Python

Programmieren in höheren Programmiersprachen

Peter Barth

Hochschule Mannheim

Fakultät Informationstechnik

16. November 2019

Programmieren in höheren Programmiersprachen, am Beispiel Python

Motivation

- Entwicklerproduktivität und Sicherheit im Fokus
- Weg von der Maschine, hin zum Entwickler
- Fokus auf die Problemstellung und Algorithmen

Ziele – Entwickeln in höheren Programmiersprachen

- Laufzeitumgebung, Referenzen und Garbage Collection
- Mächtige funktionale und objektorientierte Sprachmittel
- Integration von Bibliotheken, integrierte Bibliotheken verwenden

Ziele – Python, Skriptsprache

- Andere Syntax, kompakt, mächtig, eingebaute Datentypen
- Objektorientiert programmieren und funktional programmieren
- Unit Testing, Laufzeitumgebung, Patterns
- Batteries included, Bibliotheken, String-Verarbeitung, GUI
- C/C++ integrieren
- Numerische Berechnungen, Graphiken erstellen, Daten aufbereiten und analysieren

Organisatorisches

Vorlesung

- Freitag, 4. DStd (13:40), S221

Praktikum

- Freitag, 5. DStd (15:20), S117

Kein separates Tutorium :-(

- Nutzen Sie die Rechnerräume wann immer frei
- Sie können am Freitag auch etwas länger bleiben
- Moodle

Testat und Projekt (20 Extra-Punkte)

- Testat 60 min am Rechner; Skript, Doku (PDF/html) verfügbar
  ??.??., Freitag, 15:20, S113/117
- Mini-Projekt bis letzte Vorlesungswoche
  17.1.20, Freitag, 23:00, per Email

Klausur (80 Punkte, ab 40 Punkte bestanden)

- 90 Minuten, keine Hilfsmittel

Ablauf, Bewertung

Vorlesung als PDF, laufend aktualisiert

- Vorlesung als Foliensatz, (noch) kein Skript
- Für Sie zum Lesen und Nachprogrammieren
- In der Veranstaltung selbst auch viel live programmieren

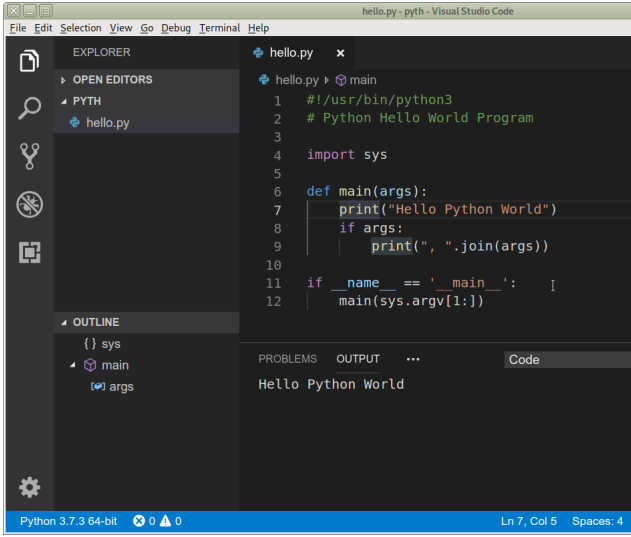
Übungsblätter als PDF

- Praktische Aufgaben zur Vertiefung des Stoffs, kleine Projektchen
- Gelerntes auf ähnliche Probleme übertragen
- Selbstständiges Arbeiten, individuell
- Spaß am Programmieren

Ein Testat, ein Mini-Projekt, 20 Punkte

- Python programmieren, GUI mit Python
- Jeweils 10 Punkte

Klausur, 80 Punkte



Ablauf im Semester, Inhalte

Programmieren in Python

- Einführung, Historie, Übersicht
- Mächtige eingebaute Datentypen inklusive Listen und Dictionaries
- Dateien, Imperatives Programmieren, Unit Testing
- Funktionales Programmieren, Primitive, List Comprehension
- Module und Pakete
- Objektorientierung und Ausnahmen, Iteratoren und Generatoren
- Threading und IO-Anwendungen
- Bibliotheken, GUI, reguläre Ausdrücke
- Integration mit C/C++
- Berechnungen (`numpy`), Diagramme (`matplotlib`)
- ...

Voraussetzungen, Erwartungen, Infrastruktur

Voraussetzungen

- Imperatives und Objektorientiertes Programmieren, Speicherverwaltung
- Erste Erfahrungen mit etwas größeren Programmieraufgaben wünschenswert (SET)

Erwartungen, Ablauf

- Regelmäßige Teilnahme (auch im Praktikum), aktive Mitarbeit, Pünktlichkeit
- Bearbeiten der Übungsaufgaben, alles selbst probieren, nicht nur in den 2 Stunden Praktikum
- Vor- und Nachbereitung des Stoffs, Lesen und Nachprogrammieren
- Klären von Fragen, Fragen Sie, Immer!!!

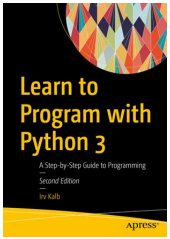
Immer und überall

- Wir verwenden Visual Studio Code (Spyder geht auch) als Python-Editor/IDE
  Anleitung zum Installieren für Linux, MacOSX und Windows (egal)
  Moodle (Python), `https://www.pbma.de/swt/`
- Jeder erhält ein Source-Code Repository (Git, 19pythxx)
- Sie können sich alles in 5 Minuten selbst installieren für zu Hause
- Sie können von zu Hause mit x2go auf einem Hochschulrechner (Linux) arbeiten
- In allen Pools ist auf allen PCs die gleiche Umgebung (Linux/Windows) verfügbar
- Es gibt keine Ausrede nicht aktiv zu arbeiten ;-)

Quellenverzeichnis

Literatur und Online-Quellen: Python

- `www.python.org`,
  `docs.python.org/3.6/`, Tutorial, HowTos, Library Reference
- Learning Python, O'Reilly, Lutz, 2013
- Python von Kopf bis Fuß, Barry, 2017
- Einführung in Python 3, Bernd Klein, Hanser, 2017
  (das ist gut, fast alles drin) [online]
  `https://www.hanser-elibrary.com/doi/book/10.3139/9783446453876`
- Einstieg in Python, Thomas Theis, Rheinwerk, 2018
- Learn to Program with Python 3, Irv Kalb, Apress, 2018
  (gaaaaanz langsam die ersten Kapitel) [online]
- Programmierung in Python, Ralph Steyer, Springer, 2018
  (die erste Hälfte) [online]



Spezielle Kapitel

- Numerisches Python : Arbeiten mit NumPy, Matplotlib und Pandas,
  Bernd Klein, Hanser, 2019 [online]

Moodle nutzen, Forum

[online] per Download an der Hochschule über Bibliothek als ebook erhältlich

Python

Pseudocode that runs

Was ist Python?

- Objektorientierte Skriptsprache („Perl by a sane person“)
- Einfach erlernbar (Einsteigersprache), interaktiv, Open Source
- Um Größenordnung schnellere Entwicklung, auch für Prototypen geeignet
- Plattformunabhängig, interpretiert, Bytecode (JIT mit pypy)

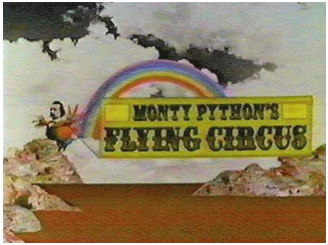
Features

- Dynamische Typisierung, Garbage Collection
- Eingebaute Datenstrukturen (Listen, Tupel, Dictionaries, String, . . .)
- Mächtige Ausdrucksweise und Werkzeuge (verketten, abbilden, slicen)
- Klassische Kontrollstrukturen (imperativ), Objektorientierung und funktionale Primitive
- Unterstützung für große Projekte (Module, Pakete, Ausnahmen, . . .)
- Integration (C/C++, Java; REST, SOAP, XML-RPC; DBMS, . . .)
- Bibliotheken (Text/Binär, ADTs, Math/Numerik, Concurrency (Threading, Async),
  Web, Email, reguläre Ausdrücke, XML/JSON/CSV, GUI, Unittesting, Bild(be|ver)arbeitung, Graphiken, Statistik, KI, . . .)

Historie

Background

- Treibende Kraft: Guido van Rossum, BDFL (-7/18)
- Name von „Monty Python Flying Circus“



Historie

90: Als Skriptsprache für verteiltes Betriebsystem konzipiert

91: Erste Veröffentlichung

94: comp.lang.python, CPython, andere Implementierungen:

- 99: Jython, Java VM
- 01: psyco, native Code → PyPy
- 06: IronPython, .Net
- 07: PyPy, Python in Python, JIT
- 13: MicroPython, Mikrocontroller

Heute

- Aktuell CPython 2.7/3.7, Jython 2.7, IronPython 2.7, PyPy2.7/3.5
- Python3 (seit 2008) hat sich durchgesetzt, wir nehmen Python 3.6
- Aktive Entwicklergemeinde, sehr viele Bibliotheken, `python.org`, `pypi.python.org`

Verbreitung

Plattform für Anwendungen

- Hunderte Skripte in `/usr/bin`
- Blender (`blender.org`): 3D-Graphik
- Mercurial, Trac (`trac.edgewall.org`): Source/Projekt Management
- Google App Engine, Django, Raspberry Pi
- `www.python.org/about/success/`
- Auf Bibliotheken aufbauend: Tensorflow

Interner Einsatz

- Google, Youtube, NASA, Dropbox, ILM
- Jede Firma, die irgendwas mit KI macht

Entwickler

- Sehr weit verbreitet
- Die meist genutzte Skriptsprache
- Top Four in TIOBE
  `www.tiobe.com/tiobe-index/`

Mar 2019	Mar 2018	Change	Programming Language	Ratings	Change
1	1		Java	14.880%	-0.06%
2	2		C	13.305%	+0.55%
3	4	^	Python	8.262%	+2.39%
4	3	v	C++	8.126%	+1.67%

Top 1 im IEEE Language Ranking 2018
`spectrum.ieee.org/static/interactive-the-top-programming-languages-2018`

Umgebung

Interpreter und Tools

- `www.python.org/ftp/python/3.6.8/`
- Integrierte Entwicklungsumgebung (IDLE), Interpreter
- `ipython`: Interactive Python, Kommandozeile (gut!)

`ipython` oder „Shell“ in IDLE

- Sprache lernen, mit Bibliothek experimentieren
- Ausdrücke auswerten, Programme Testen

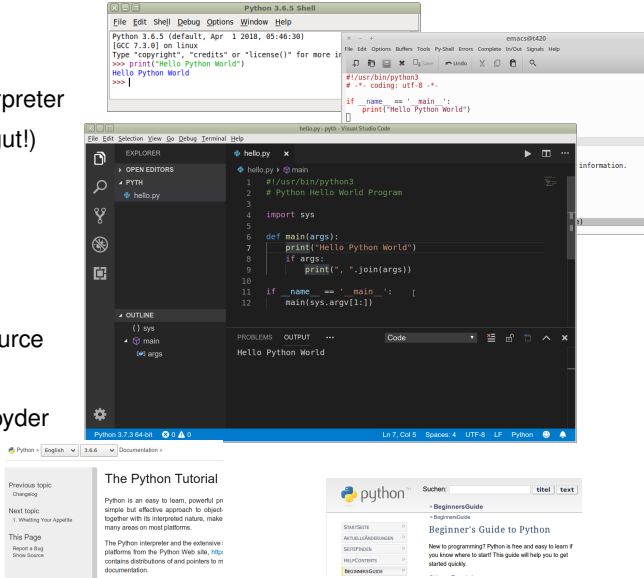
IDE, Visual Studio Code

- Gute Fonts, gute Completion, schnell, Open Source
- `code.visualstudio.com`

Alternativen: Emacs, PyCharm, Eclipse (pydev), Spyder

Online-Lektüren (neben Büchern)

- `docs.python.org/3/tutorial/`
- `wiki.python.org/moin/BeginnersGuide`
- `pythonchallenge.com`, `checkio.org`

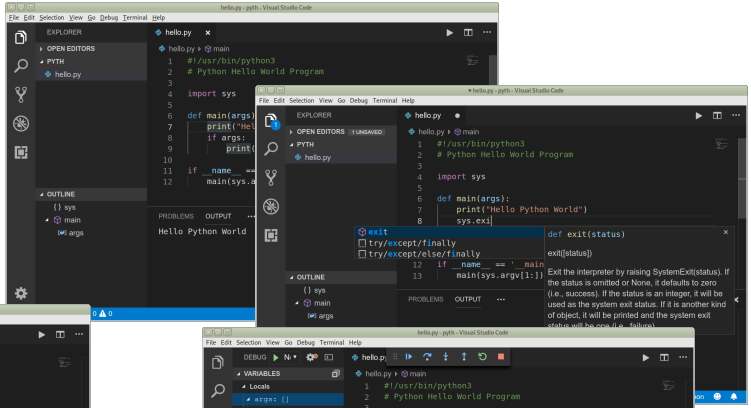


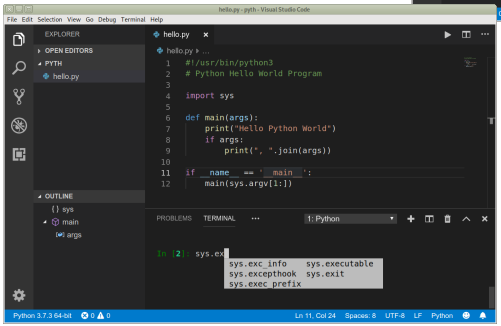
Visual Studio Code (`code`), Übersicht

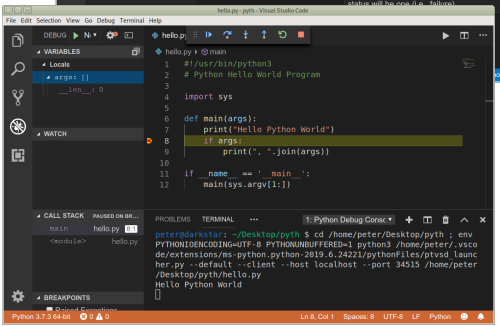
Schlanke, schnelle, moderne IDE

- Editor, Completion/Intellisense
- Refactoring, Coding Conventions
- Warnungen, Debugger . . .
- Terminal dabei, Erweiterungen
- Geniale GIT-Integration

Obwohl von Microsoft und mit Node.js







IDLE Übersicht

Interaktive Shell und Editoren

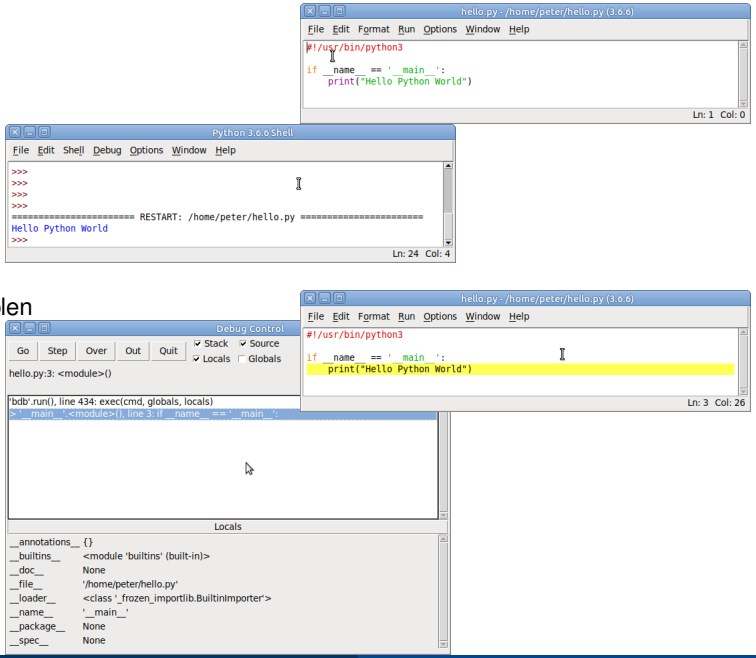
- Im Editor mit F5 Programm in interaktiver Shell ausführen
- Interaktive Shell ist kein Editor

Integrierter Debugger

- Stop, Over, Out
- Mitlaufen Source
- Anzeige Stack, lokale/globale Variablen
- Seltener benötigt als in C
  - Interaktives Testen
  - Weniger Fehler in Python ;-)

Alternativen

- Emacs: solide, interaktiv
- Spyder: okay, schlank, ipython
- Eclipse, pydev: mächtig, ipython integriert, schwerfällig
- PyCharm: ähnlich zu Eclipse



Encoding bei Python Skripten, Unicode verwenden

Angabe der Zeichenkodierung

- Erste Zeile im Skript
  `# * coding: <encoding name> *`
  oder zweite Zeile, falls erste
  `#/usr/bin/python3`
- Spezieller Kommentar, teilt python (und Editoren/IDEs) Encoding des Quelltexts mit

Verfügbare Zeichenkodierungen

- `utf-8`
  - Unicode: alle Sprachen
  - Seit Python3 Standard, muss nicht mehr angegeben werden
  - Sehr stark empfohlen, default in `code`
- `iso-8859-1`, vermeiden
  - Standard Westeuropa, `latin1`
- Böse: `Mac-Roman` (OsX), `cp1285` (Win)

```
1 #!/usr/bin/python3
2 # -*- coding: utf-8 -*-
3
4 print("Hello Python World")
```

BÄ¶ses Encoding

1. Shebang: Starte den Python-Interpreter mit den folgenden Zeilen als interaktive Eingabe
   Wir nehmen `python3`, ohne 3 ist es Python 2
2. Encoding: Der ganze Quelltext, inklusive Kommentare und den Texten in Strings, ist in dem angegebenen Encoding

Seit Python3 Zeile mit `coding` nicht nötig.

Alle ausführbaren Programme mit Zeile 1 beginnen

```
1 #!/usr/bin/python3
2
3 print("Hello Python World")
```

Erste Schritte – Zahlen und Ausdrücke

```
>>> print("Hallo Python Welt")
Hallo Python Welt
>>> # Kommentare
>>> 2 # Integer-Zahlen
2
>>> 3.14 # Fliesskommazahlen
3.14
>>> "String" # Zeichenketten
'String'

>>> 0xff # Hex
255
>>> 0o377 # Oktal
255
>>> 0377 # in 2.7 noch oktal
'Syntax Error: invalid token'
```

```
>>> 6+2 # Ausdruecke
8
>>> 6*2
12
>>> 6/2
3.0

>>> 6/5 # Vorgabe float bei Division (3.x)
1.2

>>> 6//5 # Explizit ganzzahlig
1

>>> 3**4 # Exponenten
81

>>> 3+1j # Komplexe Zahlen
(3+1j)
```

Erste Schritte – Zuweisungen und Tests

```
>>> x = 1 # Variablenzuweisung
>>> x # hat den zugewiesenen Wert
1
>>> x = y = 2 # Mehrfachzuweisung
>>> x
2
>>> y
2
>>> abs(-3) # eingebaute Funktionen
3

>>> 0 < 1 # Tests
True
>>> 1 < 0
False
>>> True == 1 # Bool neuer Typ
True
>>> False == 0
True
```

```
>>> 0 < 5 < 10 # Mehrfachtest
True

>>> True and False
False
>>> True or False
True

>>> not "eins"
False
>>> not False
True

>>> "eins" and "zwei" # letzter Wert
'zwei'
>>> False and 0 # entscheidende Wert
False
>>> 0 and False
0
```

Erste Schritte – Binäres Rechnen und Division

```
>>> x = 2 # binaer ...000010
>>> x = 0b0010 # geht direkt
>>> x << 15 # Linksshift
65536
>>> x & 0xff # binaeres Und
2
>>> x | 1 # binaeres Oder
3
>>> ~x # Komplement
-3
>>> x^3 # binaeres XOR
1
>>> 0b10 ^ 0b11
1
>>> bin(0b10 ^ 0b11)
'0b1'

>>> 10.0/3
3.3333333333333335
```

```
>>> 3/2 # Divison: float (3.x), int (2.x)
1.5
>>> 3//2 # ganzzahlige Division ([23].x)
1
>>> 3.0/2 # Hochkonvertieren
1.5
>>> 3.0//2 # Abschneiden
1.0
>>> 3./2 # 0 nicht notwendig
1.5
>>> 10 % 3 # Modulo, Rest
1
>>> 10 // 3 # ganzzahlig
3
>>> 3*(10//3) + 1*(10%3)
10

>>> 10.0//3 # erzwungen ganzzahlig
3.0         # anderer Typ
```

Große Zahlen und Konvertierung

```
>>> 3e10 # Exponentenschreibweise
30000000000.0

>>> 2**10 # Potenzieren
1024
>>> 2**20 # grosse ganze Zahl
1048576
>>> 2**1000 # sehr grosse ganze Zahl
107150860718626732094842504906000181056
140481170553360744375038837035105112493
612249319837881569585812759467291755314
682518714528569231404359845775746985748
039345677748242309854210746050623711418
779541821530464749835819412673987675591
655439460770629145711964776865421676604
298316526243868372056680693 76

>>> # keine Typ-Unterscheidung 2**10, 2**1000
>>> # in Python3
```

```
>>> float(3) # explizite Konvertierung
3.0 # in C-double (es gibt nur float)

>>> int(3.0) # und zurück
3 # in Zahl, es gibt nur int, beliebig gross

>>> int("12") # auch fuer Strings
12
>>> int("11", 16) # explizite Basis
17

>>> int("zwoelf") # nur wenn möglich
# ansonsten Ausnahme
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int()
            with base 10: 'zwoelf'
```

Zahlen und Operationen

Zahlentypen

- ganze Zahlen, `int` und `long` (beliebig lang)
- Gleitkommazahlen, `float` mit `double` Genauigkeit, komplexe Zahlen
- Intuitive Darstellung und Eingabe

Alle üblichen arithmetischen Operationen

- Standardarithmetik: `+`, `-`, `*`, `/`, `**`, `%`, `//`
- Logische Arithmetik: `~`, `^`, `|`, `&`, `<<`, `>>`
- Vorrangregeln und Klammern wie gewohnt

Boolesche Werte: `False`, `True`

- Entspricht 0 und 1 (Historie)
- Oder leerer String, nichtleerer String, . . .

Logische Ausdrücke: `and`, `or`, `not`

- Erstes bestimmendes Element wird zurückgegeben
- Die anderen Argumente werden nicht ausgewertet

```
>>> print(1, 2.3, 4.5+6.7j)
1 2.3 (4.5+6.7j)
>>> 1j**2
(-1+0j)
>>> 3//2
1
```

```
Python 3.x
>>> 3/2
1.5
```

```
Python 2.7
>>> 3/2
1
```

```
>>> 0 or ""
      or "Hallo"
      or "Welt"
'Hallo'
```

Ausdrücke

Aufrufe von Funktionen und Methoden

- Funktionsaufrufe
- Methodenaufrufe
- Interaktive Ausgabe des Wertes
- Zusammengesetzte Ausdrücke
- Bereichstests

Werte ausgeben in der interaktiven Kommandozeile

- Zuweisungen haben *keinen* Wert
- Wert von Ausdrücken wird ausgegeben
- **print** für explizite Ausgabe

```
>>> lis = [3, 1, 2] # Liste
>>> len(lis) # Funktionsaufruf
3
>>> lis.append(8) # Methode
>>> lis # Ausgabe
[3, 1, 2, 8]
>>> len(lis) < 6 and len(lis)
# Zusammengesetzt
4
>>> 1 < len(lis) < 8 # Bereich
True
```

```
>> lis
[3, 1, 2]
>>> print(lis) # 2.x, 3.x
[3, 1, 2]

# Python 2: Ohne Klammern verboten in 3.x
>>> print lis  # 2.x, war schön...
[3, 1, 2]
```

Ausgabe mit `print`

Ausgabe von Objekten mit **`print`**

- Lesbare Ausgabe von Objekten
- Ausgabe aller Objekte durch Leerzeichen getrennt
- Setzen von `end`, statt Zeilenumbruch
- Ausgabe mit Formatstring, wie in C, `%`-Operator
- Seit Python 3.x, nur noch in Klammern

Alternative

- Ausgabe über offene Dateihandles in `sys`
- Ausgabe ist nur der Text `Hallo Welt`
- 11 ist der Rückgabewert, Anzahl der Zeichen; wie in C `printf`

```
>>> lis = [1, 2, 3]
>>> print(lis, "zwei", 4)
[1, 2, 3] zwei 4
>>> print(lis, "zwei", 4); print("weiter")
[1, 2, 3] zwei 4
weiter
>>> print(lis, "zwei", 4, end=""); print("weiter")
[1, 2, 3] zwei 4weiter
>>> print("%s %s" % ("Hallo", "Welt"))
Hallo Welt
>>> print(lis)
[1, 2, 3]
```

```
>>> import sys
>>> sys.stdout.write("Hallo Welt\n")
Hallo Welt
11
```

Zeichenketten, Strings – Darstellung

Syntax

- Einfache oder doppelte Anführungszeichen
- Nicht limitierendes Anführungszeichen nutzbar
- Alternativ Escape-Zeichen verwenden Backslash \
- Typ `str`

```
>>> "hallo"
'hallo'
>>> 'hallo'
'hallo'
```

```
>>> "spam's"
"spam's"
>>> 'spam"s'
'spam"s'
>>> 'spam\'s'
"spam's"
```

Mehrzeilige Strings

- Drei Anführungszeichen am Anfang und am Ende
- Zeilenendezeichen als \n verfügbar
- Alle Anführungszeichen verwendbar

```
>>> """hallo
...   du "tolle"
...     'perfekte' Welt """
'hallo\n  du "tolle"\n    \'perfekte\' Welt '
```

Strings – Operationen

Alle üblichen Operationen vorhanden

- `+` Konkatenation
- `*` Wiederholung
- `[+i]` Indizierung
  positive Zahl, 0 ist erstes Zeichen
- `[-i]` Indizierung von hinten
  negative Zahl, -1 ist letztes Zeichen

Slicing

- `[von:bis]` Ausschnitt von inklusive, bis exklusive
- `[von:]` Ausschnitt bis Ende
- `[:bis]` Ausschnitt ab Anfang
- `[:]` Gesamt (Kopie)

```
>>> "Hallo" + " Welt"
'Hallo Welt'
>>> "Hallo"*3
'HalloHalloHallo'
>>> "Hallo"[0]
'H'
>>> "Hallo"[-1]
'o'
```

```
>>> "Hallo"[1:3]
'al'
>>> "Hallo"[1:]
'allo'
>>> "Hallo"[:-1]
'Hall'
```

Slicing

`"SLICEOFSPAM"[von:bis]`

- von, einschließend
- bis, ausschließend

Vorzeichen

- Positive Zahlen, von links, ab 0
- Negative Zahlen, von rechts, ab -1

Dritter Parameter Schrittweite

- `"SLICEOFSPAM"[von:bis:step]`
- Schrittweite, normalerweise 1

Geht auch mit Listen (später)

0	1	2	3	4	5	6	7	8	9	10
s	l	i	c	e	o	f	s	p	a	m
-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1

```
>>> "sliceofspam"[2:-1]
'iceofspa'
>>> "sliceofspam"[5:]
'ofspam'
>>> "sliceofspam"[:5]
'slice'
>>> "sliceofspam"[2:5]
'ice'
```

```
>>> "sliceofspam"[2:-1:3]
'iop'
>>> "sliceofspam"[::4]
'sep'
>>> lis = [0, 1, 2, 3, 4, 5, 6, 7]
>>> lis[::2], lis[1::2]
([0, 2, 4, 6], [1, 3, 5, 7])
```

Strings – Formatierung wie `printf`

Formatierung

- Formatstring und Argumente
- Ähnlich zu C, `printf`
- `%s` String
- `%d` Zahl
- `%x` Hexadezimal
- `%e, %f, %g` Gleitkommaformate
- `%%` Prozentzeichen

Benamtes Argument

- `%(<key>)<typ>`
- Durch `<key>` bestimmter Wert in Abbildung als Argument
- Gefolgt von Argumenttyp

```
>>> "ein %s Papagei" % "toter"
'ein toter Papagei'
>>> "%d %s Papageie" % (2, "tote")
'2 tote Papageie'
>>> "%x" % 42
'2a'
>>> "%X" % 42
'2A'
>>> "%e %f %g" % (1.1, 1.2, 1.3)
'1.100000e+00 1.200000 1.3'
>>> "%%%s%%" % "hallo"
'%hallo%'
```

```
>>> "%(name)s heisst %(name)s" %
                    {"name": "Eric"}
'Eric heisst Eric'
```

String – Formatierung mit `format`

Formatierung

- Moderne Formatierung mit `format`-Methode von String
- `"{}"` für den nächsten Parameter
- `"{i}"` für den i.ten Parameter
- `"{name}"` für Schlüsselwort-Parameter, geht auch als Dictionary

Zahlformatierung

- je Typ Modifizierer nach Doppelpunkt wie `printf`
- Einzunehmender Platz als Modifizierer
- Rechts(>), Mittig(^), Links(<)

Ab Python 3.6: F-Strings (selbst nachlesen)

```
>>> "Hello {} Python {}".format("nice", "world")
'Hello nice Python world'
>>> "Hello {0} Python, the {1} is {0}".format("nice", "world"
'Hello nice Python, the world is nice'
>>> "Hello {attr} Python, the {sub} is {attr}"
                .format(attr="nice", sub="world")
'Hello nice Python, the world is nice'
>>> dic = dict(attr="nice", sub="world")
>>> dic
{'attr': 'nice', 'sub': 'world'}
>>> "Hello {attr} Python, the {sub} is {attr}".format(**dic)
'Hello nice Python, the world is nice'
# Syntax für **dic kommt noch
>>> "{0:05.2f}".format(1.234)
'01.23'
>>> "{0:5}x{1:8}y{2:3}z".format(1, 2, 3)
'    1x       2y  3z'
>>> "{0:>5}x{1:^8}y{2:<3}z".format(1, 2, 3)
'    1x   2    y3  z'
```

Strings – Builtins und raw Strings

Eingebaute Funktionen, Builtins

- Länge mit `len`
- Standardoperationen für Vergleich
- Test auf Enthaltensein mit `in`
  - Teilstring, Substring

`raw` Strings

- Steuerzeichen quotieren
- Hilfreich bei regulären Ausdrücken, später

```
>>> len("Hallo")
5
>>> "hallo" < "wallo"
True
>>> "al" in "Hallo"
True
```

```
>>> r"raw \n string"
'raw \\n string'
>>> print(r"raw \n string")
raw \n string
>>> "normaler \n string"
'normaler \n string'
>>> print("normaler \n string")
normaler
 string
```

Strings – weitere Features

`string`-Modul

- als Modul „importieren“
- String-Methoden
  (noch einige freie Funktionen)
- Groß- und Kleinschreibung
- Suchen, finden, ersetzen
- Konvertieren, splitten, kleben
- Konkatenation automatisch
- Leerzeichen entfernen

Kommandozeilenargumente

- Modul `sys`
- Liste von Argumente `sys.argv`

```
$ python3 argv.py a b c d
['argv.py', 'a', 'b', 'c', 'd']
```

```
>>> "ab".upper()
'AB'
>>> "spammify".find("mm")
3
>>> int("42")
42
>>> "spammify".split("mm")
['spa', 'ify']
>>> "XX".join(["spa", "ify"])
'spaXXify'
>>> "spa" "XX" "ify"
'spaXXify'
>>> " Meaning of Life ".strip()
'Meaning of Life'
```

argv.py

```
1 #!/usr/bin/python3
2 import sys
3 print(sys.argv)
```

Eigenschaften von Strings

Strings sind unveränderlich (gut!)

- Operation mit Strings (+, [], etc.) erzeugen immer neue Strings
- Vorhandene Strings werden niemals verändert

Strings sind Sequenzen

- Geordnet, die Position der Elemente ist wichtig
- Klare Menge von unterstützten Operationen
- Sequenz von Zeichen

Unicode Strings

- Seit Python3 default
- u vor dem String noch erlaubt
- Typ `str` ist Unicode
- Typ `bytes` ist Byte-String fast wie Typ `str` in Python2

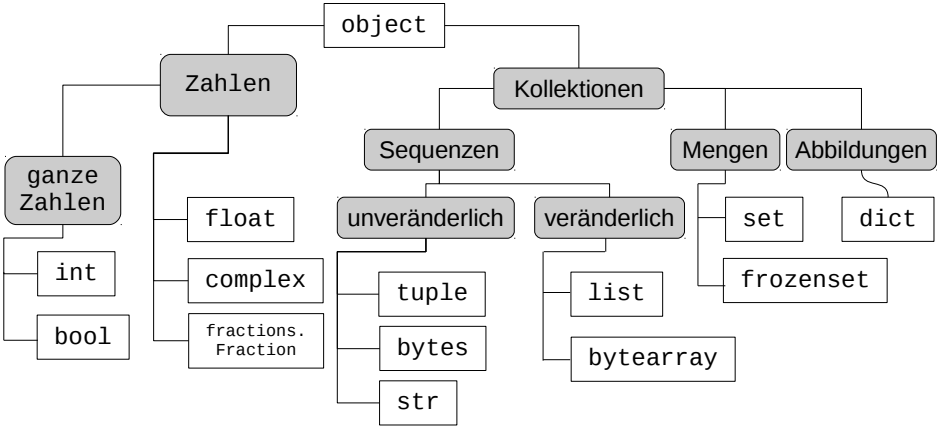
```
>>> s = "hallo"
>>> s[0] = "H"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not
           support item assignment
```

```
>>> s = "hallo"; s
'hallo'
>>> s == u"hallo"
True
>>> s == b"hallo"
False
>>> type("hi"), type(u"hi"), type(b"hi")
(<class 'str'>, <class 'str'>, <class 'bytes'>
>>> b"hä"
  File "<stdin>", line 1
SyntaxError: bytes can only contain
  ASCII literal characters.
```

Typhierarchie eingebauter Typen

`object` als Wurzel für alle Typen
`isinstance` zum Testen
Alles, auch eine Zahl, ist ein `object`

```
>>> isinstance(3, int)
True
>>> isinstance(3, object)
True
```



Kollektionen

Eingebaute Kollektionen

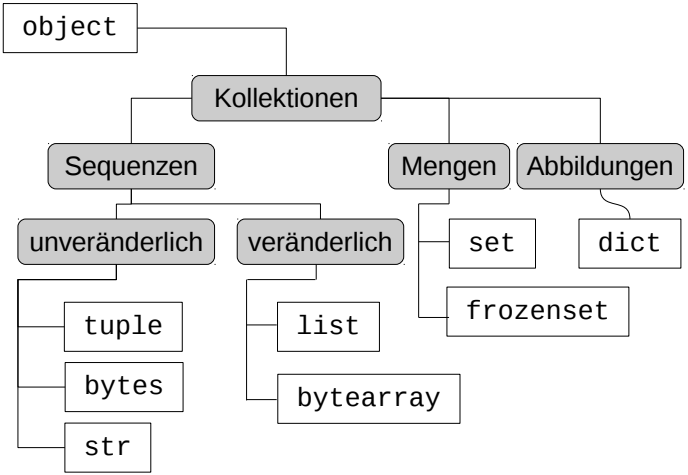
- Eingebaute Typen zum Verwalten von mehreren Objekten
- Teil der Sprache, nicht nur der Bibliothek

Vorteile

- Effiziente Implementierung
- Einfache Formulierung
- Konzentration aufs Problem, nicht auf technische Implementierungsdetails

Umfangreich

- Alles was man braucht
- Ungefähr die Container der `STL`
- Unicode-Strings, Byte-Arrays, Listen, Tupel, Mengen, Wörterbücher



Tupel – Sequenz

Tupel sind eine Sequenz

- Syntax n-Tupel, n$\geq$ 2 wie erwartet
- 1-Tupel hat spezielle Syntax
  `(1)` wäre nur ein arithmetischer Ausdruck mit Klammerung
- Leeres Tupel wie erwartet

Beliebig kombinierbar

- Beliebige Typen, schachtelbar

Indizierung

- mit `[]`
- Slicing (wie mit allen Sequenzen), Ergebnis ist wieder Tupel

```
>>> (1, 2)
(1, 2)
>>> (1, 1, 1)
(1, 1, 1)
>>> (1, )
(1,)
>>> (1)
1
>>> ()
()
>>> (1, "eins", 2.0, (34, 4))
(1, 'eins', 2.0, (34, 4))
>>> tup = (1, "zwei", (3, 4))
>>> tup[0]
1
>>> tup[2][1]
4
>>> tup[1:]
('zwei', (3, 4))
```

Tupel – Builtins

Builtins von Sequenzen

- Länge mit `len`
- Konkatenation mit `+`
- Wiederholung mit `*`
- Test auf Enthaltensein mit `in`

Automatisch Tupel

- Auch ohne Klammern

Tupelzuweisung

- Nette Syntax
- Geht auch ohne Klammern
- Gut bei Rückgabewerten

```
>>> tup = (1, "zwei", (3, 4))
>>> len(tup)
3
>>> tup + tup
(1, 'zwei', (3, 4), 1, 'zwei', (3, 4))
>>> tup*2
(1, 'zwei', (3, 4), 1, 'zwei', (3, 4))
>>> (3, 4) in tup
True
```

```
>>> 1, 2
(1, 2)
>>> (1, 2)
(1, 2)
>>> (a, b) = (1, 2)
>>> print(a, b)
1, 2
>>> b, a = a, b
>>> a, b
(2, 1)
```

Tupel – Unveränderliche Sequenz

Tupel sind unveränderlich

- Operationen mit Tupel erzeugen neue Tupel
- Achtung! Tupel kann veränderliche Werte als Tupelelemente haben

Tupel sind Sequenzen

- Geordnet
- Klare Menge unterstützter Operationen
- Sequenz von Werten beliebigen Typs, im Gegensatz zu Strings (nur Zeichen)

```
>>> tup = (1, 2, 3)
>>> tup[1] = "zwei"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not
            support item assignment
>>> tup = (1, [], 3)
>>> tup[1].append("zwei")
>>> tup
(1, ['zwei'], 3)
```

Listen – Sequenz

Syntax wie Tupel

- Umschlossen mit `[` und `]`
- Keine spezielle Syntax mehr für einelementige Listen
- Leere Liste mit `[]`

Beliebig kombinierbar und schachtelbar

- Auch mit Tupel

Sonst wie Tupel

Wie `vector` in C++

```
>>> [1, 2, 3]
[1, 2, 3]
>>> [1], []
[1], []
>>> [1, [2, "zwei"], (3, "drei")]
[1, [2, 'zwei'], (3, 'drei')]
>>> lis = [1, [2, "zwei"], (3, "drei")]
>>> lis[0]
1
>>> lis[1:]
[[2, 'zwei'], (3, 'drei')]
>>> lis[2][0], len(lis)
(3, 3)
>>> lis + [4,5]
[1, [2, 'zwei'], (3, 'drei'), 4, 5]
>>> 1 in lis
True
>>> (lis*2)[1:-2]
[[2, 'zwei'], (3, 'drei'), 1]
```

Listen – Veränderliche Sequenz

Ändern

- Zuweisung von Teilen
- Löschen

Kopien und Zuweisung

- Operationen erzeugen Kopien
- Nachträgliches Verändern der Parameter ohne Einfluss

Flache Kopien

- Kopien nur auf oberster Ebene
- Erzeugen neuer Listenstruktur
- Die Elemente werden nicht kopiert, sondern nur Referenzen
- Ändern der Elemente ist sichtbar

```
>>> lis = [1, 2, 3]
>>> lis[0] = "eins"; lis
['eins', 2, 3]
>>> del lis[0]
>>> lis
[2, 3]
```

```
>>> lis, kat = ["a", "b"], ["c", "d"]
>>> lk = lis + kat
>>> lk
['a', 'b', 'c', 'd']
>>> lis[0] = "X"
>>> lis, lk
(['X', 'b'], ['a', 'b', 'c', 'd'])
>>> tup = (lis, kat)
>>> tup
(['X', 'b'], ['c', 'd'])
>>> lis[1] = "Y"
>>> tup
(['X', 'Y'], ['c', 'd'])
```

Listen – Zuweisungen

Zuweisungskompatibilität

- Sequenzen sind untereinander zuweisungskompatibel

Slice-Zuweisung möglich

- Der Bereich wird durch Sequenz als Liste ersetzt

Destruktive Methoden

- `append` und `extend` ändern gegebene Liste, kein Rückgabewert
- **`del`** ändert Liste
- Zuweisung an Index ändert Liste

```
>>> lis = [1, 2, 3]
>>> lis[1:2] = ("zwei", "two")
>>> lis
[1, 'zwei', 'two', 3]
>>> lis[1:2] = ["zwei", "two"]
>>> lis
[1, 'zwei', 'two', 'two', 3]
>>> lis.append(4)
>>> lis
[1, 'zwei', 'two', 'two', 3, 4]
>>> lis.extend([5, 6])
>>> lis
[1, 'zwei', 'two', 'two', 3, 4, 5, 6]
>>> del lis[1:3]
>>> lis
[1, 'two', 3, 4, 5, 6]
>>> lis[1] = 2
>>> lis
[1, 2, 3, 4, 5, 6]
```

Listen – range und Methoden

`range` generiert Zahlenliste

- Anfang, erstes Argument, optional (default 0)
- Ende, zweites Argument
- Schrittweite, drittes Argument, optional (default 1)
- Nur auf Bedarf, erzwingen durch Konvertieren Liste erzeugen mit `list`

Methoden, Listen als Objekte

- `append` ist eine destruktive Methode
- Destruktive Methoden haben keinen Rückgabewert
- Destruktive Methoden haben einen Seiteneffekt
- Die Liste wird verändert

```
>>> range(5)
range(0, 5)
>>> list(range(5))
[0, 1, 2, 3, 4]
>>> list(range(1, 5))
[1, 2, 3, 4]
>>> list(range(0, 50, 10))
[0, 10, 20, 30, 40]
>>> lis = list(range(5))
>>> lis.append(5)
>>> lis
[0, 1, 2, 3, 4, 5]
```

Listen – Weitere destruktive Methoden

Kellerverhalten und mehr

- `pop`, oberstes entfernen und zurückgeben
- `insert`, Einfügen

Rumdrehen

- `reverse`, destruktiv
- `reversed`, erzeugt umgedrehtes neu, ursprüngliche Liste nicht verändert, wieder nur auf Bedarf (lazy)

Sortieren

- `sort`, destruktive Methode
- `sorted`, erzeugt sortierte Kopie, ursprüngliche Liste nicht verändert, direkt die Liste (eager)

Beispiel

```
>>> lis = list(range(6))
>>> lis
[0, 1, 2, 3, 4, 5]
```

```
>>> lis.pop()
5
>>> lis
[0, 1, 2, 3, 4]
>>> lis.insert(0, -1); lis
[-1, 0, 1, 2, 3, 4]
>>> lis.insert(4, 2.5); lis
[-1, 0, 1, 2, 2.5, 3, 4]
>>> lis.reverse(); lis
[4, 3, 2.5, 2, 1, 0, -1]
>>> reversed(lis)
<list_reverseiterator object at 0x7f7bb0c78f28>
>>> list(reversed(lis))
[-1, 0, 1, 2, 2.5, 3, 4]
>>> lis = [4, 3, 2.5, 2, 1, 0, -1]
>>> sorted(lis)
[-1, 0, 1, 2, 2.5, 3, 4]
>>> lis.sort(); lis
[-1, 0, 1, 2, 2.5, 3, 4]
```

Listen – Listen veränderlich versus Tupel unveränderlich

Listen sind veränderlich

- Sequenz-Operationen unveränderlicher Operationen mit Listen erzeugen neue Listen auf oberster Ebene
- Spezielle destruktive Listen-Operationen, ohne Wert

Listen sind Sequenzen

- Ähnlich zu Feld (`vector`) in C++
- Laufzeitverhalten ähnlich zu Feld

Warum Listen und Tupel?

- Bei gemischten Typen eher Tupel;
  bei Liste möglich, nicht intuitiv
- Unveränderliches Tupel effizienter
- Tupel kann als Schlüssel im Wörterbuch genutzt werden

```
>>> lis = [1, 2, 3]
>>> lis[1] = "zwei"
>>> lis
[1, 'zwei', 3]
>>> lis.append(4)
>>> lis
[1, 'zwei', 3, 4]
```

```
>>> (1, "ab")
(1, 'ab')
>>> [1, "ab"]
[1, 'ab']
```

Dictionaries, Wörterbücher

Dictionary, Wörterbuch

- Abbildung unveränderlicher Werte (Schlüssel) auf beliebigen Werten
- Syntax für Zugriff und Zuweisung wie beim Feld
- Realisiert über Hashing

Vorteile

- Verwendung einfach und intuitiv
- Schlüssel nicht beschränkt auf Zahlen
- Keine Feldgröße notwendig

Dictionaries und Sequenzen

- Schlüsselmenge (Sequenz), Wertemenge und Tupelmenge Schlüssel/Wert erzeugbar
- Wieder nur auf Bedarf, lazy
- Test auf Enthaltensein gegen Schlüsselmenge mit `in`

Wie `unordered_map` in C++

```
>>> dic = {} # neues leeres Dictionary
>>> dic = { 'spam' : 2 } # neues Dictionary
>>> dic['spam']
2
>>> dic['spam'] = 3
>>> dic
{'spam': 3}
>>> dic['eggs'] = 42
>>> dic
{'eggs': 42, 'spam': 3}
>>> dic.keys() # lazy
dict_keys(['eggs', 'spam'])
>>> list(dic.keys())
['eggs', 'spam']
>>> dic.values()
dict_values([42, 3])
>>> dic.items()
dict_items([('eggs', 42), ('spam', 3)])
>>> 'eggs' in dic # automatisch in keys
True
```

Dictionaries – Builtins und Schlüsseleigenschaften

Eingebaute Funktionen

- `len` für Anzahl Schlüssel
- **`del`** zum Löschen
- Keine Slicing-Operationen, Dictionary ist keine Sequenz

Schlüssel unveränderlich

- Intern mit Hashing realisiert
- Zahlen, Strings, etc.
- Tupel (auch tiefe Strukturen) sind erlaubt

Ausnahme bei nicht vorhandenem Schlüssel

```
>>> dic = {'eggs': 42, 'spam': 3}
>>> len(dic)
2
>>> del dic['eggs']; dic
{'spam': 3}
>>> dic[0:]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'slice'
>>> dic[(1, 2)] = 'tupel'
>>> dic
{(1, 2): 'tupel', 'spam': 3}
>>> dic[[1, 2]] = 'liste'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'list'
>>> dic['cheese']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'cheese'
```

Dictionaries – Veränderliche Abbildungen

Dictionaries sind veränderlich

- Zuweisung mit neuem Schlüssel fügt neue Elemente hinzu
- Zuweisung mit existierendem Schlüssel überschreibt Wert
- **`del`** löscht Schlüssel/Wert Zuordnung

Dictionaries sind die einzigen Abbildungen

- Ungeordnet (keine Sequenz), Zugriff nur über Schlüssel
- Feste Menge von unterstützten Operationen
- Zugriff zu beliebigen Objekten beliebigen Typs (Werte)
- Zugriff über beliebige unveränderliche Objekte (Schlüssel)
- Unveränderlicher Schlüssel
  - Wegen interner Implementierung mit Hashing
  - Sonst würde man Wert nicht wiederfinden
  - Siehe Algorithmen und Datenstrukturen

`collections.OrderedDict`

- Dictionary, das sich die Einfügereihenfolge merkt
- In Bibliothek

```
>>> d = dict()
>>> d[1]=2; d[3]=4; d[5]=6
>>> d[7]=8; d[9]=10
>>> d
{1:2, 3:4, 9:10, 5:6, 7:8}
>>> d["pi"] = 3.14
>>> print(d[1], d["pi"])
2 3.14
>>> d.keys()
dict_keys([1, 3, 5,
           7, 9, 'pi'])
>>> from collections
        import OrderedDict
>>> d = OrderedDict()
>>> d[1]=2; d[3]=4; d[5]=6
>>> d[7]=8; d[9]=10
>>> d
OrderedDict([(1, 2), (3, 4),
   (5, 6), (7, 8), (9, 10)])
```

Mengen – Veränderlich, keine Reihenfolge

Mengen

- Keine doppelten Elemente
- Keine Reihenfolge, keine Sequenz
- Realisiert wie Dictionaries,
  nur ist der Wert egal

Eingebaute Funktionen und Operationen

- `len` für Anzahl Elemente
- `in` für Test auf Enthaltensein
- Mengenoperationen mit `|` und `&`
- Konstruktur mit `set` oder `{ ... }`

```
>>> s = set()
>>> s.add(1)
>>> s
{1}
>>> s.add(2); s.add(3)
>>> s
{1, 2, 3}
>>> s.add(2)
>>> s
{1, 2, 3}
>>> 2 in s
True
>>> 4 in s
False
>>> len(s)
3
>>> s | {2, 3, 4}
{1, 2, 3, 4}
>>> s & {2, 3, 4}
{2, 3}
```

Dateien

Modul `os` für Systemoperationen

- `chdir`, um Verzeichnis zu wechseln
- `getcwd`, um Verzeichnis anzuzeigen
- Unix und Windows Notation

Dateien

- Öffnen mit `open`
- Dateiname und Bearbeitungsart (`r` lesen, `w` schreiben, `a` erweitern)
- `read` zum Lesen der gesamten! Datei
- `readlines` für String-Liste der Zeilen
- `close` Schließen, geht auch über Garbage Collector
- Schreiben funktioniert ähnlich
- Text-Dateien haben encoding, mit Modus `'br'` zum binären Lesen öffnen

```
>>> import os
>>> os.chdir('/usr/share/dict')
>>> os.getcwd()
'/usr/share/dict'
>>> eingabe = open("words", "r")
>>> inhalt = eingabe.read()
>>> len(inhalt)
971304
>>> inhalt[731079:731103]
"python\npython's\npythons\n"
>>> eingabe.close()
>>> os.chdir('/tmp')
>>> ausgabe = open('wordskopie', 'w')
>>> ausgabe.write(inhalt)
971304
>>> ausgabe.close()
>>> open('wordskopie')
<_io.TextIOWrapper name='wordskopie' mode='r'
  encoding='UTF-8'>
```

Typ-Überprüfung und Konvertierung

Test welcher Typ

- `type` gibt Typ zurück
- `isinstance` testet ob von Typ
- Vergleich und Auswahl von Typen

Typkonvertierung

- Typname als Funktionsaufruf

Typen

- `int`, `long`, `float`, `complex`
- `str`, `list`, `tuple`
- `dict`, ...

Konvertierung von/zu Zeichen

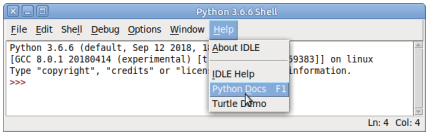
- Zeichen in Python nicht vorhanden
  (keine Zahl, meist String der Länge 1 verwendet)
- `ord` und `chr`

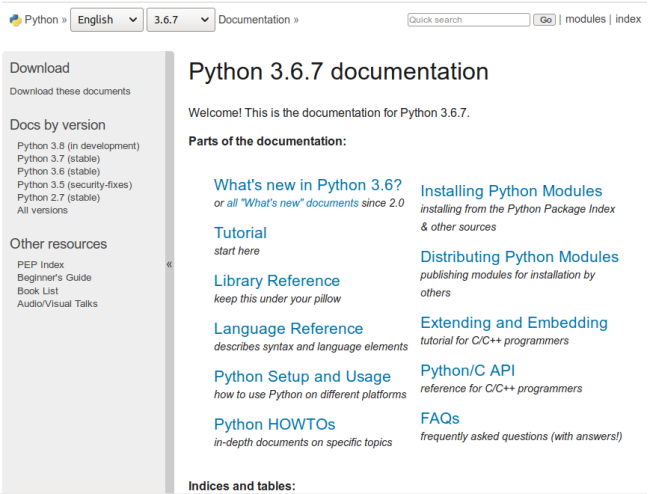
```
>>> type([])
<class 'list'>
>>> lis = [1, 2, 3]; type(lis)
<class 'list'>
>>> type(lis)==list, type(lis)==type({})
(True, False)
>>> isinstance(lis, list)
True
>>> int, float, complex
(<class 'int'>, <class 'float'>, <class 'compl
>>> str, list, tuple, dict
(<class 'str'>, <class 'list'>,
 <class 'tuple'>, <class 'dict'>)
>>> int("12")
12
>>> list("123")
['1', '2', '3']
>>> ord('A'), chr(65)
(65, 'A')
```

Die Doku ist Dein Freund

Dokumentation

- `docs.python.org`
- Tutorial, sehr guter Einstieg
- Bibliothek, die Referenz für alle eingebauten und anderen Objekte, Funktionen, Module
- Integrierte Anzeige der Dokumentation in `code`, sehr gut





Variablen

Verwendung

- Müssen nicht deklariert werden
- Müssen vor erster Verwendung initialisiert werden

Typ, dynamisch und streng

- Dynamische Typisierung,
  Variable an alles bindbar
- Strenge Typisierung,
  gebundenes Objekt hat Typ

Syntaxspezialitäten

- Zuweisung hat keinen Wert
- Mehrfachzuweisung möglich
- Tupelzuweisung

Gültige Variablenbezeichner wie in C++,
keine Schlüsselwörter

```
>>> val = 3
>>> val
3
>>> bla
NameError: name 'bla' is not defined
>>> val = 3
>>> val = "Hallo"
>>> val = 3
>>> val + 3
6
>>> val = "3"
>>> val + 3
TypeError: Can't convert 'int' object
      to str implicitly
```

```
>>> x = y = 3
>>> x, y = 3, 4
>>> x, y
(3, 4)
```

Zuweisung und Ausdrücke

Gleichheit und Identität

- Alles ist ein Objekt
- Gleichheit ==, Objektidentität `is`

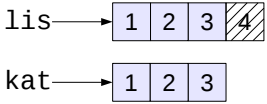
Zuweisung

- Variablen sind Namen,
  die an Objekte gebunden sind
- Rechte Seite einer Zuweisung gibt Objekt (vorhandenes oder neues) zurück
- Linke Seite einer Zuweisungen ist ein Name an den ein Objekt gebunden wird

Ausdrücke evaluieren zu neuen Objekten

- Die Sequenz von Anfang bis Ende ist ein Ausdruck
- `[:]` macht eine (flache) Kopie

```
>>> lis = [1, 2, 3]
>>> kat = [1, 2, 3]
>>> lis == kat, lis is kat
(True, False)
>>> lis.append(4)
>>> lis
[1, 2, 3, 4]
>>> kat
[1, 2, 3]
>>> lis = [1, 2, 3]
>>> kat = lis[:] # Kopie
>>> lis == kat, lis is kat
(True, False)
```



Referenzsemantik

Referenzsemantik

- Variablen sind Namen, die an Objekte gebunden sind
  - Denken Sie an Zeiger
  - Aber automatisch dereferenziert
  - Keine Zeigerarithmetik
- Rechte Seite einer Zuweisung gibt (Referenz auf) Objekt zurück
- Linke Seite einer Zuweisungen ist ein Name an den ein Objekt gebunden wird

Namen evaluieren zu dem gebundenem Objekt

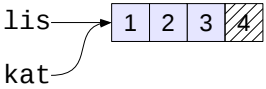
- Änderungen von Objekten über alle Referenzen sichtbar

Referenzen statt Zeiger

- Alles ist wie ein sicherer Zeiger
- Es gibt keine Werte auf dem Stack
- Freispeicher wird automatisch verwaltet, Garbage Collection

Alles ist einfach

```
>>> lis = [1, 2, 3]
>>> kat = lis
>>> lis == kat, lis is kat
(True, True)
>>> lis.append(4)
>>> lis
[1, 2, 3, 4]
>>> kat
[1, 2, 3, 4]
```



Kontrollstrukturen – Verzweigung

Verzweigung

- Schlüsselwörter: `if`, `elif`, `else`
- `<bedingung>`: Ausdruck, der als Wahrheitswert interpretiert wird
- `<anweisungen>`: Beliebige Folge von Anweisungen

Syntax

- Bedingung endet mit Doppelpunkt, keine Klammern notwendig
- **ACHTUNG!** Einrückung ist signifikant und Teil der Syntax
  - Block durch Einrücktiefe bestimmt
  - Einrückung Tabulator oder gleiche Anzahl an Leerzeichen
    Wir nehmen 4 Leerzeichen je Block
  - Keine (geschweiften) Klammern!

Einzeiler möglich, vermeiden

```
if <bedingung>:
    <anweisungen>
[elif <bedigung>:
    <anweisungen>]*
[else:
    <anweisungen>]
```

```
if 0 == 0:
    print("ja")
if "nichtleerer String":
    print("ja")
else:
    print("nein")
if 0:
    print("eins")
    print("zwei")
print("drei")
if 1: print("eins")
```

`if/else` Ausdruck statt ?-Operator

Syntax ungewöhnlich

- Erst Ergebniswert im „normalen“ Fall
- Dann Bedingung
- Dann „alternativer“ Fall

```
<wert> if <bedingung> else <alternative>
```

```
// in C/C++
<bedingung> ? <wert> : <alternative>
```

Alternative mit booleschem Ausdruck

- `and` und `or` evaluieren nur so weit wie notwendig
- Nachbau mit `and`/`or` möglich
- Problematisch wenn „normaler“ Wert äquivalent zu falsch ist
- Nicht mehr statt `if/else` verwenden

```
>>> x, y = 1, 2
>>> e = "kleiner" if x<y else "groessergl"
>>> e
'kleiner'
```

```
<bedingung> and <wert> or <alternative>
```

```
>>> x, y = 1, 2
>>> e = x<y and "kleiner" or "groessergl"
>>> e
'kleiner'
```

Kontrollstrukturen – `while`-Schleife

`while`-Schleife

- Schlüsselwörter:
  `while`, **`else`**, **`break`**, **`continue`**, **`pass`**
- **`else`**-Zweig nur, wenn Schleife ohne **`break`** verlassen wurde
- **`pass`**, leere Anweisung, macht nichts, Ersatz für `{}`

Syntax

- Wieder Einrücken als integraler Bestandteil der Schleifensyntax

```
while <bedingung>:
    <anweisungen>
[else:
    <anweisungen>]
```

```
1 lis = [1, 2, 3, 4]
2 while lis:
3     print(lis.pop())
4 else:
5     print("Ende")
```

```
4
3
2
1
Ende
```

Kontrollstrukturen – `for`-Schleife

`for`-Schleife

- Schlüsselwörter: **`for`**, **`in`**, **`break`**, **`continue`**, **`pass`**
- Iterieren über Elemente in Sequenzen
- **`in`** wählt Elemente aus Sequenz sukzessive aus und weist sie je Durchlauf der Variable zu
- Klassische Schleife über Indexe gibt es nicht
  - Nur noch als Nachbau mit `range`

`range`

- `range(von, bis, step)`
- Erzeugt einen Iterator über Zahlen `von` inklusive, `bis` exklusive
- `von` optional, default 0
- `step` optional, default 1

```
for <variable> in <sequenz>:
    <anweisungen>
[else:
    <anweisungen>]
```

```
for ele in ["Hallo", "Welt"]:
    print(ele)
for idx in range(1, 4):
    print(idx)
for ch in "Hi":
    print(ch)
```

```
Hallo
Welt
1
2
3
H
i
```

Kontrollstrukturen – `for`-Schleife, Dictionaries

`for`-Schleife und Dictionaries

- Fragt Objekte, ob sie auch eine Sequenz sein können
- Das Dictionary sagt, ja, dann bin ich eine Sequenz der Schlüssel
- Dann wird über die Schlüssel iteriert, lazy

Iterieren über items

- `items()` erzeugt Iterator über Tupel
- Tupelzuweisung greift auf Bestandteile zu
- Intuitive nette Syntax

```
1 dic = { 1: 'eins', 2: 'zwei' }
2 for key in dic: # wie dic.keys()
3     print(key, dic[key])
```

```
1, eins
2, zwei
```

```
1 dic = { 1: 'eins', 2: 'zwei' }
2 print(dic.items())
3 for key, value in dic.items():
4     print(key, value)
```

```
dict_items([(1, 'eins'), (2, 'zwei')])
1 eins
2 zwei
```

Kontrollstrukturen – `for`-Schleife, typischer Einsatz

Klassische **`for`**-Schleife

- Durchlaufen von Index-Werten
- Sequenz von Index-Werten, mit `range` erzeugt
- Selten benötigt
- C/C++ Idiom, weniger Python

```
for idx in range(100): # 0-99 in Python
    print(idx)
```

```
for(int idx=0; idx < 100; idx+=1) { // 0-99, C++
    cout << idx << endl;
}
```

Feldzugriff vermeiden

- Keine expliziten Indexvariablen
- Eleganter
- Kein Indexfehler möglich!
- Funktioniert für alle Sequenzen

```
for ele in arr: # Feldzugriff in Python
    print(ele)
```

```
// Feldzugriff in C++
for(size_t idx=0; idx < vec.size(); idx+=1) {
    cout << vec[idx] << endl;
}
```

```
for ele in (0, 1, 2, 3): # alle Sequenzen
    print(ele)
```

Falls Index und Wert notwendig `enumerate`

- Tupel mit Index und Wert
- Auspacken mit Tupelzuweisung

```
for idx, ele in enumerate(("n", "e", "z", "d")):
    print("%d,%s" % (idx, ele), end="  ")
# 0,n   1,e   2,z   3,d
```

Schleifenbeispiel – Dateiinhalte durchlaufen

Dateien mit **while**

- Endlosschleife mit **break**
- Einlesen bis Fehlschlag
- Schließen optional, aber guter Stil

Dateien mit **for**

- Eleganter
- Alles Einlesen in eine Liste

Noch Eleganter

- Default Datei-Aktion ist lesen
- Default ist zeilenweise einlesen
- Automatisch schließen, durch **with** oder da offene Datei nicht gebunden

```
# C-Style            :-(
eingabe = open("dat.txt", "r")
while True:
    line = eingabe.readline()
    if not line:
        break
    print(line.strip())
eingabe.close()
```

```
# Besser             :-|
eingabe = open("dat.txt", "r")
for line in eingabe.readlines():
    print(line.strip())
eingabe.close()
```

```
# Python-Style, alternativ      :-)
for line in open("dat.txt"):
    print(line, end="")
```

```
# Python-Style        :-)
with open("dat.txt") as datei:
    for line in datei:
        print(line, end="")
```

Hinweise zu Kontrollstrukturen

Beim Editieren

- Vergessen Sie die Doppelpunkte nicht
- Beginnen Sie immer in der ersten Spalte
- Rücken Sie konsistent ein (nur Leerzeichen, keine Tabulatoren)
- Rücken Sie immer gleich weit ein (4 Zeichen)
- Am besten mit einer passenden IDE (`code`, eclipse, IDLE, emacs)

Leerzeilen beenden im interaktiven Modus einen Anweisungsblock

Erwarten Sie nicht immer ein Ergebnis

- Beispiel: `append`

Schreiben Sie nicht C/C++ in Python!

Funktionen

Definition mit **def**

- Erstellt Funktionsobjekt und bindet Namen `<fname>` daran
- **return** für Rückgabe, ansonsten automatisch `None`
- Eins oder mehrere Argumente/Parameter `<arg>` werden per Namensbindung übergeben
- Dokumentationsstring `<docstring>` eingebaut, optionale erste Zeile

Interaktiv mit `ipython`

- Leerzeile zum Beenden
- Automatisch Doku mit ?

```
def <fname>([<arg> [, <arg> ...]]):
    [<docstring>]
    <anweisungen>
    [return <name>]
```

```
1 def mult(x, y):
2     "Multipliziere zwei Objekte x, y"
3     return x*y
```

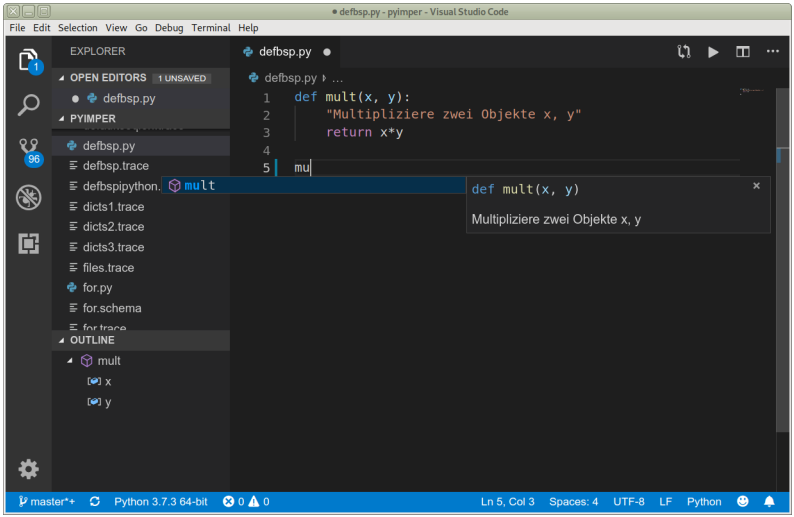
```
In [1]: def mult(x, y):
    "Multipliziere zwei Objekte x, y"
    return x*y
   ...:
In [2]: mult?
Type:       function
String Form:<function mult at 0x7faa65d52f50>
File: /home/6660815/19pyth00/<ipython-input-17>
Definition: mult(x, y)
Docstring:  Multipliziere zwei Objekte x, y
```

Funktionen – Dokumentation

Automatische Extraktion Doku in `code`

- Dokumentationsstring als Doku zugreifbar
- Bei Completion oder Hover

```
def mult(x, y):
    "Multipliziere zwei Objekte x, y"
    return x*y
```



Funktionen – Beispiel, `intersect`

Ziel

- Schnittmenge von Sequenzen
- Abstraktion als Funktion

Lösung

- Python-Funktion `intersect`
- Ergebnis initial eine leere Sequenz
- Für alle Elemente in erster Sequenz
- Suche dieses Element in zweiter Sequenz
- Wenn vorhanden, dann ist es ein gemeinsames Element, also füge es zum Ergebnis hinzu
- Gib Ergebnis zurück

Beispiele

- Strings, Listen, Tupel sind Sequenzen

```
1 def intersect(seq1, seq2):
2     """Schnittmenge von Sequenzen
3        seq1 und seq2"""
4     res = []
5     for ele in seq1:
6         if ele in seq2:
7             res.append(ele)
8     return res
```

```
>>> intersect("spam", "scam")
['s', 'a', 'm']
>>> intersect([1, 2, 3], (4, 2, 6))
[2]
```

Gültigkeitsbereiche in Funktionen

Funktionsaufrufe sind Namensraum

- Jeder Funktionsaufruf bietet einen neuen Gültigkeitsbereich (Namensraum)
- Die Definition auch

Beispiel `fun, val`

- Variable `val` wird nicht verändert
  - Weder bei der Definition
  - Noch bei dem Aufruf
- `val` in der Funktion ist eine neue lokale Variable

```
>>> val = 1
>>> def fun(x, y):
        val = 2
        return x+y

>>> val
1
>>> fun(5, 6)
11
>>> val
1
```

Gültigkeitsbereiche in Funktionen – Durchgriff

Durchgriff auf globale Variable

- Schlüsselwort **`global`**
- Variable vom „globalen“ Namensraum wird verwendet
- Durchgriff immer nur auf äußerste „globale“ Ebene möglich
- Wenn irgend möglich vermeiden

```
>>> val = 1
>>> def fun(x, y):
        global val
        val = 2
        return x+y

>>> val
1
>>> fun(5, 6)
11
>>> val
2
```

Gültigkeitsbereiche Funktionen – Überschreiben

Überschreiben Global/lokal Eigenschaft

- Überschreiben der global/lokal Eigenschaft nicht möglich
- In `fun1` ist `val` als Parameter lokal, kann nicht auch global sein

Nur lesen automatisch möglich

- Direkter Zugriff auf globalen Namensraum möglich wie in `fun2`
- Syntaktisch ok, stilistisch unschön

Nachträglich global machen

- Wenn vorher nur gelesen wurde, dann wurde globaler Wert gelesen, `fun3`
- Wenn vorher geschrieben wurde, dann geht lokaler Wert verloren, `fun4`
- Als Fehler gemeldet seit Python 3.6

```
>>> def fun1(x, val):
        global val
SyntaxError: name 'val' is parameter and global
```

```
>>> val = 1
>>> def fun2(x, y):
        return val+1
>>> fun2(5, 6)
2
```

```
>>> def fun3(x):
        y = val
        global val
SyntaxError: name 'val' is used
prior to global declaration
>>> def fun4(x):
        val = 3
        global val
SyntaxError: name 'val' is
assigned to before global declaration
```

Regeln für Gültigkeitsbereiche

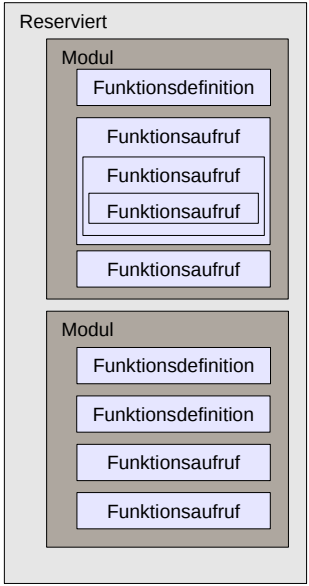
Lokal

- Jeder Funktionsaufruf bietet einen neuen lokalen Gültigkeitsbereich (Namensraum)
- Jeder Aufruf, Rekursion funktioniert
- Zugewiesene Namen sind lokal im aktuellen Gültigskeitsbereich
- können explizit `global` werden
- Blöcke (Einrückungen) sind kein eigener Gültigkeitsbereich

Global

- Das einschließende Modul ist ein globaler Gültigkeitsbereich
- Der interaktive Modus ist ein Modul, hat den Namen `'__main__'`

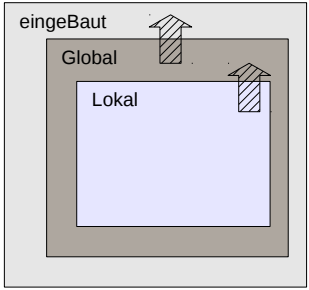
Reservierte Namen sind Teil der Sprache



Namensauflösung, LGB-Regel

Namen in drei Bereichen suchen

- Erst Lokal
  zum Beispiel Variablen und Parameter in `def`
- Dann Global
  zum Beispiel Funktionsdefinition, Variablen auf oberster Ebene
- Dann eingeBaute
  builtin, reservierte Namen, zum Beispiel `len`, `file`



Namensbindungen

- In Funktionen lokal
- In Modulen global
- Für reservierte Namen immer vermeiden

```
>>> val, zwei = 1, 2
>>> def fun():
        val = 3
        print(val, zwei)
>>> fun()
3, 2
>>> val, zwei
(1, 2)
>>> list = "kein Typ mehr, immer vermeiden"
```

Parameterübergabe

Parameterübergabe ist Namensbindung

- Parameternamen werden an übergebene Objekte gebunden
- Neue lokale Namen
- Bindung von Namen an Objekt beeinflusst Objekt nicht

Verwendung

- Call by Reference
- Das Ändern eines änderbaren Objekts hat Auswirkungen (Seiteneffekte)
- Bei unveränderlichen Objekten kann es keine Auswirkungen haben, wie call by value

```
1 def aendern(x, y):
2     x = 2
3     y[1] = 1
```

```
>>> x, y = ([1, 2], [3, 4])
>>> aendern(x, y)
>>> x, y
([1, 2], [3, 1])
```

Parameterübergabe – Vorgabewerte

Vorgabewerte (Default)

- Vorgabewert kann optional bei jedem Parameter angegeben werden
- Zusätzlich = und Ausdruck, der zu Vorgabewert evaluiert
- Parameter mit Vorgabewerten hinter die Parameter ohne Vorgabewerte
- Stellungsparameter, Positionsparameter
- Wie in C++

Aufruf

- Wert für Parameter mit Vorgabewert kann weggelassen werden
- Falls Wert weggelassen, wird Vorgabewert eingesetzt

```
1 def ink(x, a=1):
2     return x+a
```

```
>>> ink(3)
4
>>> ink(3, 1)
4
>>> ink(3, 17)
20
```

Vorgabewerte – Anwendungsbeispiel

Suche von Elementen in Sequenz

- Vorgabewerte für `start` und `ende` des Suchbereichs
- Sinnvolle Vorgabe ist 0 für `start` und Länge der Sequenz für `ende`

Nutzung

- Automatische Belegung von links nach rechts
- Ergänzung nach Position: Um `ende` zu übergeben, muss man auch `start` übergeben

```
1 def suche(seq, ele, start=0, ende=None):
2     if ende is None:
3         ende = len(seq)
4     for pos in range(start, ende):
5         if seq[pos] == ele:
6             return pos
```

```
>>> suche([1, 2, 3, 4, 5, 6], 3)
2
>>> suche([1, 2, 3, 4, 5, 6], 3, 0)
2
>>> suche([1, 2, 3, 4, 5, 6], 3, 4)
>>> suche([1, 2, 3, 4, 5, 6], 3, 0, 6)
2
>>> suche([1, 2, 3, 4, 5, 6], 3, 0, 1)
>>>
```

Vorgabewerte – Besonderheiten

Gültigkeit

- Parametername während der Definition der Vorgabewerte noch nicht nutzbar
- Umgehen durch Markieren mit `None` und dynamische Berechnung

Lebenszyklus

- Vorgabewert wird nur einmal ausgewertet, bei der Funktionsdefinition
- Überraschungen bei änderbaren Vorgabewerten
- Änderungen bleiben über Aufrufe hinweg erhalten
- Vermeiden durch Markieren mit `None` und dynamische Berechnung

```
>>> def suche(seq, ele, start=0, ende=len(seq)):
        for pos in range(start, ende):
            ...
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'seq' is not defined
```

```
>>> def oh(l, a=[]):
        a.append("ende")
        return l+a
>>> oh([1, 2], [3, 4])
[1, 2, 3, 4, 'ende']
>>> oh([1, 2], [3, 4, 5])
[1, 2, 3, 4, 5, 'ende']
>>> oh([1, 2])
[1, 2, 'ende']
>>> oh([1, 2])
[1, 2, 'ende', 'ende']
>>> oh([1, 2])
[1, 2, 'ende', 'ende', 'ende']
```

Parameterübergabe mit Schlüsselwort

Ziel

- Viele Vorgabewerte verwenden, aber einige Parameter anders
- Vermeiden alle früheren Parameter anzugeben

Schlüsselwortparameter

- Parameter über Namen zuweisen
- Name des Parameters als linke Seite einer Zuweisung im Aufruf
- Reihenfolge der Parameter beliebig
- Für alle Argumente außer den Schlüsselwortparametern Vorgabe
- Parameter ohne Vorgabewert müssen als Positions- oder Schlüsselwortparameter übergeben werden

```
1 def suche(seq, ele, start=0, ende=None):
2     if ende is None:
3         ende = len(seq)
4     for pos in range(start, ende):
5         if seq[pos] == ele:
6             return pos
```

```
>>> lis = [1, 2, 3, 4, 5, 6]
>>> suche(lis, 2, start=0, ende=len(lis))
1
>>> suche(lis, 2, 0, len(lis))
1
>>> suche(lis, 2, ende=len(lis))
1
>>> suche(lis, 2, start=1, ende=len(lis))
1
>>> suche(lis, ende=len(lis), ele=2)
1
>>> suche(lis, 2, ende=1)
>>>
```

Argumente sammeln mit * und **

Weitere Positionsparameter *

- Eine Variable am Ende kann mit einem Stern * versehen werden
- In dieser Variable werden dann alle weiteren Positionsparameter als Tupel gesammelt
- Realisiert beliebig viele Parameter, wie zum Beispiel für `print`

Weitere Schlüsselwortparameter **

- Eine Variable ganz am Ende kann mit zwei Sternen ** versehen werden
- Darin werden alle weiteren Schlüsselwortparameter als Dictionary gesammelt
- Die Schlüssel sind die Variablennamen als Strings

```
def params(x, y, *pos, **key):
    print(x, y, pos, key)
```

```
>>> params(1, 2)
1 2 () {}
>>> params(1, 2, 3, 4, 5)
1 2 (3, 4, 5) {}
>>> params(1, 2, 3, 4, drei=3, vier=4)
1 2 (3, 4) {'vier': 4, 'drei': 3}
>>> params(1, 2, 3, 4, vier=4, fuenf=5)
1 2 (3, 4) {'fuenf': 5, 'vier': 4}
>>> params(1, 2, 3, fuenf=5, 4, vier=4)
SyntaxError: positional argument follows
             keyword argument
```

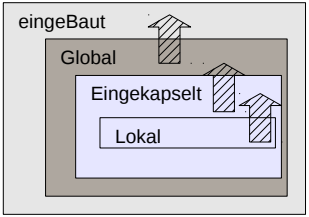
Funktionen in Funktionen und die LEGB-Regel

Funktionsdefinitionen während Funktionsausführung

- Sind erlaubt und sinnvoll
- Umfassen aktuellen Namensraum und halten ihn fest (Enclosing)
- Von innen nach außen

LEGB-Regel

- LGB-Regel erweitert
- Zwischen Lokal und Global
- Man sucht im umschließenden Namensraum, in dem die Funktion definiert wurde
- Dies ist entweder der globale Namensraum oder ein anderer Funktionsaufruf als Namensraum




```
glo 1 2 3
enc 6 5 3
loc 4 5 3
```

```python
x, y, z = 1, 2, 3
def loc():
    x, y = 4, 5
    def enc():
        x = 6
        print("enc", x, y, z)  # 6 5 3
    enc()
    print("loc", x, y, z)      # 4 5 3
print("glo", x, y, z)          # 1 2 3
loc()
```

Design-Prinzipien für Funktionen

Parameter verwenden

- Keine globalen Variablen verwenden
- Flexible Gestaltung der übergebbaren Parameter möglich
- Reduziert Abhängigkeiten

Rückgaben verwenden

- Keine globalen Variablen verwenden
- Modifikation von Parametern meiden
- Mehrere Werte als Tupel zurückgeben
- Modifikation ok in Klassen/Methoden

Verschachtelte Funktionen mit Vorsicht

- Definition einer Funktion in einer Funktion nur wenn notwendig
- Anonyme Funktionen sind meist Alternative

```
x, y = 1, 2
def sehrboese():
    global x
    x = x*y
sehrboese()

x, y = 1, 2
def besser(x, y):
    return x*y
x = besser(x, y)
```

Anonyme Funktion, Lambda-Ausdrücke

Funktionen sind auch Objekte

- Funktionsobjekte sind normale Objekte, „first class citizens“
- Funktionsnamen sind an Funktionsobjekte gebunden
- Definition von Funktionsobjekten und Bindung an einen Namen mit **def**

Funktionsobjekte

- Erzeugen eines anonymen Funktionsobjekte mit **lambda**
- **lambda** <args>: <ausdruck>
  - Funktion mit <args> als Parameter
  - Wert der Funktion durch <ausdruck>, der <args> verwendet
- Zuweisung an Namen und Aufruf möglich

```
>>> def ink(x):
        return x+1
>>> ink(3)
4
>>> ink
<function ink at 0x272a410>
>>> x+1
NameError: name 'x' is not defined
```

```
>>> lambda x: x+1
<function <lambda> at 0xfe2410>
>>> ink2 = lambda x: x+1
>>> ink2
<function <lambda> at 0xfe2488>
>>> ink2(3)
4
>>> (lambda x: x+1)(3)
4
```

Funktionales Programmieren

Funktionales Programmieren, Funktionsaufrufe

- Programm als applikativer Algorithmus, sukzessive Reduktion von Ausdrücken
- Nur Funktionsaufrufe, statt Kontrollstrukturen
- Primitive zum flexiblen Aufrufen von Funktionen mit Parametern notwendig

Flexible Parameter

- Erweitere Call Syntax
- $*$ für Positionsparameter, „ausbreiten“ der Parameter
- $**$ für Schlüsselwortparameter

```
>>> def plus(x, y):
        return x+y

>>> tup = (2, 4)
>>> plus(tup)
plus() missing 1 required
       positional argument: 'y'
>>> plus(*tup)
6
>>> ptup = (2, )
>>> plus(*(ptup + (4,)))
6
>>> dic = {'x':2, 'y':4}
>>> plus(**dic)
6
```

Funktionale Primitive – `map`

Funktion auf alle Elementen anwenden

- Für alle Objekte von Sequenzen eine Funktion anwenden
- Schlüsselwort `map`
- Erster Parameter die Funktion
- Ab zweiten Parameter die Sequenzen, so viele Sequenzen wie die Funktion Parameter hat
- Ergebnis ist iterierbares Objekt mit Funktionsergebnis je Element der Eingabesequenz
- Lazy, mit `list` eine Liste daraus machen
- Längste Sequenz bestimmt Länge des Ergebnisses, Eingabe ergänzen mit `None`

Beispiele

- Erhöhe alle Werte um 1
- Addiere Inhalte einer Liste paarweise
- In einer Schleife ist lazy gut

```
>>> def ink(x):
        return x+1
>>> map(ink, [1, 2, 3]) # lazy
<map object at 0x7f537b2e8668>
>>> list(map(ink, [1, 2, 3]))
[2, 3, 4]
>>> list(map(lambda x: x+1, [1, 2, 3]))
[2, 3, 4]
>>> list(map(lambda x: x+3, [1, 2, 3]))
[4, 5, 6]
>>> list(map(lambda x,y: x+y,
             [1, 2, 3], [4, 5, 6]))
[5, 7, 9]
>>> for e in map(ink, [1, 2, 3]): print(e)
...
2
3
4
```

Einsatzbeispiel für `map`

Aufgabe

- Berechne Liste der Quadratzahlen von 1 bis 10

Lösung

- `for`-Schleife
- Liste zusammenbauen
- Zurückgeben

Funktionale Lösung

- Einzeiler mit `map`
- Liste der Zahlen mit `range`
- Quadrieren als Lambda-Funktion
- Keine Schleifen
- Lazy, bei Bedarf eager mit `list`

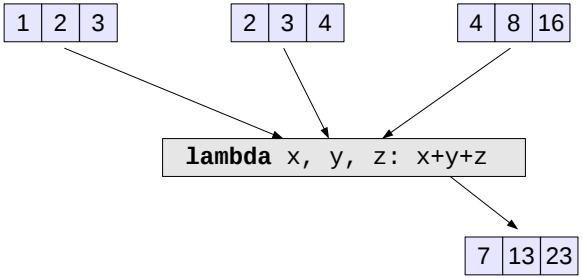
```
1 def quadratzahlen(von=1, bis=10):
2     lis = []
3     for i in range(von, bis+1):
4         lis.append(i**2)
5     return lis
6
7 print(quadratzahlen())
```

```
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

```
1 def quadratzahlen(von=1, bis=10):
2     return map(lambda x: x**2, range(von, bis+1))
3
4 print(list(quadratzahlen()))
```

map illustriert

```
>>> list(map(lambda x, y, z: x+y+z, [1, 2, 3], [2, 3, 4], [4, 8, 16]))
[7, 13, 23]
```



Funktionale Primitive – `filter`

Nur die Elemente, für die eine Funktion ok liefert

- Für alle Objekte einer Sequenz eine Funktion anwenden und nur die durchlassen, für die der Funktionswert äquivalent zu wahr ist
- Schlüsselwort `filter`
- Erster Parameter die Funktion
- Zweiter Parameter die Sequenz
- Ergebnis ist iterierbares Objekt der ursprünglichen Elemente, deren Funktionswert äquivalent zu wahr ist

Beispiele

- Teste, ob Zahl größer 0
- Teste, ob Zahl zwischen 1 und 2
- Teste, ob gerade Zahl

```
>>> def groesser0(x):
        return x>0
>>> filter(groesser0, [1,-1,2,-2,3,-3,4])
<filter object at 0x7f537b2e86a0>
>>> list(filter(groesser0,
                [1, -1, 2, -2, 3, -3, 4]))
[1, 2, 3, 4]
>>> list(filter(lambda x: x>0,
                [1, -1, 2, -2, 3, -3, 4]))
[1, 2, 3, 4]
>>> list(filter(lambda x: x>0 and x<3,
                [1, -1, 2, -2, 3, -3,  4]))
[1, 2]
>>> list(filter(lambda x: x%2 == 0,
                [1, 2, 3, 4, 5]))
[2, 4]
```

Einsatzbeispiel für `filter`

Aufgabe

- Berechne die Liste der Primzahlen zwischen 2 und 100

Lösung

- `for`-Schleife
- Liste zusammenbauen
- Zurückgeben

Funktionale Lösung

- Einzeiler mit `filter`
- Liste der Zahlen mit `range`
- Funktion testet, ob Zahl Primzahl ist, wie bei `for`-Schleife
- Keine Schleifen
- Lazy, bei Bedarf eager mit `list`

```
def is_prim(zahl):
    for teiler in range(2, zahl):
        if zahl%teiler == 0:
            return False
    return True
def prims():
    primzahlen = []
    for zahl in range(2, 100):
        if is_prim(zahl):
            primzahlen.append(zahl)
    return primzahlen
print(prims())
```

```
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29,
 31, 37, 41, 43, 47, 53, 59, 61, 67,
 71, 73, 79, 83, 89, 97]
```

```
def prims():
    return filter(is_prim, range(2, 100))
print(list(prims()))
```

Funktionale Primitive – reduce

Alle Elemente einer Sequenz mit einer Funktion kombinieren

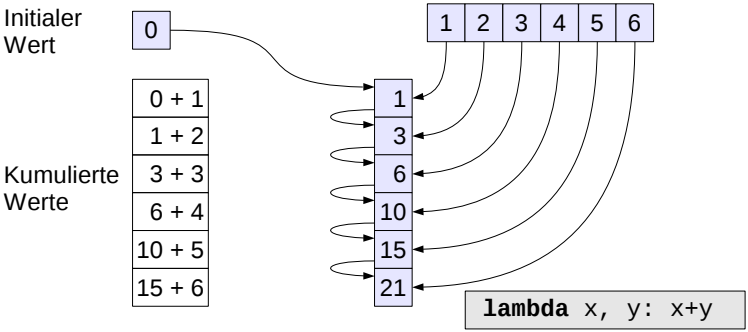
- Kumuliere alle Objekte einer Sequenz über Funktionswerte auf (`foldl`)
- Kein Schlüsselwort mehr: `reduce`
  - Seit Python3 bei BDFL(GvR) in Ungnade gefallen
  - Muss aus `functools` importiert werden
- Erster Parameter: die Funktion
- Zweiter Parameter: die Sequenz
- Dritter Parameter: der Anfangswert, meist neutral
- Ergebnis ist Auswertung der linksassoziativen sukzessiven Funktionsanwendung

Beispiele: Summieren, Fakultät, Stringkonkatenation

```
>>> def add(x, y):
        return x+y
>>> def summe(lis):
        ret = 0
        for e in lis:
            ret = ret + e
        return ret
>>> summe(range(101))
5050
>>> from functools import reduce
>>> reduce(lambda x,y : x+y, range(101), 0)
5050
>>> fak = lambda x: reduce(lambda x,y: x*y,
                           range(1, x+1), 1)
>>> fak(3), fak(6)
(6, 720)
>>> reduce(lambda x,y : x+" "+y,
           ["Hallo", "tolle", "Welt"], "")
' Hallo tolle Welt'
```

reduce illustriert

```
>>> add = lambda x,y: x+y
>>> reduce(add, [1, 2, 3, 4, 5, 6], 0)
21
>>> add(add(add(add(add(add(0, 1), 2), 3), 4), 5), 6) # foldl
21
```



Definition von `map`

Primitive für funktionales Programmieren

- Können selbst implementiert werden
- Keine Schlüsselwörter, nur vordefiniert

Eager `mymap` selbst gemacht

- `for`-Schleife
- Funktion als Parameter
- Aufbau der Ergebnisliste

Verwenden wie eingebautes `map`

- Implementierung mit mehreren Sequenzen auch einfach möglich
- Gute Hausaufgabe

Lazy `mymap` selbst gemacht

- Noch einfacher
- aber neues Schlüsselwort **`yield`**, später

```
def mymap(func, seq): # eager
    ret = []
    for ele in seq:
        ret.append(func(ele))
    return ret
```

```
>>> ink = lambda x: x+1
>>> list(map(ink, [1, 2, 3]))
[2, 3, 4]
>>> mymap(ink, [1, 2, 3])
[2, 3, 4]
>>> mymap(lambda x: x+1, [1, 2, 3])
[2, 3, 4]
```

```
def mymap(func, seq): # lazy
    for ele in seq:
        yield func(ele)
```

Definition von `filter`

Eager `myfilter` selbst gemacht

- `for`-Schleife
- Funktion als Parameter
- Aufbau der Ergebnisliste

Verwenden wie eingebautes `filter`

```
def myfilter(func, seq): # eager
    ret = []
    for ele in seq:
        if func(ele):
            ret.append(ele)
    return ret
```

```
>>> list(filter(lambda x: x>10, range(20)))
[11, 12, 13, 14, 15, 16, 17, 18, 19]
>>> list(filter(lambda x: x%2, range(20)))
[1, 3, 5, 7, 9, 11, 13, 15, 17, 19]
>>> myfilter(lambda x: x>10, range(20))
[11, 12, 13, 14, 15, 16, 17, 18, 19]
>>> myfilter(lambda x: x%2, range(20))
[1, 3, 5, 7, 9, 11, 13, 15, 17, 19]
>>> myfilter(lambda x: x%2==0, range(20))
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

Lazy `myfilter` selbst gemacht

```
def myfilter(func, seq): # lazy
    for ele in seq:
        if func(ele):
            yield(ele)
```

Definition von reduce

`myreduce` selbst gemacht

- `for`-Schleife
- Funktion als Parameter
- Aufbau des Ergebniswerts
- Ist immer eager

Verwenden wie eingebautes `reduce`

```
1 def myreduce(func, seq, init=None):
2     if init is None:
3         init = seq[0]
4         seq = seq[1:]
5     wert = init
6     for ele in seq:
7         wert = func(wert, ele)
8     return wert
```

```
>>> add = lambda x, y: x+y
>>> reduce(add, range(101))
5050
>>> myreduce(add, range(101))
5050
>>> sub = lambda x, y: x-y
>>> myreduce(sub, [1, 2, 3, 4, 5], 100)
85
>>> reduce(sub, [1, 2, 3, 4, 5], 100)
85
```

List Comprehension

Listentransformation

- Anwendungsgebiet wie funktionales Programmieren
- Einfachere Verwendung der Möglichkeiten von `map` und `filter` zusammen mit **`lambda`** und **`for`**

Syntax `[<ausdruck> for <var> in <seq>]`

- `<ausdruck>` statt **`lambda`**-Ausdruck
- Durch **`for`** wird `<var>` sukzessive ein Wert aus `<seq>` zugewiesen
- Zusätzlich **`if`** `<bedingung>`, als Filter
- Beliebig kombinierbar

Ausdrucksweisen

- Statt verschachtelte **`for`**-Schleifen
- Wie mathematische Mengenschreibweise
- Da die Reihenfolge relevant ist eigentlich Sequenzen, mit `(...)` oder `<...>`

```
>>> list(map(lambda x: x+1, [1, 2, 3, 4]))
[2, 3, 4, 5]
>>> [x+1 for x in [1, 2, 3, 4]]
[2, 3, 4, 5]
>>> list(filter(lambda x:x>3, [1, 5, 2, 6, 7]))
[5, 6, 7]
>>> [x for x in [1, 5, 2, 6, 7] if x>3]
[5, 6, 7]
```

```
>>> [x**2 for x in range(1, 11)
          if x % 2 == 0]
[4, 16, 36, 64, 100]
```

$$\{x^2 \mid 1 \leq x \leq 10, x \bmod 2 = 0\} = \{4, 16, 36, 64, 100\}$$

$$(x^2 \mid 1 \leq x \leq 10, x \bmod 2 = 0) = (4, 16, 36, 64, 100)$$

List Comprehension – Beispiele mit Zahlen

Quadratzahlen

- Zahlen von 0 bis 9; 1 bis 10
- Ausdruck statt Funktion

Komplexe Definition

- Mischen von Filtern und Funktionsapplikation
- Aufzählen durch `for`

Kartesisches Produkt

- Sequenzen kombinieren
- Kartesisches Produkt, Vollkombination
- Nicht so bei funktionalem Primitiv `map`

`zip`

- Builtin, generiert Tupel
- Lazy

```
>>> [x for x in range(10)]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> [x*x for x in range(1, 11)]
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

```
>>> [(x,y) for x in range(4)
           for y in range(0,5,2)
           if x < y]
[(0, 2), (0, 4), (1, 2), (1, 4), (2, 4), (3, 4)]
```

```
>>> list(map(lambda x,y: x+y, range(5), range(5)))
[0, 2, 4, 6, 8]
>>> [x+y for x in range(5) for y in range(5)]
[0, 1, 2, 3, 4, 1, 2, 3, 4, 5, 2, 3, 4, 5, 6,
 3, 4, 5, 6, 7, 4, 5, 6, 7, 8]
```

```
>>> [x+y for (x,y) in zip(range(5),range(5))]
[0, 2, 4, 6, 8]
>>> list(zip(range(5), range(5)))
[(0, 0), (1, 1), (2, 2), (3, 3), (4, 4)]
```

List Comprehension – Beispiele mit Text

Wörter ersetzen

- Kleine Zahlen durch Wörter ersetzen
- Für alle Worte, falls das Wort 1,2 oder 3 ist,
- dann ersetze durch Zahlwort und setze neuen String zusammen

Datei von Zahlen zeilenweise lesen

- Zeilenweise einlesen und aufteilen (splitten)
- Alle Strings aus Ziffern je Zeile in Liste von Zahlen umwandeln
- Tupel aus Liste machen

```
>>> trans = {'2': 'zwei', '3': 'drei',
             '4': 'vier'}
>>> s = """2 Sachen und 3 Sachen sind
nicht 4 Sachen"""
>>> print(" ".join(
     [trans[w] if w in trans else w
        for w in s.split()]))
zwei Sachen und drei Sachen sind
nicht vier Sachen
```

```
>>> d = "zahlen.dat"
>>> open(d).readlines()
['1 2 3\n', '42      343  12\n',
 '8   77 223\n', '23 43\n',
 '45 3   3 66 7\n']
>>> [tuple(map(int, line.split()))
              for line in open(d)]
[(1, 2, 3), (42, 343, 12), (8, 77, 223),
 (23, 43), (45, 3, 3, 66, 7)]
```

Eager und Lazy

Eager

- Mit []
- Default

Lazy

- Mit ()
- Wieder mit `list` auflösen

Eager versus lazy

- Eager erzeugt die Liste im Speicher
- Eager iteriert über die erzeugte Liste
- Lazy erzeugt nur was iterierbares
- Lazy generiert immer nur das nächste zurückzugebende Element
- Wann immer es funktioniert lazy verwenden, sonst eager
- Lazy häufiger default in Python3

```
>>> [x**2 for x in range(1, 11)] # eager
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
>>> (x**2 for x in range(1, 11)) # lazy
<generator object <genexpr> at 0x7f2ef2381b48>
>>> list((x**2 for x in range(1, 11)))
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

```
>>> l = [i for i in range(2**25)] # >1 GB Spei
>>> print(sum(l))
562949936644096
>>> del l # <10 MB Speicher
>>> l = (i for i in range(2**25))
>>> print(sum(l))
562949936644096
```

```
>>> # auf raspberrypi2, 32 Bit, 1 GB swapoff
>>> l = (i for i in range(2**27)) # schnell
>>> print(sum(l)) # dauert
9007199187632128
>>> l = [i for i in range(2**27)] # dauert
Killed
```

Funktionale Programmierung – Zusammenfassung

Vorteile

- Verminderung der Fehleranfälligkeit, zum Beispiel sind viele Indexfehler gar nicht mehr formulierbar
- Kompakter, ausdrucksstarker Code nahe an der Spezifikation
- Gut zu lesen und zu verstehen, wenn nicht zu viel in einer Zeile passiert
- Seiteneffektfrei, wenn man selbst etwas darauf achtet

Nachteile

- Syntax/Ausdrucksweise etwas gewöhnungsbedürftig
- Funktionales Denken erfordert etwas Einarbeitungszeit

Verbreitung – wieder sehr modern

- Funktionale Primitive in allen höheren Programmiersprachen zu finden (Ruby, Perl, Lisp, C#, C++)
- Lambda-Ausdrücke sind in C# für LINQ, in Java ab 8 (Stream API/Lazy), in C++ ab C++11
- MapReduce, Framework (Google) zum Suchen in riesigen Datenmengen

Online-Referenzen

- `docs.python.org/3/howto/functional.html`
- `developer.ibm.com/articles/l-prog/`
- `www.oreilly.com/programming/free/functional-programming-python.csp`

Module

Warum Module?

- Wiederverwendung von Code
- Bereitstellung von allgemein nutzbaren Diensten und Bibliotheken
- Unterteilung des Namensraums bei großen Projekten

Python Module

- Jede Python-Datei (Endung mit `.py`) ist ein Modul
- C-Erweiterungen sind ein Modul
- Nutzen mit **`import`** und **`from import`**

Suche nach Dateien/Module

- Umgebungsvariable `PYTHONPATH`
- Einträge in `sys.path`

```
>>> import sys
>>> sys.path
['', # sucht auch im aktuellen Verzeichnis
 # sucht in speziellen Zips
 '/usr/lib/python36.zip',
 '/usr/lib/python3.6',
 '/usr/lib/python3.6/lib-dynload',
 '/usr/local/lib/python3.6/dist-packages',
 '/usr/lib/python3/dist-packages']
```

Definieren von Modulen

Ein Modul ist eine Datei

- Mit der Endung `.py`
- Optional vorkompiliert
  - Unterordner `__pycache__`
  - `__pycache__/<modul>.<version>.pyc`
  - `<version>` zum Beispiel `cpython-36`
- In einem Verzeichnis
- Beliebiger Python-Code
- Keine Python-Schlüsselwörter als Dateiname

Im Suchpfad erreichbar

- Vorgabe ist aktuelles Verzeichnis und Standard-Bibliothek von Python
- `sys.path`

meinmodul.py

```
# Modul meinmodul, meinmodul.py
# Variablen und Funktionen

DEUTSCH = "deutsch"
ENGLISCH = "englisch"

SPRACHE = DEUTSCH

def drucke(x):
    print(x)

def gruss():
    if SPRACHE == DEUTSCH:
        drucke("Hallo Modul Welt")
    else:
        drucke("Hello module world")

print("meinmodul geladen")
```

Importieren von Modulen

Modul importieren

- Im aktuellen Verzeichnis
  - Wechseln mit `os.chdir()`
- Importiert Modul mit dem Namen `<modul>`
- Objekte nur über den Modul-Präfix erreichbar

Namen von Modulen importieren

- Modul selbst nicht sichtbar
- Alle Namen `<name>` im aktuellen Namensraum verfügbar
  - Aber an Objekt in Modul gebunden
- `*` für alle Namen außer Namen, die mit einem Unterstrich `\_` beginnen

```
import <modul> [, <modul> ]*
```

```
$ ls meinmodul.py
meinmodul.py
$ python3
>>> import meinmodul
meinmodul geladen
>>> meinmodul.gruss()
Hallo Modul Welt
>>> meinmodul.SPRACHE = meinmodul.ENGLISCH
>>> meinmodul.gruss()
Hello module world
>>> meinmodul.SPRACHE = ENGLISCH
NameError: name 'ENGLISCH' is not defined
```

```
from <modul> import <name> [, <name> ]*
```

```
>>> from meinmodul import gruss
>>> gruss()
Hallo Modul Welt
```

Module und Namensräume

Ausführung beim Import

- Modulanweisungen laufen nur beim ersten Import einmal ab
- Achtung bei Interaktion, Module werden nicht automatisch neu geladen

Modulattribute

- Zuweisungen und Funktionsdefinitionen auf der obersten Ebene erstellen Modulattribute
- Namen, die an Objekte gebunden sind

Module sind Namensräume

- Ein Modul ist globaler Namensraum
- Auf den Namensraum eines Moduls können Sie zugreifen
  `dir(<modul>)`
  `<modul>.__dict__`

```
>>> import meinmodul
meinmodul geladen
>>> dir(meinmodul)
['DEUTSCH', 'ENGLISCH', 'SPRACHE',
 '__builtins__', '__cached__', '__doc__', '__file__'
 '__loader__', '__name__', '__package__', '__spec__'
  'drucke', 'gruss']
>>> import meinmodul
>>> meinmodul.__dict__
{'drucke': <function drucke at 0x7f4b7f2d8488>,
 'DEUTSCH': 'deutsch',
 '__builtins__': { ....
```

Hauptprogramm mit `main`

Ziel

- Ausführen eines Code-Blocks nur, wenn es als Hauptprogramm gestartet wurde, nicht wenn es importiert wurde
- Simulieren der `main`-Funktion/Methode von C/C++
- Häufig zum Testen eines Moduls verwendet

Umsetzung, Idiom

- Der Name des interaktiven oder Haupt-Moduls ist `'__main__'`
- Test auf Name und bedingtes Ausführen

main.py

```
1 import sys
2
3 def drucke(x):
4     print("|%s|" % x)
5
6 if __name__ == '__main__':
7     print("drucker")
8     for ele in sys.argv[1:]:
9         drucke(ele)
```

```
>>> import main
>>> main.drucke("hallo")
|hallo|
>>> main.drucke("hello")
|hello|
```

```
$ python3 main.py hallo hello
drucker
|hallo|
|hello|
```

Qualifikation von Namen

Einfacher Name

- Auflösung über LEGB-Regel

Qualifikation

- `<modul>.<name>`
- Im Geltungsbereich des Moduls
- Die gleiche Notation wie bei Objekten

Qualifikationspfade

- `<modul>.<modul>.<name>`
- Nur im genau spezifizierten Geltungsbereich

Qualifikation funktioniert für alle Objekte:
Module, Klassen, Typen, . . .

qual.py

```
1 import os
2 name = "Name"
```

```
>>> import qual
>>> name
NameError: name 'name' is not defined
>>> qual.name
'Name'
>>> os.getcwd()
NameError: name 'os' is not defined
>>> qual.os.getcwd()
'/home/student/nethome/19pyth00'
```

Erzwingen von Neuladen

Import von Modulen

- Module werden beim ersten Import ausgeführt
- Beim zweiten Importieren wird nur ein neuer Namen wieder an das Modulobjekt gebunden
- Zweiter Import sinnvoll, um Modul im aktuellen Namensraum zu haben

`importlib.reload(<modul>)`

- Lädt Modul neu, erstellt neues Modulobjekt
- Wichtig bei interaktiver Entwicklung, laufender Python-Interpreter lädt Modul nicht automatisch neu
  - Weniger wichtig bei IDEs, einfach Neustart
- Code-Änderungen werden erst beim Neuladen (`reload`, oder Neustart Interpreter) wirksam

m1.py

```
1 import meinmodul
```

m2.py

```
1 import meinmodul
```

m3.py

```
1 import m1
2 import m2
3 print(m1.meinmodul.gruss)
4 print(m2.meinmodul.gruss)
5 from importlib import reload # seit Python3.4
6 m2.meinmodul = reload(m2.meinmodul)
7 print(m2.meinmodul.gruss)
```

```
$ python3 m3.py
meinmodul geladen
<function gruss at 0x7f8266392578>
<function gruss at 0x7f8266392578>
meinmodul geladen
<function gruss at 0x7f8266392488>
```

Warum nicht `from <modul> import *`

Verunreinigung des Namensraums

- Namenskonflikte möglich
- Gleiche Namen in Modulen würden im importierenden Modul überschrieben, der Letzte gewinnt

Importieren ist Namensbindung

- Ist nicht immer was erwartet wird
- Im Beispiel wird der Name `sprache` importiert
- Die Funktion `gruss` verwendet natürlich noch `sprache` im Modul `meinmodul`

`from <modul> import *` vermeiden

Wenn Modulname zu lang, `as`

```
>>> from meinmodul import *
meinmodul geladen
>>> gruss()
Hallo Modul Welt
>>> SPRACHE = ENGLISCH
>>> gruss()
Hallo Modul Welt
>>> import meinmodul
>>> meinmodul.SPRACHE = ENGLISCH
>>> gruss()
Hello module world
```

```
import <modul> as <name>
```

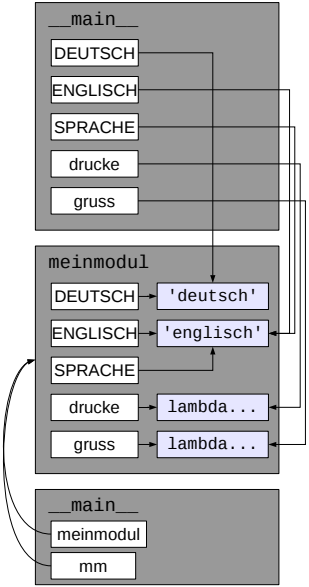
```
>>> import meinmodul as mm
meinmodul geladen
>>> mm.gruss()
Hallo Modul Welt
```

Warum nicht `from <modul> import *`

```
>>> from meinmodul import *
meinmodul geladen
>>> gruss()
Hallo Modul Welt
>>> SPRACHE = ENGLISCH
>>> gruss()
Hallo Modul Welt
>>> import meinmodul
>>> meinmodul.SPRACHE = ENGLISCH
>>> gruss()
Hello module world
```

```
import <modul> as <name>
```

```
>>> import meinmodul as mm
meinmodul geladen
>>> mm.gruss()
Hallo Modul Welt
```



Datenkapselung und Daten verstecken

Datenkapselung

- Kapselung ist Ziel von Modulen
- Nicht erzwungen in Python, kein `public`, `private`, `protected`
- Nur „unhöflich“, gegen Konvention

Daten verstecken

- Konvention mit _
- Durchgriff möglich
- Nicht importiert bei *

Konventionen

- Keine Modulattribute ändern, nur lesen
- Nur Funktionen verwenden
- Keine Funktionen mit _ am Anfang

meinmodul.py (Erweiterung)

```
1 #...
2 def andere_sprache(s):
3     global SPRACHE
4     if s in (DEUTSCH, ENGLISCH):
5         SPRACHE = s
6     else:
7         raise Exception("Sprache %s" % s)
```

```
>>> import meinmodulgross as mm
>>> mm.gruss()
Hallo Modul Welt
>>> mm.andere_sprache(mm.ENGLISCH)
>>> mm.gruss()
Hello module world
>>> mm.andere_sprache("portugiesisch")
Exception: Sprache portugiesisch
```

Pakete/Packages – Ordner als Module

Flache Struktur

- Viele Module flach im Ordner ok
- In großem Projekt nicht ausreichend
- Standardbibliotheken so nicht gut strukturierbar
- Besser ist Hierarchie von Ordnern

Package

- Ordner als Hierachiestufen bei der Namensqualifikation
- Wenn ein Ordner die Datei `__init__.py` enthält, dann ist der Ordner ein Package
- `__init__.py` kann Initialisierungscode des Package enthalten, zum Beispiel Module nachladen und Namensraum ausgestalten

paket/eins.py

```
1 MSG = "hallo"
2 def machgross(msg):
3     return msg.upper()
```

paket/zwei.py

```
1 import string
2 def machcap(msg):
3     return string.capwords(msg)
```

paket/__init__.py

```
1 # relativer import
2 from . import eins
3 from .eins import MSG
```

Relative und Absolute Imports

Python3, relative und absolute imports

- `import` statements absolut, suchen in `sys.path`
- Relativ zum aktuellen Verzeichnis suchen (in einem Paket) mit relative imports
  - `.` im aktuellen Verzeichnis,
  - `..` eins darüber,
  - `...` zwei darüber
- In Paket Untermodul/-paket relativ nachladen

Python3, `__init__.py`

```
# relativer import
from . import eins
from .eins import MSG
```

Python2, anders

- Keine Syntax für relative/absolute imports
- Sucht immer in `sys.path`
- In Paket als sei Paketpfad anfangs in `sys.path`
- In Paket Untermodul/-paket nachladen

Python2, `__init__.py`

```
# aktuelles Verzeichnis wie am
# Anfang von sys.path
import eins
from eins import MSG
```

Pakete/Packages – Beispiel Verwendung

paket/eins.py

```
1 MSG = "hallo"
2 def machgross(msg):
3     return msg.upper()
```

paket/zwei.py

```
1 import string
2 def machcap(msg):
3     return string.capwords(msg)
```

paket/__init__.py

```
1 # relativer import
2 from . import eins
3 from .eins import MSG
```

```
>>> import paket
>>> paket.eins
<module 'paket.eins' from 'paket/eins.py'>
>>> paket.eins.machgross("ab")
'AB'
>>> paket.eins.MSG, paket.MSG
('hallo', 'hallo')
>>> paket.zwei
  AttributeError: 'module' object has
    no attribute 'zwei'
>>> import paket.zwei
>>> paket.zwei
<module 'paket.zwei' from 'paket/zwei.py'>
>>> paket.zwei.string.capwords("hallo welt")
'Hallo Welt'
```

Objektorientierung in Python

Wie bei C++ nur Zeiger, alles **`virtual`**

- Schlüsselworte, Referenzsemantik
- Attribute, Vererbung, Polymorphismus
- Einfach, Sprache mit OO konzipiert

Unterschiede, neue Features

- Dynamische Attribute, Attributsuche
- Mehrfachvererbung, keine Schnittstellen
- Kapselung (Komposition) nur per Konvention, aber Properties
- Operatorenüberladung, Duck-Typing (reagieren auf Index-Zugriff, Ausgabe, Slicen)
- Selbst erstellte Klassen verhalten sich wie eingebaute Typen
- Alles! ist ein Objekt (Zahlen, Module, . . .)

```
>>> i=3
>>> type(i)
<class 'int'>
>>> isinstance(i, int)
True
>>> i.__add__
<method-wrapper '__add__'
 of int object at 0x2152788>
>>> i.__add__(4)
7
>>> i.__mul__(4)
12
>>> object
<class 'object'>
>>> issubclass(int, object)
True
```

Klassen

Klassendefinition

- `class`: Schlüsselwort
- `<klassenname>`: Name für die Klasse
- `object`: Wurzelklasse, Angabe optional
- `<docstring>`: Dokumentationsstring, optional
- `<definitionen>`: Funktionsdefinitionen für Methoden

Besonderheiten

- Erstes Methodenargument ist **immer** Instanzobjekt
  - Per Konvention der Name `self`
  - in C++ wäre es `this`, aber man **darf es nicht** angeben
- Spezielle Methode `__init__` statt Konstruktor
- Attribute sind Klassenattribute und nicht, wie von C++ erwartet, Instanzattribute
- Instanzattribute dynamisch (`__init__`) in Namensraum `self` schreiben

```
class <klassenname>(object):
      [<docstring>]
      <definitionen>
```

klasse.py

```
1 class AClass(object):
2     "Eine Klasse AClass"
3     def __init__(self):
4         self.aval = 17
5     def inca(self):
6         self.aval += 1
7 print(AClass)
```

```
$ python3 klasse.py
<class '__main__.AClass'>
```

Ausführung erzeugt Klassenobjekt (Klassen sind Objekte!), binden Klassenname an Klassenobjekt

Klassendefinition – Beispiel Stack

Stack auf Basis von Listen (Komposition)

- Initialisierung der Instanzvariable `liste` mit leerer Liste
- `self` ist immer das erste Argument bei Methoden
- `self` muss explizit aufgeführt werden bei der Definition
- `self` ist Namensraum einer Instanz
- Weitere Argumente bei Methoden erlaubt
- Manipulation der Instanzattribute in `self` sollte nur durch Methoden erfolgen
  - Keine `get/set` Methoden schreiben!
  - Später mit Properties eingreifen

```python
class Stack:
    "Stack-Klasse mit Python Listen"

    def __init__(self):
        self.liste = []

    def push(self, elem):
        self.liste.append(elem)

    def pop(self):
        del self.liste[-1]

    def top(self):
        return self.liste[-1]

    def clear(self):
        self.liste = []

    def empty(self):
        return self.liste == []
```

Generieren und Nutzen von Instanzen

Klassendefinition

- Klassenname ist Name gebunden an Klassenobjekt
- Im Beispiel `Stack` in `stack.py`

Instanziierung

- Neue Instanz durch Aufruf Klassennamen
  - Kein new
  - Implizit wird `__init__` aufgerufen
- Zuweisung der Instanz an Namen
  - Name referenziert dann Instanzobjekt

Nutzen wie gewohnt

- Kapselung nicht erzwungen aber Konvention

```
>>> from stack import Stack
>>> Stack
<class 'stack.Stack'>
>>> st = Stack()
>>> st
<stack.Stack object at 0x7f8a3b604748>
>>> st.empty()
True
>>> st.push(1); st.push(2); st.push(3)
>>> # nicht machen, Durchgriff Kapselung
>>> st.liste
[1, 2, 3]
>>> st.empty()
False
>>> st.top()
3
>>> st.pop()
>>> st.top()
2
```

Vererbung und Mehrfachvererbung

Vererbung

- Superklassen in Klammern dahinter
- Mehrfachvererbung möglich

Mehrfachvererbung

- Suche nach Attribut oder Methodenname
  - Erst in aktueller Klasse
  - Dann in Superklassen wie definiert
  - Erst alle Superklassen der Superklassen
- Kurz: von links nach rechts, Tiefensuche*
- Vermeide das mehrmalige Auftauchen derselben Klasse in der Vererbungshierarchie
  - Kurz: Baum, nicht Graph

```
class <name>(<superkl> [, <superkl> ]*):
        ...
```

fancystack.py

```
1 from stack import Stack
2
3 class FancyStack(Stack):
4     def peek(self, idx):
5         return self.liste[idx]
```

```
>>> from fancystack import FancyStack
>>> st = FancyStack()
>>> st
<fancystack.FancyStack object at 0x1c31f90>
>>> st.push(1); st.push(2); st.push(3)
>>> st.peek(2)
3
>>> st.peek(0)
1
```

*In Wirklichkeit etwas komplizierter
www.python.org/download/releases/2.3/mro/

Klassen- und Instanzvariablen

Instanzvariablen

- Zuweisung an durch `self` aufgespannten Namensraum
- Unabhängig von allen anderen Namen
- Entspricht Instanzvariablen in C++

Klassenvariablen

- Initialisierung als Attribute
- In durch Klassennamen aufgespannten Namensraum
- Entspricht `static` Variablen in C++
- Achtung: Syntax wie Instanzvariablen in C++

```
1 class Klasse:
2     verbose = 1
3     def __init__(self, verb):
4         self.verbose = verb
5     def show_verbose(self):
6         print("Klasse %s" % str(Klasse.verbose))
7         print("Instanz %s" % str(self.verbose))
```

```
>>> Klasse.verbose
1
>>> instanz = Klasse(2)
>>> instanz.verbose
2
>>> Klasse.verbose
1
>>> instanz.show_verbose()
Klasse 1
Instanz 2
```

Klassen- und Instanzvariablen – Attributsuche

Suchordnung

- Erst Instanz, dann Klasse, dann Superklassen

Suche in `self`

- Wenn keine Instanzvariable vorliegt, wird die der Klasse gelesen

Zur Laufzeit änderbar

- Instanzvariable zur Laufzeit hinzufügbar
- Einfache Zuweisung, keine Deklaration

Klassenvariable nur noch qualifiziert

- Zugriff über Namensraum der Klasse

```
1 class Klasse:
2     verbose = 1
3     def __init__(self):
4         pass
```

```
>>> instanz = Klasse()
# Suche in self
>>> instanz.verbose
1
# Zur Laufzeit änderbar
>>> instanz.verbose = 2
>>> instanz.verbose
2
>>> # Klassenvariable
>>> # nur noch qualifiziert
>>> Klasse.verbose
1
```

Klassenmethoden

Statische Methoden

- Klassenmethode, bezieht sich nicht auf Instanz, in C++ **`static`**
- Syntax: Dekorator `@staticmethod`
- Kein erstes Argument `self` mehr vorhanden

Klassenmethode

- Klassenmethode, wobei das erste Argument die Klasse ist
- Syntax: Dekorator `@classmethod`
- Überschreiben möglich, fortgeschrittenes Konzept, erst mal vermeiden

```
class Klasse:
    zaehle = 1
    def __init__(self):
        self.zahl = 17
    @staticmethod
    def statisch(x): # kein self mehr
        Klasse.zaehle += x
    @classmethod
    def klassenmethode(cls, x):
        cls.zaehle += x
```

```
>>> instanz = Klasse()
>>> instanz.zaehle, Klasse.zaehle
(1, 1)
>>> instanz.statisch(3)
>>> Klasse.zaehle
4
>>> instanz.klassenmethode(3)
>>> Klasse.zaehle
7
```

Dekorator

Dekoratorfunktion

- Funktion, die mindestens eine Funktion als Argument hat und eine Funktion zurückgibt; einpacken
- Ziel: Verhalten und Syntax einer Funktion ändern
- Beispieleinsatz: statische Methoden
- Beispiel: `statisch` in Zeilen 6-8, Ohne Dekorator einzige Alternative

Dekoratorsyntax

- `@<decfun>` vor Definition führt Dekoratorfunktion `<decfun>` nach der Methodendefinition aus und überschreibt die alte Methode
- Zeilen 2–4 mit `@staticmethod` äquivalent zu Zeilen 6–8 oben
- Dekorator ist schicke Syntax

```
1 class Klasse:
2     zaehle = 1
3     def __init__(self):
4         self.zahl = 17
5
6     def statisch(x):
7         Klasse.zaehle += x
8     statisch = staticmethod(statisch)
```

```
1 ...
2     @staticmethod
3     def statisch(x):
4         Klasse.zaehle += x
```

Alternative zu Klassenmethoden

Klassenscope meist nicht notwendig

- In C++ sind Namensräume und static Variablen ähnlich zur Ablage von Werten und Definition von Funktionen auf globaler Ebene
- Statische Methoden und Klassenvariablen, in C++ auch möglich
- In Python auch möglich, aber selten notwendig

Nutzen Modulkonzept

- Statt Klassenvariable Modulvariable
- Statt statische Methode Modulfunktion
- Häufig passender

```
1 _KZAEHLE = 1 # nicht importiert bei *
2 def statisch(x):
3     global _KZAEHLE
4     _KZAEHLE += x
5 class Klasse:
6     def __init__(self):
7         self.zahl = 17
```

```
1 def statisch(x):
2     Klasse.zaehle += x
3 class Klasse:
4     zaehle = 1
5     def __init__(self):
6         self.zahl = 17
```

```
>>> instanz = Klasse()
>>> statisch(3)
>>> Klasse.zaehle
4
```

Klassen und Instanzen sind Namensräume

Nicht qualifizierte Namen nach Geltungsbereichen

- `val = wert`: weist an lokalen Namen zu
- `val`: Zugriff mit LEGB-Regel

Qualifizierte Namen

- Namensraum des Objekts
- Suche erst in aktueller Instanz,
  beinhaltet was durch Superklassen an Instanz geändert wurde,
  dann erst in Klassenscope
- Nicht im definierenden Modul

```
1
2
3
AttributeError: 'Klasse2' object
     has no attribute 'val'
```

```
1  val = 4
2
3  class Klasse1:
4      def __init__(self):
5          self.val = 1
6
7  class Klasse2(Klasse1):
8      val = 3
9      def __init__(self):
10         Klasse1.__init__(self)
11     def fun(self):
12         self.val = 2
13     def show_val(self):
14         print(self.val)
15
16 inst = Klasse2(); inst.show_val()
17 inst.fun();       inst.show_val()
18 del inst.val;     inst.show_val()
19 del Klasse2.val;  inst.show_val()
```

Klassen, Selbstbeobachtung

Umsetzung, Interna

- Attribute mit doppelten Unterstrichen
- Direkter Zugriff nur wenn notwendig

Klassen

- Name, Superklassen

Instanzen

- Klasse der Instanz
- Namensraum mit `__dict__`
- Realisiert mit Dictionaries

Selbst ausprobieren!

```
class Klasse:
    verbose = 1
    def __init__(self):
        pass
```

klasse.py

```
>>> from klasse import Klasse
>>> Klasse.__name__ # der Name
'Klasse'
>>> Klasse.__bases__ # die Superklassen
(<class 'object'>,)
>>> instanz = Klasse()
>>> instanz.__class__, instanz.__dict__
(<class 'klasse.Klasse'>, {})
>>> dir(instanz)
['__class__', '__delattr__', '__dict__', '__dir__',
 '__doc__', '__eq__', '__format__', '__ge__',
 '__getattribute__', '__gt__', '__hash__', '__init__',
 '__le__', '__lt__', '__module__', '__ne__', '__new__',
 '__reduce__', '__reduce_ex__', '__repr__', '__setattr__',
 '__sizeof__', '__str__', '__subclasshook__',
 '__weakref__', 'verbose']
```

Properties statt Getter und Setter

Durchgriff auf Attribute

- Ist in Python **in Ordnung**
- **Keine** Getter und Setter
- Man kann nachträglich ändern und Kontrolle wieder erlangen

Properties

- Eigentliches Attribut verstecken, _ am Namensanfang als Hinweis
  - Dann ist Durchgriff unhöflich
  - Unhöflich ist schlimmer als verboten
- Definition Getter, Setter und Löschen
- Definition des Attributs mit `property(getter,setter,delete,doc)`

```
1  class Name:
2
3      def __init__(self):
4          self._vorname = "Max"
5          self.__nachname = "Headroom"
6
7      def _get_vn(self):
8          return self._vorname
9
10     def _set_vn(self, value):
11         if type(value) != str:
12             msg = "vorname kein String"
13             raise AttributeError(msg)
14         self._vorname = value
15
16     vorname = property(_get_vn, _set_vn, None,
17        "Vorname als String/Unicode")
```

Name-Demangling bei __ am Namensanfang

- Bei zwei Unterstrichen Name ändern
- Attributname (von außen) ist `_Name__nachname` statt `__nachname`
- Vermeiden

```
>>> n = Name()
>>> n.vorname
'Max'
>>> n._vorname
'Max'
>>> n.__nachname
AttributeError: 'Name' object
   has no attribute '__nachname'
>>> dir(n)
['_Name__nachname','__class__', ...
 '_get_vn','_set_vn','_vorname','vorname']
>>> n._Name__nachname
'Headroom'
```

```
1  class Name:
2
3      def __init__(self):
4          self._vorname = "Max"
5          self.__nachname = "Headroom"
6
7      def _get_vn(self):
8          return self._vorname
9
10     def _set_vn(self, value):
11         if type(value) != str:
12             msg = "vorname kein String"
13             raise AttributeError(msg)
14         self._vorname = value
15
16     vorname = property(_get_vn, _set_vn, None,
17        "Vorname als String/Unicode")
```

Properties mit `@property`-Dekorator

Properties mit Dekorator

- `@property` zum Lesen
  Name der Methode muss Name der property sein
  muss zuerst kommen, definiert Property
- `@<name>.setter` zum Schreiben
- `@<name>.deleter` zum Löschen
- Doc bei Definition (zum Lesen)

`@property`–Dekorator

- Schick
- Verwenden

```
1  class Name:
2
3      def __init__(self):
4          self._vorname = "Max"
5          self.__nachname = "Headroom"
6
7      @property
8      def vorname(self):
9          "Vorname als String/Unicode"
10         return self._vorname
11
12     @vorname.setter
13     def vorname(self, value):
14         if type(value) != str:
15             msg = "vorname kein String"
16             raise AttributeError(msg)
17         self._vorname = value
18
19     # @vorname.deleter zum Löschen
```

Properties verwenden

Auf Klassenebene zugewiesenes Attribut wie Instanzvariable verwenden

- Automatisch Aufruf passende Methode
- Wenn nicht angegeben, dann Verwendung nicht möglich

Durchgriff: Immer noch möglich, nicht machen

Doppelte Unterstriche vermeiden

- Obscurity
- Wozu?

```
>>> name = Name()
>>> name.vorname
'Max'
>>> name.vorname = 'Susi'
>>> name.vorname
'Susi'
>>> name.vorname = 17
AttributeError: vorname kein String
>>> name.vorname
'Susi'
>>> del name.vorname
AttributeError: can't delete attribute
>>> name.vorname
'Susi'

>>> name._vorname = 17
>>> name.vorname
17
```

Überladen von Operatoren – Motivation

Ziel

- Rechnen Modulo einer Zahl `mod`
- Ergebnisse aller Rechenoperationen liegen zwischen `0` und `mod-1`

Lösung

- Klasse `Modulo`
- Initialisiert mit Wert `mod`
- Implementierung der Operationen durch Standardoperationen, aber Anpassung modulo
- Funktioniert, Benutzung nicht intuitiv

Wunsch

- Benutzen der `Modulo`-Instanzen wie eingebaute Zahlen
- Wie in C++ mit Operatorüberladung

```
1 class Modulo:
2     "Rechnen modulo einer Zahl"
3     def __init__(self, mod, zahl=0):
4         self.mod = mod
5         self.zahl = zahl
6     def wert(self):
7         return self.zahl
8     def add(self, z):
9         self.zahl = (self.zahl+z) % self.mod
```

```
>>> m = Modulo(7)
>>> m.wert()
0
>>> m.add(4)
>>> m.wert()
4
>>> m.add(5)
>>> m.wert()
2
```

Überladen von Operatoren – Umsetzung

Vordefinierte Funktionsnamen

- Werden gerufen bei Anwendung von Operatoren
- Beginnen und enden mit zwei Unterstrichen
- Definition neuer Operatoren nicht möglich

Beispiele

- `__repr__`: Repräsentation bei interaktiver Ausgabe, es wird String erwartet
- `__str__`: Stringrepräsentation, wie `std::to_string` von C++
- `__add__`: Wird gerufen bei linksseitigem + Operator
  - Eingebautes + liefert immer ein neues Objekt
  - Selbst auch so realisisren

```
    ...
    def __repr__(self):
        return str(self.zahl)
    __str__ = __repr__
    def __add__(self, z):
        n = Modulo(self.mod, self.zahl)
        n.add(z)
        return n
```

```
>>> m = Modulo(7)
>>> m
0
>>> m.add(4)
>>> m
4
>>> m+5
2
>>> m
4
```

Überladbare Operatoren – Auszug

- `__init__`: Konstruktorersatz, `__init__` der Superklasse nicht automatisch gerufen
- `__del__`: Destruktorersatz, beim explizitem Löschen und bei Garbage-Collection
- `__add__`, `__or__`: Operator `+`, `|`
- `__repr__`: Drucken (`print`), Konvertierung
- `__call__`: Funktionsaufruf `()`
- `__getattr__`, `__getattribute__`: Qualifikation
- `__getitem__`: Indizierung, auch von `in`-Operator verwendet,
  geht mit Slices (Python3)
- `__setitem__`: Zuweisung über Index/Slices
- `__delitem__`: Löschen über Index/Slices
- `__len__`: Länge
- `__lt__`: kleiner Vergleich (<, less than)
- `__eq__`, `__ne__`: Gleichheit (==), Ungleichheit(`!=`)
- `__radd__`: Rechtsseitiger Operator `+`

Details und weitere Operatoren in der Dokumentation

Von eingebauten Klassen ableiten

Beispiel eigenes Wörterbuch

- Verhalten wie `dict`, aber nur Strings als Schlüssel
- Alles andere gleich

Umsetzung

- Von `dict` ableiten, Vererben alternativ zu Komposition
- Überladen von `__setitem__`
- Nutzen von `__setitem__` der Superklasse
  - Achtung: sonst unendliche Rekursion

```
1 class strdict(dict):
2     def __setitem__(self, key, value):
3         if type(key) != str:
4             msg = "nur String als Key"
5             raise AttributeError(msg)
6         dict.__setitem__(self, key, value)
```

```
>>> dic = dict()
>>> dic[1] = 2
>>> dic["eins"] = 2
>>> dic
{1: 2, 'eins': 2}
>>> strdic = strdict()
>>> strdic[1] = 2
AttributeError: nur String als Key
>>> strdic["eins"] = 2
>>> strdic
{'eins': 2}
```

Verwenden von `super`

`super` bei einfacher Vererbung

- In C++ nicht verfügbar (Mehrfachvererbung)
- `self` ist dann nicht notwendig (implizit)
- Alternativ zu Nennen der Oberklasse, etwas flexibler bei Namensänderung der Oberklasse
- vermeiden, stattdessen Angabe des Namens der Oberklasse und explizit `self` angeben
  Zeile 6: `dict.__setitem__(self, key, value)`

`super` bei Mehrfachvererbung

- Mächtig aber komplex*
- `super(a-type, an-object-or-type)`
  `a-type` ist der Typ der Klasse, optional
  `an-object-or-type` ist optional, begrenzt Suche in Hierarchie
- Einfachere Default-Syntax in Python3

```
1 class strdict(dict):
2     def __setitem__(self, key, value):
3         if type(key) != str:
4             msg = "nur String als Key"
5             raise AttributeError(msg)
6         super().__setitem__(key, value)
```

```
>>> dic = dict()
>>> dic[1] = 2
>>> dic["eins"] = 2
>>> dic
{1: 2, 'eins': 2}
>>> strdic = strdict()
>>> strdic[1] = 2
AttributeError: nur String als Key
>>> strdic["eins"] = 2
>>> strdic
{'eins': 2}
```

*rhettinger.wordpress.com/2011/05/26/super-considered-super/

Ausnahmen

Ausnahmen, spezieller Kontrollfluss

- Fokus auf Fehlerbehandlung
- Erlaubt saubere Trennung von
  - funktionalen Bestandteilen und
  - „Rest“ wie Fehlerbehandlung und Spezialfälle
- Sollte nicht für allgemeinen Kontrollfluss verwendet werden

Ausnahmen in Python

- In Sprache integriert, einfach zu verwenden
- Ähnlich zu C++

Syntax

- `try`, `else`, `except`, `finally`
  - Rahmen für Code-Strecken mit potenziellen Ausnahmen, Fangen und Verarbeiten der Ausnahmen
- `raise`: Werfen von Ausnahmen
- `assert`: Wie in C

Ausnahmen fangen, Schema

- `<anweisungen>` im **`try`**-Block ausführen
- Falls Ausführung Ausnahme wirft mit `expect` fangen
- Eine oder mehrere Ausnahmen als Tupel möglich
- Mehrere **`except`** möglich
- Mit optionalem **`as`** Ausnahme-Instanz an `<name>` binden
- Falls Ausnahme gefangen entsprechenden **`except`**-Block ausführen
- **`else`**-Block (optional) falls keine Ausnahme geworfen wurde
- **`finally`**-Block (optional) immer, ob Ausnahme geworfen wurde (und gefangen wurde oder nicht) oder nicht

```
try:
    <anweisungen>
except <ausnahme>:
    <anweisungen>
except (<ausnahme>[, <ausnahme>]*):
    <anweisungen>
except <ausnahme> as <name>:
    <anweisungen>
except (<ausnahme>[, <ausnahme>]*) as <name>:
    <anweisungen>
else:
    <anweisungen>
finally:
    <anweisungen>
```

Ausnahmen fangen, Beispiele, `except`, `else`

```
>>> 3/0
ZeroDivisionError: division by zero
>>> try:
        3/0
    except ZeroDivisionError:
        print("nicht durch Null")
nicht durch Null
>>> try:
        3/0
    except ZeroDivisionError as e:
        print(type(e), e)
(<class 'ZeroDivisionError'> division by zero
>>> try:
        3/0
    except IndexError:
        print("IE")
    except ZeroDivisionError:
        print("ZDE")
ZDE
```

```
>>> try:
        3/0
    except ZeroDivisionError:
        print("ZDE")
    else:
        print("OK")
ZDE
>>> try:
        3/1
    except ZeroDivisionError:
        print("ZDE")
    else:
        print("OK")
3.0
OK
```

`ZeroDivisionError`, builtin Ausnahmetyp

Nicht gefangene Ausnahme propagiert zum TopLevel

Ausnahmen fangen, Beispiele, `finally`

```
>> try:
...     3/0
... except ZeroDivisionError:
...     print("ZDE")
... finally:
...     print("immer")
...
ZDE
immer
>>> try:
...     3/1
... except ZeroDivisionError:
...     print("ZDE")
... finally:
...     print("immer")
...
3.0
immer
```

```
>>> try:
...     3/0
... except IndexError:
...     print("IE")
... finally:
...     print("immer")
...
immer
ZeroDivisionError: division by zero
```

`finally` wird immer ausgeführt

Beinhaltet meist Aufräumaktionen

Ausnahmen fangen, Datei-Beispiel

Mit Ausnahmen arbeiten

- Code unter der Annahme schreiben, dass alles funktioniert
- Ausnahme und Behandlung bei Fehler

Mit Dateien arbeiten

- Zähle Zeilen in Datei
- -1, wenn Datei nicht vorhanden
  - Das sollte man natürlich nicht machen
  - Wir nehmen an wir wollen eine schlechte externe Vorgabe umsetzen

```
1 def countlinesfile(filename):
2     try:
3         count = 0
4         for _ in open(filename):
5             count += 1
6     except IOError:
7         count = -1
8     return count
```

```
>>> countlinesfile("/etc/passwd")
44
>>> countlinesfile("/GIBTESNICHT")
-1
```

Ausnahmeklassen, `Exception`, Hierarchie

```
BaseException
 +-- SystemExit
 +-- KeyboardInterrupt
 +-- GeneratorExit
 +-- Exception
      +-- StopIteration
      +-- ArithmeticError
      |    +-- FloatingPointError
      |    +-- OverflowError
      |    +-- ZeroDivisionError
      +-- AssertionError
      +-- AttributeError
      +-- BufferError
      +-- EnvironmentError
      |    +-- IOError
      |    +-- OSError
      |         +-- WindowsError (Windows)
      +-- EOFError
      +-- ImportError
      +-- LookupError
      |    +-- IndexError
      |    +-- KeyError
      +-- MemoryError
      +-- NameError
      |    +-- UnboundLocalError
      +-- ReferenceError
      +-- RuntimeError
      |    +-- NotImplementedError
      +-- SyntaxError
      |    +-- IndentationError
      |         +-- TabError
      +-- SystemError
      +-- TypeError
      +-- ValueError
      |    +-- UnicodeError
      |         +-- UnicodeDecodeError
      |         +-- UnicodeEncodeError
      |         +-- UnicodeTranslateError
      +-- Warning
           +-- DeprecationWarning
           +-- PendingDeprecationWarning
           +-- RuntimeWarning
           +-- SyntaxWarning
           +-- UserWarning
           +-- FutureWarning
           +-- ImportWarning
           +-- UnicodeWarning
           +-- BytesWarning
           +-- ResourceWarning
```

Eigene Ausnahmeklassen passend ableiten

Ausnahmen werfen

Ausnahmen werfen mit **raise**

- **raise** <klasse>: Wirft Ausnahme der Klasse <klasse>, muss von **BaseException** erben
- **raise** <instanz>: Wirft Ausnahme der Instanz <instanz>, muss Instanz von **BaseException** sein
- Instanz werfen bevorzugt
- Von **Exception** erben bevorzugt

Instanzen oder Klassen werfen

- Instanz, um Daten mitzugeben
- Häufig sind Daten ein String, vordefinierte Ausnahmen verwendbar

```
>>> raise Exception("nix geht")
Exception: nix geht
```

```
>>> class MeineAusnahme(object):
        pass
>>> raise MeineAusnahme
TypeError: exceptions must derive
           from BaseException

>>> class MeineAusnahme(Exception):
        pass
>>> raise MeineAusnahme
__main__.MeineAusnahme

>>> try:
        msg = "Weil es nicht geht"
        raise MeineAusnahme(msg)
    except MeineAusnahme as inst:
        print("Warum? %s" % str(inst))
Warum? Weil es nicht geht
```

Ausnahmen mit `assert`

Zusicherungen mit **`assert`**

- `<bedingung>`: ein Test
- Sobald Test äquivalent zu `False` evaluiert, **`AssertionError`**

Optimierung

- Gesteuert durch globale Variable `__debug__`
- Beim Übersetzen mit `-O` wird `__debug__ = False` gesetzt und dann der nicht erreichbare Bytecode entfernt

Benutzung

- Schleifeninvarianten, Vor- und Nachbedingungen
- Frühzeitige Fehlererkennung
- Lauffähige Dokumentation

```
assert <bedingung>, <data>
```

```
if __debug__:
    if not <bedingung>:
        raise AssertionError(<data>)
```

Ausnahmen mit `assert`, Beispiel

Zerlegen von einem String

- Wir gehen davon aus, dass der String immer genau einen Doppelpunkt enthält
- Was passiert im Fehlerfall?

Defensiv programmieren

- Früh fehlschlagen,
  statt erst mal weiter machen;
  `assert`
- Fehlermeldungen werden besser

```
1 def zerlege_nonsafe(s):
2     l = s.split(":")
3     return (l[0], l[1])
4 def zerlege_safe(s):
5     l = s.split(":")
6     assert len(l) == 2, l
7     return (l[0], l[1])
```

```
>>> zerlege_nonsafe("a:b:c")
('a', 'b')
>>> zerlege_safe("a:b:c")
    assert len(l) == 2, l
AssertionError: ['a', 'b', 'c']
>>> zerlege_nonsafe("a")
    line 3: return (l[0], l[1])
IndexError: list index out of range
>>> zerlege_safe("a")
    line 6: assert len(l) == 2, l
AssertionError: ['a']
```

Ausnahmen statt „Check then Act“-Fehler

Aufgabe

- Überprüfe die Existenz einer Datei
- Falls vorhanden, erste Zeile zurück
- Falls nicht vorhanden, hart beenden
- Es darf kein `IOError` geworfen werden

Ansätze

- „Check then act“, fehlerhaft
  - Es kann passieren, dass jemand zwischen Check Zeile 2 und verwenden, die Datei entfernt
  - Es wird dann doch `IOError` geworfen
- Mach es; melde, wenn es nicht geht
  - Kein Problem, sobald die Datei geöffnet werden konnte, ist sie da

Ein `IOError` wäre natürlich doch die bessere Variante, statt hart zu beenden

```
1 def get_status(fname):
2     if not os.path.exists(fname):
3         print("file not found")
4         sys.exit(-1)
5     return open(fname).readline()
```

```
1 def get_status(fname):
2     try:
3         return open(fname).readline()
4     except (IOError, OSError) as e:
5         print("problem, exiting: "+str(e))
6         sys.exit(-1)
```

Iteratoren für eigene Klassen

for-Kontrollstruktur

- Einfaches Iterieren über Sequenzen
- Funktioniert mit Iterator-Objekt
  - `next`-Funktion gibt nächstes Element zurück
  - Ausnahme **`StopIteration`**, wenn kein Element mehr verfügbar
- Ähnlich zu `iterator` Konvention in C++, aber mit Ausnahmen und nicht Werte an Grenzen

`iter`-Funktion, eingebaut

- Holt von Objekt Iterator, wenn möglich
- Implizit und explizit verwendbar
- Beispiel mit Listen

```
>>> lis = [1, 2, 3]
>>> for ele in lis:
        print(ele)
1
2
3
```

```
>>> lis = [1, 2, 3]
>>> it = iter(lis)
>>> it
<listiterator object at 0x186fad0>
>>> next(it)
1
>>> next(it)
2
>>> next(it)
3
>>> next(it)
StopIteration
```

Iteratoren für eigene Klassen – Instanz selbst

Selbst implementieren in der Klasse

- Überladen von `__iter__`
- Erwartet Iterator-Objekt
  - Zum Beispiel sich selbst, wenn Methode `__next__` definiert ist
  - Keine Schnittstellen, Duck-Typing

Bewertung

- Einfach zu realisieren
- Nachteil: Nur ein Iterator gleichzeitig je Instanz, da aktueller Zustand nur einmal in der Instanz gespeichert ist

```
class Iterselbst:
    def __init__(self):
        self.lis = [1, 2, 3]
    def __iter__(self):
        self.idx = 0
        return self
    def __next__(self):
        if self.idx >= len(self.lis):
            raise StopIteration
        self.idx += 1
        return self.lis[self.idx-1]
```

```
>>> its = Iterselbst()
>>> for ele in its:
        print(ele, end=" ")
1 2 3
>>> its = Iterselbst()
>>> it1, it2 = iter(its), iter(its)
>>> next(it1), next(it2)
(1, 2)
```

Iteratoren für eigene Klassen – Iterator-Klasse

Separate Iterator-Klasse

- Neu definierte Iterator-Klasse `itk`
- Kann durchaus im normalen Codeblock dynamisch erfolgen
- Instanz der Klasse initialisieren und zurückgeben

Bewertung

- Etwas komplexer zu realisieren
- Erlaubt parallele Iteration
- Ändern des Objekts während der Iteration erlaubt, da Kopie erstellt wurde
  - Kopie erstellen im Beispiel bewusste Entscheidung
  - Geht auch ohne, anderes Verhalten bei Änderung

```
1  class Iterklasse:
2      def __init__(self):
3          self.lis = [1, 2, 3]
4      def __iter__(self):
5          class itk(object):
6              def __init__(self, lis):
7                  self.idx = 0
8                  self.lis = lis[:]
9              def __next__(self):
10                 if self.idx >= len(self.lis):
11                     raise StopIteration
12                 self.idx += 1
13                 return self.lis[self.idx-1]
14         return itk(self.lis)
```

```
>>> itc = Iterklasse()
>>> it1, it2 = iter(itc), iter(itc)
>>> next(it1), next(it2)
(1, 1)
```

Iteratoren für eigene Klassen – Iterator von Standardtyp

Iterator von Standardtyp verwenden

- Standardtyp nehmen oder aufbauen
- Davon Iterator zurückgeben

Bewertung

- Einfach, robust
- Bevorzugen

`in` Tests

- Sobald Iterator implementiert ist, ist auch automatisch der `in`-Operator realisiert
- Iteriert über alle Element und vergleicht
- Bessere Implementierung darf man angeben

```
class Iterstd:
    def __init__(self):
        self.lis = [1, 2, 3]
    def __iter__(self):
        return iter(self.lis)
```

```
>>> its = Iterstd()
>>> it1, it2 = iter(its), iter(its)
>>> next(it1), next(it2)
(1, 1)
```

```
>>> its = Iterstd()
>>> 3 in its, 17 in its
(True, False)
```

Generatoren – Motivation

Iteratoren – Umsetzungskonzept

- Iterator: Zwischen jedem `next` Anwendungscode ausführen
- Iteratorklasse: Zustand halten und bei jedem `next` aktualisieren
- Statt Iteratorklasse?
  - Einfache Schleife klappt nicht
  - Man kann Ausführung nicht anhalten und Anwendungscode ausführen

Lösungsansätze

- Man kann den Anwendungscode, den man für jedes Objekt ausführen will „mitgeben“; Funktionales Programmieren
- Man kann die Schleife doch anhalten; Generatoren

```
i = 0
def naechstes():
    global i
    if i >= len(lis):
        return # nichts zurückgeben, Ende
    ele = lis[i]
    i += 1
    return ele
```

```
for ele in lis:
    # naechstes ele, Anwendungscode?
    pass
```

```
def f(x):
    pass # Anwendungscode
###
map(f, lis)
```

Generatoren – Konzept

Generatoren in Python

- `yield` statt `return` in Funktion
  - Sobald einmal ein `yield` vorkommt ist es ein Generator
- Bei Funktionsaufruf wird automatisch Generatorobjekt erzeugt, das sich wie Iterator verhält
- Bei `next` wird Wert bei `yield` zurückgegeben und Ausführung eingefroren
- Bei jedem `next` zum nächsten Wert
- Aufrufer sieht `StopIteration` am Ende
  - Es muss kein Ende geben, unendliche Generatoren

Vorteile

- Keine großen Datenstrukturen im Speicher halten
- Natürliche Realisierung von Iteratoren
- Grundlage für die meisten Builtins

quads.py

```
1 def quads():
2     i = 1
3     while True:
4         yield i**2
5         i += 1
```

```
>>> q = quads()
>>> next(q)
1
>>> next(q)
4
>>> next(q)
9
>>> ...
```

Konzept der Koroutine

- Uralt, schon in Pascal/Modula, wieder modern
- Nebenläufigkeit nichtblockierend, asynchron
- In C++ ab C++20 mit `co_await`, `co_yield`

Generatoren/Iteratoren – Grundlage für viele Builtins

Beispiel: Eine offene Datei ist ein Generator/Iterator, der die Zeilen in der Datei sukzessive zurückgibt

```
>>> datei = open("quads.py"); datei
<_io.TextIOWrapper name='quads.py'
         mode='r' encoding='UTF-8'>
>>> next(datei)
'def quads():\n'
>>> next(datei)
'    i = 1\n'
....
>>> next(datei)
'        i += 1\n'
>>> next(datei)
StopIteration
```

Beispiel: Ergebnis eines Aufrufs von `reversed` ist ein Generator/Iterator, der Elemente rückwärts aufzählt

```
>>> lis = [1, 2, 3]
>>> gen = reversed(lis); gen
<listreverseiterator object at 0x..4b90>
>>> next(gen)
3
>>> next(gen)
2
>>> next(gen)
1
>>> next(gen)
StopIteration
```

Generator/Iterator-Objekte bei `map`, `filter`, `dict_keys`, `range`, ...
Generator als einfache Möglichkeit Iteratoren zu implementieren

Generatoren – Verwendung und weitere Beispiele

Verwendung

- Summe der Quadratzahlen,
  die zum ersten Mal 1.000.000 erreicht

```
sum = 0
for quad in quads():
    sum += quad
    if sum > 1000000:
        break
print(sum) # 1005720
```

`nquads`

- `quads` war unendlich, beschränkt geht auch
- Mit `list` wird wieder eine Liste daraus,
  läuft ewig für unendliche Generatoren
- Verwendung von `for` möglich, sinnvoll bei potenziell unendlich lange laufenden Generatoren
  - Abbruch mit `return`,
    `raise StopIteration` seit 3.6+ deprecated
  - Aufrufer sieht `StopIteration`
  - Abbruch nicht notwendig

```
def nquads(n):
    for i in range(1, n+1):
        yield i**2
```

```
def hoch4():
    for ele in quads():
        yield ele**2
```

```
def nquads(n):
    grenze = n**2
    for ele in quads():
        yield ele
        if ele >= grenze:
            return # Ende
```

```
>>> list(nquads(10))
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

Generatoren – `raise StopIteration` wird abgeschafft

Gut: **`return`**, um zu beenden

- Sie rufen **`return`**, um Generator zu beenden
- Der Aufrufer sieht die Ausnahme **`StopIteration`**

Falsch: **`StopIteration`** werfen, um zu beenden

- Der Generator wirft die Ausnahme **`StopIteration`**, um zu beenden
- Seit 3.6 kein Propagieren von **`StopIteration`** mehr (Deprecation, 3.7 Fehler), PEP 479
- 3.6 noch inkonsistent; mal Warnung, mal nicht

Beispiel

- Erzwinge Verhalten von 3.7
- Werfe Fehler, falls **`StopIteration`** geworfen
- Abbruch

```python
#!/usr/bin/python3
from __future__ import generator_stop # 3.7
def g1():
    yield 1
    raise StopIteration # hier return!
for ele in g1():
    print(ele)
```

```
1
Traceback (most recent call last):
  File "stopiteration.py", line 5, in g1
    raise StopIteration
StopIteration
The above exception was the direct cause
    of the following exception:
Traceback (most recent call last):
  File "stopiteration.py", line 6, in <module>
    for ele in g1():
RuntimeError: generator raised StopIteration
```

Generatoren – Leak `StopIteration` bei Verschachtelung

`raise StopIteration` verschachtelt

- Ein Generator kann ein **`StopIteration`** in einer Verschachtelung werfen
  - `g` ist ein Generator als Parameter in `g1`
  - `g1` erzeugt einen Generator unter Nutzung von `g`
- Darf nicht propagieren, Leak
- Nicht gleich ersichtlich

Fix: Fangen und **`return`**

- Falls Aufruf **`StopIteration`** werfen kann
- Fangen in **`try`**/**`except`**-Block
- Mit **`return`** Generator beenden

Fix: **`for`**-Schleifen fangen **`StopIteration`**

Verschachtelt, fixed mit **`for`**

```
#!/usr/bin/python3
from __future__ import generator_stop # 3.7
def g1(g):
    for ele in g:
        yield ele
```

Verschachtelt, fehlerhaft

```
#!/usr/bin/python3
from __future__ import generator_stop # 3.7
def g1(g):
    while True:
        yield next(g)
for ele in g1(iter(range(3))):
    print(ele)
```

Verschachtelt, fixed mit **`try`**/**`except`**

```
#!/usr/bin/python3
from __future__ import generator_stop # 3.7
def g1(g):
    while True:
        try:
            yield next(g)
        except StopIteration:
            return
for ele in g1(iter(range(3))):
    print(ele)
```

Generatoren – Fibonacci

Folge der Fibonacci-Zahlen

- Startwerte, zweimal die Eins
- Die nächste Fibonacci-Zahl ist die Summe der beiden Vorherigen

Umsetzung

- Aktuelle zurück geben und dabei Zustand einfrieren
- Aktuelle und vorherige Fibonacci-Zahl halten

Verwendung

- Die ersten 10 Fibonacci-Zahlen

```
1 def fib():
2     fn1 = fn0 = 1 # fn0 aktuell
3     while True:
4         yield fn0
5         fn0, fn1 = fn0+fn1, fn0
```

```
>>> f = fib()
>>> for i in range(10):
        print(next(f), end=" ")
1 2 3 5 8 13 21 34 55 89
```

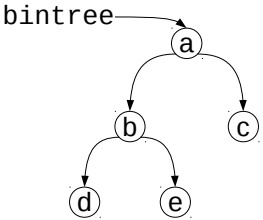
Generatoren – Baum traversieren

Baum

- Knoten als Drei-Tupel, Wert ist String
- Iterator für In-Order normalerweise aufwendig, Warteschlange

Baum traversieren mit Generatoren

- Natürliche Formulierung
- Nutzen von Schleifen mit neuen Generatoren im rekursiven Abstieg




```python
def inorder(node):
    if node is None:
        return
    left, value, right = node
    for valueleft in inorder(left):
        yield valueleft
    yield value
    for valueright in inorder(right):
        yield valueright
```

```python
bintree = (
    ((None, "d", None),
     "b",
     (None, "e", None)),
    "a",
    (None, "c", None))
```

```
>>> for value in inorder(bintree):
        print(value, end=" ")
d b e a c
```

Generatoren – `itertools`, Slicen

Generatoren sind keine Sequenzen

- Generatoren sind potenziell unendlich
- Umformung in Listen und Slicing nicht sinnvoll
- Standard-Features von Sequenzen (Slicing, Syntax) nicht verfügbar

Modul `itertools`

- Wichtigste Funktionalität von Sequenzen für Generatoren zur Verfügung stellen
- Slicen, Funktional Programmieren

Slicen mit `islice`

- Aufruf von `islice` mit `start`, `stop`, `step`
- Nachbau Funktionalität `[start:stop:step]`

```
islice(<iterator>
       [, <start>], <stop>[, <step>])
```

```
>>> list(quads())
... # 1 GB, 2 GB, 3 GB, ...
... # ca. 3 Minuten auf 32 Bit-Rechner
MemoryError
```

```
>>> from itertools import islice
>>> q = quads()
>>> iq = islice(q, 0, 10)
>>> next(iq), next(iq), next(iq)
(1, 4, 9)
>>> list(iq)
[16, 25, 36, 49, 64, 81, 100]
>>> list(islice(q, 0, 10))
[121,144,169,196,225,256,289,324,361,400]
>>> list(islice(q, 0, 10))
[441,484,529,576,625,676,729,784,841,900]
>>> list(islice(quads(), 0, 10))
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

Generatoren – `itertools`, Lazy Funktional Progammieren

Funktional Programmieren

- Etwas für alle Elemente einer Liste tun
- Problematisch bei unendlichen Listen

Spezielle Primitive für Generatoren, Lazy

- Primitive mit `i` am Anfang, `islice`
- Lazy default in Python3, eager in Python2
  in Python2: `imap`, `ifilter`, `izip`
  im Modul `itertools`
- Geben Generatoren zurück
- Funktionalität funktionaler Primitive,
  lazy statt eager

```
>>> from itertools import islice
>>> [i for i in islice(quads(), 0, 20)
       if i%2==0]
[4, 16, 36, 64, 100, 144, 196, 256, 324, 400]
>>> list(islice(filter(lambda x: x%2 == 0,
                       quads()), 0, 10))
[4, 16, 36, 64, 100, 144, 196, 256, 324, 400]
>>> gen = filter(lambda x: x%2 == 0, quads())
>>> gen
<filter object at 0xe7c0d0>
>>> next(gen)
4
>>> next(gen), next(gen), next(gen), next(gen)
(16, 36, 64, 100)
>>> import math
>>> list(islice(map(math.sqrt, quads()), 9))
[1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0]
```

Generatoren – Generator Expressions

List Comprehension für Generatoren

- List Comprehension direkt wieder nicht sinnvoll, da potenziell unendlich
- Ausdruck mit runden Klammern `()` statt eckigen Klammern `[]`
- Erzeugt dann Generator/Iterator

Mächtiges elegantes Konzept, nutzen!

```
>>> gq = (i**2 for i in range(10))
>>> gq
<generator object <genexpr> at 0x1b6b690>
>>> next(gq)
0
>>> next(gq), next(gq), next(gq)
(1, 4, 9)
>>> list(gq)
[16, 25, 36, 49, 64, 81]
>>> gquads = (i for i in quads() if i%2 == 0)
>>> gquads
<generator object <genexpr> at 0x1b6bb40>
>>> next(gquads)
4
>>> next(gquads), next(gquads), next(gquads)
(16, 36, 64)
>>> from itertools import islice
>>> list(islice((i for i in quads() if i%2==0)
                ,0,10))
[4, 16, 36, 64, 100, 144, 196, 256, 324, 400]
```

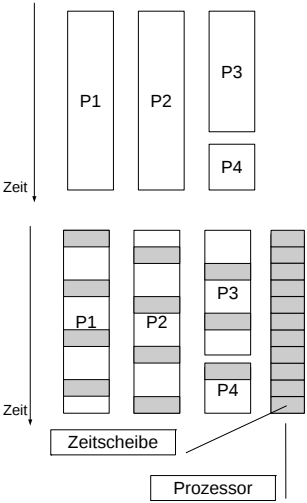
Multitasking

Multitasking

- Mehrere Dinge „gleichzeitig“ tun
- Erwartetes Verhalten auf Rechnern
  - Musik, Web (+ Animation), Download
- Ein „Ding“ ist meist ein Prozess
  - Eigener Speicher
  - Eigener Ausführungsstrang/-faden
- Beispiel: P1, P2 und P3/4 parallel; P4 nach P3

Prozessoren/Kerne und Prozesse

- Ausführung der Prozesse auf Prozessoren/Kernen
- Multitasking auch bei einem Kern
  - Zeitscheiben unterteilen Prozessorzeit
  - Jeder Prozess erhält abwechselnd eine (sehr sehr kleine) Zeitscheibe
- Bei vielen Kernen mehr (parallele Zeitscheiben)
- Aus Benutzersicht alles quasi gleichzeitig



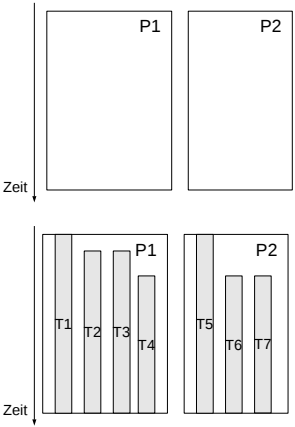
Threads

Prozesse

- Eigener Speicher, eigener Ausführungsfaden
- Fehler in einem Prozess betrifft nur diesen Prozess
  - Bei C mit Speicherfehler (Segmentation fault) läuft das System, außer dem bösen Prozess, weiter
- Schwergewichtige Umschaltung der Zeitscheiben, viel Aufwand je Kontextwechsel
- Modell für unabhängige Knoten in einem verteilten System

Threads

- Teilen sich Speicher, eigener Ausführungsfaden
- Threads innerhalb eines Prozesses
- Leichtgewichtiges Umschalten der Zeitscheiben, weniger Aufwand je Kontextwechsel
- Verteilung Zeitscheiben auf alle Threads aller Prozesse (bei Betriebssystemunterstützung)
- Nebenläufigkeit als Programmiermodell



Langsamer Primzahl-Generator als Beispiel

Primzahlgenerator

- Ein Zahl ist eine Primzahl, wenn Sie keine Teiler hat außer 1 und sich selbst
- Sehr ineffizient; gut, um Last zu generieren
- Je mehr Primzahlen, desto mehr Last

Umsetzung

- Berechne alle Teiler mit `get_divs`
  - Alle Zahlen bis eins unter der zu testenden Zahl
  - Nur Teiler herausfiltern
  - Liste daraus machen
- Nur wenn es keine Teiler gibt, ist es eine Primzahl
  - Außer der 2 sind alle Primzahlen ungerade

```python
def get_divs(x):
    return [div for div in range(2, x)
                if x%div == 0]

def primegen():
    yield 2
    p = 3
    while True: # intentionally slow
        # no divisors, is prime
        if get_divs(p) == []:
            yield p
        p += 2
```

Primzahlgenerator als Thread in Python

`threading`-Modul und `Thread`-Klasse

- `threading.Thread`, Teil der Standardbibliothek, Systemunterstützung
- Erweitere `Thread` und überschreibe `run`
- Methode `run` ist was der Thread in seinem Ausführungsfaden ausführt

Beispiel: Berechne ite Primzahl

- Primzahlgenerator `primegen`
- `IthPrime` erbt von `Thread`
- Merke Wert von `i` bei Initialisierung
- Methode `run` erzeugt Primzahlen bis ite Primzahl erreicht wurde
- Attribut `ith_prime` hält Ergebnis
- `i`, `ith_prime` sind Instanzvariablen, potenziell geteilt (geshared)
- `pg` ist lokale Variable des Threads (Stack)

```
1  import threading
2
3  class IthPrime(threading.Thread):
4
5      def __init__(self, i):
6          super().__init__()
7          self.i = i
8          assert self.i > 0
9          self.ith_prime = None # result
10
11     def run(self):
12         pg = primegen()
13         for _ in range(self.i-1):
14             next(pg)
15         self.ith_prime = next(pg)
16         print("done: %d" % self.i)
```

Thread starten

Thread-Instanz erstellen

- Instanziiere `IthPrime`
- Erbt von `threading.Thread`

Starte die Thread-Instanz

- Rufe die Methode `start`, (**nicht** `run`)
- Asynchroner Aufruf, kommt sofort zurück
- Seiteneffekt ist ein (bald) laufender Thread

Ende eines Threads (Instanz)

- Ein Thread endet sobald `run` endet
- Es gibt keinen anderen Fall
- `Thread.join` wartet auf Ende eines Threads außerhalb des Threads
  - Methode akzeptiert optional `timeout`, maximale Wartezeit (Sekunden) auf Thread-Ende
- Lesen der iten Primzahlen aus (beendeten) Thread-Objekt

```
i = 1717
ipt = IthPrime(i)
ipt.start()
print("Something useful")
ipt.join()
p = ipt.ith_prime
print("%dth prime: %d" % (i, p))
```

```
Something useful
done: 1717
1717th prime: 14657
```

Berechnung läuft nebenläufig

- Erst Ausgabe von `Something useful`

Thread, Syntax-Alternative, Callable

Callable (Runnable), Funktion

- Instanziiere original `Thread` Klasse
- Übergebe etwas „Aufrufbares“ als `target` beim Instanziieren
  - Instanz einer Klasse, die `__call__` implementiert
  - Eine Funktion

Funktion

- Bei Funktion keine Instanz für Ergebnis
  - Wir haben bei einer Funktion keine Instanz
  - Wir können uns das Ergebis nicht merken
  - Der Rückgabewert kann nicht genutzt werden
- Lösungsansatz, nutze Callback
  - Eine Funktion, die mit dem Ergebnis aufgerufen wird
  - Wird vom Aufrufer mit übergeben
  - Speichert/Nutzt das Ergebnis wie vom Aufrufer gewünscht

```
class IthPrimeCallable:
    # Initialisierung bleibt gleich
    def __call__(self): # statt run
        # Rest bleibt gleich
```

```
ipc = IthPrimeCallable(i)
ipt = threading.Thread(target=ipc)
ipt.start()
print("Something useful")
ipt.join()  # ?InterruptedException
print("%dth prime: %d" % (i, ipc.ith_prime))
```

```
def ithprime_fun(i, callback):
    pg = primegen()
    for _ in range(i-1):
        next(pg)
    callback(i, next(pg))
```

Thread, Syntax-Alternative, Aufruf Funktion

Aufruf mit Funktion

- 2-3: Callback realisieren
  - Merkt sich Ergebnis in Liste*
- 4: Callable/Runnable realisieren
  - Aufruf der Funktion mit passenden Parametern
  - **lambda**, damit es ohne Parameter aufgerufen werden kann
- 9: Verwenden des Ergebnisses im Main-Thread
- Rest bleibt gleich

```
1  def ithprime_fun(i, callback):
2      pg = primegen()
3      for _ in range(i-1):
4          next(pg)
5      callback(i, next(pg))
```

```
1  results = []
2  def callback(_, result):
3      results.append(result)
4  runnable = lambda : ithprime_fun(i, callback)
5  ipt = threading.Thread(target=runnable)
6  ipt.start()
7  print("Something useful")
8  ipt.join()  # ?InterruptedException
9  p = results[0]
10 print("%dth prime: %d" % (i, p))
```

*Achtung: Noch eine RaceCondition im Zugriff für `results`, kommt gleich

Zustände eines Threads

Neu

- Thread-Instanz wurde erstellt
- `start` wurde noch nicht gerufen

Runnable

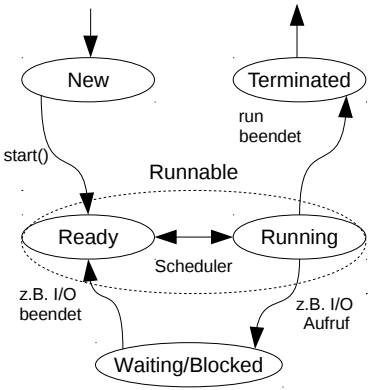
- Thread ist bereit zu laufen oder läuft (nutzt eine Zeitscheibe des Betriebssystems)
- Entscheidung durch Betriebssystem-Scheduler

Waiting/Blocked

- Thread ist nicht bereit eine Zeitscheibe zu nutzen
- Blockierender Aufruf, warten auf Ergebnis

Terminated

- `run` hat sich beendet, durchgelaufen
- Thread-Objekt selbst noch verfügbar
- Keine Möglichkeit den Thread neu zu starten



- Jeder Wechsel von/nach *Running* ist ein „Kontextwechsel“
- Kontextwechsel sind teuer

Mehrere Threads

Mehrere Threads für mehrere Tasks

- Beispiel: Threads für 1717te, 1718te und 1719te Primzahl
- Warte auf das Ende aller Threads
- Gebe die Ergebnisse aus, erwartete Reihenfolge

Keine feste Reihenfolge bei Thread-Ausführung

- Nichtdeterministisches Verhalten
- Reihenfolge der `done`-Ausgaben variiert

```
ivals = [1717, 1718, 1719]
ipts = [IthPrime(i) for i in ivals]
for ipt in ipts:
    ipt.start()
for ipt in ipts:
    ipt.join()
    i, p = ipt.i, ipt.ith_prime
    print("%dth prime: %d" % (i, p))
```

```
done: 1717
1717th prime: 14657
done: 1719
done: 1718
1718th prime: 14669
1719th prime: 14683
```

3 Threads parallel auf verfügbaren Kernen

Mehrere Threads - Python/GIL, the dark side

Global Interpreter Lock (GIL)

- Immer nur ein Thread gleichzeitig führt Python-Bytecode aus
- Eigenheit der CPython-Implementierung
- Kein Problem bei Threading für I/O-Warten
- Kann mehrere CPUs für reine Berechnung in Python-Bytecode nicht voll ausnutzen

`multiprocessing` als Lösung

- Mehrere Prozesse mit einem Thread statt ein Prozess mit vielen Threads
- Geteilte Variablen explizit definieren:
  - `Value`, `Array`, `Queue` in Shared-Memory
- Verwendung an Threading-API angelehnt
  - `Process` statt `Thread`
  - Übergabe von z.B. `Queue`-Instanz
  - Schreiben mit `put`, Lesen mit `get`, „safe“
- Ausnutzen aller Kerne

```
1  def ithprime_mp(i, results):
2      pg = primegen()
3      for _ in range(i-1):
4          next(pg)
5      results.put((i, next(pg)))
```

```
1  from multiprocessing import Process, Queue
2  ivals = [1717, 1718, 1719]
3  results = Queue() # gemeinsame Variable
4  ipjobs = []
5  for i in ivals:
6      job = Process(target=ithprime_mp,
7                    args=(i, results))
8      ipjobs.append(job)
9  for job in ipjobs: job.start()
10 for job in ipjobs: job.join()
11 for _ in range(len(ivals)):
12     i, p = results.get()
13     print("%dth prime: %d" % (i, p))
```

Threads – Ablaufkontrolle

Unmöglich Thread von außen zu stoppen

- Es **kann** keine Möglichkeit geben, die immer funktioniert
- Anwendungsentwickler muss entsprechende Möglichkeiten vorsehen

Kooperatives Beenden

- Methoden zum Setzen von Variablen
- Periodische Überprüfung der Variablen

Read/Write von gemeinsamen Variablen

- Gemeinsam heißt Thread-übergreifend
- `threading.Event`: `set, clear, is_set`
  `wait` warten bis gesetzt
- Wert in allen Threads sichtbar, setzen/löschen unteilbar

```
1  class Incer(threading.Thread):
2
3      def __init__(self):
4          super().__init__()
5          self.done = threading.Event()
6
7      def run(self):
8          x = 0
9          while not self.done.is_set():
10             x += 1
11             time.sleep(1.0)
12             print("%s: x=%d" % (util.mm(), x))
13
14     def quit(self):
15         self.done.set()
```

`util.mm`

- Aktuelle Zeit in Millisekundenauflösung alle 10 Sekunden, zum Abstände sehen

Threads – Ablaufkontrolle, verwenden

Beispiel Ausgabe

- Inkrementieren in einem Hintergrund-Thread
- Nach 3 Sekunden der Wunsch Thread zu verlassen
- Nach einer gewissen Zeit, maximal 1 Sekunde, wird der Wunsch erfüllt

Methoden von `Thread`-Instanzen

- `is_alive`, ist der Thread noch nicht in Terminated
- `Thread.join` blockiert aktuellen Thread (Wait) bis der Thread auf dem die Methode aufgerufen wurde terminiert ist
- Es dauert fast eine Sekunde bis Reaktion
- Ziel: Zeit zwischen `quitting` und eigentlichem Aufhören reduzieren

```
incer = Incer()
incer.start()
time.sleep(2.3)
print("%s: quitting" % util.mm())
incer.quit()
print("Alive1?: %s" % incer.is_alive())
incer.join()
print("Alive2?: %s" % incer.is_alive())
```

```
5.355s: x=1
6.425s: x=2
6.655s: quitting
Alive1?: True
7.426s: x=3
Alive2?: False
```

Wait – Schnell reagieren

`threading.Event.wait`

- Steuerungsklasse `threading.Event` hat eingebaute Methode zum Warten
- Methode `wait` statt `time.sleep`
- Optionaler Parameter Sekunden (`float`) wie lange man maximal warten soll
- Geht sofort aus Blockieren raus, wenn es von anderem Thread aus gesetzt (`set`) wird

```
0.002s: x=1
1.003s: x=2
1.305s: quitting
Alive1?: True
1.305s: x=3
Alive2?: False
```

```
class IncerWait(threading.Thread):
    def __init__(self):
        super().__init__()
        self.done = threading.Event()
    def run(self):
        x = 0
        while not self.done.is_set():
            x += 1
            self.done.wait(1.0) # obergrenze
            print("%s: x=%d" % (util.mm(), x))
    def quit(self):
        self.done.set()
```

```
incer = IncerWait()
incer.start()
time.sleep(2.3)
print("%s: quitting" % util.mm())
incer.quit()
print("Alive1?: %s" % incer.is_alive())
incer.join()
print("Alive2?: %s" % incer.is_alive())
```

Kritischer Wettlauf – Beispiel `Incer`

Globale Variable x

- Initial `0`
- Durch zwei Threads um 1 erhöht

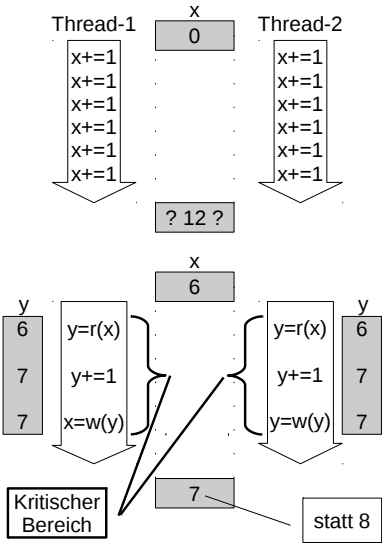
Problem

- x`+=1` sind drei unterbrechbare Anweisungen
  - Lese (read/r) x in y
  - Berechne y `=` y `+ 1`
  - Schreibe (write/w) y in x
- Lesen und Schreiben der gleichen Variable x, kritischer Wettlauf
  - y unkritisch, Thread-lokal, zwei Mal y

Kritischer Wettlauf

- Quasi gleichzeitiges Ausführen in kritischem Bereich
- Durch Reihenfolge der Ausführung unerwartete Ergebnisse

Synchronisation notwendig, Sperren



Unsicherer Counter

Unsicherer Counter

- Eine Instanz von `Counter`
- Zwei (oder mehr) Threads greifen auf die gleiche Instanz zu und ändern diese, `inc`
- Undefiniertes Verhalten

```
class Counter:
    def __init__(self):
        self.val = 0
    def inc(self):
        self.val += 1
    @property
    def value(self):
        return self.val
```

Beispiel:

- 2 Threads, je Thread 1 Million mal `inc`
- Nur eine Instanz eines `Counter`
- Jedes Mal anderer Wert (!= 2000000)

```
def run_counter(counter, times=1000000, incers=2):
    def incmany():
        for _ in range(times):
            counter.inc()
    threads = [threading.Thread(target=incmany)
                    for _ in range(incers)]
    util.reset_time()
    for thread in threads: # startall
        thread.start()
    for thread in threads: # joinall
        thread.join()
    class_name = counter.__class__.__name__
    value = counter.value
    used = util.get_times()
    print("%14s: %d, %s" % (class_name, value, used)
```

```
Counter: 1655103, 0.301s
```

Synchronisation mit Sperren

Kritischer Bereich `x+=1` mit Sperre/Lock sichern

- Sichere Sperre, um atomare Ausführung (ununterbrechbar) zu garantieren
- (Betriebs-)Systemunterstützung für Holen und Freigeben von Sperren
  - In Python eine Instanz von `threading.Lock`
  - Methoden `acquire` und `release`
- Sperre vor Eintritt in kritischen Bereich, Sperre loslassen nach Verlassen des kritischen Bereichs
- Eine schon belegte Sperre holen legt den rufenden Thread schlafen (suspendieren, Wait)
- Auch lesender Zugriff auf Wert ist nur sicher, wenn dabei Sperre gehalten wird (Sichtbarkeit)

```python
class SafeCounter:

    def __init__(self):
        self.lock = threading.Lock()
        self.val = 0

    def inc(self):
        self.lock.acquire()
        try:
            self.val += 1
        finally:
            self.lock.release()

    @property
    def value(self):
        self.lock.acquire()
        try:
            return self.val
        finally:
            self.lock.release()
```

Sicherer Zähler als Monitor

Monitor

- Sperre ist interner (privater) Teil der Klasse
- Verwenden der Sperre automatisch passend bei den Methodenaufrufen
- Alle geschützen Variablen auch privat
- Es ist nicht möglich einen Monitor nicht Thread-safe zu verwenden, gut

Syntax `with`

- Holt bei Eintritt die Sperre
- Lässt bei Austritt die Sperre los, egal wie man austritt
- Spart das `try` ... `finally`

Thread-Safety is not Composable

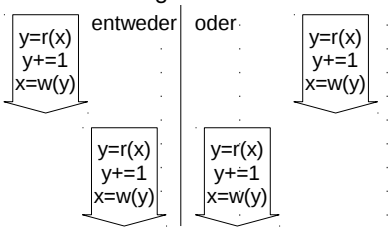
- Zwei Monitore zusammen sind kein Monitor
- Monitor ist gut, löst aber nicht alle Probleme

```
1  class MonitorCounter:
2
3      def __init__(self):
4          self._lock = threading.Lock()
5          self._val = 0
6
7      def inc(self):
8          with self._lock:
9              self._val += 1
10
11     @property
12     def value(self):
13         with self._lock:
14             return self._val
```

Sicheren Zähler verwenden

Sperren ist sicher

- Nur ein Thread gleichzeitig in kritischem Bereich
- Kein Interleaving (Abwechslung), keine verlorenen Erhöhungen



Jeder serielle Schedule

Sperren ist teuer

- Laufzeit explodiert, trotzdem
- . . . „better safe than sorry“
- Teuer in dem Beispiel, bei I/O häufig kein Problem

```
run_counter(Counter())
run_counter(SafeCounter())
run_counter(MonitorCounter())
```

2 Threads, je Thread 1 Million `inc`-Aufrufe
Erwartete Ausgabe: 2000000

```
        Counter: 1655103, 0.270s
    SafeCounter: 2000000, 3.343s
MonitorCounter: 2000000, 3.423s
```

Typisches Verhalten:

- Bei hoher Contention (gleichzeitiger Zugriff auf Sperre)
- Um 1-2 Größenordnungen höhere Laufzeit

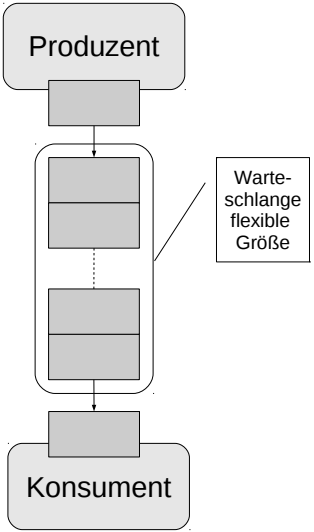
Producer/Consumer

Situation

- Ein Produzent erzeugt Werte
- Ein Konsument verwendet Werte
- Beide können unabhängig voneinander laufen
- Problem: Konsument muss warten, bis Produzent fertig; Produzent muss warten, bis Konsument fertig

Warteschlange

- Entkopplung von Produzent und Konsument
- Produzent legt fertige Werte in Warteschlange
- Konsument holt Werte aus der Warteschlange
- Warteschlange ist gemeinsames Objekt, Zugriffe müssen synchronisiert werden
  - Bei leerer Warteschlange muss Konsument warten
  - Bei voller Warteschlange (begrenzte Größe) muss Produzent warten
  - Einfügen/Entfernen nur einer gleichzeitig



Producer

Task

- Erzeuge Zahlen von 1 bis `how_many` (inklusive)

Klasse `Producer` in separatem Thread

Gemeinsame Objekte

- Warteschlange `queue`
- Sperre für die Warteschlange `qlock`

Ausführung in `run`

- Bisschen warten mit `slow(n)`
  - Irgendeine Berechnung, damit langsamer produziert wird
- Modifizieren der Warteschlange beinhaltet read und write, ist also ein kritischer Bereich, muss daher mit `qlock` geschützt werden
- Nächstes generiertes Element einfügen

```
1  from threading import Thread, Event, Lock
2
3  class Producer(Thread):
4
5      def __init__(self, queue, qlock, how_many):
6          super().__init__()
7          self.queue = queue
8          self.qlock = qlock
9          self.how_many = how_many
10
11     def run(self):
12         for i in range(1, self.how_many+1):
13             slow(42) # verlangsamt Erzeuger
14             with self.qlock:
15                 self.queue.append(i)
```

Consumer

Task

- Summe aller verfügbaren Elemente

Klasse `Consumer`: Separater Thread

Gemeinsame Objekte

- Warteschlange `queue`
- Sperre für die Warteschlange `qlock`
- `done` mit `Event`

Ausführung in `run`

- So lange noch nicht fertig (`done`)
- Falls was verfügbar, dann hole nächstes Element von `queue` und summiere es auf
- Falls die `queue` leer ist, dann zähle den Fehlschlag
- Anmerkung: Sperre so kurz wie möglich gehalten

```
1  class Consumer(Thread):
2
3      def __init__(self, queue, qlock):
4          super().__init__()
5          self.queue = queue
6          self.qlock = qlock
7          self.done = Event()
8          self.sum = 0
9          self.misses = 0
10
11     def run(self):
12         while not self.done.is_set():
13             value = None
14             with self.qlock:
15                 if self.queue: # not empty
16                     value = self.queue.pop(0)
17             if value is not None:
18                 self.sum += value
19             else:
20                 self.misses += 1
```

Producer und Consumer

Erzeuge gemeinsame Objekte

- Warteschlange als `list`
- Sperre mit `Lock`

Erzeuge Producer/Consumer und starte

- Erst beide erzeugen
- Dann beide starten (kurze Zeitdifferenz)

Producer, Consumer `join`

- Warte auf Producer sich zu beenden, da nur endlich viele Zahlen erzeugt werden sollen
- Sage Consumer Bescheid, wenn der Producer fertig ist
- Warte auf den Consumer sich zu beenden

Die Summe kann geholt werden

- Die Warteschlange war schon leer manchmal

```
1  how_many = 100_000
2  queue = []
3  qlock = Lock()
4
5  producer = Producer(queue, qlock, how_many)
6  consumer = Consumer(queue, qlock)
7  producer.start()
8  consumer.start()
9
10 producer.join()
11 consumer.done.set()
12 consumer.join()
13
14 print("sum=%d, misses=%d" %
15          (consumer.sum, consumer.misses))
```

```
sum=5000050000, misses=52324
```

Performance Probleme

Producer produziert zu schnell

- Viel Speicher verschwendet
- Lösung: Eine Warteschlange mit begrenzt vielen Elementen, OK

Warteschlange ist voll/leer

- Warteschlange voll: Producer produziert nicht, OK
- Warteschlange leer: Consumer holt sich keinen Wert
  - Aber wir laufen durch die Schleife (Fehlschläge)
  - Aktives Warten (Busy wait)! CPU-Zyklen ohne Fortschritt

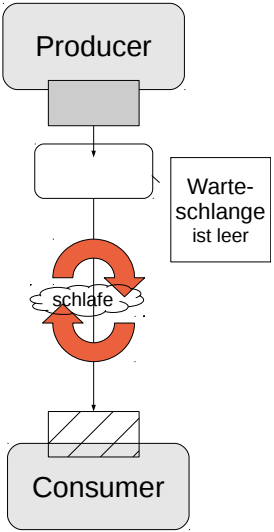
Partieller Fix: Ein bisschen schlafen, um Kontextwechsel zu fördern

- Ist **kein** Fix, nur beruhigen

Echte Lösung: Schlafen und aufwecken

- Falls Warteschlange leer ist, dann Consumer schlafen legen
- Sobald Element eingefügt wurde, den Consumer wecken

Konstrukte in Umgebung (Plattform), um zu schlafen und aufzuwecken: `wait`, `notify`



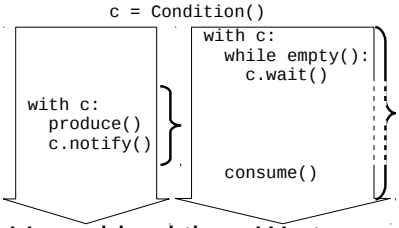
wait/notify

`threading.Condition`

- Hält intern Sperre (oder erhält Sperre beim Initialisieren)
- Implementiert `threading.Lock`-Interface `acquire/release`
- `wait/notify` als zusätzliche Methoden

`wait()`: Legt rufenden Thread schlafen

- Die Sperre muss aktuell gehalten sein
- Der aktuell rufende Thread wird schlafen gelegt (suspend) und verliert die Sperre!
- Beides in einem atomaren ununterbrechbarem Schritt
- Wiederaufwachen (resume) möglich, wenn anderer Thread auf dieser Condition `notify` ruft
- Sperre/Condition wird automatisch vor Aufwachen wieder gehalten!



Vermeide aktives Warten

`notify()`: Weckt einen schlafenden Thread

- Die Sperre/Condition muss aktuell gehalten sein
- Signalisiert **einem** wartenden Thread, der darauf schläft (`wait`), aufzuwachen
- Idiom: Direkt danach diese Sperre loslassen
- Anmerkung: Thread läuft nicht direkt, aber wird wieder verfügbar, um eine Zeitscheibe zu erhalten

`notifyAll()`: Alle auf dieser Sperre schlafenden Threads wecken

Producer mit `notify`

`notify`: Signalisiert, dass ein Element verfügbar wurde

- Sperre muss gehalten sein
- `notify`, ein Signal. In unserem Anwendungskontext: Ein Element wurde in die Warteschlange eingefügt und ist verfügbar zum Konsumieren
- System reagiert auf `notify`, indem es einen beliebigen Thread, der auf dieser Condition/Sperre schläft, weckt
  - Thread wecken heißt vom Zustand „waiting“ auf „ready“

Anmerkung: Zeilen 13/14 können ohne Probleme getauscht werden (wäre nur nicht intuitiv)

```
1  class Producer(Thread):
2
3      def __init__(self, queue, cond, how_many):
4          super().__init__()
5          self.queue = queue
6          self.cond = cond
7          self.how_many = how_many
8
9      def run(self):
10         for i in range(1, self.how_many+1):
11             with self.cond:
12                 # slow(42)
13                 self.queue.append(i)
14                 self.cond.notify()
```

Consumer mit `wait`

Schlafe auf leerer Warteschlange

- Sperre ist gehalten
- Falls Warteschlange leer, dann `wait`, lege aktuell laufenden Consumer-Thread schlafen
- Benutze immer **`while`** nicht **`if`**
  - Prüfe Bedingung in **`while`**-Schleife
  - Nie über Zustand spekulieren auf Basis der Tatsache, dass ein Thread geweckt wurde
  - Es kann und darf Weckvorgänge geben, die nichts mit dem eigenen Programm zu tun haben

Am Ende des Producers:

- Prüfe auf `value` immer noch notwendig, da am Ende es wegen `done` aufhört
- Achtung: Der Consumer kann hängen

Nur die Implementierung von `run`, Rest ist gleich

```
1  while not self.done.is_set():
2      value = None
3      with self.cond:
4          # noch relevant und nicht leer
5          while (not self.done.is_set() and
6                     not self.queue):
7              self.cond.wait()
8          if self.queue:
9              value = self.queue.pop(0)
10     if value is not None:
11         self.sum += value
12     else:
13         self.misses += 1
```

Producer/Consumer mit `wait/notify`

Verwenden

- Fast keine Änderung

Für Beendigung

- Zeile 12/13, zusätzlich `notify`, wenn Consumer hängt
- Signalisiert das Ende der Produktion
- Vermeidet endloses Blockieren auf leerer Warteschlange

Beachte: Keine Fehlschläge mehr, außer dem letzten bei Beendigung

```
sum=5000050000, misses=1
```

```
1  how_many = 100_000
2  queue = []
3  cond = Condition()
4
5  producer = Producer(queue, cond, how_many)
6  consumer = Consumer(queue, cond)
7  producer.start()
8  consumer.start()
9
10 producer.join()
11 consumer.done.set()
12 with cond: # koennte am Ende warten
13     cond.notify()
14 consumer.join()
15
16 print("sum=%d, misses=%d" %
17           (consumer.sum, consumer.misses))
```

Producer/Consumer mit `Queue`

Eingebaute Datentypen

- Eingebauter Datentypen für nebenläufige Ausführung
- Monitore, Sperre im Bauch
- Oft passend wait/notify im Bauch
- Wenn möglich eingebaute Datentypen verwenden

`queue.Queue`, Warteschlange

- Optional Größenbegrenzt
- Blockierend (`get/put`), blockierend mit Timeout und nichtblockierend (`get_nowait/put_nowait`)
- Auch Methoden zur Task-Kontrolle (Beenden)

Verwendung

- Einfacher
- Maximal 100 Elemente

```
1  how_many = 100_000
2  queue = Queue(100) # optional max size
3
4  producer = Producer(queue, how_many)
5  consumer = Consumer(queue)
6  producer.start()
7  consumer.start()
8
9  producer.join()
10 consumer.done.set()
11 consumer.join()
12
13 print("sum=%d" % consumer.sum)
```

```
sum=5000050000
```

Producer mit Queue

`Producer`

- Ganz einfach
- Blockierendes `put`, nie mehr als 100 Elemente in der Warteschlange
- Kein `slow` mehr sinnvoll oder notwendig

```
1  from queue import Queue, Empty
2
3  class Producer(Thread):
4
5      def __init__(self, queue, how_many):
6          super().__init__()
7          self.queue = queue
8          self.how_many = how_many
9
10     def run(self):
11         for i in range(1, self.how_many+1):
12             self.queue.put(i) # blocking
```

Consumer mit `Queue`

`Producer`

- Einfacher
- Blockierendes `get`, mit Timeout
  - Timeout für am Ende
  - Ansonsten eventuell ewig im `get` warten
  - Falls leer (Timeout) harmlose `Empty`-Ausnahme
  - Ist aber (unkritisches) Busy/Wait

ACHTUNG

- Alle Beispiele bisher waren **nicht** korrekt
- Falls am Ende mehr als 1 Element in der Warteschlange ist, dann werden diese Element nicht angearbeitet
- Wenn die Summe vollständig sein soll ist das ein Bug, der nur ab und zu vorkommt, Nebenläufigkeitsbug
- Fix (auch für alle vorherigen): Warteschlange am Ende leer machen (17/18)

```
1  class Consumer(Thread):
2
3      def __init__(self, queue):
4          super().__init__()
5          self.queue = queue
6          self.done = Event()
7          self.sum = 0
8
9      def run(self):
10         while not self.done.is_set():
11             value = None
12             try:
13                 value = self.queue.get(timeout=0.1)
14                 self.sum += value
15             except Empty:
16                 pass
17         while not self.queue.empty(): # drain
18             self.sum += self.queue.get()
```

Concurrent Programming

Fertige Datenstrukturen

- Concurrent Blocking Queues wie `Queue`
- Latches, Barrier, ...
- Tasks, Thread-Pools, ...
- Non-Blocking

Patterns zur Verwendung

- (Multi-)Producer/(Multi-)Consumer
- Thread Pooling
- Poison Pill
- Non-Blocking, Event-Loops
- ...

Programmieren mit Nebenläufigkeit

- Ist notwendig
  - Mehr Kerne aber nicht schnellere Kerne
  - Mehr Last da mehr Client-Devices (IoT)
  - Langsame Netze (mobil)
- Ist schwer, kritischer Wettlauf, Deadlocks, ...
- Bei Fehler passieren böse Dinge
  - 1987, Therac-25, Maschine für radiologische Therapie, 3 Tote, viele Verletzte
  - 2003, Großer Stromausfall in Nordamerika
- Kann man beherrschen

Separate Veranstaltung

`Concurrent Programming`

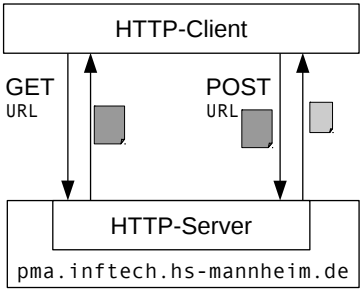
Web-Anfragen

Kommunikation über Hypertext Transfer Protocol (HTTP)

- Einfaches Protokoll, um Dokumente (Text/binär) über das Internet auszutauschen
- Basis des Web, GET Holen, POST Senden
- Basiert auf Internet-Protokoll TCP (Transmission Control Protocol) $\rightsquigarrow$ CN2

Uniform Resource Locator (URL)

- Eindeutige Adresse im Web
  `https://pma.inftech.hs-mannheim.de/static/bild.png`
  `https://pma.inftech.hs-mannheim.de/wsgi/rand?min=10&max=20`
  - Protokoll, `http` oder `https`
  - Hostname, `pma.inftech.hs-mannheim.de`
  - Pfad, `/static/bild.png`, `/wsgi/rand`
  - Query-Parameter: Mit `?` angehängte Schlüssel/Wert-Paare mit `&` separiert, `?min=10&max=20`
- In Python `urllib.request`
  - HTTP-Anfrage mit einer URL
  - Aus String aufbauen oder erweitern



Zufallszahl holen

URL bauen und Holen

- URL zusammenbauen, Default-Methode `GET`
- URL Query-Parameter mit `urlencode` aus Dictionary
  - `urlencode` wegen Sonderzeichen (z.B. Leerzeichen +)
- Holen mit `urlopen`, `response` als Antwort, wie offene Datei/Strom
  - `reponse.status` 200 OK, 404 not found
  - ...
- Strom vollständig lesen und konvertieren

```
1  import urllib.request
2  import urllib.parse
3
4  BASE_URL = "https://pma.inftech.hs-mannheim.de"
5  def read_random(min=None, max=None, debug=False):
6      randurl = BASE_URL+"/wsgi/rand"
7      dict = {}
8      if min is not None:
9          dict["min"] = min
10     if max is not None:
11         dict["max"] = max
12     if dict:
13         randurl += "?" + urllib.parse.urlencode(dict)
14     if debug:
15         print(urllib.parse.urlparse(randurl))
16     response = urllib.request.urlopen(randurl)
17     return int(response.read())
```

Zufallszahl holen

Verwenden

- Einfacher Aufruf mit optionalen Parametern
- Zufallszahlen, bei jedem Aufruf ein anderes Ergebnis

Ergebnis von `urlparse`

- Named tuple, wie Dictionary

```
1 rand = read_random()
2 randmm = read_random(17, 42, debug=True);
3 print("  rand=%d" % rand)
4 print("randmm=%d" % randmm)
```

```
  rand=79019942
randmm=38
```

```
ParseResult(scheme='https',
  netloc='pma.inftech.hs-mannheim.de',
  path='/wsgi/rand',
  params='',
  query='min=17&max=42',
  fragment='')
```

POST-Anfrage

Anwendung

- Ein Speicher auf einem Web-Server
- Unterstützt Lesen (GET) und Schreiben (POST)
- Lesen, wie vorher

Daten aufbereiten

- In `byte`-String konvertieren
- Passendes Encoding, UTF-8

POST-Anfrage schicken

- Eine `POST`-Anfrage
  - wenn bei `Request` `data` gesetzt
- `Content-Type` muss sein!
- Schicken mit `urlopen`
- Antwort nur wenn es geklappt hat, aktuell gespeichertes zurück

```
1  PMA = "https://pma.inftech.hs-mannheim.de"
2  PMASTORE = PMA + "/wsgi/store"
3  def store_get():
4      resp = urllib.request.urlopen(PMASTORE)
5      if not resp.status == 200:
6          return None
7      return resp.read().decode("UTF-8")
```

```
1  def store_post(msg):
2      data = msg.encode("UTF-8")
3      # mit data automatisch POST
4      req = urllib.request.Request(PMASTORE,
5                                   data=data)
6      req.add_header("Content-Type", "text/plain")
7      resp = urllib.request.urlopen(req)
8      if not 200 <= resp.status <= 299: # 2xx is ok
9          return None
10     return resp.read().decode("UTF-8")
```

POST-Anfrage verwenden

Verwenden

- Beliebigen Text auf Server ablegen
- Zustand auf dem Server, kann sich zu jedem Zeitpunkt beliebig ändern bei nebenläufigen Anfragen

Andere Verwendungen

- Formulare
  - Content-Type `application/x-www-form-urlencoded`
  - wie Query-String, dann bytes
  - HTML-Formulare schicken das
- Irgendwas anderes
  - Bilder, JSON, ...
  - Passender Content-Type

Ausblick: Web-Services, REST

```
now = "Hello %d" % int(time.time()*1000)
print("storing %s" % now)
posted = store_post(now)
if posted != now:
    print("something changed that fast?\n")
    print("  we posted: |%s|" % now)
    print("  we got   : |%s|" % posted)
time.sleep(1.0)
msg = store_get()
print("getting %s" % msg)
```

```
storing Hello 1563862027899
getting Hello 1563862027899
```

Ausblick: Netzwerkprogrammierung mit Sockets

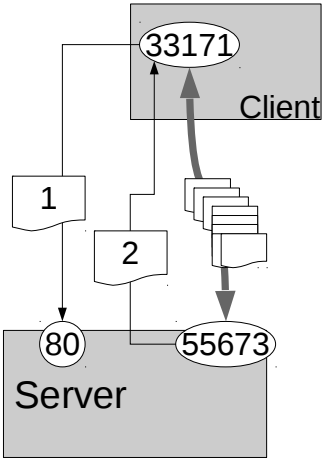
Sockets

- Abstraktion (bidirektionaler) Verbindung, TCP
- Grundlage für fast alles andere
- Server akzeptiert auf einem Port (z.B. 80 für HTTP oder 443 für HTTPS)
- Öffnet bidirektionale Verbindung, auf zwei anderen Ports Ströme
- Server kann weiter neue akzeptieren

Ströme

- Input und Output von Verbindung
- Schicken von beliebigen Byte-Blöcken
- Selbst um Protokoll kümmern
- Ein Protokoll war HTTP (HTTPS)

Eine Ebene tiefer, `CN2`



Bibliotheken: Module in der Standardbibliothek

Python-Bibliotheken – viel im Standard

- Als Module/Pakete verfügbar
- Breites Angebot, „Batteries included“, gute Doku

Beispiele:

- `os`: Systemnah, Verzeichnisse/Dateien
  `sys`: Python-Umgebung
- `unittest`: Unit Testing
  `pickle`: Objektpersistenz
  `urllib, cgi, imap, ftp, http`: Web, Internet
  `PIL(pillow)`: Bildbearbeitung (nicht standard)
  `turtle`: Einfachstgraphik, `tkinter`
  `re`: Reguläre Ausdrücke

Zusätzliche Bibliotheken, Binding an Abertausende

- GUI, alle OS: `wx, PySide2 (Qt5), vtk`, ...
- Math: `numpy, matplotlib, scipy, simpy`, ...
- KI: `tensorflow, keras, scikit`, ...
- Web, Dokumente (`pygments`), μC `micropython`,

```
>>> import os
>>> os.getcwd()
'/home/student/nethome'
>>> os.chdir("/usr/share/games/fortunes")
>>> os.getcwd()
'/usr/share/games/fortunes'
>>> [x for x in os.listdir(os.getcwd())
       if x.endswith(".dat")]
['fortunes.dat', 'riddles.dat',
 'literature.dat']
>>> import sys
>>> sys.argv
['']
>>> import re
>>> re.findall(r'ab+c', "abbcdsabcab")
['abbc', 'abc']
```

Unit Testing

Ziel

- Funktionalitäten feingranular testen
- Automatisierung, einfache Verwaltung von Testfällen

Umsetzung – `unittest`, Verwendung

- Anlehnung an `JUnit`
- Von Klasse `TestCase` erben
- Testfunktionen mit `test_` am Anfang des Namens
- Zusicherung, `self.<name>`: Erwartet
  - `assertTrue`: True, wie **`assert`**
  - `assertEqual`: Zwei gleich Parameter
  - `assertRaises`: Passende Ausnahme
- `setUp, tearDown`: Vor/nach jedem Test, Test Fixture
- Aufrufen mit `unittest.main()`

```
import unittest
class <testfall>(unittest.TestCase):
    [def setUp(self):
        ...
    ]
    [def tearDown(self):
        ...
    ]
    [def test<einzeltest>(self):
        ...
        [self.assert_(<testausdruck>) |
        [self.assertEqual(<o1>, <o2>) |
        [self.assertRaises(<ausnahme>)]*
        ...
      ]*

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

Anpassbar, flexibel, weitere Optionen

Unit Testing – Beispiel

Beispieltest für „Ist Primzahl?“

prim.py

```
1 def is_prim(x):
2     if x < 2: return False
3     ts = [t for t in range(2, x)
4                 if x%t==0]
5     return not ts
```

- Zu testende Funktion `is_prim`, in Modul `prim`
- Testfunktionen, Start der Tests
- Ausgabe: Zwei Tests ok durchgelaufen

```
$ python3 primtest.py
..
------------------------------
Ran 2 tests in 0.001s

OK
```

primtest.py

```
1  import unittest
2  from prim import is_prim
3
4  class PrimTest(unittest.TestCase):
5      def setUp(self):
6          self.nums = range(1, 101) # 1..100
7          self.primnums = 25 # 25 Primzahlen
8      def testPrim1(self):
9          self.assertTrue(is_prim(2))
10         self.assertTrue(not is_prim(4))
11         self.assertFalse(is_prim(6))
12         self.assertEqual(True, is_prim(7))
13     def testPrim2(self):
14         prims = list(filter(is_prim, self.nums))
15         self.assertEqual(self.primnums, len(prims))
16
17 if __name__ == '__main__':
18     unittest.main()
```

Pickle – Einfache Objektpersistenz

Ziel

- Serialisierung von Objektinstanzen
- Plattform- und versionsunabhängig
- Kein Persistenzframework

Umsetzung – `pickle`

- Für eingebaute Standardtypen
- Hierarchien und Referenzen
- Objektorientierung, aber nur Instanzen
- Ähnlich zu `Serializable` in Java

Beispiel

- Schreiben mit `dump`
- Lesen mit `load`

aclass.py

```
1  class AKlasse:
2      def __init__(self, end=3):
3          self.lis = list(range(1, end))
4      def __str__(self):
5          return "A["+str(self.lis)+"]"
6      __repr__ = __str__
```

```
1  import pickle
2  from aclass import AKlasse
3
4  tup = (1, 1.0, 1+1j, (1,), "1", {1:2})
5  lis = [1,2,3]
6  lis.append(lis)
7  alis = [AKlasse(i) for i in range(1, 6)]
8  pickle.dump((tup, lis),
9              open("tuplis.pickle", "wb"))
10 pickle.dump(alis,
11             open("alis.pickle", "wb"))
```

Pickle – Einlesen

Beispiel

- Referenzen, auch zirkuläre, werden richtig wiederhergestellt
- Instanzen von Klassen werden richtig wieder instanziiert
  - Klasse/Modul muss verfügbar sein, wird nicht gepickelt
  - Einladen über `import` muss klappen

Erweiterungen

- Pickeln von Instanzen über Methoden beeinflussbar
  - Pickeln von Klassen oder Funktionen nicht möglich
- Unterschiedliche Protokollversionen
- `shelve`: key/value store

```
>>> import pickle
>>> of = open("tuplis.pickle", "rb")
>>> tup, lis = pickle.load(of)
>>> tup
(1, 1.0, (1+1j), (1,), '1', {'1': 2})
>>> lis
[1, 2, 3, [...]]
>>> alis = pickle.load(open("alis.pickle", "rb"))
>>> alis
[A[[]], A[[1]], A[[1, 2]], A[[1, 2, 3]],
 A[[1, 2, 3, 4]]]
>>> alis[0].__class__
<class 'aclass.AKlasse'>
```

Internetdaten verarbeiten

- `email`, `mimetypes`, `mailcap`, `imaplib`, `mailbox`, `smtplib`, `nntplib`:
  Email- und News-spezifische Bibliotheken
- `uu`, `binascii`: Konvertierungen
- `socketserver`, `http`, `http.server`, `ftplib`: Protokollimplementierungen
- `urllib.(parse|request)`: URL-Manipulation, Anfragen

```
>>> import urllib.request, re
>>> s = urllib.request.urlopen("https://www.heise.de/newsticker/").read().decode("utf-8")
>>> s[:100]
'<!DOCTYPE html>\n<html lang="de"\n      data-responsive\n      >\n\n  <head>\n
 \n    <title>7-Tage-News '
>>> m = re.search(r'<a\n *href="/.*/meldung/(.*?).html"', s).group(1); m
'StopSpyingOnUs-Buergerrechtler-fordern-Massnahmen-gegen-Werbeauktionen-4439456'
>>> len(re.findall(r'<a\n *href="/.*/meldung/(.*?).html"', s))
163
>>> s[s.find(m)-124:s.find(m)+200]
'<li class="archiv-liste__item"\n          >\n            <article>\n              <a\n
href="/newsticker/meldung/StopSpyingOnUs-Buergerrechtler-fordern-Massnahmen-gegen-Werbeaukti
title="#StopSpyingOnUs: Bürgerrechtler gegen Kontrollverlust bei Online-Werbung"\n
```

(X|HT)ML mit passenden Tools

Keine regulären Ausdrücke für (X|HT)ML verwenden

- Die Auszeichnungssprachen sind kontextfreie Sprachen, nicht pfuschen
- `stackoverflow.com/questions/1732348/regex-match-open-tags-except-xhtml-self-contained-tags/1732454#1732454`

Parsen mit BeautifulSoup (externe Bibliothek)

```
>>> import bs4, urllib.request
>>> s = urllib.request.urlopen("https://www.heise.de/newsticker/").read().decode("utf-8")
>>> doc = bs4.BeautifulSoup(s, "lxml")
>>> hls = [a.h1.getText().split("\n")[2].strip() for a in doc.select("article")]
>>> for t in [t for t in hls if "Linux" in t]: print(t)
Do svidaniya Windows: Russisches Militär kann komplett auf Astra Linux umsteigen
```

Installieren von BeautifulSoup

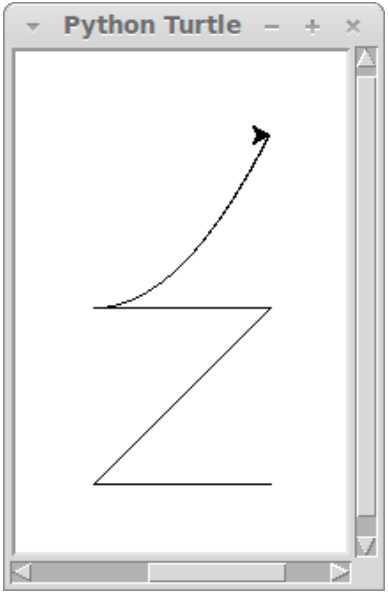
- `sudo apt-get install python3-bs4` oder `pip3 install beautifulsoup4`
- `www.crummy.com/software/BeautifulSoup/bs4/doc/`

Turtle Graphics

Turtle Graphics

- Einfache Graphiken wie mit einem Stift erstellen, „Logo, Programming for Kids“
- Zeichenfläche mit Mitte 0,0
- Kommandos `up` (Stift hoch), `down` (Stift auf Papier), `goto` (Bewege Stift), ...

```
from turtle import *
down()
goto(100, 0)
goto(0, -100)
goto(100, -100)
up()
goto(0, 0)
down()
for i in range(100):
    goto(i, i**2/100.)
```

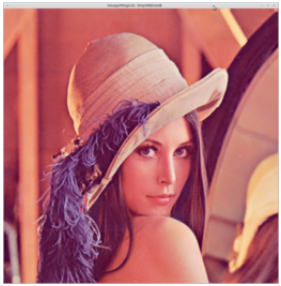


Bildbearbeitung mit PIL

Python Imaging Library (PIL)

- C-Erweiterung, nicht im Standard
- Einfach, mächtig, `pillow` in Python3

```
>>> from PIL import Image
>>> im = Image.open("lena.png")
>>> im.show()
>>> im2 = im.resize(map(lambda x: x*2, im.size))
>>> im2.show()
>>> im3 = im.rotate(45)
>>> im3.show()
>>> from PIL import ImageFilter
>>> im4 = im.filter(ImageFilter.FIND_EDGES)
>>> im4.show()
```



Bildbearbeitung mit PIL pixelweise

Mache Graustufenbild

- Durch alle Pixel, je Pixel
- Berechne Graustufenwert (nach Wahrnehmung)
- Speichere Graustufenwert

```
#!/usr/bin/python3
from PIL import Image

im = Image.open("lena.png")
width, height = im.size
RF, GF, BF = 0.3, 0.6, 0.1 # besser als je 1/3
for i, j in ((i, j) for i in range(width)
                    for j in range(height)):
    r, g, b = im.getpixel((i, j))
    gval = int(r*RF + g*GF + b*BF)
    im.putpixel((i, j), (gval, )*3)
im.save("lena_grey.png")
```

Integration mit C, `ctypes`

`ctypes`

- Standard Python Modul zur Integration von C-Funktionalität
- Dynamische Bibliotheken zur Laufzeit
- Typen, teilweise automatische Konvertierung
- Bidirektionale Aufrufe möglich

Verwendung

- Dynamische Bib. laden (.dll Win)
- Funktionen in Bib. verwenden
  - Python Strings 'utf-8', C erwartet `char`
- Anlegen von C-Variablen, Übergabe als Zeiger, änderbarer Wert, mit `.value`
- Felder
- Automatische Konvertierung von `int`, `double`, `char *`

```
>>> import ctypes
>>> libc = ctypes.cdll.LoadLibrary("libc.so.6")
>>> libc
<CDLL 'libc.so.6',
      handle 7fc68018c4e8 at 0x7fc67b37d9b0>
>>> libc.printf(b"hello %s world", b"ctypes")
18
hello ctypes world
>>> i, f = ctypes.c_int(), ctypes.c_float()
>>> libc.sscanf(b"42 3.1415 rest", b"%d %f",
                ctypes.byref(i), ctypes.byref(f))
2
>>> i.value, f.value
(42, 3.1414999961853027)
>>> cs = string.ascii_letters.encode("ascii")
>>> arr = (ctypes.c_char*53)(*cs)
>>> n = libc.strlen(arr)
>>> n
52
```

Integration mit C, Fallbeispiel `MaxSubArray`

Aufgabenstellung: `MaxSubArray`

- Suche in Feld von ganzen Zahlen ein zusammenhängendes Teilfeld, dessen Summe maximal ist
- Einfacher quadratischer Algorithmus
- Nur `int`-Arithmetik

Problem

- Python für solche Aufgaben nicht gemacht
- Python3 wurde für solche Aufgaben sogar langsamer!
  - Man soll so etwas nicht in Python machen

Lösung

- Solche Basisalgorithmen (Pixel schubsen, Signalverarbeitung, ...) in C machen und integrieren

```
1  def maxsub_py(lis):
2      max_sum, start, end = 0, 0, -1
3      n = len(lis)
4      for i in range(n):
5          sum = 0
6          for j in range(i, n):
7              sum += lis[j] # sum == sum(lis[i:(j+1)])
8              if sum > max_sum:
9                  max_sum = sum
10                 start, end = i, j
11     return max_sum, start, end
```

Beispiel

- Alle 6666 Zahlen zwischen -3332 und 3333
- Durcheinandergewürfelt mit `random.shuffle`
- `MaxSubArray` lösen, Zeit messen

```
maxsub_py : n= 6666,
val=163394, start= 255, end= 4751, took= 2.307s
```

Integration mit C, Fallbeispiel `MaxSubArray`

`MaxSubArray` in C

- Identisch zu realisieren
- Weder schöner noch hässlicher
- C gut geeignet für solche Aufgaben

Übersetzen als dynamische Bibliothek

- `gcc -O3 -std=c99 -Wall -Wvla -pedantic -fPIC -shared maxsub.c -o maxsub.so`
- Hohe Optimierung, C99 Standard, alle Warnungen
- Positionsunabhängiger Code als dynamische Bib.
- Quellcode und Ausgabe als `.so`

Kann so schon integriert werden

- Beachte Parameter- und Rückgabetypen

```c
int maxsub_c(int *arr, int n, int *start, int *end)
    int max_sum = 0;
    *start = 0;
    *end = -1;
    for (int i=0; i < n; i++) {
        int sum = 0;
        for (int j=i; j < n; j++) {
            sum += arr[j]; // sum == sum(arr[i..j])
            if (sum > max_sum) {
                max_sum = sum;
                *start = i;
                *end = j;
            }
        }
    }
    return max_sum;
}
```

Integration mit C, Fallbeispiel `MaxSubArray`

Laden der dynamischen Bibliothek

- Absoluter Pfadname, sicherer
- Laden, einmal würde reichen
- Argumenttypen genau festlegen, sicherer

Verwenden

- Liste in C-Array konvertieren
  - Anlegen
  - Kopieren
- Variablen für Start und Ende
- Aufruf wie erwartet, Ergebnis automatisch konvertiert zu **`int`**

Effekt: viel schneller

- Ca. 2 Größenordnungen (100 Mal) schneller
- Anteil der Konvertierung der Liste, `0.003s`

```
1 def get_cmaxsub():
2     path = os.path.join(os.getcwd(), 'maxsub.so')
3     clib = ctypes.cdll.LoadLibrary(path)
4     c_int_p = ctypes.POINTER(ctypes.c_int)
5     c_int = ctypes.c_int # shorter
6     argtypes = [c_int_p, c_int, c_int_p, c_int_p]
7     clib.maxsub_c.argtypes = argtypes
8     return clib.maxsub_c
```

```
1 def maxsub_c(lis):
2     n = len(lis)
3     arr = (ctypes.c_int*n)(*lis) # expensive
4     start, end = ctypes.c_int(0), ctypes.c_int(-1)
5     cmaxsub = get_cmaxsub()
6     max_sum = cmaxsub(arr, n, ctypes.byref(start),
7                                ctypes.byref(end))
8     return max_sum, start.value, end.value
```

```
maxsub_c : n= 6666,
val=163394, start= 255, end= 4751, took= 0.017s
```

Integration mit C, Fallbeispiel `MaxSubArray`

Alternative: Anderes Python

- `pypy`, Schnellere Python-Umgebung mit JIT-Compiler
- Laufzeit ca. 0,1s
- Nur noch eine Größenordnung langsamer
- Häufig ausreichend, wenn alles unterstützt
- `cython`, einfachere C-Integration

Bessere Alternative: Besserer Algorithmus

- Scanline-Verfahren
- $O(n)$ statt $O(n^2)$
- Bessere Komplexität, viel schneller
- C-Variante jetzt unrentabel, Konvertierung auf jeden Fall teurer

```
1  def maxsub_spy(lis):
2      n = len(lis)
3      count = 0
4      bis_sum, bis_start, bis_end = 0, 0, -1
5      sca_sum, sca_start, sca_end = 0, 0, -1
6      for i in range(n):
7          count += 1
8          sca_end = i
9          sca_sum += lis[i]
10         if sca_sum <= 0:
11             sca_sum,sca_start,sca_end = 0,i+1,i
12         elif sca_sum > bis_sum:
13             tup = sca_sum, sca_start, sca_end
14             bis_sum,bis_start,bis_end = tup
15     return bis_sum, bis_start, bis_end
```

```
maxsub_c : n= 6666,
val=163394, start= 255, end= 4751, took= 0.002s
```

Python und C

Allgemeine Optimierungsregeln

- Don't
- Don't Yet
- Measure

`PrematureOptimization is the root of all evil`, *Don Knuth*

Optimierung in Python

- Erst in Python realisisieren
  - Eingebaute Datentypen verwenden
  - Bibliotheken verwenden
- Dann messen, Problemstelle verifizieren
  - Bessere Algorithmen immer besser
- C nur für kleine Blöcke oder Integration von existierenden Code
  - Da wo C passt
  - Bits schubsen, Basisalgorithmen, viele einfache Iterationen

```
# ipython3, integrated microbenchmarking
>>> def sum1(lis): return sum(lis)
>>> def sum2(lis):
    s = 0
    for i in range(len(lis)):
        s += lis[i]
    return sum
>>> %timeit sum1(range(10001))
1000 loops, best of 3: 171 µs per loop
>>> %timeit sum2(range(10001))
100 loops, best of 3: 1.55 ms per loop
# python3, manual microbenchmarking
>>> import timeit
>>> setup = "from __main__ import sum1, sum2"
>>> timeit.repeat("sum1(range(10001))",
                  setup=setup, number=1000)
[0.2467009740, 0.17174179598, 0.17239759300]
>>> timeit.repeat("sum2(range(10001))",
                  setup=setup, number=100)
[0.2343949269, 0.15916752500, 0.15549547702]
```

Reguläre Ausdrücke

Reguläre Ausdrücke

- „Matchen“, finde passenden String
- Flexibles Matchen auf „Pattern“ mit Sonderzeichen
- Erweiterte reguläre Ausdrücke, nicht Wildcards
- In Java, Python, ... verfügbar, Kommandozeile mit `egrep`

Reguläre Ausdrücke in Skriptsprachen

- Spezielle Bibliotheken (Python, Java, C++) oder Teil der Sprache (Perl, awk)
- Kompilation des Patterns, Ausführen auf C-basierter Engine

Einsatz

- Stringverarbeitung, Extraktion, Suchen/Ersetzen
- Test auf syntaktische Korrektheit (Email-Adresse, Telefonnummer)

Matchen mit einfachen Patterns

Teil des Strings

- Ein Pattern beschreibt einen Teil des zu durchsuchenden Strings

Pattern – Zeichen matchen sich selbst

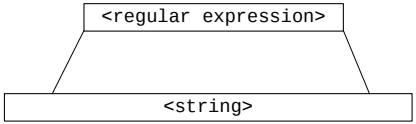
- Zum Beispiel matcht das Pattern `fahr` den String `erfahren`

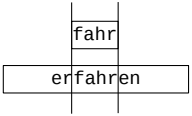
Groß- und Kleinschreibung wird unterschieden

- Das Pattern `Fahr` matcht **nicht** den String `erfahren`

Sonderzeichen haben spezielle Bedeutung

- `. ^ $ * + ? { } [ ] \ | ( )`





Auswahlbereiche für ein Zeichen

Auswahl mit `[]`

`[abcde]`	Einzelnes Zeichen: Matcht entweder `a`, `b`, `c`, `d` oder `e`
`[a-z]`	Bereich: Matcht jeden (einen) Kleinbuchstaben (ASCII)
`[a-zA-Z]`	Bereich: Matcht Groß- und Kleinbuchstaben
`[abc$^]`	Sonderzeichen nicht aktiv in Bereichen. Matcht `a`, `b`, `c`, `$` oder `^`
`[^a-z]`	`^` am Anfang negiert den Bereich. Matcht alles außer Kleinbuchstaben
`[ab\cd]`	hebt die besondere Bedeutung für `\` auf. Matcht `a`, `b`, `\`, `c` oder `d`

Vordefinierte Auswahlbereiche

`\d`	Alle Ziffern, `[0-9]`
`\D`	Alles außer Ziffern, `[^0-9]`
`\s`	Whitespace, `[ \t\n\r\f\v]`
`\S`	Alles außer Whitespace, `[^ \t\n\r\f\v]`
`\w`	Alphanumerische Zeichen, `[a-zA-Z0-9]`
`\W`	Alles außer alphanumerischen Zeichen, `[^a-zA-Z0-9]`

Auch für Nicht-ASCII Zeichensätze

- Beispiel Deutsch: Umlaute `äöüÄÖÜß` werden von `\w` erkannt
- Achtung: `locale`-Einstellungen werden berücksichtigt
- Unicode? wieder alles anders...

Wiederholungen und Gruppierungen

<pattern>	matcht	<beispiel1>, <beispiel2>, ...
`Pa*gei`		`Pgei, Pagei, Paagei, ...`
`Pa+gei`		`Pagei, Paagei, ...`
`Pa?gei`		`Pgei, Pagei` und sonst nichts
`P[ae]*gei`, wie `P(a|e)*gei`		`Pgei, Pagei, Pegei, Paagei, Paegei, Peagei Peegei, Paaagei, ...`
`(Pap|Mam)agei`		`Papagei, Mamagei` und sonst nichts
`(Pa)+gei`		`Pagei`, `PaPagei`, `PaPaPagei`, ...
`Pa(pa)+gei`		`Papagei`, `Papapagei`, ...
`Pa(pa){1,3}gei`		`Papagei`, `Papapagei`, `Papapapagei` und sonst nichts

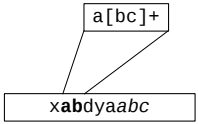
Besondere Zeichen und Bedeutung

`.`	beliebiges Zeichen	`{m,n}`	Wiederholung, mindestens `m` mal und maximal `n` mal
`*`	Wiederholung 0, 1 oder mehrfach	`{n}`	Exakt `n` mal
`?`	Wiederholung 0 oder 1 mal	`[`	Beginn Auswahl aus Zeichen
`+`	Wiederholung 1 oder mehrfach	`]`	Ende Auswahl aus Zeichen
`(`	Beginn einer Gruppierung	`^`	Anfang String/Zeile
`)`	Ende einer Gruppierung	`[^`	Negation/Auswahl
`{`	Beginn explizit gezählte Wiederholung	`$`	Ende String/Zeile
`}`	Ende explizit gezählte Wiederholung	`|`	Oder-Auswahl von Gruppierungen

Gierige Ausführung

Von links nach rechts

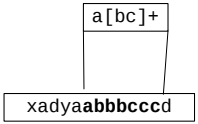
- Der erste passende Teilausdruck wird gematcht
- Später passende Teilausdrücke werden ignoriert



matcht **ab**,
nicht *abc*

So viel wie möglich

- Bei dem ersten passenden Teilausdruck
- So viel wie möglich Zeichen in den Match bringen



matcht **abbbccc**,
nicht Teil davon

Beispiele

Binärzahlen	`(0|1[01]*)`	ohne führende Nullen
Ganze Zahlen	`0|([1-9][0-9]*)`	ohne führende Nullen
Pi	`3\.14`	
Emailadresse	`\w+([.-]\w+)*@[a-z]\w+(\.[a-z]\w+)+`	

Einfaches Ausprobieren in der Shell (`bash`)

- `echo <String> | egrep "<RE>"` # matcht Teilstring
- `echo <String> | egrep "^(<RE>)$"` # exakter Match

Beispiel

- ```
  $ echo "11110" | egrep "^(0|(1[01]*))$"
  11110
  $
  ```
- ```
  $ echo "011110" | egrep "^(0|(1[01]*))$"
  $
  ```

Reguläre Ausdrücke und Wildcards

Reguläre Ausdrücke

Matchen von Strings

In Perl, Python, egrep, ...

(Meist) erweiterte reguläre Ausdrücke

Beispiel Unterschiede

- a∗ passt auf beliebig viele a's; ∗ steht für die Wiederholung des vorherigen Zeichens
- a? passt auf leeres Wort oder a

...

Wildcards

Matchen von Strings

In Shell, DOS, grep

Standard Wildcards

Beispiel Unterschiede

- a∗ passt auf jeden String der mit a beginnt; ∗ steht für beliebige Zeichenfolge
- a? passt auf jeden zwei Zeichen langen String, der mit a beginnt

Weitere Probleme

- Groß-/Kleinschreibung unterschieden?
- Das $-Zeichen als Beginn einer Variable

Literatur und Online-Quellen für Reguläre Ausdrücke

Reguläre Ausdrücke allgemein

- `regex101.com`
- Die Referenz:
  Jeffrey E.F. Friedel
  Reguläre Ausdrücke
  O'Reilly

Reguläre Ausdrücke in Python

- `docs.python.org/3.6/library/re.html`
- `docs.python.org/dev/howto/regex.html`

re-Modul – Reguläre Ausdrücke in Python

Reguläre Ausdrücke in Python

- `re`-Modul
- Matche, modifiziere, zerteile, ersetze, ...
- Kompilation des „Patterns“
- Ausführen auf C-Engine

Ausführlich dokumentiert – Lesen!

Einfaches Beispiel

- `compile`: Pattern kompilieren
- `match`: Matchen
- `group`: Ergebnis (Matches) ausgeben
- ...oder alles auf einmal

```
>>> import re
>>> pattern = re.compile("[a-z]+")
>>> match = pattern.match("abc")
>>> match.group()
'abc'

>>> re.match(r'[a-z]+', "abc").group()
'abc'
```

Beispiel – Kompilieren und Matchen

Kompilieren

- Modul-Funktion `compile`
- Kompilieren eines regulären Ausdrucks erzeugt Patternobjekt

Matchen

- Methoden des Patternobjekts
- Mit `match` Suche in einem String ab Anfang
- Mit `search` Suche in einem String irgendwo
- Ergebnis ist Matchobjekt wenn erfolgreich, sonst `None`
- Aus Match-Objekt mit `group` den Match (String) extrahieren

```
>>> import re
>>> pattern = re.compile(r'[a-z]+')
```

```
>>> match = pattern.match('abcABC')
>>> match.group()
'abc'
>>> match = pattern.match('ABCabcABC')
>>> match
>>> print(match)
None
>>> match.group()
AttributeError: 'NoneType' object
          has no attribute 'group'
>>> match = pattern.search('ABCabcABC')
>>> match.group()
'abc'
```

Beispiel – Optionen und Kompakt

Optionen

- `re.IGNORECASE` ignorieren von Groß-/Kleinschreibung
- Weitere Optionen: `re.LOCALE`, `re.MULTILINE`, `re.DOTALL`, `re.UNICODE`, ...
- Optionen mit Oder | verknüpft

Kompakter Code

- `re`-Modul stellt Funktionen zur Verfügung
- Alles in einem Ausdruck möglich

```
>>> pattern = re.compile(r'[a-z]+', re.IGNORECASE)
>>> match = pattern.search('ABCabcABC')
>>> match.group()
'ABCabcABC'
```

```
>>> re.search('[a-z]+', "ABCabcABC")
<_sre.SRE_Match object; span=(3, 6), match='abc'>
>>> re.search('[a-z]+', "ABCabcABC").group()
'abc'
>>> re.search('[a-z]+', "ABCabcABC",
              re.IGNORECASE).group()
'ABCabcABC'
```

Backslash-Zirkus

Problem

- Mit \ (Backslash) werden Sonderzeichen Ihres besonderen Status beraubt
- Viele Backslashes in Python sind ein Problem
  - Text `\section` matchen benötigt `\\\\section` als Pattern
  - Ein \, um das Backslash im Python String zu haben
  - Die beiden anderen, um das zweite Backslash für den regulären Ausdruck zu haben

Lösung

- Benutze „raw“ Strings in dem alle Zeichen ohne Sonderstatus vorkommen
- `r"\\section"` entspricht `"\\\\section"`

```
>>> re.search('\\\\section', "dies ist \\section wichtig").group()
'\\section'
>>> print(re.search('\\\\section', "dies ist \\section wichtig").group())
\section
>>> re.search(r'\\section', "dies ist \\section wichtig").group()
'\\section'
```

Gruppierungen

Gruppierungen

- Teilbereich im Pattern, die referenziert werden können
- Klammern im Pattern

Zugriff auf Gruppierungen

- `groups()`: Alle Gruppierungen
- `groups(i)`: Die `i`.-te Gruppe, definiert durch `i` öffnende Klammern im Pattern
- `group(0)`, `group()`: Der gesamte Match

Zurückreferenzieren durch `\i` im Pattern

- Beispiel: Suche doppeltes Vorkommen

```
>>> pat = r'(a(b)c)d'
>>> s = "abcd"
>>> re.search(pat, s).groups()
('abc', 'b')
>>> re.search(pat, s).group()
'abcd'
>>> re.search(pat, s).group(0)
'abcd'
>>> re.search(pat, s).group(1)
'abc'
>>> re.search(pat, s).group(2)
'b'
>>> re.search(pat, s).group(3)
IndexError: no such group
>>> re.search(pat, s).group(0, 1, 2)
('abcd', 'abc', 'b')
>>> s = "Ist doppelt doppelt drin"
>>> pat = r'([a-z]+ )\1'
>>> print(re.search(pat, s).group())
doppelt doppelt
```

Modifizieren von Strings

Zerteilen von Strings

- `re.split(<pattern>, <string>)`
- Gibt Liste von Strings zurück
- Zerteilt `<string>` bei jedem Match von `<pattern>`

Suchen und Ersetzen

- `re.sub(<pat>,<neu>,<string>)`
- Gibt neuen String zurück
- Bei jedem Match von `<pattern>` in `<string>` wird `<neu>` stattdessen eingesetzt

```
>>> pat = r'\W+'
>>> s = "Das ist ein netter kleiner Test"
>>> re.split(pat, s)
['Das','ist','ein','netter','kleiner','Test']
```

```
>>> pat = '(blaue|weisse|rote)'
>>> neu = "bunte"
>>> s = "Die blaue Socke und der rote Ball"
>>> re.sub(pat, neu, s)
'Die bunte Socke und der bunte Ball'
```

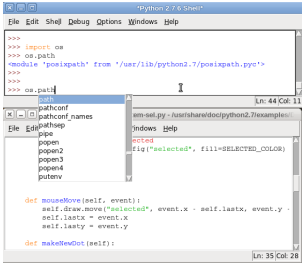
GUI-Programmierung mit Python

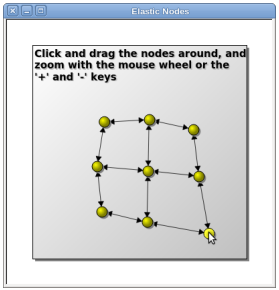
Anwendungsentwicklung mit Python

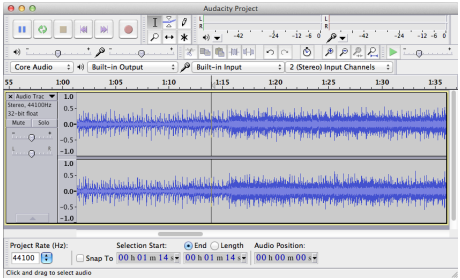
- Sehr gut geeignet, hohe Abstraktionsebene
- Offen und integrativ, viele Toolkits/Paradigmen
  - Web-basierte Anwendungen (Django, Flask, . . .)
  - GUI-basierte Anwendungen (Tk, Qt, wx, vtk)

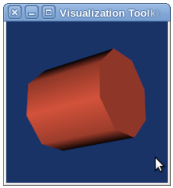
GUI-Toolkit Bindings – plattformunabhängig, nativ

- Tcl/Tk, Tkinter: C/Tcl, standard, high level, einfach
- Qt: C++, professionell, mächtig, komplex
- wx: C++, Alternative zu Qt, offen
- vtk: C++, Visualization Toolkit, 3D









Tkinter

Tcl (Tool Command Language) / Tk (Tool Kit), `www.tcl.tk`

- 1990: John Ousterhout, Berkeley, GUI gebraucht, public distribution
- 1993: Verbreitung, erstes einfach zu nutzende GUI-Umgebung
- 1994: Sun, 1998: Scriptics, 1999: Tcl/Tk 8.1, 2000: Community
- 2007: Tcl/Tk: 8.5, 2012: Tcl/Tk: 8.6, 2018: Tcl/Tk 8.6.9
- Widgets, Canvas, plattform-unabhängige Entwicklung aber native Look & Feel, themes (ttk), Bindings (OpenGL), Fonts (AntiAlias), ...

Tkinter

- Standard GUI-Toolkit in Python, Fredrik Lundh, 199x
- Dünner Layer zwischen Tcl/Tk und Python
- Einfach zu bedienen und zu verstehen
- Seit Python 2.7/3.1 `ttk`, themed (native look & feel)

Dokumentation

- `docs.python.org/3/library/tkinter.html`
- `www.tkdocs.com/tutorial/index.html`

GUI-Programmierung, Frameworks

Framework

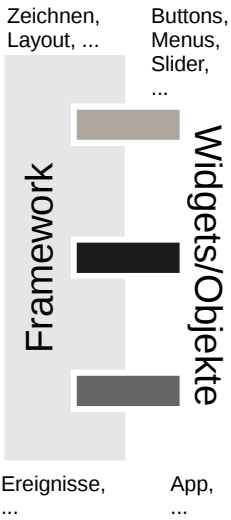
- Ablaufumgebung für spezielle Objekte
- Standardkonzept für graphische Benutzungsoberflächen

Widgets (Objekte), LayoutManager

- Widgets sind Visualisierungs- und Interaktionskomponenten
- Button, Slider, Label, CheckBox, . . .
- Stellen Methoden zur Verfügung (Vererbung)
- LayoutManager zur Anordnung der Widgets relativ zueinander

Events, Event-Binding

- Ereignisse sind meist Interaktionen (Tastendruck, Mausbewegung, Klick, . . .)
- Binden von Ereignissen an Methoden (Widgets, Controller)
- Hollywood-Principle: “Don’t call us, we call you”
- Verwendung meist mit Model View Controller Pattern



GUI-Programmierung mit Tkinter

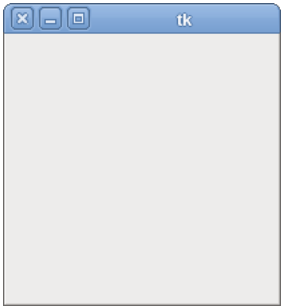
Erstes Tkinter Programm

- Zeile 1: Modul importieren, kurzer Name
- Zeile 2: Hauptfenster (`root`) öffnen, initialisieren
- Zeile 3: Ein Button mit Hauptfenster als Vater, Attribute über Schlüsselwortparameter
- Zeile 4: Einfachster Window-Manager `pack`, von oben nach unten
- Zeile 5: Eine Funktion, wird als Callback benutzt
- Zeile 6: Callback registrieren, beim Klicken passiert was an `command` hängt
- Zeile 7-9: Ausgabe wenn man drei Mal klickt

Philosophie

- Einfaches bleibt einfach, nur was man braucht
- Verschiedene Programmierstile (Objektorientierung optional)
- Nutzung der Python-Möglichkeiten

```
1 >>> import tkinter as tk
2 >>> root = tk.Tk()
3 >>> btn = tk.Button(root, text="Drück mich")
4 >>> btn.pack()
5 >>> def cmd(): print("Aua")
6 >>> btn["command"] = cmd
7 Aua
8 Aua
9 Aua
```



Tkinter Beispiel – Grüße

Anforderungen: Gruß-GUI

- Grüße in verschiedenen Sprachen
- Die Grüße immer in einer anderen Sprache
- Bewusstes Auslösen des Sprachwechsels für den Gruß (Button)
- Bewusstes Auslösen des Beendens (Button)

Besonderheiten

- Der Beenden-Button soll einen roten Text haben
- Die Grüße sollen an einer Stelle verwaltet werden und leicht zu ändern sein

Tkinter Grüße 1/2

Verwaltung der Grüße

- Zeile 1-2: Globale Liste von erlaubten Grüßen
- Zeile 3: Globaler Index (aktueller Zustand) des anzuzeigenden Grußes

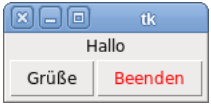
Callback-Funktion, Zeile 5 ff

- Zeile 6: Der Zustand (welcher Gruß) kann geändert werden, also eine globale Variable
- Zeile 7: Den Index immer um eins im Kreis erhöhen
- Zeile 8: Den Inhalt des Labels mit dem aktuellen Gruß aktualisieren (wir brauchen noch ein label, kommt gleich)

```
1 GREETINGS = ["Hallo", "Hello", "Aloita",
2               "Oi", "Ni Hao"]
3 IDX = 0
4
5 def callback():
6     global IDX
7     IDX = (IDX + 1) % len(GREETINGS)
8     label["text"] = GREETINGS[IDX]
```

Anmerkung

- Shebang, encoding und Import von Tkinter nicht gezeigt
- Wir nehmen an, dass das in Zukunft immer vorhanden ist



Tkinter Grüße, 2/2

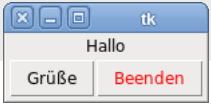
Hauptfunktion, Zeile 1 ff

- Zeile 2: Das Label wird gesetzt, also global
- Zeile 3: Erstelle Label mit initialem Gruß als Text
- Zeile 5: Ein separates Frame um die Buttons
- Zeile 4,6: Windowmanager pack, von oben nach unten
- Zeile 7: Der Button für die Grüße, Callback setzen
- Zeile 10: Beenden-Button, vorgefertige Tkinter-Funktion `destroy` als Callback
- Zeile 9,13: Buttons von links nach rechts in Frame setzen, Konstante `tk.LEFT == 'left'`

Hauptprogramm, Zeile 15 ff

- Zeile 16: Wurzelfenster und initialisieren
- Zeile 17: Hauptfunktion rufen
- Zeile 18: Kontrolle an das Framework (nicht beenden), Eventloop

```
1  def main(root):
2      global label
3      label = tk.Label(root, text=GREETINGS[IDX]
4      label.pack()
5      frame = tk.Frame(root)
6      frame.pack()
7      gruss = tk.Button(frame, text="Grüße",
8                        command=callback)
9      gruss.pack(side=tk.LEFT)
10     beende = tk.Button(frame, text="Beenden",
11                         command=root.destroy)
12     beende["fg"] = "red"
13     beende.pack(side=tk.LEFT)
14
15 if __name__ == '__main__':
16     root = tk.Tk()
17     main(root)
18     root.mainloop()
```



Tkinter Grüße, Objektorientiert 1/3

Anwendung, Klasse `Greetings`

- Anwendungslogik über separate Klasse
- Keine Abhängigkeit von Tkinter
- Separat testbar, wiederverwendbar, ...

GUI, Klasse `Main`

- Instanziiert `Greetings`
- Abhängig von Tkinter

Hauptprogramm, immer gleich

- Tkinter initialisieren und `root` holen
- GUI instanziieren mit root als Master (Toplevel Fenster)
- Event-Loop, Schleife bis Programmende

```
1  class Greetings(object):
2
3      GREETINGS = ["Hallo", "Hello", "Aloita",
4                   "Oi", "Ni Hao"]
5
6      def __init__(self):
7          self.idx = 0
8
9      def next(self):
10         self.idx = (self.idx + 1) % len(self.GREETINGS)
11
12     def _get_greeting(self):
13         return self.GREETINGS[self.idx]
14
15     greet = property(_get_greeting, None, None, "greet")
```

```
1  if __name__ == '__main__':
2      root = tk.Tk()
3      main = Main(root)
4      root.mainloop()
```

Tkinter Grüße, Objektorientiert 2/3

GUI, Klasse `Main`

- Zeile 1: Von Widget erben, für Main meist `Frame`
- Zeile 3: Konvention, Vater heißt `master`, Default None (neues Toplevel-Fenster)
- Zeile 4: Nicht vergessen Oberklasse zu initialisieren, alternativ `super().__init__(master)`
- Zeile 5: Instanziiere Anwendungslogik
- Zeile 6: Erstelle und verbinde Widgets
- Zeile 7: Fülle `root` mit `self` vollständig aus
- Zeile 8-10: callback, Zugriff auf `self`, weiterrücken und auslesen

```
1  class Main(tk.Frame):
2
3      def __init__(self, master=None):
4          tk.Frame.__init__(self, master)
5          self.greetings = Greetings()
6          self.create_widgets()
7          self.pack(expand=True, fill=tk.BOTH)
8
9      def callback(self):
10         self.greetings.next()
11         self.label["text"] = self.greetings.greet
```

Tkinter Grüße, Objektorientiert 3/3

`create_widgets`

- Methode von `Main`
- Wie vorher, Zugriff über `self`
- Merken des Labels, wird in `callback` gebraucht
- Rest nicht merken, nur in `self`, also einem `tk.Frame`, einhängen
- `btn_frame` ist verschachteltes Frame

Zeile 5-16: Widgets wie vorher

```
1  def create_widgets(self):
2      self.label = tk.Label(self,
3                            text=self.greetings.greet)
4      self.label.pack()
5      btn_frame = tk.Frame(self)
6      btn_frame.pack()
7      gruss = tk.Button(btn_frame, text="Grüße",
8             command=self.callback)
9      gruss.pack(side=tk.LEFT)
10     beende = tk.Button(btn_frame, text="Beenden",
11              command=self._root().destroy)
12     beende["fg"] = "red"
13     beende.pack(side=tk.LEFT)
```

Layoutmanager

Layoutmanager

- Ordnet alle Kinder eines Widgets (meist `tk.Frame`) auf der zur Verfügung stehenden Fläche an
- Für die Kind-Widgets eines Vater-Widgets immer nur einen Layoutmanager nehmen, sonst Aua
- Verschiedene Optionen für Padding, Stretchen, ...

`pack`

- Relative Anordnung
- Bei wenigen Widgets einfach, sonst schnell komplex

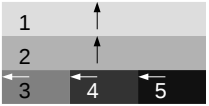
`grid`

- Anordnung in einem Grid, einer Tabelle
- Zeilen (row) und Spalten (column)
- Meistens verwenden, insbesondere bei vielen Widgets

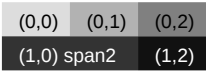
`place`

- Absolute Positionierung
- So gut wie nie verwenden

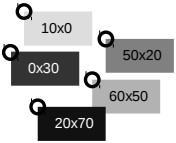
`pack`



`grid`



`place`



Layoutmanager `pack`

`pack`

- Relative Anordnung, default von oben nach unten
- `fill`: nach X oder Y Richtung (oder BOTH) die Größe des Widgets ausdehnen
  - `tk.X=='x', tk.Y=='y', tk.BOTH=='both'`
- `padx, pady`: Äußerer x-, y-Rahmen in Pixel
- `ipadx, ipady`: Innerer x-, y-Rahmen in Pixel, im Beispiel durch Hintergrund sichtbar
- `side`: Ansatzpunkt Anordung, default `TOP`, verfügbar `BOTTOM`, `LEFT`, `RIGHT`
  - Wenn zu viel gemischt, sehr sehr unübersichtlich

Einsatzgebiet

- Einfach Anordnung von Widgets in eine, max zwei Richtungen, Beispiel ButtonBar
- Anordnung von Haupt-Fenstern in GUIs
- Lieber verschachteln statt spekulieren, wenn zu komplex dann `grid`

```
labels = [tk.Label(root, bg="lightgreen",
    text="Label%d"%i) for i in range(8)]
labels[0].pack()
labels[1].pack(fill=tk.X)
labels[2].pack(fill=tk.X, padx=3, pady=3)
labels[3].pack(fill=tk.X, ipady=10)
labels[4].pack(side=tk.LEFT)
labels[5].pack(side=tk.LEFT, fill=tk.Y)
labels[6].pack()
labels[7].pack()
```

`expand=True`

- Diesen Widgets Restplatz gleichmäßig zuteilen

Layoutmanager `grid`

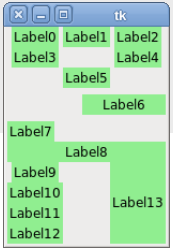
`grid`

- Anordnung in Tabelle
- `row`: Zeile
- `column`: Spalte
- `...span`: Mehr als eine
- `sticky`: Ausrichtung nach Himmelsrichtungen (NESW)
  - default ist zentriert
  - auch gegenüberliegend (N+S, W+E) füllt aus (stretch)
  - alle Himmelsrichtungen okay
- `i?pad[xy]`: Wie `pack`

Einsatzgebiet

- Für fast alles gut geeignet
- Alle komplexeren Layouts

```
labels = [tk.Label(root, bg="lightgreen",
    text="Label%d"%i) for i in range(14)]
for i in range(3):
    labels[i].grid(row=0, column=i)
labels[3].grid(row=1, column=0)
labels[4].grid(row=1, column=2)
labels[5].grid(row=2, column=0, columnspan=3)
labels[6].grid(row=3, column=0, columnspan=3,
                sticky=tk.E, pady=6, ipadx=17)
labels[7].grid(row=4, column=0, columnspan=3, sticky=tk.W)
labels[8].grid(row=5, column=0, columnspan=3,
                sticky=tk.W+tk.E)
for i in range(4):
    labels[9+i].grid(row=6+i, column=0)
labels[13].grid(row=6, rowspan=4, column=2,
                 sticky=tk.N+tk.S)
```



`grid_forget/grid_remove`

- Versteckt Widget, vergisst/behält Optionen
- Kann wieder mit `grid` angezeigt werden

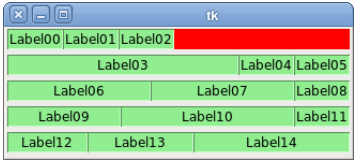
Layoutmanager `grid` – Gewichtungen

Ziel – Aufteilung des überschüssigen Platzes

- Im umschließenden Widget (Vater) `rowconfigure`, `columnconfigure` mit Attribut `weight`, default 0
- Verteilung des Platzes je nach Anteil

Beispiel: 0: keine Aufteilung

- 1: Alles dem 0.
- 2: Gleichverteilt 0.+1.
- 3: 0. ein Drittel, 1. Rest
- 4: 0. 1/7, 1. 2/7., 2. 4/7




```
root.minsize(320, 120)
frames = [tk.Frame(root, bg="red")
                for i in range(5)]
for i in range(5):
    frames[i].pack(fill=tk.X, expand=True)
labels = [tk.Label(frames[i//3],
    bg="lightgreen", relief=tk.SUNKEN,
    text="Label%02d"%i) for i in range(15)]
for i in range(5):
    for j in range(3):
        labels[i*3+j].grid(row=i, column=j,
                           sticky=tk.W+tk.E)
frames[1].columnconfigure(0, weight=1)
frames[2].columnconfigure(0, weight=1)
frames[2].columnconfigure(1, weight=1)
frames[3].columnconfigure(0, weight=1)
frames[3].columnconfigure(1, weight=2)
frames[4].columnconfigure(0, weight=1)
frames[4].columnconfigure(1, weight=2)
frames[4].columnconfigure(2, weight=4)
```

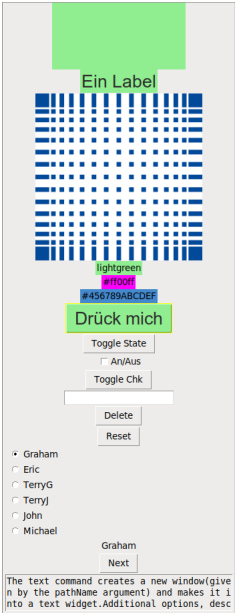
Standard Widgets, Übersicht

Standard Widgets

- Frame, Grundfläche
- Label, Text und Bilder
- (Farben und Zeichensätze)
- Button, um Aktionen auszulösen
- Checkbutton, toggle mit Zustand, Mehrere aus Mehreren
- Entry, Eingabefeld
- Radiobutton, Eins aus Mehreren
- (IntVar, Tkinter-Variablen, Observer)
- Text, Mini-Editor

Dokumentation

- Informationen (Widgets, Optionen) sind unvollständig
- Sinnvolle und obskure Details in der Dokumentation nachlesen



Frame

`Frame`, ein Container

- Alle Widgets sind in Containern, meist `Frame`
- `root` hat auch ein `Frame`
- Jedes TopLevel-Fenster hat ein `Frame`

Nutzen in Anwendung

- `class Anwendung(tk.Frame)`
- Verwenden als Superklasse
- `self` ist Vater aller direkten Widgets
- Meist auch Basisklasse selbst-definierter Widgets

```
frame = tk.Frame(root, width=200, height=100)
frame["bg"] = "lightgreen"
frame.pack()
```

Label

`Label`

- Ein `Label`, ein `Frame` mit Text

Konfiguration

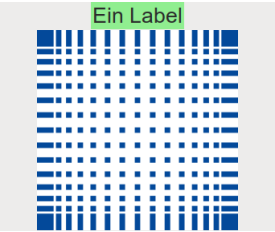
- Initialisierungsparameter
- Dictionary-Zuweisung mit Namen
- Konfiguration für alle Widgets auf diese beiden Arten

Konfiguration Aussehen und Inhalt `Label`

- `text`, `image`: Inhalt, Text oder Bild
- `bg`: Hintergrundfarbe
- `fg`: Vordergrundfarbe
- `font`: Zeichensatz; Namen, Größe

Besonderheiten

- Zeile 9: Halte Referenz auf Bild für die Lebenszeit des Widgets, ansonsten räumt Garbage Collector Bild gleich wieder weg




```
1  label = tk.Label(root, text="Ein Label")
2  label["bg"] = "lightgreen"
3  label["fg"] = "gray17"
4  label["font"] = "Arial 20"
5  label.pack()
6  img = Image.open("hsmalogo.png")
7  photo = ImageTk.PhotoImage(img)
8  imglabel = tk.Label(root, image=photo)
9  imglabel.photo = photo # keep reference
10 imglabel.pack()
```

Alternative `Message`, mehrzeilig

Farben

lightgreen
#ff00ff
#456789ABCDEF

```
color = "lightgreen"
tk.Label(root, text=color, bg=color).pack()
color = "#ff00ff"
tk.Label(root, text=color, bg=color).pack()
color = "#456789ABCDEF"
tk.Label(root, text=color, bg=color).pack()
```

Farbwahl

- Angabe HEX-String: \#RGB, \#RRGGBB, \#RRRGGGBBB, \#RRRRGGGGBBBB
- Ein paar vordefinierte Namen

Farben in Widgets

- `background, foreground`: Standard, Abk. `bg, fg`
- `activebackground, activeforeground`: wenn aktiv
- `disabledforeground`: wenn disabled
- `highlightbackground, highlightcolor`: bei Fokus
- `selectbackground, selectforeground`: Wenn ausgewählt (nur wenn Selektion unterstützt)

Alle Farben mit Namen anzeigen

```
CN, columns = COLOR_NAMES, 12
rows = (len(CN)//columns)+1
for row in range(rows):
    for col in range(columns):
        idx = row*columns+col
        if idx < len(CN):
            color = CN[row*columns+col]
            lbl = tk.Label(root,
                    text=color,bg=color)
            if is_dark(root, color):
                lbl["fg"] = "white"
            lbl.grid(row=row,column=col,
                    sticky=tk.W+tk.E)
```

Farbnamen

snow	ghost white	white smoke	gainsboro	floral white	old lace	linen	antique white	papaya whip	blanched almond	bisque	peach puff
navajo white	lemon chiffon	mint cream	azure	alice blue	lavender	lavender blush	misty rose	dark slate gray	dim gray	slate gray	light slate gray
gray	light grey	midnight blue	navy	cornflower blue	dark slate blue	slate blue	medium slate blue	light slate blue	medium blue	royal blue	blue
dodger blue	deep sky blue	sky blue	light sky blue	steel blue	light steel blue	light blue	powder blue	pale turquoise	dark turquoise	medium turquoise	turquoise
cyan	light cyan	cadet blue	medium aquamarine	aquamarine	dark green	dark olive green	dark sea green	sea green	medium sea green	light sea green	pale green
spring green	lawn green	medium spring green	green yellow	lime green	yellow green	forest green	olive drab	dark khaki	khaki	pale goldenrod	light goldenrod yellow
light yellow	yellow	gold	light goldenrod	goldenrod	dark goldenrod	rosy brown	indian red	saddle brown	sandy brown	dark salmon	salmon
light salmon	orange	dark orange	coral	light coral	tomato	orange red	red	hot pink	deep pink	pink	light pink
pale violet red	maroon	medium violet red	violet red	medium orchid	dark orchid	dark violet	blue violet	purple	medium purple	thistle	snow2
snow3	snow4	seashell2	seashell3	seashell4	AntiqueWhite1	AntiqueWhite2	AntiqueWhite3	AntiqueWhite4	bisque2	bisque3	bisque4
PeachPuff2	PeachPuff3	PeachPuff4	NavajoWhite2	NavajoWhite3	NavajoWhite4	LemonChiffon2	LemonChiffon3	LemonChiffon4	cornsilk2	cornsilk3	cornsilk4
ivory2	ivory3	ivory4	honeydew2	honeydew3	honeydew4	LavenderBlush2	LavenderBlush3	LavenderBlush4	MistyRose2	MistyRose3	MistyRose4
azure2	azure3	azure4	SlateBlue1	SlateBlue2	SlateBlue3	SlateBlue4	RoyalBlue1	RoyalBlue2	RoyalBlue3	RoyalBlue4	blue2
blue4	DodgerBlue2	DodgerBlue3	DodgerBlue4	SteelBlue1	SteelBlue2	SteelBlue3	SteelBlue4	DeepSkyBlue2	DeepSkyBlue3	DeepSkyBlue4	SkyBlue1
SkyBlue2	SkyBlue3	SkyBlue4	LightSkyBlue1	LightSkyBlue2	LightSkyBlue3	LightSkyBlue4	SlateGray1	SlateGray2	SlateGray3	SlateGray4	LightSteelBlue1
LightSteelBlue2	LightSteelBlue3	LightSteelBlue4	LightBlue1	LightBlue2	LightBlue3	LightBlue4	LightCyan2	LightCyan3	LightCyan4	PaleTurquoise1	PaleTurquoise2
PaleTurquoise3	PaleTurquoise4	CadetBlue1	CadetBlue2	CadetBlue3	CadetBlue4	turquoise1	turquoise2	turquoise3	turquoise4	cyan2	cyan3
cyan4	DarkSlateGray1	DarkSlateGray2	DarkSlateGray3	DarkSlateGray4	aquamarine2	aquamarine4	DarkSeaGreen1	DarkSeaGreen2	DarkSeaGreen3	DarkSeaGreen4	SeaGreen1
SeaGreen2	SeaGreen3	PaleGreen1	PaleGreen2	PaleGreen3	PaleGreen4	SpringGreen2	SpringGreen3	SpringGreen4	green2	green3	green4
chartreuse2	chartreuse3	chartreuse4	OliveDrab1	OliveDrab2	OliveDrab4	DarkOliveGreen1	DarkOliveGreen2	DarkOliveGreen3	DarkOliveGreen4	khaki1	khaki2
khaki3	khaki4	LightGoldenrod1	LightGoldenrod2	LightGoldenrod3	LightGoldenrod4	LightYellow2	LightYellow3	LightYellow4	yellow2	yellow3	yellow4
gold2	gold3	gold4	goldenrod1	goldenrod2	goldenrod3	goldenrod4	DarkGoldenrod1	DarkGoldenrod2	DarkGoldenrod3	DarkGoldenrod4	RosyBrown1
RosyBrown2	RosyBrown3	RosyBrown4	IndianRed1	IndianRed2	IndianRed3	IndianRed4	sienna1	sienna2	sienna3	sienna4	burlywood1
burlywood2	burlywood3	burlywood4	wheat1	wheat2	wheat3	wheat4	tan1	tan2	tan4	chocolate1	chocolate2
chocolate3	firebrick1	firebrick2	firebrick3	firebrick4	brown1	brown2	brown3	brown4	salmon1	salmon2	salmon3
salmon4	LightSalmon2	LightSalmon3	LightSalmon4	orange2	orange3	orange4	DarkOrange1	DarkOrange2	DarkOrange3	DarkOrange4	coral1
coral2	coral3	coral4	tomato2	tomato3	tomato4	OrangeRed2	OrangeRed3	OrangeRed4	red2	red3	red4
DeepPink2	DeepPink3	DeepPink4	HotPink1	HotPink2	HotPink3	HotPink4	pink1	pink2	pink3	pink4	LightPink1
LightPink2	LightPink3	LightPink4	PaleVioletRed1	PaleVioletRed2	PaleVioletRed3	PaleVioletRed4	maroon1	maroon2	maroon3	maroon4	VioletRed1
VioletRed2	VioletRed3	VioletRed4	magenta2	magenta3	magenta4	orchid1	orchid2	orchid3	orchid4	plum1	plum2
plum3	plum4	MediumOrchid1	MediumOrchid2	MediumOrchid3	MediumOrchid4	DarkOrchid1	DarkOrchid2	DarkOrchid3	DarkOrchid4	purple1	purple2
purple3	purple4	MediumPurple1	MediumPurple2	MediumPurple3	MediumPurple4	thistle1	thistle2	thistle3	thistle4	gray1	gray2
gray3	gray4	gray5	gray6	gray7	gray8	gray9	gray10	gray11	gray12	gray13	gray14
gray15	gray16	gray17	gray18	gray19	gray20	gray21	gray22	gray23	gray24	gray25	gray26
gray27	gray28	gray29	gray30	gray31	gray32	gray33	gray34	gray35	gray36	gray37	gray38
gray39	gray40	gray42	gray43	gray44	gray45	gray46	gray47	gray48	gray49	gray50	gray51
gray52	gray53	gray54	gray55	gray56	gray57	gray58	gray59	gray60	gray61	gray62	gray63
gray64	gray65	gray66	gray67	gray68	gray69	gray70	gray71	gray72	gray73	gray74	gray75
gray76	gray77	gray78	gray79	gray80	gray81	gray82	gray83	gray84	gray85	gray86	gray87
gray88	gray89	gray90	gray91	gray92	gray93	gray94	gray95	gray97	gray98	gray99	

Zeichensätze, Fonts

```
import tkinter.font
```

- `tk.font`, Klasse `Font`
- Verfügbare Familien je nach Plattform

Zeichensatzwahl

- im Widget: Familie Größe `font="Arial 12"`
- sonst `Font`-Instanz mit Schlüsselwörter `family`, `size`, `weight`, `slant`, `underline`, `overstrike` bei Initialisierung

Besonderheiten

- Zeile 1: `row`, einelementige Liste, veränderbar
- Zeile 5: Aufpassen, dass Tcl/Tk den Font nicht recycelt

```
1  both, row, rowoff = tk.W+tk.E, [0], 6
2  def do_label(text, **opts):
3      font = tk.font.Font(**opts)
4      lbl = tk.Label(root, text=text+str(opts), font=font)
5      lbl.font = font # prevent reuse
6      lbl.grid(row=row[0], column=0, columnspan=len(SIZE))
7      row[0] += 1
8  do_label("default=normal ", weight='bold')
9  do_label("default=roman ", slant='italic')
10 do_label("", weight='bold', slant='italic')
11 do_label("", underline=True)
12 do_label("", overstrike=True)
13 for row in range(len(FAMILY)):
14     family = FAMILY[row]
15     for col in range(len(SIZE)):
16         size = SIZE[col]
17         font = tk.font.Font(family=family, size=size)
18         fontstr = "%s %s" % (family, size)
19         lbl = tk.Label(root, text=fontstr, font=font)
20         lbl.grid(row=row+rowoff, column=col, sticky=both)
```

Fonts, Beispiel

default=normal {'weight': 'bold'}
default=roman {'slant': 'italic'}
{'weight': 'bold', 'slant': 'italic'}
{'underline': True}
~~{'overstrike': True}~~

Arial 10	Arial 11	Arial 12	Arial 14	Arial 16	Arial 18
Helvetica 10	Helvetica 11	Helvetica 12	Helvetica 14	Helvetica 16	Helvetica 18
Verdana 10	Verdana 11	Verdana 12	Verdana 14	Verdana 16	Verdana 18
Times 10	Times 11	Times 12	Times 14	Times 16	Times 18
Times New Roman 10	Times New Roman 11	Times New Roman 12	Times New Roman 14	Times New Roman 16	Times New Roman 18
Georgia 10	Georgia 11	Georgia 12	Georgia 14	Georgia 16	Georgia 18
Courier 10	Courier 11	Courier 12	Courier 14	Courier 16	Courier 18
Courier New 10	Courier New 11	Courier New 12	Courier New 14	Courier New 16	Courier New 18
Andale Mono 10	Andale Mono 11	Andale Mono 12	Andale Mono 14	Andale Mono 16	Andale Mono 18
Bookman 10	Bookman 11	Bookman 12	Bookman 14	Bookman 16	Bookman 18
Avantgarde 10	Avantgarde 11	Avantgarde 12	Avantgarde 14	Avantgarde 16	Avantgarde 18
Palatino 10	Palatino 11	Palatino 12	Palatino 14	Palatino 16	Palatino 18
new century schoolbook 10	new century schoolbook 11	new century schoolbook 12	new century schoolbook 14	new century schoolbook 16	new century schoolbook 18
itc zapf chancery 10	*itc zapf chancery 11*	*itc zapf chancery 12*	*itc zapf chancery 14*	*itc zapf chancery 16*	*itc zapf chancery 18*
bitstream charter 10	bitstream charter 11	bitstream charter 12	bitstream charter 14	bitstream charter 16	bitstream charter 18
urw gothic l 10	urw gothic l 11	urw gothic l 12	urw gothic l 14	urw gothic l 16	urw gothic l 18

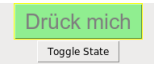
Button

`Button`: Ein Druckknopf, Konfiguration

- `command`: Kommando bei Klick, eine Funktion
- `state`: `normal`, `disabled`, `active`
- Je nach `state` andere Darstellung

Normal

Disabled



Normal, Aktiv

```
button = tk.Button(root, text="Drück mich")
button["font"] = "Arial 20"
button["fg"] = "gray17"
button["bg"] = "lightgreen"
button["disabledforeground"] = "gray50"
button["highlightbackground"] = "yellow"
button["highlightcolor"] = "purple"
button["activeforeground"] = "magenta"
def callback():
    print("klicked")
button["command"] = callback
button.pack(fill="none")
def togglestate():
    newstate = "normal"
    if button["state"] == "normal":
        newstate = "disabled"
    button["state"] = newstate
toggle_btn = tk.Button(root, text="Toggle State")
toggle_btn["command"] = togglestate
toggle_btn.pack()
```

Checkbutton

`Checkbutton`

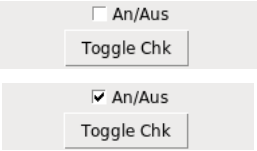
- Knopf mit Label und sichtbarem Zustand
- An oder aus
- Extra Zustandsvariable von Tkinter

Zustandsvariable

- `tk.BooleanVar` (False, True), `tk.IntVar` (0, -1, 1, -2, 2, ...), ...
- `get`: Holen des Werts
- `set`: Setzen des Werts

Beispiel

- Ändern einer Booleschen Variable
- Ändern auch über extra Funktion von außen möglich, Zustand wird trotzdem reflektiert




```
bvar = tk.BooleanVar()

def callback():
    print(bvar.get())
chkbtn = tk.Checkbutton(root, text="An/Aus",
    variable=bvar, command=callback)
chkbtn.pack()

def toggle():
    bvar.set(1-bvar.get())
btn = tk.Button(root, text="Toggle Chk",
    command=toggle)
btn.pack()
```

Entry

`Entry` Eingabefeld

- Einzeilige Eingabe
- Interaktiv
- Methoden
  - `get`: Holt Text
  - `delete`: Löschen von/bis Position
  - `insert`: An Position

Position

- `idx`: 0 ist Start, ansonsten die wievielte Stelle
- `tk.END`: spezielle Position Ende

```
entry = tk.Entry(root, bg="white")
entry.pack()

def e1cmd():
    print(entry.get())
    entry.delete(0, tk.END)
tk.Button(root, text="Delete", command=e1cmd).pack()

def e2cmd():
    entry.delete(0, tk.END)
    entry.insert(0, "Vorgabe")
tk.Button(root, text="Reset", command=e2cmd).pack()
```

Radiobutton

`Radiobutton`: Eins aus Mehreren

- Anzeigetext
- Variable für Auswahl, Wert wenn gewählt

Besonderheiten

- Zeile 10: Zeige aktuell gewählten Namen in einem Label
- Zeile 14: Folge jeder Änderung (w=write), aktualisiere Label
- Zeile 16: Aktualisieren von außen, Anzeige automatisch aktualisiert

```
1  names = ["Graham", "Eric", "TerryG", "TerryJ",
2           "John", "Michael"]
3  radios, var = [], tk.IntVar()
4  for idx, name in enumerate(names):
5      radio = tk.Radiobutton(root, text=name,
6          variable=var, value=idx)
7      radios.append(radio)
8      radio.pack(anchor=tk.W)
9  var.set(0)
10 lbl = tk.Label(root, text=names[var.get()])
11 lbl.pack()
12 def update(*args):
13     lbl["text"] = names[var.get()]
14 var.trace("w", update) # follow changes
15 def next():
16     var.set((var.get()+1)%len(names))
17 tk.Button(root, text="Next", command=next).pack()
```

Tkinter-Variablen

Tkinter-Variablen (Tk-Variablen)

- Aufgabe: Kommunikation von Werten zwischen Skriptsprache (Python) und GUI
- Standardvariablen können dazu nicht benutzt werden (kein Observer)
- Tkinter-Variablen sind Instanzen von Klassen
- Verfügbar sind: `IntVar`, `BooleanVar`, `StringVar`, `DoubleVar`

Methoden

- `get()`: Holt Wert, aktueller Wert wie in von Variable abhängigen Widgets angezeigt
- `set(value)`: Setzt Wert, abhängige Widgets aktualisieren sich
- `trace*`: Observer
  Callbacks erlauben

Callbacks, Observer für `var`

- `var.trace(mode, callback)`: Beginnt ein Trace
  - mode: `w` write, `r` read, `u` undefine
  - Gibt Trace-Namen zurück
- `var.trace_vdelete(name)`: Löscht Trace mit diesem Namen
- `var.trace_vinfo()`: Gibt Liste von Tupel mit `mode` und Trace-Namen zurück, alle aktiven Traces

```
var = tk.IntVar()
def tracer(*args):
    print("trace: %d" % var.get())
var.set(17)
tracename = var.trace("w", tracer)
var.set(42) # trace: 42
print(var.trace_vinfo())
# [('w', '140256148065488tracer')]
var.trace_vdelete("w", tracename)
var.set(69)
```

Text

`Text`, kleiner Text-Editor, all inclusive

- Sehr mächtig, hier nur kleiner Ausschnitt
- Klassisches Editieren möglich
- Cut, Copy, Paste, Undo
- Markieren, Formatieren der Markierungen
- Bilder embedden, ...

Editieren, Grundlagen

- `insert(tk.END, text)`: Am Ende einfügen
- `get(1.0, tk.END)`: Alles holen, ...

Index

- `'r.c'`: Zeile `r`, 1-basiert; Spalte `c`, 0-basiert
  - Für Spalte auch `'end'`, Zeilenende
- `tk.CURRENT`: Stelle am nächsten zum Mauszeiger
- `tk.INSERT`: Aktuelle Einfügestelle
- `'@x.y'`: Fensterkoordinaten ...

```
The text command creates a new window(give
n by the pathName argument) and makes it i
nto a text widget.Additional options, desc
```

```
text = tk.Text(root, height=3, width=42)
text.pack()
text.insert(tk.END, "The text command "
"creates a new window (given by the "
"pathName argument) and makes it into "
"a text widget. Additional options, "
"described above, may be specified on ...")
```

Besonderheiten

- 3 Zeilen, 42 Zeichen je Zeile
- Im Beispiel zu viel Text, um ihn anzuzeigen
- Text ist noch vorhanden, aber nicht sichtbar

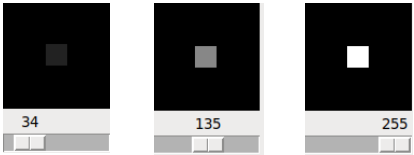
Scale

`Scale`, Schieberegler

- von, bis
  - `from_` mit Unterstrich, da **`from`** ein Schlüsselwort ist
- Orientierung, default vertikal

Beispiel

- Lichtpunkt (ein Frame) Helligkeit
- Lichtpunkt mit `place` in die Mitte von `f1`
  - Keine gemischten Layoutmanager!
  - Aber `f1` ist Wurzel für `f2`, sonst keine Widgets in `f1`
- Auswirkung von Änderungen mit `command`
  - `event` ist Wert des `scale` als String




```
1  f1 = tk.Frame(root, width=100, height=100,
2                bg='black')
3  f1.pack()
4  f2 = tk.Frame(f1, width=20, height=20,
5                bg='white')
6  f2.place(relx=0.5, rely=0.5,
7           anchor=tk.CENTER)
8  sc = tk.Scale(root, from_=0, to=255,
9                orient=tk.HORIZONTAL)
10 sc.pack()
11 def update(event=None):
12     f2["bg"] = "#"+("%02x" % sc.get())*3
13 sc["command"] = update
14 sc.set(255)
```

Listbox

`ListBox`

- Eins oder Mehrere aus Mehreren
- `insert`: Einfügen von Einträgen an Index-Position, `tk.END` am Ende
- `delete`: Einträge in Bereichen löschen

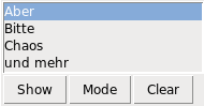
Interaktive Auswahl

- `curselection()`: Lesen der aktuellen Auswahl, Tupel mit ausgewählten Indizes
- `selection_set`: Setzen einer Auswahl

Modus, `'selectmode'`

- `tk.BROWSE`: Eins aus Mehreren
- `tk.EXTENDED`: Mehrere aus Mehreren, Klicken mit gedrückter CTRL-Taste

```
(0,)
(0, 1, 3)
```



```
listbox = tk.Listbox(root, height=4, width=23)
listbox.pack()
entries = ["Aber", "Bitte", "Chaos",
           "und mehr"]
for entry in entries:
    listbox.insert(tk.END, entry)
def cmdsel():
    print(listbox.curselection())
tk.Button(root, text="Show",
          command=cmdsel).pack(side=tk.LEFT)
def cmdmode():
    if listbox["selectmode"] == tk.BROWSE:
        listbox["selectmode"] = tk.EXTENDED
    else:
        listbox["selectmode"] = tk.BROWSE
tk.Button(root,text="Mode",
          command=cmdmode).pack(side=tk.LEFT)
def clear(): listbox.select_clear(0, tk.END)
tk.Button(root,text="Clear",
          command=clear).pack(side=tk.LEFT)
```

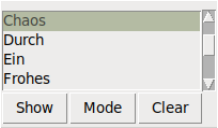
Scrollbar

Scrolling – Sichtbarer Bereich

- Es ist möglich, dass nicht der vollständige Inhalt eines Widgets angezeigt werden kann
- Widgets können den Inhalt meist trotzdem verwalten
- Anzeige eines Ausschnitts
- Manipulation des Ausschnitts durch Scrolling

Scrollbar – Widget um Scrolling zu steuern

- Arbeitet mit Widget zusammen
- Scrollbar → Widget
  - `command` auf `xview` oder `yview`
  - Meist `'moveto', index` mit `index` ∈ [0.0, 1.0]
- Widget → Scrollbar
  - `yscrollcommand` auf `set`
  - Meist `(von, bis)` anzuzeigender Teil mit `von`, `bis` ∈ [0.0, 1.0]




```
listbox = tk.Listbox(lbfrm,
                     height=4, width=23)
listbox.pack(side=tk.LEFT)
entries = ["Aber", "Bitte", "Chaos", "Durch",
           "Ein", "Frohes", "Großes", "Haus",
           "und", "so"]
for entry in entries:
    listbox.insert(tk.END, entry)
scrollbar = tk.Scrollbar(lbfrm)
scrollbar.pack(side=tk.RIGHT, fill=tk.Y)
scrollbar["command"] = listbox.yview
listbox["yscrollcommand"] = scrollbar.set
```

Manuelle Kombination von zwei Widgets für Scrolling

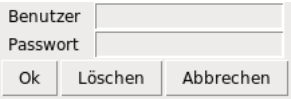
Eigene Komponenten

Eigene Klasse

- Erbt meist von `Frame`, darin Anordnung und eigenes Verhalten
- Übergabeparameter oder Attribute zur Konfiguration

Beispiel – Login-Komponente

- Benutzername/Password
- Verhalten: Abbrechen bricht ab, Löschen leert die Eingabefelder, Ok versucht zu authentifizieren mit einem Callback




```
1  class Login(tk.Frame):
2      def __init__(self, master=None, callback=None):
3          tk.Frame.__init__(self, master)
4          tk.Label(self, text="Benutzer").grid(row=0, column=0)
5          tk.Label(self, text="Passwort").grid(row=1, column=0)
6          self.userentry = tk.Entry(self, width=17)
7          self.passwordentry=tk.Entry(self, width=17, show='*')
8          self.userentry.grid(row=0, column=1)
9          self.passwordentry.grid(row=1, column=1)
10         bf = tk.Frame(self);bf.grid(row=2, column=0, columnspan=2)
11         tk.Button(bf,text="Ok",command=self.ok).pack(side=tk.LEFT)
12         tk.Button(bf,text="Löschen",command=self.clear).pack(side=tk
13         tk.Button(bf,text="Abbrechen",command=self.cancel).pack(side
14         self.callback = callback
15     def ok(self):
16         self.callback(True, self.userentry.get(),
17                       self.passwordentry.get())
18     def clear(self):
19         self.userentry.delete(0, tk.END)
20         self.passwordentry.delete(0, tk.END)
21     def cancel(self):
22         self.callback(False, None, None)
```

Benutzen der Login-Komponente

Benutzen

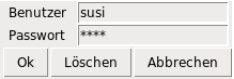
- In Toplevel-Fenster rendern
- Callback
  - Bei Abbrechen Programm beenden
  - Bei Ok versuchen zu authentifizieren
  - Falls beides gleich, dann ok und beenden, sonst nochmal probieren
- Callback-Konfiguration über Attribut

Ablauf:

`susi/rudi` Ok
`> failed, try again`
`susi/susi` Ok
`> accepted`

`x/y` Abbrechen
`>`

```
def callback(state, user, password):
    if not state:
        root.destroy()
    elif state and user == password:
        print("accepted")
        root.destroy()
        return
    else:
        print("failed, try again")
login = Login(root, callback)
login.pack()
```



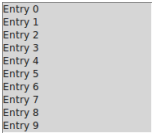
Toplevel-Fenster und Popup

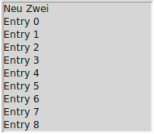
`Toplevel`

- Ein neues Toplevel-Fenster, vom Window-Manager (WM) des Desktops verwaltet
- `overrideredirect(True)`: WM-Dekorationen aus
- `geometry`: An Position zwingen
- `destroy`: Schließen

Beispiel: Popup-Fenster

- In aktuellem Fenster, rechter Mausklick
- Nach Callbacks des Buttons Fenster schließen






```
1  listbox = tk.Listbox(root, height=10, width=23)
2  listbox.pack()
3  for idx in range(10):
4      listbox.insert(tk.END, "Entry %d" % idx)
5  def callback(s):
6      listbox.insert(0, "Neu %s" % s)
7      top.destroy()
8  def popup(event):
9      global top
10     top = tk.Toplevel()
11     top.overrideredirect(True)
12     top.geometry("+%d+%d"%(event.x_root,event.y_root
13     for s in ("Eins", "Zwei", "Drei"):
14         cmd = lambda s=s: callback(s)
15         tk.Button(top, text=s, command=cmd).pack()
16 listbox.bind("<ButtonRelease-3>", popup)
```

Besonderheiten:

- Zeile 14: `s=s`, vermeide Neubindung für alle
- Zeile 16: Binden an Event

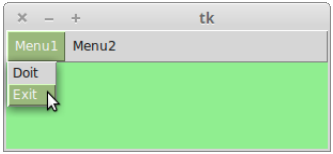
Menus

Menu

- Eigenschaft eines Toplevel-Fensters
- Kaskadierend, beliebig viele Untermenus möglich
- `tearoff`: unabhängig von Fenster, macht man nicht mehr

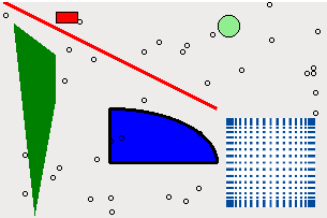
Weitere Menuarten

- `add_separator`: Strich
- `add_radiobutton`: Auswahl, Variable
- `add_checkbutton`: An/Aus-Option, Variable




```
1  tk.Frame(root, width=300, height=80, bg="lightgreen").pack()
2  menubar = tk.Menu(root)
3  root.config(menu=menubar)
4  def doit(): print("doit")
5  menu1 = tk.Menu(menubar, tearoff=False)
6  menu1.add_command(label="Doit", command=doit)
7  menu1.add_command(label="Exit", command=root.destroy)
8  menubar.add_cascade(label="Menu1", menu=menu1)
9  menu2 = tk.Menu(menubar, tearoff=False)
10 menu2.add_command(label="Help")
11 menu2.add_command(label="About")
12 menubar.add_cascade(label="Menu2", menu=menu2)
```

Canvas

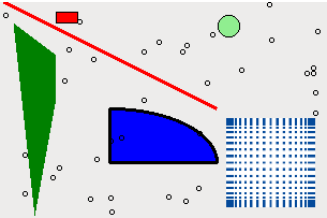


Sehr mächtiger Canvas

- Linien: Startpunkt, Endpunkt
- Rechtecke
- Kreisbogen (Bounding Box)
- Kreise
- Polygonzüge
- Bilder
- ...

```
cv = tk.Canvas(root, width=300, height=200)
cv.pack()
cv.create_line(0, 0, 200, 100,
               fill="red", width=3)
cv.create_rectangle(50, 10, 70, 20, fill="red")
cv.create_arc((0, 100, 200, 200),
              start=0, extent=90,
              fill="blue", width=3)
for i in range(42):
    x, y = random.randrange(300), random.randrange(200)
    cv.create_oval((x-2, y-2, x+2, y+2))
points = [(10, 20), (50, 50), (50, 90), (30, 200)]
cv.create_polygon(points, fill="green")
img = Image.open("hsmalogo.png")
img = img.resize(tuple(map(lambda x: x//3, img.size)))
cv.image = ImageTk.PhotoImage(img) # safe ref
cv.create_image((250, 150), image=cv.image)
```

Canvas, Animation



Animation

- Namen für Objekte, Änderungen nachträglich
- `Widget.move`: ändert Start
- `Canvas.update`: aktuellen Zustand auf Screen bringen

Beispiel

- Hellgrüner Kreis
- Bewegt sich zufällig innerhalb des grauen Canvas

```
x, y = 200, 20
dot = cv.create_oval((x, y, x+20, y+20), fill="lightgreen")
def tick():
    "on animation step, a random move"
    dx = random.randrange(-3, 4)
    dy = random.randrange(-3, 4)
    cv.move(dot, dx, dy)
    x, y, _, _ = cv.coords(dot)
    cv.move(dot, 3 if x<0 else -3 if x>300 else 0, 0)
    cv.move(dot, 0, 3 if y<0 else -3 if y>200 else 0)
    cv.update() # redraw
    cv._root().after(50, tick) # 50ms -> 20 FPS
tick() # kicks off endless animation
```

Endlos-Animation

- Mit `after(ms, fun)` von `root`-Objekt führt man nach (mindestens) `ms` Millisekunden die Funktion `fun` aus
- `tick` als Funktion, die einen Fort-Schritt (der Animation) berechnet
- Am Ende von `tick` wieder ein weiteres `tick` nach 50 `ms` registrieren

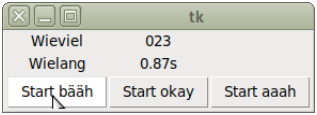
Threading

Threading

- Langläufer **nie** im Main-Thread
- Extra-Thread für IO, Netzwerk, ...
- Ziel: GUI bleibt responsive

Beispiel

- GUI, um Zufallszahlen vom Web zu lesen
- Solange Zufallszahlen zwischen 1 und 100 bis die 17 kommt
- Drei Implementierungen ...

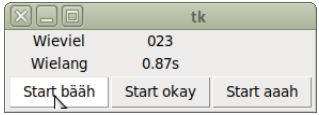



```
def __init__(self, master=None):
    super().__init__(master)
    self.pack(fill=tk.BOTH)
    tk.Label(self, text="Wieviel").grid(row=0, column=0)
    tk.Label(self, text="Wielang").grid(row=1, column=0)
    self.howmany = tk.Label(self)
    self.howmany.grid(row=0, column=1)
    self.took = tk.Label(self)
    self.took.grid(row=1, column=1)
    self.btn1 = tk.Button(self, text="Start bääh",
                          command=self.start)
    self.btn1.grid(row=2, column=0)
    self.btn2 = tk.Button(self, text="Start okay",
                          command=self.start_thread)
    self.btn2.grid(row=2, column=1)
    self.btn3 = tk.Button(self, text="Start aaah",
                           command=self.start_thread_cb)
    self.btn3.grid(row=2, column=2)
```

Threading, GUI-Thread blockieren ist böse!

Start bääh, nicht gut

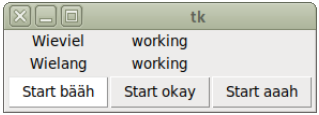
- GUI auf „working“
- Solange vom Internet lesen, bis die 17 kommt
- Wenn die 17 kommt messen und anzeigen




```
def start(self):
    self.took["text"] = "working"
    self.howmany["text"] = "working"
    self._root().update() # force screen update
    s, c = time.time(), 0
    read = 0
    while read != 17:
        read = read_random(0, 100)
        c += 1
    self.took["text"] = "%4.2fs" % (time.time()-s)
    self.howmany["text"] = "%03d" % c
```

Problem: GUI ist blockiert

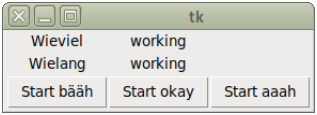
- 4: Damit man „working“ sieht, zwingen
- GUI während der kompletten Schleife nicht „responsive“ (oder nicht reactive)



Threading, Nicht den GUI-Thread blockieren, gut

Start_thread okay, gut

- GUI auf „working“
- Solange vom Internet lesen, bis die 17 kommt
- Wenn die 17 kommt messen und anzeigen



Kein Problem: GUI ist nicht blockiert

- Netzwerk IO in separatem Thread, okay
- GUI ist responsive
- Mehrmals starten möglich

```
1  def start(self):
2      self.took["text"] = "working"
3      self.howmany["text"] = "working"
4      self._root().update() # force screen update
5      s, c = time.time(), 0
6      read = 0
7      while read != 17:
8          read = read_random(0, 100)
9          c += 1
10     self.took["text"] = "%4.2fs" % (time.time()-s)
11     self.howmany["text"] = "%03d" % c
```

```
1  def start_thread(self):
2      Thread(target=self.start).start()
```

Anmerkung:

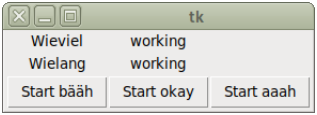
- 4: Damit man „working“ sieht, zwingen

Aber: Widgets im Background-Thread aktualisiert, :-(

Threading, Nie im Background-Thread GUI aktualisieren

Start_thread_cp, sehr gut, aaah

- Achtung Zeile 8-11 ist anders
- Nie im Background-Thread GUI aktualisieren
- Widgets nur im GUI-Thread aktualisieren
  - Sonst crash, außer bei Tkinter und Qt
  - `Tkinter` geht, GIL+intern
  - `Qt` geht, Thread-Wechsel über Signal/Slot
- 11: `update` sobald Zeit ist
- 15/16: Kein erzwungenes Refresh mehr notwendig




```
1  def start_cb(self):
2      s, c = time.time(), 0
3      read = 0
4      while read != 17:
5          read = read_random(0, 100)
6          c += 1
7      t = (time.time()-s)
8      def update():
9          self.took["text"] = "%4.2fs" % t
10         self.howmany["text"] = "%03d" % c
11     self._root().after_idle(update)
12
13 def start_thread_cb(self): # im main thread
14     self.took["text"] = "working"
15     self.howmany["text"] = "working"
16     Thread(target=self.start_cb).start()
```

Aber... leider Zeile 11 (`after_idle`) nicht thread safe

- Egal in `cpython`, crash in z.B. `pypy`
- Echter Mist (Bug?), andere Lösung muss her

Eigene Queue

Lösung, neu dazu, Overkill

- Eigene `Queue`
- `Queue` periodisch abgefragen
  - Z3-12: `event_loop`
  - Abfrage erfolgt im GUI-Thread
  - Z12: Der Animationstrick mit `after`, alle 100 ms im Beispiel
- In `Queue` werden Funktionen gesteckt
- Z8: Funktion im GUI-Thread rufen
- Z17: Neues main, vor `mainloop` noch Animationsloop starten
- Z19: Am Ende Animationsloop beenden über `Event`

Hinzufügen zur `Queue`

- Automatisch thread safe (nächste Seite)

```
1  queue = Queue()
2  done = Event()
3  def event_loop(root):
4      if not done.is_set():
5          try:
6              while True: # drain the queue
7                  fun = queue.get_nowait()
8                  fun()
9          except Empty:
10             pass
11         # do again after 100 msecs
12         root.after(100, lambda: event_loop(root))
13
14 if __name__ == '__main__':
15     root = tk.Tk()
16     main = Main(root)
17     event_loop(root) # establish anim loop
18     root.mainloop()
19     done.set() # no longer necessary
```

Eigene Queue verwenden

Verwenden einfach

- Nur Zeile 12 statt Zeile 11
- Statt `after_idle` jetzt in die `Queue` eine Funktion stecken
- Funktion wird spätestens in 100 ms, wenn GUI noch läuft, ausgeführt

Effekt, alles gut

- Klappt jetzt mit jeder Implementierung, auch `pypy`

Vor-/Nachteile

- Viel mehr Code, aber nur an einer Stelle
- Verwendung einfach, Setup aufwendig
- Keine sofortige Reaktion, doofes Pollen

```
1  def start_cb(self):
2      s, c = time.time(), 0
3      read = 0
4      while read != 17:
5          read = read_random(0, 100)
6          c += 1
7      t = (time.time()-s)
8      def update():
9          self.took["text"] = "%4.2fs" % t
10         self.howmany["text"] = "%03d" % c
11     # self._root().after_idle(update)
12     queue.put(update)
13
14 def start_thread_cb(self): # im main thread
15     self.took["text"] = "working"
16     self.howmany["text"] = "working"
17     Thread(target=self.start_cb).start()
```

My 0.02 EUR, they should fix it!

Nicht mehr richtig Ereignis-getrieben

Ereignisse/Events und Binding

Ereignis-getriebenes Programmieren

- Framework-Prinzip
- Applikation wird gerufen, wenn ein passendes Ereignis/Event eintritt
- Vordefiniertes Verhalten vieler Widgets, zum Beispiel Klick bei Buttons

Flexible Konfiguration, Binden in Tkinter

- Ereignis: `'<[Modifier-]type[-detail]>'`
  - `<Button-1>` Linksklick, `<ButtonRelease-3>` Rechte Maustaste loslassen, `<Enter>` Maus in Widget, `Control-KeyPress-h` Tastaturevent, ...
- Ereignis-Handler: Eine Funktion, Event als Parameter
- Binding: Binden Ereigniss an Funktion
  - Reichweite: `bind` Instanz, `bind_class` alle Instanzen, `bind_all` Anwendung

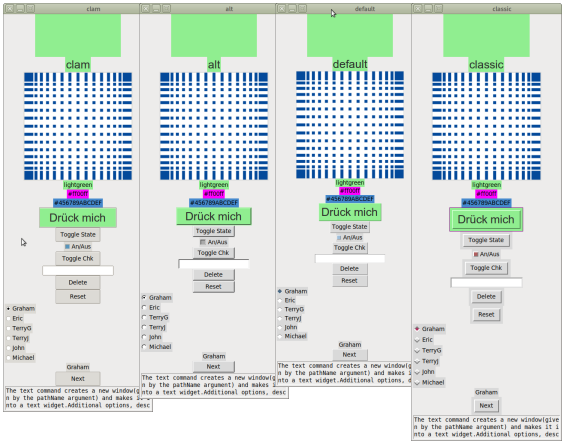
```
def callback(event=None):
    tup = () if not event else (event.x, event.y)
    print("callback", tup)
f1 = tk.Frame(root, width=100, height=20); f1.pack()
f1.bind("<Button-3>", callback)
f2 = tk.Frame(root, width=100, height=50); f2.pack()
tk.Button(f2, text="B1", command=callback).pack()
btn = tk.Button(f2, text="B2", command=callback)
btn.pack()
btn.bind("<Button-3>", callback)
root.bind_class('Label', "<Button-3>", callback)
tk.Label(f2, text="Label1").pack()
tk.Label(f2, text="Label2").pack()
root.bind_all('<Key-q>', lambda e: root.destroy())
```

```
('callback', (46, 14))
('callback', ())
('callback', ())
('callback', (24, 15))
('callback', (23, 9))
('callback', (24, 11))
```

ttk, Themed Widgets, Styles

Separiere Verhalten von Aussehen

- `from tkinter import tkk`
- Verwende oft `ttk` statt `tk`
- Anderes Styling, erbt von `Widget`




```python
import tkinter as tk
import tkinter.ttk as ttk
```

```python
style = ttk.Style()
style.theme_use(theme) # "default"
style.configure("main.TFrame",
                background="lightgreen")
frame = ttk.Frame(root, width=200, height=100,
                  style="main.TFrame")
frame.pack()

style.configure("main.TLabel",
                background="lightgreen",
                foreground="gray17",
                font="Arial 20")
label = ttk.Label(root, text=theme,
                  style="main.TLabel")
label.pack()
```

ttk, Kombinierte Widgets

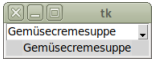
Kombinierte Widgets

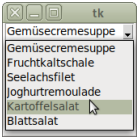
- Combobox
- Spinbox
- Notebook mit Tabs
- Progressbar
- TreeView

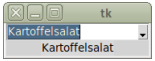
Idee

- Häufig gebrauchte Kombinationen von Widgets
- Nicht immer wieder Rad neu erfinden
- Immer noch sehr rudimentär, da ist Qt deutlich umfangreicher

```
values = ["Gemüsecremesuppe", "Fruchtkaltschale",
          "Seelachsfilet", "Joghurtremoulade",
          "Kartoffelsalat", "Blattsalat"]
cb = ttk.Combobox(frame, values=values)
cb.pack()
lb = ttk.Label(frame)
lb.pack()
def update(event=None):
    lb["text"] = cb.get()
cb.bind("<<ComboboxSelected>>", update)
cb.current(0)
update()
```







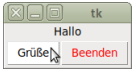
Templates für Tkinter-Apps

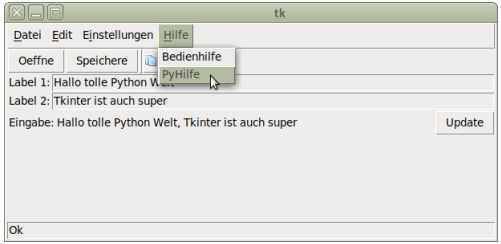
Einfaches Template

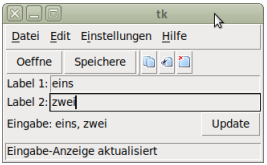
- Siehe Beispiel für Tkinter Grüße, Objektorientiert
- `create_widgets`: Initialisierung
- `end_it`: Sauberes Beenden der Anwendung
- Beispieldatei: `tktpl.py`

Eleganter, `App` (erben)

- Menus, Buttonbar, Statusbar, Icons, Shortcuts
- `create_widgets`,
  optional: `create_menubar`, `create_buttonbar`
- Helper: `MenuConfig`, `ButtonConfig`
- Dateien: `etkapp.py`, `etkapp_sample.py`







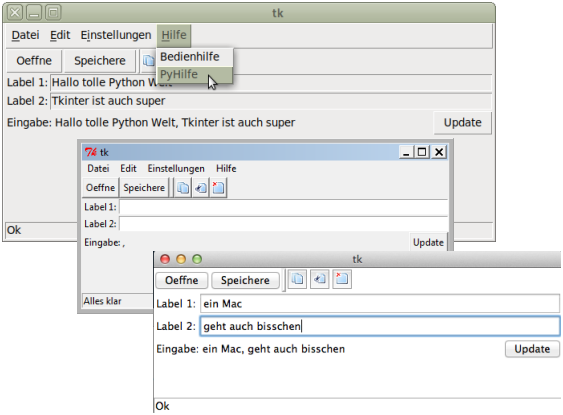
Noch mehr...

Features, Anwendungen

- Clipboard, Copy/Paste
- Animationen, . . .

Plattformunabhängigkeit

- Native Look and Feel



Python für Numerische Berechnungen und Visualisierung

Python-Bibliotheken für Ingenieure/Wissenschaftler

- Datenbeschaffung, -verarbeitung sowie 2D- und 3D-Visualisierung
- Numerische Berechnungen, Statistik, künstliche Intelligenz
- Vielfältig, dokumentiert, verbreitet “matlab”-ish

Basis-Bibliotheken

- numpy: `numpy.org`, Numerical Python, multidimensionale Arrays, schnelle Operationen auf Arrays, schicke API für Algorithmen in C/C++
- matplotlib: `matplotlib.org`, Erstellen von 2D-Visualisierungen (Druck/interaktiv)
- pandas: `pandas.pydata.org` Python Data Analysis Library, Beschaffung/Auswertung tabellarischer Daten, statistische Auswertungen
- simpy: `sympy.org`, Symbolische Mathematik, Computer Algebra

Pakete, die Basis-Bibliotheken beinhalten

- SciPy: Scientific Python, `scipy.org`
- Anaconda: Python-Distribution für wissenschaftliches Arbeiten, plus IDE spyder, . . .
- WinPython: Python-Distribution für wissenschaftliches Arbeiten, plus IDE spyder, unter Windows, nicht ganz so aufgeblasen und widerspenstig wie Anaconda . . .

Fokus hier: `numpy` und `matplotlib`

- Grundlage für numerische Berechnungen
- Ordentliche Graphiken, (auch) für Abschlussarbeit

NumPy, Numerical Python

Ziel, für numerische Berechnungen

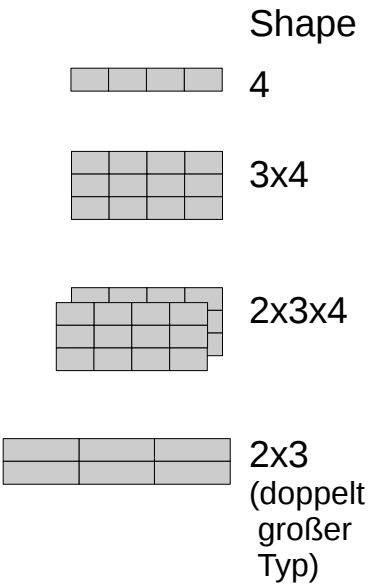
- Ausdrucksmächtigkeit von Python
- Ausführungsgeschwindkeit von C/C++

Historie

- Numeric (1995), Numarray (2001)
- Firma Enthought, Scientific Computing
- matplotlib (2001)
- NumPy (2005)

Kern-Feature: Array

- Matrix, mehrdimensionales Feld von Werten eines Typs, C/C++-Layout
- Slicing, Shaping, Extraktion, Sichten
- Schnelle mathematische Operationen: Vektorisierung, Mapping, Aggregation



Motivation: Eleganz und Performance

Eleganz

```python
# with python3
def inpy3(a):
    return sum([math.sin(x) for x in a])
```

Performance

```
>>> a = list(map(float, range(123456)))
>>> print("sum=%g" % inpy3(a))
sum=1.9007
>>> %timeit inpy3(a)
100 loops, best of 3: 12.6 ms per loop
>>> a = list(map(float, range(1234567)))
>>> print("sum=%g" % inpy3(a))
sum=1.58531
>>> %timeit inpy3(a)
10 loops, best of 3: 135 ms per loop
```

Eleganz

```python
# with numpy
def withnp(a):
    return np.sum(np.sin(a))
```

Performance

```
>>> a = np.arange(123456, dtype=float)
>>> print("sum=%g" % withnp(a))
sum=1.9007
>>> %timeit withnp(a)
1000 loops, best of 3: 1.84 ms per loop
>>> a = np.arange(1234567, dtype=float)
>>> print("sum=%g" % withnp(a))
sum=1.58531
>>> %timeit withnp(a)
100 loops, best of 3: 18.6 ms per loop
```

Ca. eine Größenordnung schneller (kann mehr werden)

`% timeit`: IPython Builtin für Microbenchmarking
i7-6700, Python 3.6.5, numpy 1.13.3, Ubuntu 18.04

numpy verwenden

```
import numpy as np
```

- Externes Modul importieren
- Nicht Teil der Standardbibliothek
- Etablierter abkürzender Name: `np`

Datenstruktur `array`

- Konstruktor
- Aus Python Liste ein Array machen
- Mehrdimensional

Verwendung

- Zugriff wie Liste
- Slicing
- Slicing macht eine Sicht, keine Kopie!
- Operationen sind Vektoroperation

```
>>> import numpy as np
>>> np.array([1,2,3])
array([1, 2, 3])
>>> a = np.array([1,2,3])
>>> print(type(a), a)
<class 'numpy.ndarray'> [1 2 3]
>>> a[0], a[1], a[2]
(1, 2, 3)
>>> a[0] = 17
>>> a
array([17, 2, 3])
>>> a[:2]
array([17, 2])
>>> a[:2] = 42; a
array([42, 42, 3])
>>> b = a[:2]; b
array([42, 42])
>>> b[0] = 17
>>> a
array([17, 42,  3])
```

Arrays erzeugen

Aus Listen

- Name als Konstruktor
- Parameter ist Liste/Tupel/Iterable
- Mehrdimensional möglich, Eigenschaft `shape`
- Achtung: gleiche Dimensionen, sonst Quatsch

Spezial: Nullen oder Einsen

Datentypen, Positionsparameter `dtype`

- Klassische Datentypen `int`, `float`
- NumPy diskrete Datentypen:
  `int`, `int8`, `int16`, `int32`, `int64`,
  `int0 (intp)` ist **`size_t`**, unsigned `uint`, ...
- NumPy Fließpunkt-Datentypen:
  `float`, `float64` (**`double`**), float32 (**`float`**)
- Strukturierte Typen (**`struct`**) möglich
- `align`-Flag (C-Compiler nachahmen)

```
>>> import numpy as np
>>> a = np.array((1,2,3,4))
>>> a.shape
(4,)
>>> a = np.array([[1,2,3,4], [4,5,6,7]])
>>> a
array([[1, 2, 3, 4], [4, 5, 6, 7]])
>>> a.shape, a.dtype
((2, 4), dtype('int64'))
>>> np.zeros((3,4))
array([[ 0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.]])
>>> np.ones((2,3))
array([[ 1.,  1.,  1.],
       [ 1.,  1.,  1.]])
>>> np.array([[1,2,3,4], [4,5,6]]) # bad
array([[1, 2, 3, 4], [4, 5, 6]], dtype=object)
>>> np.array(range(5), dtype=float)
array([ 0.,  1.,  2.,  3.,  4.])
```

Arrays erzeugen und formen

`arange`: 1-dimensionales Array, wie `range`
`reshape`: Sicht auf Array mit passender Dimension, keine Kopie
`flatten`: Kopie in 1-dimensionales Array

```
>>> a = np.arange(8); a
array([0, 1, 2, 3, 4, 5, 6, 7])
>>> b = a.reshape(2, 4); b
array([[0, 1, 2, 3],
       [4, 5, 6, 7]])
>>> a[0] = 17
>>> b
array([[17,  1,  2,  3],
       [ 4,  5,  6,  7]])
>>> c = b.flatten(); c
array([17,  1,  2,  3,  4,  5,  6,  7])
>>> c[0] = 42
>>> a
array([17,  1,  2,  3,  4,  5,  6,  7])
```

```
>>> a = np.arange(10); a
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> a.reshape((2,5))
array([[0, 1, 2, 3, 4],
       [5, 6, 7, 8, 9]])
>>> a.shape
(10, )
>>> a = a.reshape((2,5)); a.shape
(2, 5)
>>> a.reshape((2, 3))
ValueError: cannot reshape arrof 10 into (2,3)
>>> a = np.arange(24).reshape(2,3,4); a
array([[[ 0,  1,  2,  3], [ 4,  5,  6,  7],
        [ 8,  9, 10, 11]],
       [[12, 13, 14, 15], [16, 17, 18, 19],
        [20, 21, 22, 23]]])
>>> a.flatten()
array([ 0,   1,  2,  3,  4,  5,  6,  7,  8,
        9,  10, 11, 12, 13, 14, 15, 16, 17,
       18, 19, 20, 21, 22, 23])
```

Arrays elegant erzeugen

`linspace(fro, to, n)`

- `float`-Array mit gleichem Abstand zwischen den Werten erzeugen
- `fro`: Anfangswert, inklusive
- `to`: Endwert exklusive
- `n`: Anzahl der Werte

`logspace(fro, to, n, base=10.0)`

- `float`-Array mit logarithmischen Abstand zwischen den Werten erzeugen
- Werte sind Exponente, linear in den Exponenten
- Basis kann angegeben werden

```
>>> np.linspace(1, 3, 6)
array([ 1. ,  1.4,  1.8,  2.2,  2.6,  3. ])
>>> np.linspace(0, 1, 10)
array([ 0.        ,  0.11111111,  0.22222222,
        0.33333333,  0.44444444,  0.55555556,
        0.66666667,  0.77777778,  0.88888889,
        1.        ])
>>> np.linspace(0, 1, 11)
array([ 0. ,  0.1,  0.2,  0.3,  0.4,  0.5,
        0.6,  0.7,  0.8,  0.9,  1. ])
np.linspace(0, 1, 10, endpoint=False)
array([ 0. ,  0.1,  0.2,  0.3,  0.4,  0.5,
        0.6,  0.7,  0.8,  0.9])
>>> np.logspace(0, 10, 11, base=2)
array([  1.00000000e+00,   2.00000000e+00,
         4.00000000e+00,   8.00000000e+00,
         1.60000000e+01,   3.20000000e+01,
         6.40000000e+01,   1.28000000e+02,
         2.56000000e+02,   5.12000000e+02,
         1.02400000e+03])
```

Memory-Layout von Arrays

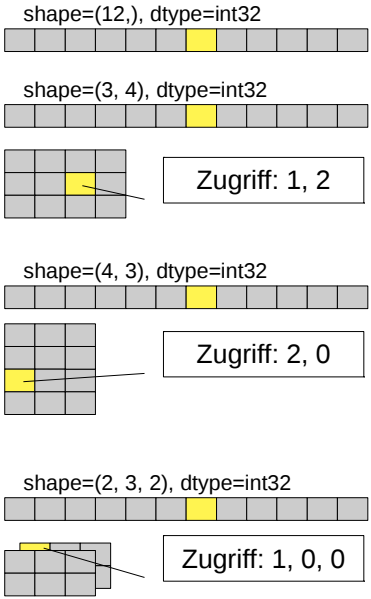
Flaches Memory-Layout, wie in C

- Unabhängig vom Shape
- Shape separat, wird für Zugriff verwendet
- Oben echt, unten Zugriffsmodell

Beispiel: Feld mit 12 Mal Integer (a 4 Byte)

- Flach, wie in Memory: Zugriff Element an Index 6
- 3x4: $1 \cdot 4 + 2 = 6$
- 4x3: $2 \cdot 3 + 0 = 6$
- 2x3x2: $1 \cdot 6 + 0 \cdot 3 + 0 = 6$

```
a = np.arange(12)
print(a[6]) # 6
a = np.arange(12).reshape((3, 4))
print(a[1,2]) # 6
a = np.arange(12).reshape((4, 3))
print(a[2,0]) # 6
a = np.arange(12).reshape((2, 3, 2))
print(a[1,0,0]) # 6
```



Zugriff auf Arrays - Einfach

Zugriff

- Kommasepariert in einer Klammer []
- Sichten auf altes Feld

```
>>> a = np.arange(12).reshape(3, 4)
>>> a[1]
array([4, 5, 6, 7])
>>> a[1, 2]
6
>>> a[1:3, 1]
array([5, 9])
>>> a[0:2, 1:3]
array([[1, 2],
       [5, 6]])
>>> a[[0,-1]]
array([[ 0,  1,  2,  3],
       [ 8,  9, 10, 11]])
```

a

0	1	2	3
4	5	6	7
8	9	10	11

a[1, 2]

0	1	2	3
4	5	6	7
8	9	10	11

a[1:3, 1]

0	1	2	3
4	5	6	7
8	9	10	11

a[0:2, 1:3]

0	1	2	3
4	5	6	7
8	9	10	11

a[[0, -1]]

0	1	2	3
4	5	6	7
8	9	10	11

Zugriff auf Arrays - Elegant Unterarrays

```
>>> a = np.arange(12).reshape(3, 4)
>>> a[[0,-1]]
array([[ 0,  1,  2,  3],
       [ 8,  9, 10, 11]])
>>> a[[0,-1], 0] # vector remaining
array([0, 8])
>>> a[[0,-1], [0, -1]] # matching pairs
array([ 0, 11])
>>> a[[0,-1]][:,[0,-1]] # slice of result
array([[ 0,  3],
       [ 8, 11]])
>>> a.shape[0]
3
>>> a[range(3), range(3)] # matching pairs
array([ 0,  5, 10])
```

a

0	1	2	3
4	5	6	7
8	9	10	11

a[[0, -1]]

0	1	2	3
4	5	6	7
8	9	10	11

a[[0, -1], 0]

0	1	2	3
4	5	6	7
8	9	10	11

a[[0, -1], [0, -1]]

0	1	2	3
4	5	6	7
8	9	10	11

a[[0, -1]][:, [0, -1]]

0	1	2	3
4	5	6	7
8	9	10	11

a[range(3), range(3)]

0	1	2	3
4	5	6	7
8	9	10	11

Zugriff auf Arrays - Strides

```
>>> a = np.arange(12).reshape(3, 4)
>>> a[::2]
array([[ 0,  1,  2,  3],
       [ 8,  9, 10, 11]])
>>> a[::2, ::2]
array([[ 0,  2],
       [ 8, 10]])
>>> a[::2, ::3]
array([[ 0,  3],
       [ 8, 11]])
# corners matrix
>>> a[::a.shape[0]-1, ::a.shape[1]-1]
array([[ 0,  3],
       [ 8, 11]])
```

a

0	1	2	3
4	5	6	7
8	9	10	11

a[::2]

0	1	2	3
4	5	6	7
8	9	10	11

a[::2, ::2]

0	1	2	3
4	5	6	7
8	9	10	11

a[::2, ::3]

0	1	2	3
4	5	6	7
8	9	10	11

Ändern von Arrays, Vektoroperationen

```
# reset auf Original nach jedem Beispiel
>>> a = np.arange(12).reshape(3, 4)
>>> a[1, 2] = 17
>>> a
array([[ 0,  1,  2,  3],
       [ 4,  5, 17,  7],
       [ 8,  9, 10, 11]])

>>> a[1] = 17
>>> a
array([[ 0,  1,  2,  3],
       [17, 17, 17, 17],
       [ 8,  9, 10, 11]])

>>> a[::2, ::3] = 17
>>> a
array([[17,  1,  2, 17],
       [ 4,  5,  6,  7],
       [17,  9, 10, 17]])
```

a

0	1	2	3
4	5	6	7
8	9	10	11

a[1, 2] = 17

0	1	2	3
4	5	17	7
8	9	10	11

a[1] = 17

0	1	2	3
17	17	17	17
8	9	10	11

a[::2, ::3] = 17

17	1	2	17
4	5	6	7
17	9	10	17

Ändern von Arrays, elegant

```
# reset auf Original nach jedem Beispiel
>>> a = np.arange(12).reshape(3, 4)

>>> a[[0,1,2,2], 0]
array([0, 4, 8, 8])
>>> a[1] = a[[0,1,2,2], 0]
>>> a
array([[ 0,  1,  2,  3],
       [ 0,  4,  8,  8],
       [ 8,  9, 10, 11]])

>>> a[::2, ::3] = a[1].reshape(2, 2)
>>> a
array([[ 4,  1,  2,  5],
       [ 4,  5,  6,  7],
       [ 6,  9, 10,  7]])
```

a

0	1	2	3
4	5	6	7
8	9	10	11

a[1] = a[[0,1,2,2], 0]

0	1	2	3
0	4	8	8
8	9	10	11

a[::2, ::3] = a[1].reshape(2,2)

4	1	2	5
4	5	6	7
6	9	10	7

Slices/Strides sind Sichten auf ursprüngliche Arrays

```
>>> a = np.arange(12).reshape(3, 4)
>>> b = a[::2, ::3]
>>> b
array([[ 0,  3],
       [ 8, 11]])
>>> b[:] = a[1].reshape((2, 2))
>>> a
array([[ 4,  1,  2,  5],
       [ 4,  5,  6,  7],
       [ 6,  9, 10,  7]])

>>> b.flags.owndata
False
```

a

0	1	2	3
4	5	6	7
8	9	10	11

b = a[::2, b::3]

0	1	2	3
4	5	6	7
8	9	10	11

b[:]=a[1].reshape(2,2)

4	1	2	5
4	5	6	7
6	9	10	7

Operationen auf Arrays

Komponentenweise

- Üblicherweise sind alle Operationen komponentenweise
- Auch wenn eher unüblich (Beispiel: *)

Skalare Operationen, Vektoroperationen

- Für Multiplikation und Power klar
- Geht auch unüblich als Argument

Spezialoperationen

- Dedizierte Funktionsnamen
- Beispiel: Skalarprodukt (dot product)

```
>>> a, b = np.array([1,2,3,4]), np.array([5,6,7,8])
>>> a, b
(array([1, 2, 3, 4]), array([5, 6, 7, 8]))
# Komponentenweise
>>> a+b
array([ 6,  8, 10, 12])
>>> a*b
array([ 5, 12, 21, 32])

# Skalar
>>> a, a*2
(array([1, 2, 3, 4]), array([2, 4, 6, 8]))
>>> a**2
array([ 1,  4,  9, 16])
>>> 2**a
array([ 2,  4,  8, 16])

# Spezialoperationen, Skalarprodukt
>>> np.dot(a, b), np.sum(a*b)
(70, 70)
```

Viele eingebauten Funktionen

Eingebaute Funktionen, Auszug

- Vergleiche: `<, <=, ==, !=, >= >`
- Arithmetik: `+, -, *, /, reciprocal, square`
- Exponential: `exp, exp2, expm1, log, log10, log2, power, sqrt`
- Trigonometrisch: `sin, cos, tan, arc...`
- Hyperbolisch: `sinh, cosh, tanh, arc...`
- Bits: `&, |, ~, ^, left_shift, right_shift`
- Logisch: `logical_and, ..._or, ..._not, ..._xor`
- Prädikate: `isfinite, isinf, isnan, signbit`
- Sonst: `abs, ceil, floor, mod, modf, round, sinc, sign, trunc`

```
>>> a = np.array([1,2,3,4,5])
>>> a>3
array([False, False, False,  True,  True],
      dtype=bool)
>>> a%2==1 # odd
array([ True, False,  True, False,  True],
      dtype=bool)
>>> np.sin(a)
array([ 0.84147098,  0.90929743,  0.14112001,
       -0.7568025 , -0.95892427])
>>> a-3
array([-2, -1,  0,  1,  2])
>>> np.sign(a-3)
array([-1, -1,  0,  1,  1])
>>> np.abs(a-3)
array([2, 1, 0, 1, 2])
>>> np.left_shift(a, 2)
array([ 4,  8, 12, 16, 20])
```

Funktionen und Achsen

Reduktionsachse

- Aggregationsoperationen, `sum`
- Angabe der Reduktionsachsen, Dimension um 1 erniedrigt

```
>>> a = np.arange(12).reshape((3, 4))
>>> a
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
>>> np.sum(a)
66
>>> np.sum(a, axis=0)
array([12, 15, 18, 21])
>>> np.sum(a, axis=1)
array([ 6, 22, 38])
```

0	1	2	3
4	5	6	7
8	9	10	11

6
22
38

12	15	18	21

66

Broadcasting kompatibler Arrays

Broadcast

- Operationen klappen bei gleicher Dimension
- Wenn eine Dimension 1 ist, dann behandeln als wenn Fehlendes wiederholt wird

```
>>> a = np.arange(12).reshape((3, 4))
>>> b = np.array([1,2,3])
>>> a + b
ValueError: operands could not be broadcast
            together with shapes (3,4) (3,)
>>> b = np.array([1,2,3]).reshape((3, 1))
>>> b
array([[1, 2, 3]])
>>> a + b
array([[ 1,  2,  3,  4],
       [ 6,  7,  8,  9],
       [11, 12, 13, 14]])
```

0	1	2	3
4	5	6	7
8	9	10	11

1
2
3

1	1	1	1
2	2	2	2
3	3	3	3

1	2	3	4
6	7	8	10
11	12	13	14

Prädikate, Reduktionen, Manipulationen, ...

Prädikate

- `all`: Alle Werte äquivalent zu `True`
- `any`: Mindestens ein Wert äquivalent zu `True`

Weitere Reduktionen: . . .

- `trace`: Summe Diagonale

Weitere Manipulationen: . . .

```
>>> a.transpose()
array([[ 0,  4,  8],
       [ 1,  5,  9],
       [ 2,  6, 10],
       [ 3,  7, 11]])
>>> a.clip(5, 7)
array([[ 5, 5, 5, 5],
       [ 5, 5, 6, 7],
       [ 7, 7, 7, 7]])
```

Es geht erst los, . . . , Dokumentation!

```
>>> a = np.arange(12).reshape((3, 4))
>>> a > 10
array([[False, False, False, False],
       [False, False, False, False],
       [False, False, False,  True]],
       dtype=bool)
>>> (a > 10).all(), (a > 10).any()
(False, True)
>>> a.min(), a.mean(), a.max(), a.trace()
(0, 5.5, 11, 15)
>>> a.cumsum()
array([ 0,  1,  3,  6, 10, 15, 21, 28, 36, 45,
       55, 66])
>>> a[1:]
array([[ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
>>> a[1:].cumprod()
array([      4,      20,     120,     840,
        6720,   60480,  604800, 6652800])
```

Daten visualisieren mit `matplotlib`

Ziel: Datenvisualisierung

- Professionale 2D-Graphiken skripten mit Python und `numpy`
- Ursprünglich nahe zu Matlab, aber pythonic

Features

- Sehr hohe Qualität, Buchdruckstandard
- Viele Backends: PDF, PS, SVG, PNG, Tkinter, ...
- Natürlich plattformunabhängig

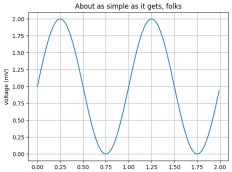
Historie: `www.matplotlib.org`

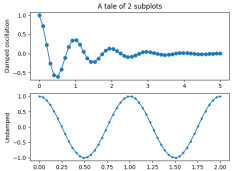
- 2003: Nachbilden Matlab-Graphiken
- 2010: Version 1.0, de facto Standard für 2D-Vis.
- ab 2012: Matplotlib development team

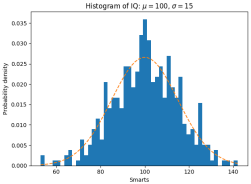
Was fehlt (?)

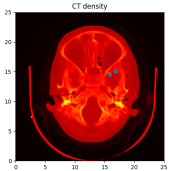
- Skripten, nicht alles automatisch, nicht Statistik geschenkt: → pandas (basiert auf matplotlib)
- Graphiken, nicht interaktives Explorieren im Browser: → bokeh, plot.ly

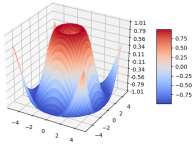
Beispiele

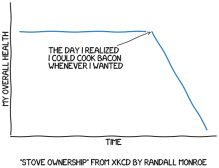












matplotlib.org/tutorials/introductory/sample_plots.html

matplotlib verwenden

```
import matploblib.pyplot as plt
```

- Externes Modul importieren
- Nicht Teil der Standardbibliothelk
- Etablierter Namen für die reine Plotting-Funktionalität: `plt`
- Wird meist zusammen mit `numpy` verwendet

Daten aufbereiten mit `numpy`

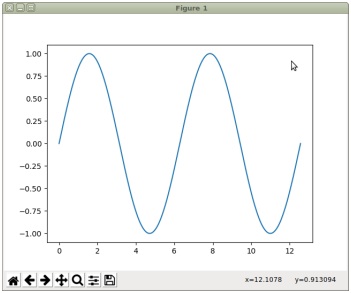
- 1000 Samples zwischen 0 bis $4 \cdot \pi$
- Sinus plotten (sind dann zwei Perioden)

Daten anzeigen mit `matplotlib`

- x und y Werte an `plt.plot`-Funktion übergeben
- Interaktiv anzeigen mit `plt.show()`

```python
#!/usr/bin/python3
import matplotlib.pyplot as plt
import numpy as np

x = np.linspace(0, 4*np.pi, 1000)
y = np.sin(x)
plt.plot(x, y)
plt.show()
```



Einfache Interaktion mit Plot

Settings, Slider klicken

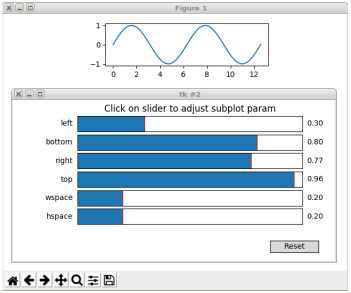
- 6 einfache Einstellungen
- Außendimendionen des Bildes

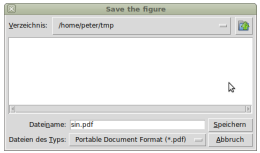
Speichern als

- PNG (vermeiden, pixelig!)
- PDF, PS, EPS, SVG (gut)
- PGF (LaTeX)
- RAW, TIFF (vermeiden, pixelig, unkomprimiert)

Sonstiges

- Achsen verschieben (pan)
- Zoom





Bitte keine pixeligen Graphiken!

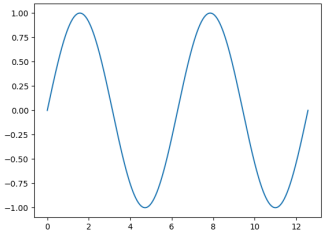
Graphikformate

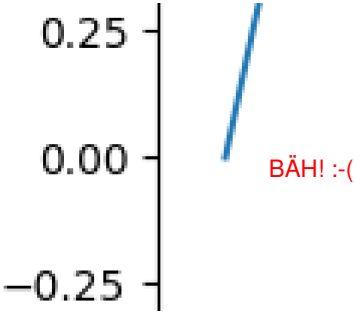
- für Screenshot: PNG
- für Bilder (Photo): JPG
- für Graphiken: Vektorgrafikformat, wie PDF

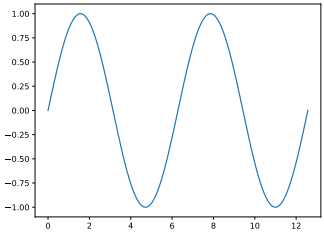
Pixelig ist **BÄH**

- Skalierungseffekte
- Bei Druck auf hochwertigem Drucker sichtbar
- PS-Berichte und Abschlussarbeiten sind Pixelig-freie Zonen!

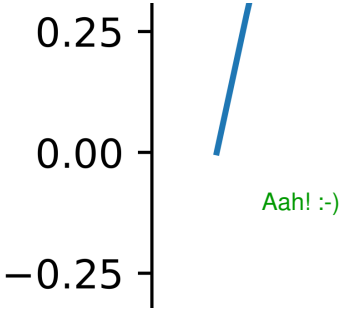
PNG







PDF



Erstellen von Graphiken automatisieren

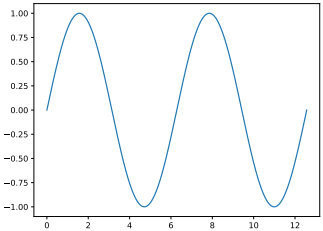
Graphiken in Skript

- Erstellen, Speichern, auch mehr als eine
- bbox tight, weniger Rand, Verwendung in Druck
- Mehrere Plots möglich

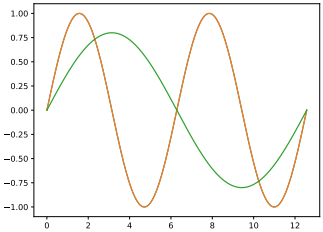
```
#!/usr/bin/python3
import matplotlib.pyplot as plt
import numpy as np

x = np.linspace(0, 4*np.pi, 1000)
y1 = np.sin(x)
plt.plot(x, y1)
plt.savefig("pltsimple_save.pdf",
                bbox_inches='tight')

y2 = np.sin(.5*x)*.8
plt.plot(x, y1, x, y2)
plt.savefig("pltsimple2_save.pdf",
                bbox_inches='tight')
```



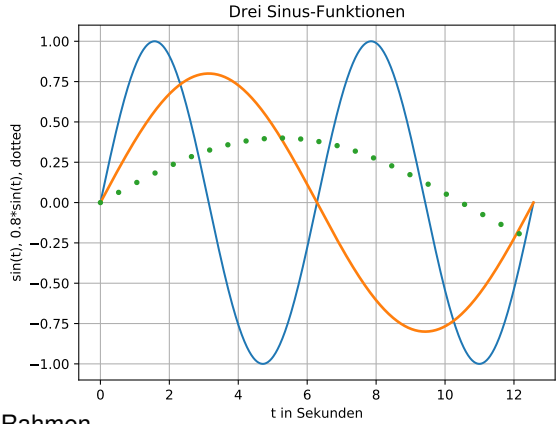
pltsimple_save.pdf



pltsimple2_save.pdf

Plots konfigurieren

```
x = np.linspace(0, 4*np.pi, 1000)
y1 = np.sin(x)
y2 = np.sin(.5*x)*.8
y3 = np.sin(.3*x[::42])*.4
plt.plot(x, y1)
plt.plot(x, y2, linewidth=2)
plt.plot(x[::42], y3, '.')
plt.title("Drei Sinus-Funktionen")
plt.xlabel("t in Sekunden")
plt.ylabel("sin(t), 0.8*sin(t), dotted")
plt.grid(True)
```



Mehrere Plots

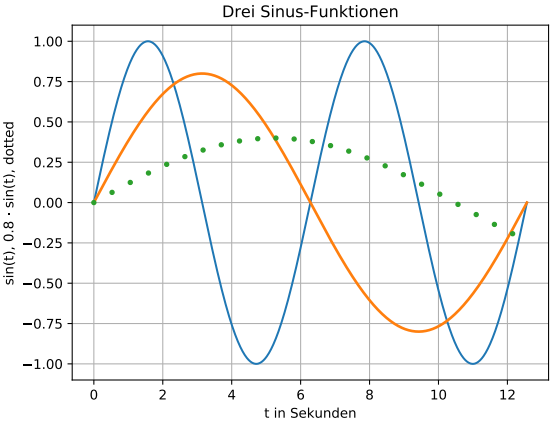
- Mehrere Aufrufe
- Separat konfiguriert
- Zum Beispiel Linienbreite, Anzeige als Punkte
- Dokumentation!

Rahmen

- Titel, Beschriftung Achsen
- Ein Grid im Hintergrund
- Dokumentation

OO-Style Verwendung

```
x = np.linspace(0, 4*np.pi, 1000)
y1 = np.sin(x)
y2 = np.sin(.5*x)*.8
y3 = np.sin(.3*x[::42])*.4
fig, ax = plt.subplots()
ax.plot(x, y1)
ax.plot(x, y2, linewidth=2)
ax.plot(x[::42], y3, '.')
ax.set(xlabel="t in Sekunden",
  ylabel="sin(t), 0.8 $\cdot$ sin(t), dotted",
  title="Drei Sinus-Funktionen")
ax.grid()
```



Vorher Matlab-Style

- Zustandsbehaftet
- Anfangs einfach, später Hölle

Jetzt OO-Style: Wie vom Programmieren gewohnt

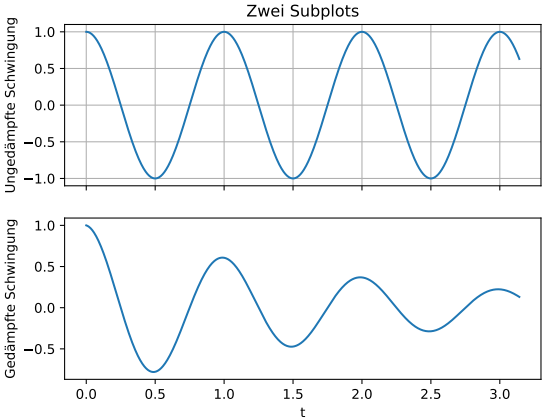
- Objekte halten Zustand
- Objekte manipulieren

LaTeX nutzen

- Nicht $0.8 * sin(t)$, sondern $0.8 \cdot sin(t)$
- Man kann LaTeX nutzen und tippt
  `"... 0.8 $\cdot $ sin(t)..."`

Mehrere Teilplots

```
x = np.linspace(0, 1*np.pi, 1000)
unged = np.cos(2*np.pi*x)
ged   = np.cos(2*np.pi*x)*np.exp(-.5*x)
fig, axs = plt.subplots(nrows=2, ncols=1,
                        sharex=True)
axs[0].plot(x, unged)
axs[0].set(ylabel="Ungedämpfte Schwingung")
axs[1].plot(x, ged)
axs[1].set(xlabel="t",
        ylabel="Gedämpfte Schwingung")
axs[0].set(title="Zwei Subplots")
axs[0].grid()
```

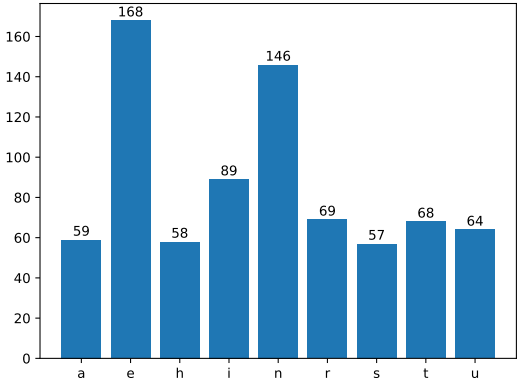


- Teilplots durch Angabe der Anzahl je Achse
- Rückgabe der Rahmen-Figure und Liste der Teilplots
- Man beachte `numpy`-Verwendung

- Zwei Teilplots
- Unabhängig, außer explizit gesharter X-Achse
- Grid nur oben

Balkendiagramm

```
print(leitbild[:42]) # diehochschulemannheimhatihrezentraleaufgab
abc = string.ascii_lowercase
items = [(c, leitbild.count(c)) for c in abc if c in leitbild]
items = [(c, v) for c, v in items if v > 50]
keys, values = zip(*items)
fig, ax = plt.subplots()
xind = np.array(range(len(keys)))
ax.set_xticks(xind)
ax.set_xticklabels(keys)
ax.bar(xind, values)
for i, v in enumerate(values):
    ax.text(i, v+2, str(v), ha='center')
```

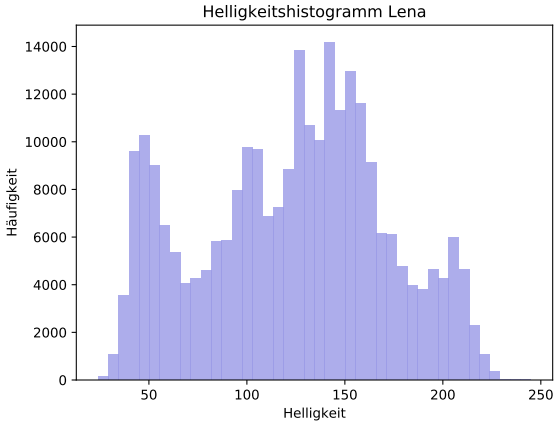


- bar chart, mit Beschriftung
- Angabe der Werte an Bar untypisch, skripten

Alle Zeichen, die mehr als 50 mal im Leitbild vorkommen

Histogramm

```
img = Image.open("lena.png").convert('L')
lena = np.array(img.getdata(),
                dtype=np.uint8)
fig, ax = plt.subplots()
n, bins, patches = ax.hist(lena, 42,
    facecolor=(.6, .6, .9),
    alpha=0.80)
ax.set(xlabel="Helligkeit")
ax.set(ylabel="Häufigkeit")
ax.set(title="Helligkeitshistogramm Lena")
```

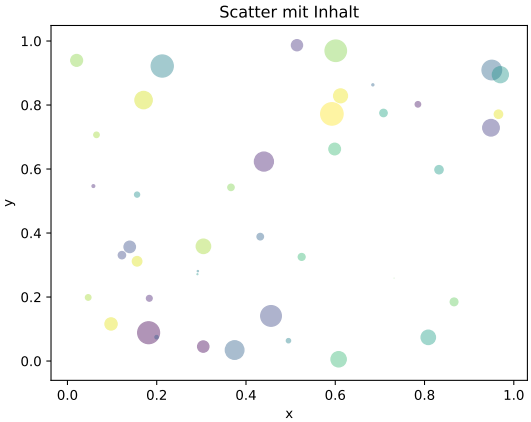


- Histogramm
- Angabe der Klassifikation, in 42 Teile
- Bars mit Alpha-Kanal (Durchlässigkeit)

Einlesen, Datentyp für Graustufen, `char` (`uint8`)

Scatter-Plot

```
np.random.seed(42) # immer gleich
N = 42
x = np.random.rand(N)
y = np.random.rand(N)
colors = np.random.rand(N)
area = (17*np.random.rand(N))**2
fig, ax = plt.subplots()
ax.scatter(x, y, s=area, c=colors,
           alpha=0.42)
ax.set(xlabel="x")
ax.set(ylabel="y")
ax.set(title="Scatter mit Inhalt")
```



- xy-Scatter-Plot
- Angabe der einzelnen Werte für x, y, Flächeninhalt, Farben
- Alpha, damit man Überlappung besser sieht

Matplotlib in Tkinter

Verwenden von Plots in einen Tkinter-GUI

- Einfach machbar
- Dedizierte Integrations-Wrapper verfügbar
- Animationen möglich

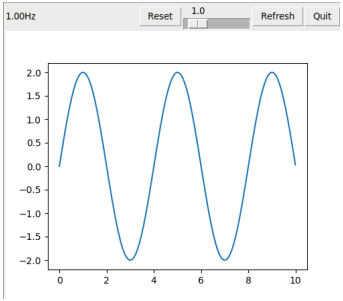
Beispiel

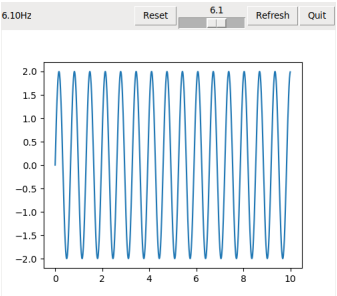
- Einfacher Sinusplot
- Frequenz einstellbar über Slider (`tk.Scale`)

Umsetzung/1

- Importieren der Integration
- `FigureCanvasTkAgg`, ein Plot aus Tkinter-Sicht
- `Figure`, ein Plot aus Matplotlib-Sicht

```
from matplotlib.backends.backend_tkagg import (
    FigureCanvasTkAgg)
from matplotlib.figure import Figure
```





Matplotlib in Tkinter, Umsetzung/2

Erstellen der Widgets

- Z2: die x-Werte
- Z3: ein Unterplot in den wir zeichnen
- Z4: wir zeichnen auf einem Canvas
- Z5-6: den Canvas ausfüllend layouten
- Z7: Initiale Frequenz, 1Hz
- Z8: Zum ersten Mal anzeigen

Frequenz aktualisieren bei Slider-Event

- Wert merken und neu zeichnen

Neue Werte berechnen

- Sinus aller gemerkten x-Werte

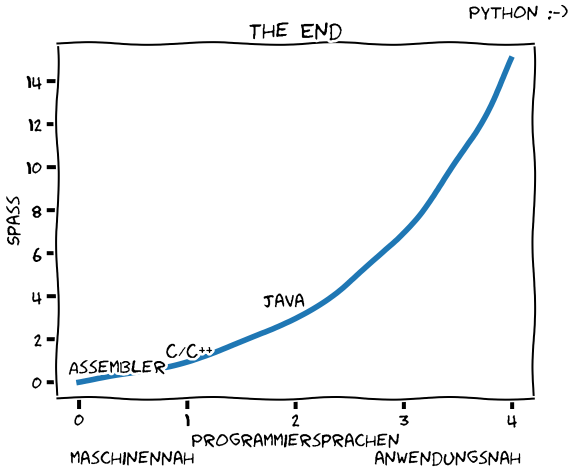
Plot neu zeichnen

- Z7-9: Werte berechnen, löschen, plotten
- Z10: Frequenzanzeige (links) aktualisieren
- Z11: Plot auf Canvas zeichnen

in Klasse `Main`, in Methode `create_widgets`

```
1  fig = Figure()
2  self.t = np.arange(0, 10, .01)
3  self.a = fig.add_subplot(111) # 1 row, 1 column, @1
4  self.canvas = FigureCanvasTkAgg(fig, master=self)
5  self.canvas.get_tk_widget().pack(side=tk.TOP,
6                 fill=tk.BOTH, expand=True)
7  self.scale.set(self.frequenz) # var des sliders
8  self.refresh()
```

```
1  def update_scale(self, event=None): # vom slider
2      self.frequenz = float(event)
3      self.refresh()
4  def fun(self):
5      return 2*np.sin(np.pi/2.0*self.t*self.frequenz)
6  def refresh(self):
7      vals = self.fun()
8      self.a.clear()
9      self.a.plot(self.t, vals)
10     self.label_freq["text"] = "%4.2fHz" % self.frequ
11     self.canvas.draw()
```

The End

```python
plt.xkcd()
x = np.linspace(0, 4, 100)
y = (2**x)-1
fig, ax = plt.subplots()
ax.plot(x, y, linewidth=4.2)
s = "Programmiersprachen\n"
s += "maschinennah"+(" "*20)+"anwendungsnah"
ax.set(xlabel=s)
ax.set(ylabel="Spass")
ax.set(title="The End")
ax.annotate('Assembler', xy=(0, 0),
            xytext=(-0.1, .5))
ax.annotate('C/C++', xy=(0, 0),
            xytext=(0.8, 1.3))
ax.annotate('Java', xy=(0, 0),
            xytext=(1.7, 3.6))
ax.annotate('Python :-)', xy=(0, 0),
            xytext=(3.6, 17))
```