz).


 Die For-Schleife kann auf direkt jedem beliebigen iterierbaren Objekt benutzt werden!

Iteratoren (2)


 Die üblichen Verdächtigen: for und while

 While:


```
 x = 1  
y = 0  
while x < 100:  
    y = y + 1  
else:  
    print „Y=“,y
```

 Else-Block definiert, was passiert, wenn die Bedingung nicht mehr wahr ist.


Iteratoren (3)

 Wie kann ich schnell eine Schleife n mal durchlaufen, wenn ich nur über Sequenzen iterieren darf?

 also kein: `for(int i = 0; i < n; i++)`

 Lösung: `range()`

 `range()` liefert eine Liste von 0 bis n-1:



`for i in range(10):`
`print „Python rockt!“`

Funktionen


- ☞ Funktionen werden als „callable types“ bezeichnet. Sie wie Datentypen behandelt.
- ☞ Return-Werte müssen nicht explizit angegeben werden.
- ☞ Funktionsparameter sind typenlos und können mit default-Werten belegt werden.
- ☞ Es können auch andere Funktionen als Parameter übergeben werden. Dabei sind eventuelle weitere Parameter berücksichtigt.

Funktionsaufrufe


 Normaler Aufruf:

```
 sayHello(„Hello World!“)
```

 Aufruf mit default-Werten:

```
 def sayHello(text=“Hello World!“):...  
    sayHello()
```


 Aufruf mit anderen Funktionen als Parameter


```
 def berechne(f, n):  
    for i in range(n):  
        f(i)
```

 Aufruf mit beliebiger Anzahl von Parametern

```
 def postkarte(adressat, *grussVon):...
```

Funktionsaufrufe (2)

```
 def postkarte(adressat, *grussVon):  
    print „Hallo „, adressat, „!“  
    print „Bier kalt, Wasser warm, alles roger im  
    Urlaub!“  
    print „Viele Gruesse von“  
    for i in range(len(grussVon))  
        print grussVon[i],
```

```
 Aufruf:  
postkarte(„Hillary“, „Bill“, „Monika“, „ und George“)
```


```
 Ausgabe?
```


Parametertyp bestimmt Rückgabetyt


 Funktionsparameter sind zunächst typenlos

 Python entscheidet selbst, welcher Typ der beste ist.

 Das muss bei bestimmten Funktionsaufrufen beachtet werden, um Fehler zu vermeiden:

```
 def kehrwert(x):  
    return 1/x
```

```
 kehrwert(3)  
0 ??
```

```
 richtig: kehrwert(3.0)  
0.333...
```

Lokale Funktionen

👉 LF sind Funktionen innerhalb anderer Funktionen.

👉 Bekannt aus Java (z.B. ActionListener)
🦶 (nur halt als Klasse)

👉 LF sind nur innerhalb der eigenen Funktion sichtbar:

```
🦎 def makeMeHappy():  
    def happy():  
        print „You are now happy!“  
    happy()
```

🦎 happy() ist nur innerhalb von makeMeHappy()


Generator-Funktionen

- 👉 Extrem geiles Feature
- 👉 Extrem mächtiges Instrument
- 👉 Liefert einen Generator, der Elemente „just-in-time“ berechnet:


```
🐍 def genQuadrat(n):  
    for i in range(n):  
        yield i*i
```


🐍 Wenn „yield“ in einer Funktion auftaucht, liefert die Funktion einen Generator für den Ausdruck hinter yield.


Makros in Python: lambda

 Makros sind schick, schnell und bekannt aus C

 In Python können Makros als anonyme „namenlose“ Funktionen schnell mal zwischendurch generiert werden:

 `(lambda x,y: x * x + y * y) (2,3)`
13

 Erzeugt eine lambda-Funktion und führt sie gleich aus. Wenn die Funktion keinen Namen erhält wird sie gelöscht:

 `f = lambda x,y: x*x + y*y`

Objektorientierung unter Python

- 👉 Python unterstützt OOP voll
- 👉 Schlüsselwort: class (wie immer)
- 👉 Python kann Mehrfachvererbung
- 👉 Private != Private:
 - 👉 Starke Privatheit
 - 👉 Schwache Privatheit (analog: protected)
- 👉 this=self
- 👉 Methoden haben immer self als Argument


Klassen

Erzeugung von Klassen:

 Normal:

 `class MyClass:`


 Einfach vererbt:


 `class MyClass(MySuperClass):`

 Mehrfach vererbt:

 `class MyClass(MySuperClass, AnotherClass):`

 Konstruktor:


 `def __init__(self):`


 Destruktor:

 `def __del__(self):`

Public, Private, Protected

 Etwas verwirrend: Es gibt weder public, private noch protected.

 aber: Die „Privacy“ hängt von der Anzahl vorangestellter Underscores („_“) ab:


```
 class C:  
    a = „public“  
    _b = „protected“  
    __c = „private“
```

Methoden

- ☞ Methoden können ebenso öffentlich, privat (`_`) und ganz privat (`__`) sein.
 - ☞ Das erste Argument der Deklaration muss immer „self“ sein.
 - ☞ Beim Aufruf wird self weggelassen. Der Interpreter setzt dafür die Instanz der Klasse ein.
- ```
class Klasse:
 def normaleMethode(self, param):
 def _protectedMeth(self, p1,p2):
 def __privateMeth(self, abc):
```


# Module und Pakete

 Man kennt das ja von Java...


 `import package bla`  
oder:

 `package bla`

 In Python:

 `import modul`  
Lädt ganzes Modul

 `from modul import Klasse[,Klasse2,...]`  
Lädt nur eine Klasse (oder mehrere)

 `from modul import *`  
Wie `import modul`

# Pakete

 „Module von Modulen“

 paket/

unterpaket1/

\_\_init\_\_.py

modul1.py

unterpaket2/

\_\_init\_\_.py

modul1.py

modul2.py


 Importiere so:

- import paket.unterpaket1
- oder: import paket.unterpaket2.modul1

# Javadoc – Auch in Python?

 JA! - Aber nicht so advanced:


 „“ Dies ist ein Hilfetext „“

 """ Dies hier auch """

 Kann für Klassen, Funktionen und Methoden definiert werden:

 class Klasse:

    """ Dies ist die Klasse, die ich am meisten hasse """


 def func():

    """ Func is eine funky Funktion! """

 Abruf über:

    help(item)

# GUIs mit Tk

 Tkinter-Modul stellt einfache Tk-GUIs für Python zur Verfügung.

 `import Tkinter`


 Es gibt diverse weitere Module für andere Bibliotheken (GTK, QT, WxWindows, etc..)

 Mit Python kann superschnell ein Fensterchen gemacht werden:

 `wnd = Tkinter.Tk()`

 `[wnd.mainloop()]`

# GUI-Elemente einfach hinzufügen


```
 from Tkinter import *
def hallo():
 print „Hallo“
wnd = Tk()
label = Label(wnd, text=„Begruessung“)
button = Button(wnd, text=„Sage Hallo“,
command=hallo)
label.pack()
button.pack()
root.mainloop()
```


 Das sollte hoffentlich laufen...


# Layout-Manager

 Bekannt aus AWT/Swing

 In Tk gibt es:


 pack()

 Packt neue Widgets oben, unten, rechts oder links von den anderen.

 place()

 Setzt neue Widgets pixelgenau in ihre Container.

 grid()

 Benutzt ein Raster mit Zeilen und Spalten zum Anordnen.

# Natürlich gibt's auch MessageBoxes

 Es gibt verschiedene:

 `showerror(title, message)`

 `showinfo(...)`

 `askokcancel(...)`

 `askyesno(...)`



Ende

# Noch Fragen?

