Einführung in Python

Rebecca Breu

Verteilte Systeme und Grid-Computing
JSC
Forschungszentrum Jülich

May 2011

Contents

Einführung in Python

Rebecca Breu

Verteilte Systeme und Grid-Computing
JSC
Forschungszentrum Jülich

May 2011

Contents — Part 1

Introduction

What is Python?

Python: dynamic programming language which supports several different programing paradigms:

- procedural programming
- object oriented programming
- functional programming

Standard: Python byte code is executed in the Python interpreter (similar to Java)
→ platform independent code

Why Python?

- syntax is clear, easy to read and learn (almost pseudo code)
- intuitive object oriented programming
- full modularity, hierarchical packages
- error handling via exceptions
- dynamic, high level data types
- comprehensive standard library for many tasks
- simply extendable via C/C++, wrapping of C/C++ libraries

Focus: programming speed

Is Python fast enough?

- for compute intensive algorithms: Fortran, C, C++ might be better
- for user programs: Python is fast enough!
- most parts of Python are written in C
- performance-critical parts can be re-implemented in C/C++ if necessary
- first analyse, then optimise!

Hello World!

```
#!/usr/bin/env python

# This is a commentary
print "Hello world!"
```

```
$ python hello_world.py
Hello world!
$
```

```
$ chmod 755 hello_world.py
$ ./hello_world.py
Hello world!
$
```

Hello User

```
#!/usr/bin/env python

name = raw_input("What's your name? ")
print "Hello", name
```

```
$ ./hello_user.py
What's your name? Rebecca
Hello Rebecca
$
```

Strong and Dynamic Typing

Strong Typing:

- Object is of exactly one type! A string is always a string, an integer always an integer
- Counter examples: PHP, JavaScript, C: `char` can be interpreted as `short`, `void *` can be everything

Dynamic Typing:

- no variable declaration
- variable names can be assigned to different data types in the course of a program
- An object's attributes are checked only at run time

Strong and Dynamic Typing

```
number = 3
print number, type(number)
print number + 42
number = "3"
print number, type(number)
print number + 42
```

```
3 <type 'int'>
45
3 <type 'str'>
Traceback (most recent call last):
  File "test.py", line 6, in ?
    print number + 42
TypeError: cannot concatenate 'str' and
'int' objects
```

Interactive Mode

The interpreter can be started in interactive mode:

```
$ python
Python 2.6 (r26:66714, Feb  3 2009, 20:52:03)
[GCC 4.3.2] on linux2
Type "help", "copyright", "credits" or ...
>>> print "hello world"
hello world
>>> a = 3 + 4
>>> print a
7
>>> 3 + 4
7
>>>
```

Documentation

Online help in the interpreter:

- `help()`: general Python help
- `help(obj)`: help regarding an object, e.g. a function or a module
- `dir()`: all used names
- `dir(obj)`: all attributes of an object

Official documentation: http://docs.python.org/

Documentation

```
>>> help(dir)
Help on built-in function dir:
...
>>> a = 3
>>> dir()
['__builtins__', '__doc__', '__file__',
'__name__', 'a']
>>> help(a)
Help on int object:
...
```

Data Types I

Numerical Data Types

- `int`: corresponds to `long` in C
- `long`: unlimited range of values
- `float`: corresponds to `double` in C
- `complex`: complex numbers

```
a = 1
b = 1L
c = 1.0; c = 1e0
d = 1 + 0j
```

Integers are automatically converted to `long` if necessary!

Operators on Numbers

- Basic arithmetics: `+`, `-`, `*`, `/`
- Div and modulo operator: `//`, `%`, `divmod(x, y)`
- Absolute value: `abs(x)`
- Rounding: `round(x)`
- Conversion: `int(x)`, `long(x)`, `float(x)`, `s complex(re [, im=0])`
- Conjugate of a complex number: `x.conjugate()`
- Power: `x ** y`, `pow(x, y)`

Result of a composition of different data types is of the "bigger" data type.

Strings

Data type: `str`

- `s = 'spam'`, `s = "spam"`
- Multiline strings: `s = """spam"""`
- No interpretation of escape sequences: `s = r"spam"`
- Generate strings from other data types: `str(1.0)`

```
>>> s = """hello
... world"""
>>> print s
hello
world
>>> print "sp\nam"
sp
am
>>> print r"sp\nam"  # or: print "sp\\nam"
sp\nam
```

String Methods

- Count appearance of substrings:
  `s.count(sub [, start[, end]])`
- Begins/ends with a substring?
  `s.startswith(sub[, start[, end]])`,
  `s.endswith(sub[, start[, end]])`
- All capital/lowercase letters: `s.upper()`, `s.lower()`
- Remove whitespace: `s.strip([chars])`
- Split at substring: `s.split([sub [,maxsplit]])`
- Find position of substring: `s.index(sub[, start[, end]])`
- Replace a substring: `s.replace(old, new[, count])`

More methods: `help(str)`, `dir(str)`

Lists

Data type: `list`

- `s = [1, "spam", 9.0, 42]`, `s = []`
- Append an element: `s.append(x)`
- Extend with a second list: `s.extend(s2)`
- Count appearance of an element: `s.count(x)`
- Position of an element: `s.index(x[, min[, max]])`
- Insert element at position: `s.insert(i, x)`
- Remove and return element at position: `s.pop([i])`
- Remove element: `s.remove(x)`
- Reverse list: `s.reverse()`
- Sort: `s.sort([cmp[, key[, reverse]]])`
- Sum of the elements: `sum(s)`

Operations on Sequences

Strings and lists have much in common: They are sequences.

- Does/doesn't s contain an element?
  `x in s`, `x not in s`
- Concatenate sequences: `s + t`
- Multiply sequences: `n * s`, `s * n`
- i-th element: `s[i]`, i-th to last element: `s[-i]`
- Subsequence: `s[i:j]`, with step size k: `s[i:j:k]`
- Subsequence from beginning/to end: `s[:-i]`, `s[i:]`, `s[:]`
- Length: `len(s)`
- smallest/largest element: `min(s)`, `max(s)`
- Assignments: `(a, b, c) = s`
  → `a = s[0]`, `b = s[1]`, `c = s[2]`

Sequences

- Another sequence: data type `tuple`: `a = (1, 2.0, "3")`
- List are mutable
- Strings and tuples are immutable
  - No assignment `s[i] = ...`
  - No appending and removing of elements
  - Functions like `upper` return a new string!

```
>>> s1 = "spam"
>>> s2 = s1.upper()
>>> s1
'spam'
>>> s2
'SPAM'
```

References

- In Python, everything is a reference to an object!
- Careful with assignments:

```
>>> s1 = [1, 2, 3, 4]
>>> s2 = s1
>>> s2[1] = 17
>>> s1
[1, 17, 3, 4]
>>> s2
[1, 17, 3, 4]
```

Flat copy of a list: `s2 = s1[:]` or `s2 = list(s1)`

Boolean Values

Data type bool: `True`, `False`
Values that are evaluated to `False`:

- `None`
- `False`
- `0` (in every numerical data type)
- empty strings, lists and tuples: `''`, `[]`, `()`
- empty dictionaries: {}
- empty sets

All other Objects of built-in data types are evaluated to `True`!

```
>>> bool([1, 2, 3])
True
>>> bool("")
False
```

Statements

The If Statement

```
if a == 3:
    print "Aha!"
```

- Blocks are defined by indentation!
- Standard: Indentation with four spaces

```
if a == 3:
    print "spam"
elif a == 10:
    print "eggs"
elif a == -3:
    print "bacon"
else:
    print "something else"
```

Relational Operators

- Comparison of content: ==, <, >, <=, >=, !=
- Comparison of object identity: `a is b`, `a is not b`
- And/or operator: `a and b`, `a or b`
- Negation: `not a`

```
if not (a==b) and (c<3):
    pass
```

For Loops

```
for i in range(10):
    print i    # 0, 1, 2, 3, ..., 9

for i in range(3, 10):
   print i     # 3, 4, 5, ..., 9

for i in range(0, 10, 2):
   print i    # 0, 2, 4, ..., 8
else:
   print "Loop completed."
```

- End loop prematurely: `break`
- Next iteration: `continue`
- `else` is executed when loop didn't end prematurely

Iterating directly over sequences (without using an index):

```
for item in ["spam", "eggs", "bacon"]:
    print item
```

The range function generates a list, too:

```
>>> range(0, 10, 2)
[0, 2, 4, 6, 8]
```

If indexes are necessary:

```
for (i, char) in enumerate("hello world"):
    print i, char
```

While Loops

```
while i < 10:
    i += 1
```

`break` and `continue` work for while loops, too.

Substitute for do-while loop:

```
while True:
    # important code
    if condition:
        break
```

Functions

Functions

```
def add(a, b):
    """Returns the sum of a and b."""

    mysum = a + b
    return mysum
```

```
>>> result = add(3, 5)
>>> print result
8
>>> help(add)
Help on function add in module __main__:

add(a, b)
    Returns the sum of a and b.
```

Return Values and Parameters

- Functions accept arbitrary objects as parameters and return values
- Types of parameters and return values are unspecified
- Functions without explicit return value return `None`

```
def hello_world():
    print "Hello World!"

a = hello_world()
print a
```

```
$ my_program.py
Hello World
None
```

Multiple Return Values

Multiple return values are realised using tuples or lists:

```
def foo():
    a = 17
    b = 42
    return (a, b)

ret = foo()
(x, y) = foo()
```

Keywords and Default Values

Parameters can be passed to a function in a different order than specified:

```
def foo(a, b, c):
    print a, b, c

foo(b=3, c=1, a="hello")
```

Defining default values:

```
def foo(a, b, c=1.3):
    print a, b, c

foo(1, 2)
foo(1, 17, 42)
```

Functions Are Objects

Functions are objects and as such can be assigned and passed on:

```
>>> a = float
>>> a(22)
22.0
```

```
>>> def foo(fkt):
...     print fkt(33)
...
>>> foo(float)
33.0
```

Input/Output

String Formatting

String formatting similar to C:

```
print "The answer is %i." % 42
s = "%s: %3.4f" % ("spam", 3.14)
```

- Integer decimal: d, i
- Integer octal: o
- Integer hexadecimal: x, X
- Float: f, F
- Float in exponential form: e, E, g, G
- Single character: c
- String: s

Use %% to output a single % character.

Command Line Input

User input:

```
user_input = raw_input("Type something: ")
```

Command line parameters:

```
import sys
print sys.argv
```

```
$ ./params.py spam
['params.py', 'spam']
```

Files

```
file1 = open("spam", "r")
file2 = open("/tmp/eggs", "wb")
```

- Read mode: r
- Write mode: w
- Handling binary files: b
- Read mode, appending to the end: a
- Read and write: r+

```
for line in file1:
    print line
```

Operations on Files

- read: `f.read([size])`
- Read a line: `f.readline()`
- Read multiple lines: `f.readlines([sizehint])`
- write: `f.write(str)`
- write multiple lines: `f.writelines(sequence)`
- Close file: `f.close()`

```
file1 = open("test", "w")
lines = ["spam\n", "eggs\n", "ham\n"]
file1.writelines(lines)
file1.close()
```

Python automatically converts `\n` into the correct line ending!

Modules and Packages

Importing Modules

Functions, classes and object thematically belonging together are grouped in modules.

```
import math
s = math.sin(math.pi)
```

```
import math as m
s = m.sin(m.pi)
```

```
from math import pi as PI, sin
s = sin(PI)
```

```
from math import *
s = sin(pi)
```

Modules

- Help: `dir(math)`, `help(math)`
- Modules are searched for in (see `sys.path`):
  - the directory of the running script
  - directories in the environment variable `PYTHONPATH`
  - installation-dependent directories

```
>>> import sys
>>> sys.path
['', '/usr/lib/python26.zip',
 '/usr/lib/python2.6',
 '/usr/lib/python2.6/site-packages', ...]
```

Importing Packages

Modules can be grouped into hierarchically structured packages.

```
import email
msg = email.mime.text.MIMEText("Hallo Welt!")
```

```
from email.mime import text as mtext
msg = mtext.MIMEText("Hallo Welt!")
```

```
from email.mime.text import MIMEText
msg = MIMEText("Hallo Welt!")
```

Own Modules

Every Python program can be imported as a module.

```
"""My first module: my_module.py"""

def add(a, b):
    """Add a and b."""
    return a + b

print add(2, 3)
```

```
>>> import my_module
5
>>> my_module.add(17, 42)
59
```

Top level instructions are executed during import!

Own Modules

If instructions should only be executed when running as a script, not importing it:

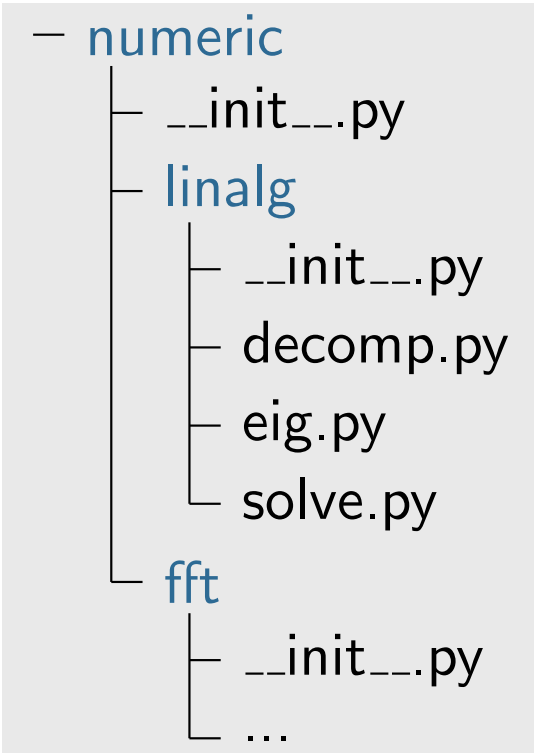
```
def add(a, b):
    return a + b

def main():
    print add(2, 3)

if __name__ == "__main__":
    main()
```

Useful e.g. for testing parts of the module.

Own Packages



- Packages are subdirectories
- In each package directory: `__init__.py` (may be empty)

```
import numeric
numeric.foo() #Aus __init__.py
numeric.linalg.eig.foo()
```

```
from numeric.linalg import eig
eig.foo()
```

Errors and Exceptions

Syntax Errors, Indentation Errors

Errors while parsing: Program doesn't get executed. E.g.:

- Mismatched or missing parenthesis
- Missing or misplaced semicolons, colons, commas
- Indentation errors

```
print "I'm running..."
def add(a, b)
    return a + b
```

```
$ ./add.py
  File "add.py", line 2
    def add(a, b)
                ^
SyntaxError: invalid syntax
```

Exceptions

Exceptions occur at runtime:

```
import math
print "I'm running..."
math.foo()
```

```
$ ./test.py
I'm running...
Traceback (most recent call last):
  File "test.py", line 3, in ?
    math.foo()
AttributeError: 'module' object has no
attribute 'foo'
```

Handling Exceptions

```
try:
    s = raw_input("Enter a number: ")
    number = float(s)
except ValueError:
    print "That's not a number!"
```

- `except` block is executed when the code in the code in the `try`-Block throws an according exception
- afterwards, the program continues normally
- unhandled exceptions force the program to exit.

Handling different kinds of exceptions:

```
except (ValueError, TypeError, NameError):
```

Handling Exceptions

```
try:
    s = raw_input("Enter a number: ")
    number = 1/float(s)
except ValueError:
    print "That's not a number!"
except ZeroDivisionError:
    print "You can't divide by zero!"
except:
    print "Oops, what's happened?"
```

- Several `except` statements for different exceptions
- Last `except` can be used without specifying the kind of exception: Catches all remaining exceptions
  - Careful: Can mask unintended programming errors!

Handling Exceptions

- `else` is executed if no exception occurred
- `finally` is executed in any case

```
try:
    f = open("spam")
except IOError:
    print "Cannot open file"
else:
    print f.read()
    f.close()
finally:
    print "End of try."
```

Exception Objects

Access to exception objects:

```
try:
    f = open("spam")
except IOError, e:
    print e.errno, e.strerror
    print e
```

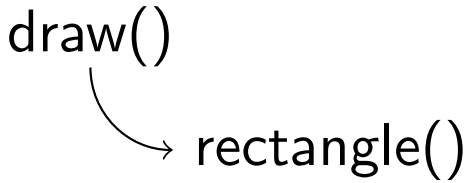
```
$ python test.py
2 No such file or directory
[Errno 2] No such file or directory: 'spam'
```

Exceptions in Function Calls

draw()

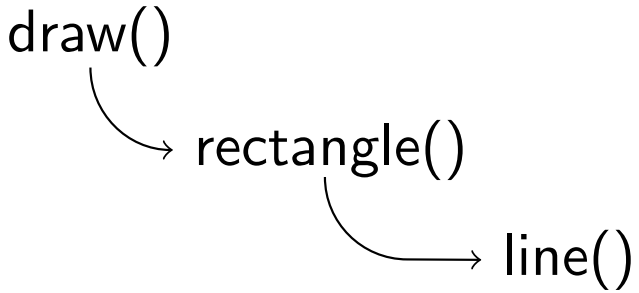
- Function calls another function.
- That function raises an exception.
- Is exception handled?
- No: Pass exception to calling function.

Exceptions in Function Calls



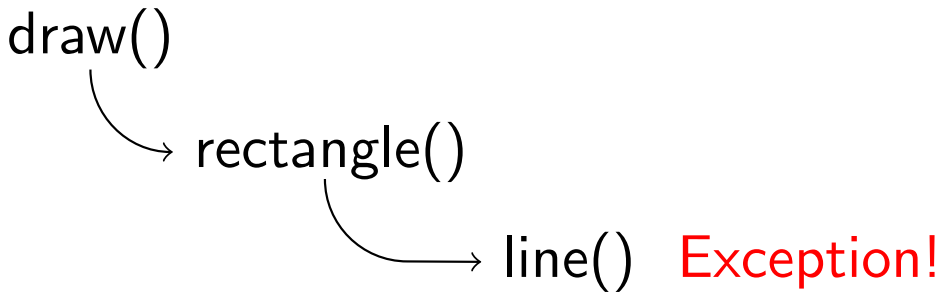
- Function calls another function.
- That function raises an exception.
- Is exception handled?
- No: Pass exception to calling function.

Exceptions in Function Calls



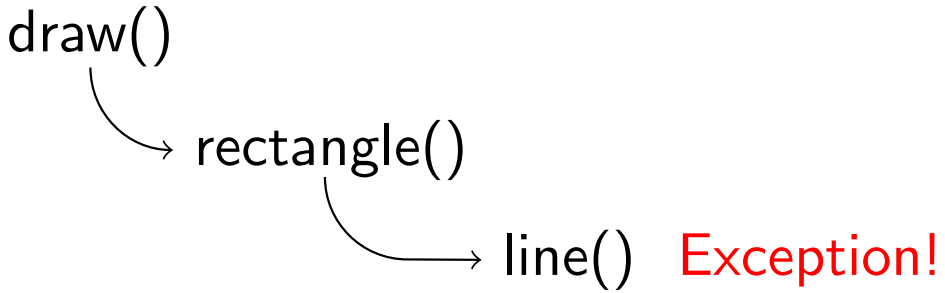
- Function calls another function.
- That function raises an exception.
- Is exception handled?
- No: Pass exception to calling function.

Exceptions in Function Calls



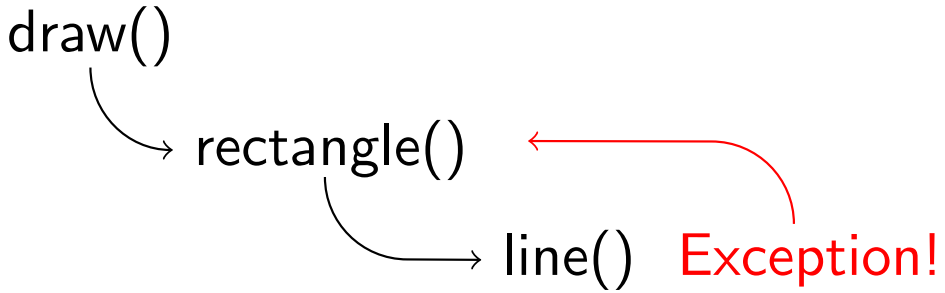
- Function calls another function.
- That function raises an exception.
- Is exception handled?
- No: Pass exception to calling function.

Exceptions in Function Calls



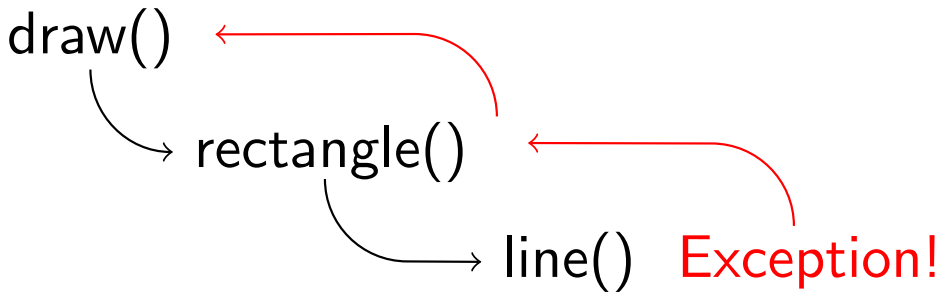
- Function calls another function.
- That function raises an exception.
- Is exception handled?
- No: Pass exception to calling function.

Exceptions in Function Calls



- Function calls another function.
- That function raises an exception.
- Is exception handled?
- No: Pass exception to calling function.

Exceptions in Function Calls



- Function calls another function.
- That function raises an exception.
- Is exception handled?
- No: Pass exception to calling function.

Raising Exceptions

Passing exceptions on:

```
try:
    f = open("spam")
except IOError:
    print "Problem while opening file!"
    raise
```

Raising exceptions:

```
def gauss_solver(matrix):
    # Important code
    raise ValueError("Singular matrix")
```

Exceptions vs. Checking Values Beforehand

Exceptions are preferable!

```
def square(x):
    if type(x) == int or type(x) == float:
        return x ** 2
    else:
        return None
```

Bad!

- What about other numerical data types (complex numbers, own data types)? Better: Try to compute the power and catch possible exceptions! → Duck-Typing
- Caller of a function might forget to check return values for validity. Better: Raise an exception!

Exceptions vs. Checking Values Beforehand

Exceptions are preferable!

```
def square(x):
    if type(x) == int or type(x) == float:
        return x ** 2
    else:
        return None
```

Bad!

- What about other numerical data types (complex numbers, own data types)? Better: Try to compute the power and catch possible exceptions! → Duck-Typing
- Caller of a function might forget to check return values for validity. Better: Raise an exception!

The `with` Statement

Some objects offer context management, which provides a more convenient way to write `try ... finally` blocks:

```
with open("test.txt") as f:
    for line in f:
        print line
```

After the `with` block the file object is guaranteed to be closed properly, no matter what exceptions occurred within the block.

In Python 2.5 this needs the following import:

```
from __future__ import with_statement
```

Viel Spaß mit

Einführung in Python

Rebecca Breu

Verteilte Systeme und Grid-Computing
JSC
Forschungszentrum Jülich

May 2011

Contents — Part 2

Data Types II

Sets

Set: unordered, no duplicated elements

- `s = set([sequence])`
- Subset: `s.issubset(t)`, `s <= t`, proper s.: `s < t`
- Superset: `s.issuperset(t)`, `s >= t`, proper s.: `s > t`
- Union: `s.union(t)`, `s | t`
- Intersection: `s.intersection(t)`, `s & t`
- Difference: `s.difference(t)`, `s - t`
- Symmetic Difference: `s.symmetric_difference(t)`, `s ^ t`
- Copy: `s.copy()`

As with secuences, the following works: `x in set`, `len(set)`, `for x in set`, `add`, `remove`

Dictionaries

Dictionary: Mapping of key → value

```
>>> d = { "spam": 1, "eggs": 17}
>>> d["eggs"]
17
>>> d["bacon"] = 42
>>> d
{'eggs': 17, 'bacon': 42, 'spam': 1}
```

Iterating over dictionaries:

```
for key in d:
    print key, d[key]
```

Operations on Dictionaries

- Delete an entry: `del`
- Delete all entries: `d.clear()`
- Copy: `d.copy()`
- Does it contain a key? `d.has_key(k)`, `k in d`
- List of all (key, value) tuples: `d.items()`
- List of all keys: `d.keys()`
- List all values: `d.values()`
- Get an entry: `d.get(k[, x])`
- Remove and return entry: `d.pop(k[, x])`
- Remove and return arbitrary entry: `d.popitem()`

Objektorientierte Programmierung

Objektorientierte Programmierung

- Bisher: prozedurale Programmierung
  - Daten
  - Funktionen, die Daten als Parameter entgegennehmen und Ergebnis zurückliefern
- Alternative: Fasse zusammengehörige Daten und Funktionen zusammen zu eigenen Datentypen
- $\rightarrow$ Erweiterung von Strukturen/Datenverbünden aus C/Fortran

Einfache Klassen als Structs verwenden

```
class Point:
    pass

p = Point()
p.x = 2.0
p.y = 3.3
```

- Klasse: Eigener Datentyp (hier: `Point`)
- Objekt: Instanz der Klasse (hier: `p`)
- Attribute (hier `x`, `y`) können dynamisch hinzugefügt werden

Klassen

```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

p = Point(2.0, 3.0)
print p.x, p.y
p.x = 2.5
p.z = 42
```

- `__init__`: Wird automatisch nach Erzeugung eines Objekts aufgerufen

Methoden auf Objekten

```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def norm(self):
        n = math.sqrt(self.x**2 + self.y**2)
        return n

p = Point(2.0, 3.0)
print p.x, p.y, p.norm()
```

- Methodenaufruf: automatisch das Objekt als erster Parameter
- $\rightarrow$ wird üblicherweise `self` genannt
- Achtung: Kein Überladen von Methoden möglich!

Objekte in Strings konvertieren

Standard-Rückgabe von `str(...)` für eigene Objekte:

```
>>> p = Point(2.0, 3.0)
>>> print p  # --> print str(p)
<__main__.Point instance at 0x402d7a8c>
```

Das kann man anpassen:

```
def __str__(self):
    return "(%i, %i)" % (self.x, self.y)
```

```
>>> print p
(2, 3)
```

Objekte in Strings konvertieren

Standard-Rückgabe von `str(...)` für eigene Objekte:

```
>>> p = Point(2.0, 3.0)
>>> print p  # --> print str(p)
<__main__.Point instance at 0x402d7a8c>
```

Das kann man anpassen:

```
def __str__(self):
    return "(%i, %i)" % (self.x, self.y)
```

```
>>> print p
(2, 3)
```

Objekte vergleichen

Standard: == prüft Objekte eigener Klassen auf Identität.

```
>>> p1 = Point(2.0, 3.0)
>>> p2 = Point(2.0, 3.0)
>>> p1 == p2
False
```

Das kann man anpassen:

```
def __eq__(self, other):
    return (self.x == other.x) and
           (self.y == other.y)
```

```
>>> p1 == p2 # Check for equal values
True
>>> p1 is p2 # Check for identity
False
```

Objekte vergleichen

Standard: == prüft Objekte eigener Klassen auf Identität.

```
>>> p1 = Point(2.0, 3.0)
>>> p2 = Point(2.0, 3.0)
>>> p1 == p2
False
```

Das kann man anpassen:

```
def __eq__(self, other):
    return (self.x == other.x) and
           (self.y == other.y)
```

```
>>> p1 == p2 # Check for equal values
True
>>> p1 is p2 # Check for identity
False
```

Objekte vergleichen

Weitere Vergleichsoperatoren:

- `<` : `__lt__(self, other)`
- `<=` : `__le__(self, other)`
- `!=` : `__ne__(self, other)`
- `>` : `__gt__(self, other)`
- `>=` : `__ge__(self, other)`

Alternativ: `__cmp__(self, other)`, gibt zurück:

- negativen Integer, wenn `self < other`
- null, wenn `self == other`
- positiven Integer, wenn `self > other`

Datentypen emulieren

Man kann mit Klassen vorhandene Datentypen emulieren:

- Zahlen: Rechenoperationen, `int(myobj)`, `float(myobj)`, ...
- Funktionen: `myobj(...)`
- Sequenzen: `len(myobj)`, `myobj[...]`, `x in myobj`, ...
- Iteratoren: `for i in myobj`

Siehe dazu Dokumentation:
http://docs.python.org/ref/specialnames.html

Klassenvariablen

Haben für alle Objekte einer Klasse stets den gleichen Wert:

```
class Point:
    count = 0   # Count all point objects
    def __init__(self, x, y):
        self.__class__.count += 1
        ...
```

```
>>> p1 = Point(2, 3); p2 = Point(3, 4)
>>> p1.count
2
>>> p2.count
2
>>> Point.count
2
```

Klassenmethoden und statische Methoden

```
class Spam:
    spam = "I don't like spam."

    @classmethod
    def cmethod(cls):
        print cls.spam

    @staticmethod
    def smethod():
        print "Blah blah."
```

```
Spam.cmethod()
Spam.smethod()
s = Spam()
s.cmethod()
s.smethod()
```

Vererbung

Oft hat man verschiedene Klassen, die einander ähneln.
Vererbung erlaubt:

- Hierarchische Klassenstruktur (Ist-ein-Beziehung)
- Wiederverwenden von ähnlichem Code

Beispiel: Verschiedene Telefon-Arten

- Telefon
- Handy (ist ein Telefon mit zusätzlichen Funktionen)
- SmartPhone (ist ein Handy mit zusätzlichen Funktionen)

Vererbung

```
class Phone:
    def call(self):
        pass

class MobilePhone(Phone):
    def send_text(self):
        pass
```

Handy erbt jetzt Methoden und Attribute von Telefon.

```
h = MobilePhone()
h.call() # inherited from Phone
h.send_text() # own method
```

Methoden überschreiben

In der abgeleiteten Klasse können die Methoden der Elternklasse überschrieben werden:

```
class MobilePhone(Telefon):
    def call(self):
        find_signal()
        Phone.call(self)
```

Mehrfachvererbung

Klassen können von mehreren Elternklassen erben. Bsp:

- SmartPhone ist ein Handy
- SmartPhone ist eine Kamera

```
class SmartPhone(MobilePhone, Camera)
    pass

h = SmartPhone()
h.call()  # inherited from MobilePhone
h.take_photo() # inherited from Camera
```

Attribute werden in folgender Reihenfolge gesucht:
SmartPhone, MobilePhone, Elterklasse von MobilePhone (rekursiv), Camera, Elternklasse von Camera (rekursiv).

Private Attribute

- In Python gibt es keine privaten Variablen oder Methoden.
- Konvention: Attribute, auf die nicht von außen zugegriffen werden sollte, beginnen mit einem Unterstrich: `_foo`.
- Um Namenskonflikte zu vermeiden: Namen der Form `__foo` werden durch `_klassenname__foo` ersetzt:

```
class Spam:
    __eggs = 3
```

```
>>> dir(Spam)
>>> ['_Spam__eggs', '__doc__', '__module__']
```

Properties

Sollen beim Zugriff auf eine Variable noch Berechnungen oder Überprüfungen durchgeführt werden: Getter und Setter

```
class Spam(object):
    def __init__(self):
        self._value = 0

    def get_value(self):
        return self._value

    def set_value(self, value):
        if value <= 0:  self._value = 0
        else:  self._value = value

    value = property(get_value, set_value)
```

Properties

Auf Properties wird wie auf gewöhnliche Attribute zugegriffen:

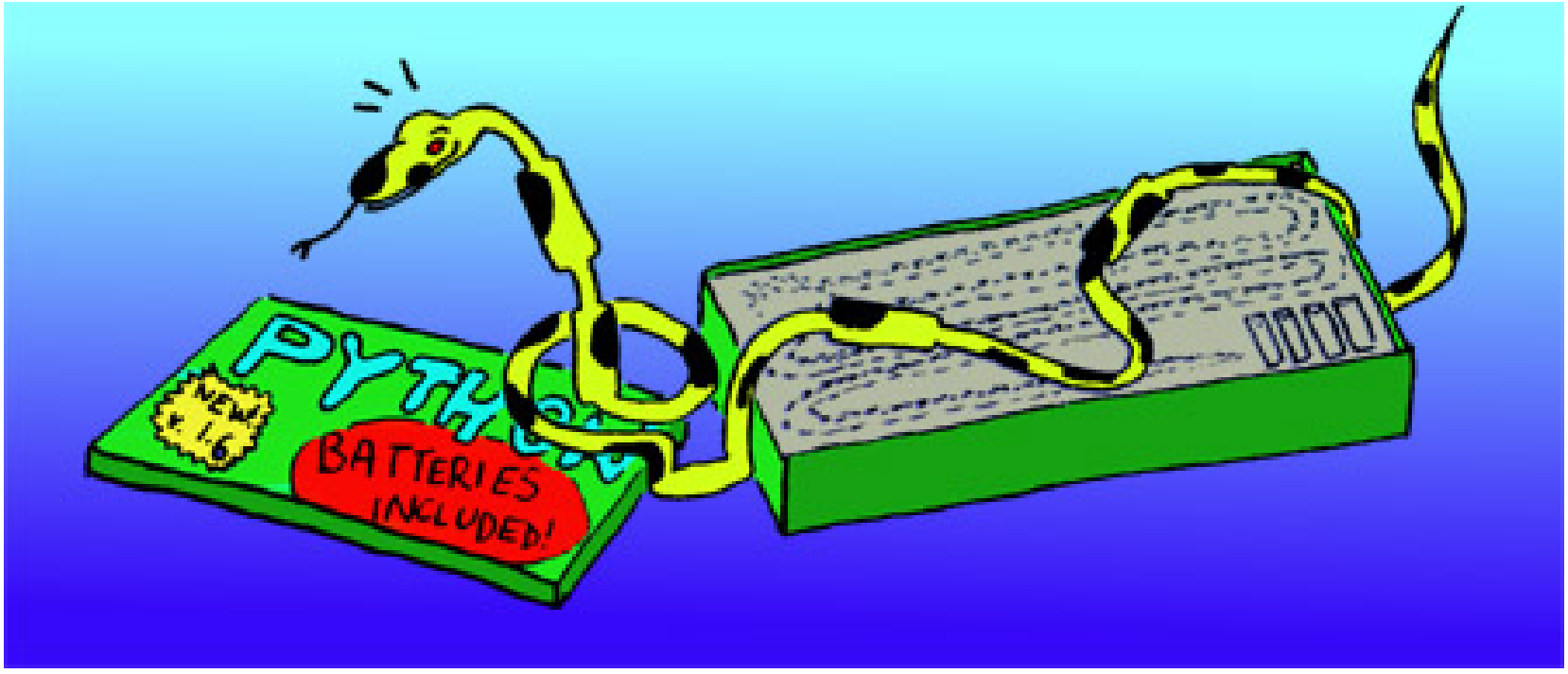
```
>>> s = Spam()
>>> s.value = 6    # set_value(6)
>>> s.value        # get_value()
>>> 6
>>> s.value = -6   # set_value(-6)
>>> s.value        # get_value()
>>> 0
```

- Getter und Setter können nachträglich hinzugefügt werden, ohne die API zu verändern.
- Zugriff auf `_value` immer noch möglich

Pythons Standardbibliothek

Pythons Standardbibliothek

„Batteries included“: umfassende Standardbibliothek für die verschiedensten Aufgaben



Mathematik: `math`

- Konstanten: `e`, `pi`
- Auf- und Abrunden: `floor(x)`, `ceil(x)`
- Exponentialfunktion: `exp(x)`
- Logarithmus: `log(x[, base])`, `log10(x)`
- Potenz und Quadratwurzel: `pow(x, y)`, `sqrt(x)`
- Trigonometrische Funktionen: `sin(x)`, `cos(x)`, `tan(x)`
- Kovertierung Winkel $\leftrightarrow$ Radiant: `degrees(x)`, `radians(x)`

```
>>> import math
>>> math.sin(math.pi)
1.2246063538223773e-16
>>> math.cos(math.radians(30))
0.8660254037844387
```

Zufall: `random`

- Zufällige Integers:
  `randint(a, b)`, `randrange([start,] stop[, step])`
- Zufällige Floats (Gleichverteilg.): `random()`, `uniform(a, b)`
- Andere Verteilungen: `expovariate(lambd)`,
  `gammavariate(alpha, beta)`, `gauss(mu, sigma)`, ...
- Zufälliges Element einer Sequenz: `choice(seq)`
- Mehrere eindeutige, zufällige Elemente einer Sequenz:
  `sample(population, k)`
- Sequenz mischen: `shuffle(seq[, random])`

```
>>> s = [1, 2, 3, 4, 5]
>>> random.shuffle(s)
>>> s
[2, 5, 4, 3, 1]
>>> random.choice("Hello world!")
'e'
```

Exakte Dezimalzahlen: `fractions`

```
>>> from fractions import Fraction
>>> a = Fraction(2, 3)
>>> b = Fraction(2, 5)
>>> print float(a), float(b)
0.6666666666666663 0.4000000000000002
>>> a+b
Fraction(16, 15)
>>> a/b
Fraction(5, 3)
```

(Siehe auch: Modul `decimal`.)

Datum und Zeit: datetime

Zeit- und Datumsangaben:

```
d1 = datetime.date(2008, 3, 21)
d2 = datetime.date(2008, 6, 22)
dt = datetime.datetime(2011, 8, 26, 12, 30)
t = datetime.time(12, 30)
```

Mit Zeit und Datum rechnen:

```
print d1 < d2
delta = d2 - d1
print delta.days
print d2 + datetime.timedelta(days=44)
```

Weitere Datentypen: `collections`

defaulttict: Dictionary, welches Defaultwerte für nicht vorhandene Schlüsselwörter erzeugen kann:

```
haeufigkeiten = defaultdict(lambda: 0)

for c in "Hallo Welt!":
    haeufigkeiten[c] += 1
```

Paramter (optional): Funktion zum Erzeugen der Defaultwerte

Operationen auf Verzeichnisnamen: os.path

- Pfade: `abspath(path)`, `basename(path)`, `normpath(path)`, `realpath(path)`
- Pfad zusammensetzen: `join(path1[, path2[, ...]])`
- Pfade aufspalten: `split(path)`, `splitext(path)`
- Datei-Informationen: `isfile(path)`, `isdir(path)`, `islink(path)`, `getsize(path)`, ...
- Home-Verzeichnis vervollständigen: `expanduser(path)`
- Umgebungsvariablen vervollständigen: `expandvars(path)`

```
>>> os.path.join("spam", "eggs", "ham.txt")
'spam/eggs/ham.txt'
>>> os.path.splitext("spam/eggs.py")
('spam/eggs', '.py')
>>> os.path.expanduser("~/spam")
'/home/rbreu/spam'
>>> os.path.expandvars("/blah/$TEST")
'/bla/test.py'
```

Dateien und Verzeichnisse: os

- Working directory: `getcwd()`, `chdir(path)`
- Dateirechte ändern: `chmod(path, mode)`
- Besitzer ändern: `chown(path, uid, gid)`
- Verzeichnis erstellen: `mkdir(path[, mode])`, `makedirs(path[, mode])`
- Dateien löschen: `remove(path)`, `removedirs(path)`
- Dateien umbenennen: `rename(src, dst)`, `renames(old, new)`
- Liste von Dateien in Verzeichnis: `listdir(path)`

```
for myfile in os.listdir("mydir"):
    os.chmod(os.path.join("mydir", myfile),
             stat.S_IRUSR|stat.S_IWUSR)
```

Verzeichnislisting: `glob`

Liste von Dateien in Verzeichnis, mit Unix-artiger Wildcard-Vervollständigung: `glob(path)`

```
>>> glob.glob("python/[a-c]*.py")
['python/confitest.py',
 'python/basics.py',
 'python/curses_test2.py',
 'python/curses_keys.py',
 'python/cmp.py',
 'python/button_test.py',
 'python/argument.py',
 'python/curses_test.py']
```

Dateien und Verzeichnisse: `shutil`

Higher Level-Operationen auf Dateien und Verzeichnissen.

- Datei kopieren: `copyfile(src, dst)`, `copy(src, dst)`
- Rekursiv kopieren; `copytree(src, dst[, symlinks])`
- Rekursiv löschen:
  `rmtree(path[, ignore_errors[, onerror]])`
- Rekursiv verschieben: `move(src, dst)`

```
shutil.copytree("spam/eggs", "../beans",
                symlinks=True)
```

Andere Prozesse starten: subprocess

Einfaches Ausführen eines Programmes:

```
p = subprocess.Popen(["ls", "-l", "mydir"])
returncode = p.wait() # wait for p to end
```

Zugriff auf die Ausgabe eines Programmes:

```
p = Popen(["ls"], stdout=PIPE, stderr=STDOUT)
p.wait()
output = p.stdout.read()
```

Pipes zwischen Prozessen (`ls -l | grep txt`)

```
p1 = Popen(["ls", "-l"], stdout=PIPE)
p2 = Popen(["grep", "txt"], stdin=p1.stdout)
```

Zugriff auf Kommandozeilenparameter: optparse

- Einfach: Liste mit Parametern → `sys.argv`
- Komfortabler für mehrere Optionen: `OptionParser`

```
parser = optparse.OptionParser()
parser.add_option("-f", "--file",
                  dest="filename",
                  default="out.txt",
                  help="output file")
parser.add_option("-v", "--verbose",
                  action="store_true",
                  dest="verbose",
                  default=False,
                  help="verbose output")

(options, args) = parser.parse_args()
print options.filename, options.verbose
print args
```

Zugriff auf Kommandozeilenparameter: optparse

So wird ein optparse-Programm verwendet:

```
$ ./test.py -h
usage: test.py [options]

options:
  -h, --help            show this help message and exit
  -f FILENAME, --file=FILENAME
                        output file
  -v, --verbose         verbose output
```

```
$ ./test.py -f aa bb cc
aa False
['bb', 'cc']
```

Konfigurationsdateien: `ConfigParser`

Einfaches Format zum Speichern von Konfigurationen u.Ä.:
Windows INI-Format

```
[font]
font = Times New Roman
# comment (or: ! as comment symbol)
size = 16

[colors]
font = black
pointer = %(font)s
background = white
```

Konfigurationsdateien: ConfigParser

Config-Datei lesen:

```
parser = ConfigParser.SafeConfigParser()
parser.readfp(open("config.ini", "r"))
print parser.get("colors", "font")
```

Weitere Parser-Methoden:

- Liste aller Sections: `sections()`
- Liste aller Optionen: `options(section)`
- Liste aller Optionen und Werte: `items(section)`
- Werte lesen: `get(sect, opt)`, `getint(sect, opt)`, `getfloat(sect, opt)`, `getboolean(sect, opt)`

Konfigurationsdateien: ConfigParser

Config-Datei schreiben:

```
parser = ConfigParser.SafeConfigParser()
parser.add_section("colors")
parser.set("colors", "font", "black")
parser.write(open("config.ini", "w"))
```

Weitere Parser-Methoden:

- Section hinzufügen: `add_section(section)`
- Section löschen: `remove_section(section)`
- Option hinzufügen: `set(section, option, value)`
- Option entfernen: `remove_option(section, option)`

CSV-Dateien: csv

CSV: Comma-seperated values

- Tabellendaten im ASCII-Format
- Spalten durch ein festgelegtes Zeichen (meist Komma) getrennt

```
reader = csv.reader(open("test.csv", "rb"))
for row in reader:
    for item in row:
        print item
```

```
writer = csv.writer(open(outfile, "wb"))
writer.writerow([1, 2, 3, 4])
```

CSV-Dateien: csv

Mit verschiedenen Formaten (Dialekten) umgehen:

```
reader(csvfile, dialect='excel') # Default
writer(csvfile, dialect='excel_tab')
```

Einzelne Formatparameter angeben:

```
reader(csvfile, delimiter=";")
```

Weitere Formatparameter: `lineterminator`, `quotechar`, `skipinitialspace`, ...

Objekte serialisieren: pickle

Beliebige Objekte in Dateien speichern:

```
obj = {"hello": "world", "spam":1}
pickle.dump(obj, open("blah.bin", "wb"))
# ...
obj = pickle.load(open("blah.bin", "rb"))
```

Objekt in String unwandeln (z.B. zum Verschicken über Streams):

```
s = pickle.dumps(obj)
# ...
obj = pickle.loads(s)
```

Persistente Dictionaries: `shelve`

Ein Shelve benutzt man wie ein Dictionary, es speichert seinen Inhalt in eine Datei.

```
d = shelve.open("blah")
d["spam"] = "eggs"
d["blah"] = 1
del d["foo"]
d.close()
```

Leichtgewichtige Datenbank: sqlite3

Datenbank in Datei oder im Memory, ab Python 2.5 in der stdlib.

```
conn = sqlite3.connect("bla.db")
c = conn.cursor()

c.execute("""CREATE TABLE Friends
             (firstname TEXT, lastname TEXT)""")
c.execute("""INSERT INTO Friends
             VALUES("Jane", "Doe")""")
conn.commit()
```

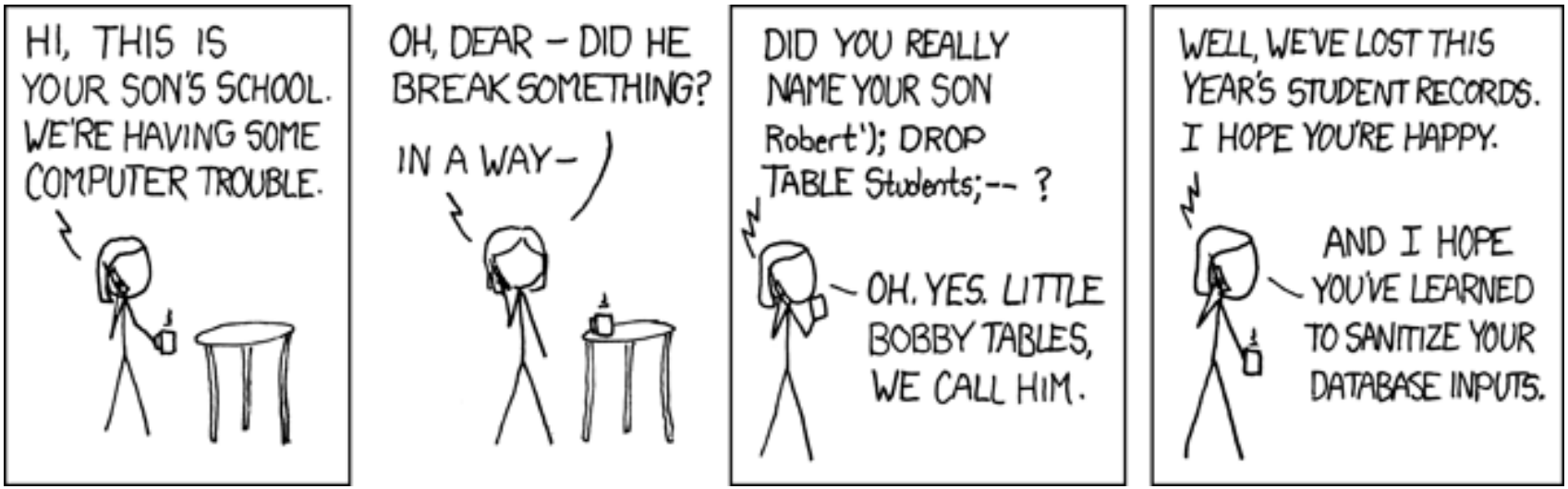
```
c.execute("""SELECT * FROM Friends""")
for row in c: print row

c.close(); conn.close()
```

Leichtgewichtige Datenbank: `sqlite3`

String-Formatter sind unsicher, da beliebiger SQL-Code eingeschleust werden kann!

```
# Never do this!
symbol = "Jane"
c.execute("... WHERE firstname = '%s'" % symbol)
```



Leichtgewichtige Datenbank: sqlite3

Stattdessen die Platzhalter der Datenbank-API benutzen:

```
c.execute("... WHERE name = ?", symbol)
```

```
f = (("Janis", "Joplin"), ("Bob", "Dylan"))
for item in friends:
    c.execute("""INSERT INTO Friends
                 VALUES (?,?)""", item)
```

Tar-Archive: `tarfile`

Ein tgz entpacken:

```
tar = tarfile.open("spam.tgz")
tar.extractall()
tar.close()
```

Ein tgz erstellen:

```
tar = tarfile.open("spam.tgz", "w:gz")
tar.add("/home/rbreu/test")
tar.close()
```

Log-Ausgaben: `logging`

Flexible Ausgabe von Informationen, kann schnell angepasst werden.

```
import logging
logging.debug("Very special information.")
logging.info("I am doing this and that.")
logging.warning("You should know this.")
```

```
WARNING:root:You should know this.
```

- Messages bekommen einen Rang (Dringlichkeit): `CRITICAL`, `ERROR`, `WARNING`, `INFO`, `DEBUG`
- Default: Nur Messages mit Rang `WARNING` oder höher werden ausgegeben

Log-Ausgaben: logging

Beispiel: Ausgabe in Datei, benutzerdefiniertes Format, feineres Log-Level:

```
logging.basicConfig(level=logging.DEBUG,
  format="%(asctime)s %(levelname)-8s %(message)s",
  datefmt="%Y-%m-%d %H:%M:%S",
  filename='/tmp/spam.log', filemode='w')
```

```
$ cat /tmp/spam.log
2007-05-07 16:25:14 DEBUG    Very special information.
2007-05-07 16:25:14 INFO     I am doing this and that.
2007-05-07 16:25:14 WARNING  You should know this.
```

Es können auch verschiedene Loginstanzen gleichzeitig benutzt werden, siehe Python-Dokumentation.

Reguläre Ausdrücke: `re`

Einfaches Suchen nach Mustern:

```
>>> re.findall(r"\[.*?\]", "a[bc]g[hal]def")
['[bc]', '[hal]']
```

Ersetzen von Mustern:

```
>>> re.sub(r"\[.*?\]", "!", "a[bc]g[hal]def")
'a!g!def'
```

Wird ein Regex-Muster mehrfach verwendet, sollte es aus Geschwindigkeitsgründen compiliert werden:

```
>>> pattern = re.compile(r"\[.*?\]")
>>> pattern.findall("a[bc]g[hal]def")
['[bc]', '[hal]']
```

Reguläre Ausdrücke: `re`

Umgang mit Gruppen:

```
>>> re.findall("(\[.*?\])|(<.*?>)",
               "[hi]s<b>sdd<hal>")
[('[hi]', ''), ('', '<b>'), ('', '<hal>')]
```

Flags, die das Verhalten des Matching beeinflussen:

```
>>> re.findall("^a", "abc\nAbc", re.I|re.M)
>>> ['a', 'A']
```

- `re.I`: Groß-/Kleinschreibung ingnorieren
- `re.M`: `^` bzw. `$` matchen am Anfang/Ende jeder Zeile (nicht nur am Anfang des Strings)
- `re.S`: `.` matcht auch Zeilenumbruch

URLs lesen: urllib2

Einfaches lesen:

```
import urllib2
r = urllib2.urlopen('http://www.fz-juelich.de')
print r.read()
print r.headers["content-type"]
```

Man kann den Request vorm Absenden anpassen:

```
opener = urllib2.build_opener()
opener.addheaders = [('User-agent',
                      'Mozilla/5.0')]
r = opener.open('http://www.fz-juelich.de')
```

Es werden Cookies, Authentifizierung, Proxies etc unterstützt.

XML-RPC-Client: `xmlrpclib`

- XML-RPC: Remote Procedure Call via XML und HTTP
- unabhänging von Plattform und Programmiersprache

```
import xmlrpclib

s = xmlrpclib.Server("http://localhost:8000")
print s.add(2,3)
print s.sub(5,2)
```

Konvertierungen für die gängigen Datentypen geschehen automatisch: Booleans, Integer, Floats, Strings, Tupel, Listen, Dictionaries mit Strings als Keys, ...

XML-RPC-Server: SimpleXMLRPCServer

```
from SimpleXMLRPCServer import SimpleXMLRPCServer

# Methoden, die der Server zur Verfuegung
# stellen soll:
class MyFuncs:
    def add(self, x, y):
        return x + y
    def sub(self, x, y):
        return x - y

# Erstelle und starte Server:
server = SimpleXMLRPCServer(("localhost", 8000))
server.register_instance(MyFuncs())
server.serve_forever()
```

Viel Spaß mit

Einführung in Python

Rebecca Breu

Verteilte Systeme und Grid-Computing
JSC
Forschungszentrum Jülich

May 2011

Contents — Part 3

Fortgeschrittene Techniken

Conditional Expressions

Kurze Schreibweise für bedingte Zuweisung. Statt:

```
if zahl<0:
    s = "Negativ"
else:
    s = "Positiv"
```

kann man schreiben:

```
s = "Negativ" if zahl<0 else "Positiv"
```

Funktionsparameter aus Listen und Dictionaries

```
def spam(a, b, c, d):
    print a, b, c, d
```

Man kann positionale Parameter aus Listen erzeugen:

```
>>> args = [3, 6, 2, 3]
>>> spam(*args)
3 6 2 3
```

Man kann Keyword-Paramter aus Dictionaries erzeugen:

```
>>> kwargs = {"c": 5, "a": 2, "b": 4, "d":1}
>>> spam(**kwargs)
2 4 5 1
```

Funktionen mit beliebigen Parametern

```
def spam(*args, **kwargs):
    for i in args:
        print i
    for i in kwargs:
        print i, kwargs[i]
```

```
>>> spam(1, 2, c=3, d=4)
1
2
c 3
d 4
```

List Comprehension

Abkürzende Schreibweise zum Erstellen von Listen aus for-Schleifen. Statt:

```
a = []
for i in range(10):
    a.append(i**2)
```

kann man schreiben:

```
a = [i**2 for i in range(10)]
```

Mit Bedingung:

```
a = [i**2 for i in range(10) if i != 4]
```

Anonyme Funktionen: Lambda

```
>>> f = lambda x, y: x + y
>>> f(2, 3)
5
>>> (lambda x: x**2)(3)
9
```

Nützlich, wenn einfache Funktionen als Parameter übergeben werden sollen.

```
l = ["alice", "Bob"]
l.sort()
l.sort(lambda a,b: cmp(a.upper(), b.upper()))
```

Map

Anwenden einer Funktion auf alle Elemente einer Liste:

```
>>> li = [1, 4, 81, 9]
>>> map(math.sqrt, li)
[1.0, 2.0, 9.0, 3.0]
>>> map(lambda x: x * 2, li)
[2, 8, 162, 18]
```

Wenn die Funktion mehr als einen Parameter nimmt, kann je zusätzlichem Parameter eine weitere Liste übergeben werden:

```
>>> map(math.pow, li, [1, 2, 3, 4])
[1.0, 16.0, 531441.0, 6561.0]
```

Filter

Wie Map, jedoch enthält die Egebnisliste nur die Elemente, welche wahr sind:

```
>>> li = [1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> filter(lambda x: x % 2, li)
[1, 3, 5, 7, 9]
```

Zip

Zusammenfügen mehrer Sequenzen zu einer Liste von Tupeln:

```
>>> zip("ABC", "123")
[('A', '1'), ('B', '2'), ('C', '3')]
>>> zip([1, 2, 3], "ABC", "XYZ")
[(1, 'A', 'X'), (2, 'B', 'Y'), (3, 'C', 'Z')]
```

Nützlich, wenn man über mehrere Sequenzen parallel iterieren möchte

Iteratoren

Was passiert, wenn `for` auf einem Objekt aufgerufen wird?

```
for i in obj:
    pass
```

- Auf `obj` wird die `__iter__`-Methode aufgerufen, welche einen Iterator zurückgibt
- Auf dem Iterator wird bei jedem Durchlauf `next()` aufgerufen
- Eine `StopIteration`-Ausnahme beendet die for-Schleife

Iteratoren

```
class Reverse:
    def __init__(self, data):
        self.data = data
        self.index = len(data)
    def __iter__(self):
        return self
    def next(self):
        if self.index == 0:
            raise StopIteration
        self.index = self.index - 1
        return self.data[self.index]
```

```
>>> for char in Reverse("spam"):
...     print char,
...
m a p s
```

Generatoren

Einfache Weise, Iteratoren zu erzeugen:

- Werden wie Funktionen definiert
- `yield`-Statement, um Daten zurückzugeben und beim nächsten `next`-Aufruf dort weiterzumachen

```
def reverse(data):
    for element in data[::-1]:
        yield element
```

```
>>> for char in reverse("spam"):
...     print char,
...
m a p s
```

Generator-Audrücke

Ähnlich zu List Comprehensions kann man anonyme Iteratoren erzeugen:

```
>>> data = "spam"
>>> for c in (elem for elem in data[::-1]):
...     print c,
...
m a p s
```

Dynamische Attribute

Erinnerung: Man kann Attribute von Objekten zur Laufzeit hinzufügen:

```
class Empty:
    pass

a = Empty()
a.spam = 42
a.eggs = 17
```

Und entfernen:

```
del a.spam
```

getattr, setattr

Man kann Attribute von Objekten als Strings ansprechen:

```
import math
f = getattr(math, "sin")
print f(x) # sin(x)
```

```
a = Empty()
setattr(a, "spam", 42)
print a.spam
```

Nützlich, wenn man z.B. Attributnamen aus User-Input oder Dateien liest.

Neues in Python 2.7

Neues in Python 2.7

- Neue Syntax für Mengen: Statt `set(1, 2, 3)` kann man schreiben: `{1,2,3}`
- Analog zu List Comprehensions gibt es Set Comprehensions und Dictionary Comprehensions zum schnellen Erzeugen:

```
s = {i*2 for in in range(3)}
d = {i: i*2 for i in range(3)}
```

- `collections.OrderedDict`: Beim normalen Dictionary haben die Einträge eine willkürliche Reihenfolge. OrderedDict kann die Einträge sortiert hinzufügen.

wxPython

Grafische Benutzeroberflächen (GUIs)

Verbreitete GUI-Toolkits mit Bindings für (u.A.) Python:

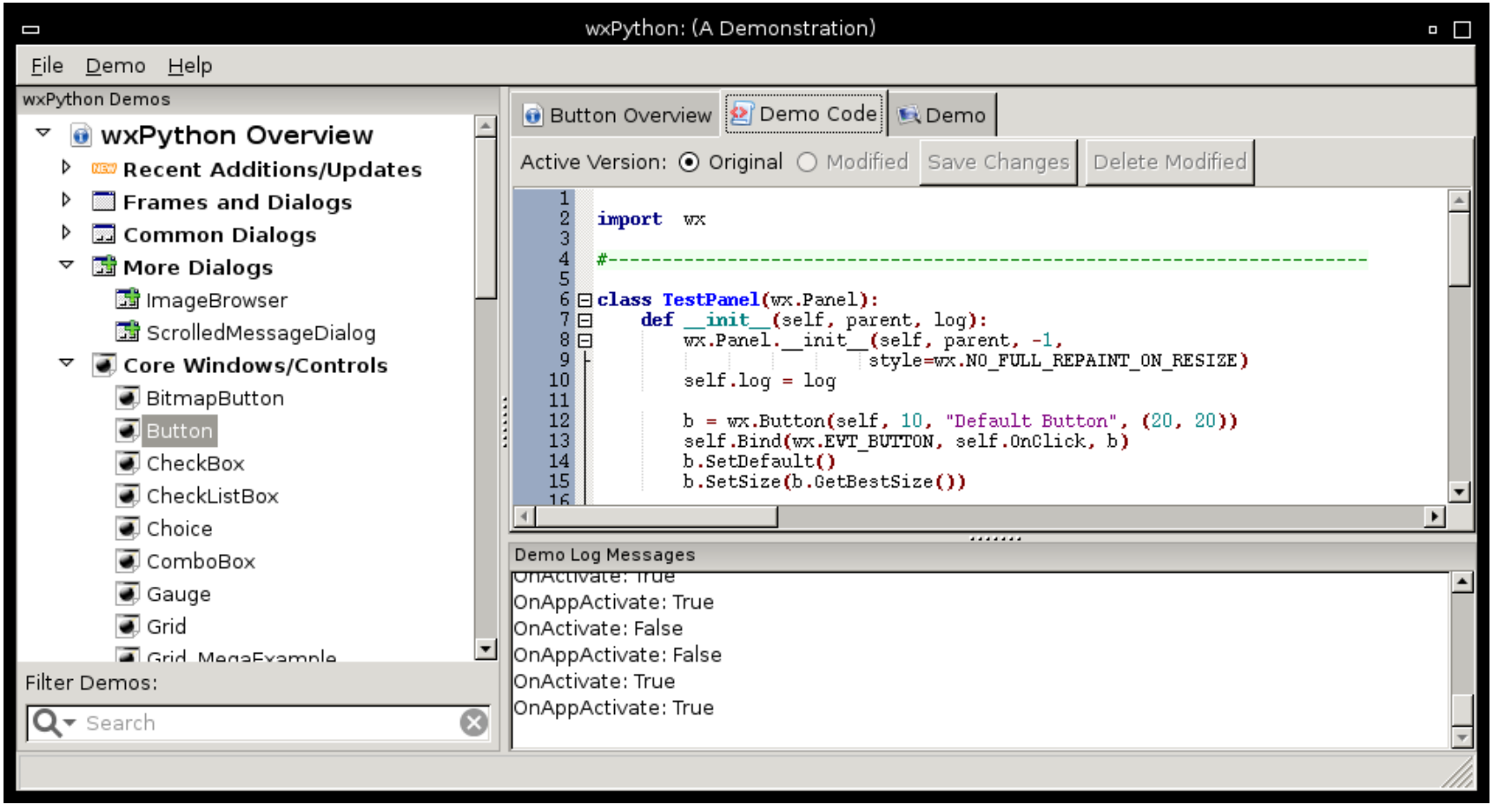
- Tk: In Pythons Standardbibliothek, simpel (ungeeignet für komplexe Anwendungen), veraltetes Aussehen
- GTK: z.B. Gnome Desktop, GIMP, Eclipse, ...
- QT: KDE Desktop, Skype, Scribus, ...

Alle werden auf den gängingen Betriebssystemen unterstützt.

- wxWidgets: Benutzt Windows-, Mac OS-Bibliotheken oder GTK $\rightarrow$ Look and Feel des jeweiligen Betriebssystems

Die wxPython-Demo

`/usr/share/doc/wx2.8-examples/examples/wxPython/demo.py`



Kurze Beschreibung aller Features mit Live-Demo und Beispiel-Code

Hello World

```
import wx

class MainFrame(wx.Frame):
    def __init__(self):
        wx.Frame.__init__(self, parent=None,
                          title="Hello World")
        self.Show(True)

app = wx.PySimpleApp()
frame = MainFrame()
app.MainLoop()
```

Erzeugt ein leeres Fenster mit Titel „Hello World“.

Die Basis: Application und Top Level Windows

Application:

- Kern eines wx-Programms, betreibt die Hauptschleife
- Hauptschleife verarbeitet alle Events (Mausbewegung, Tastaturanschlag, ...)
- `PySimpleApp`: Für einfache Anwendungen mit nur einem Top Level Window

Zur Application gehört mindestens ein Top Level Window:

- Präsentiert dem Anwender die wichtigsten Daten und Kontrollelemente
- Wird das letzte Top Level Window geschlossen, beendet sich die Application (die Hauptschleife wird verlassen)

Das allgemeinste Widget: wx.Frame

```
wx.Frame(parent, id=-1, title="",
         pos=(-1,-1), size=(-1,-1), ...)
```

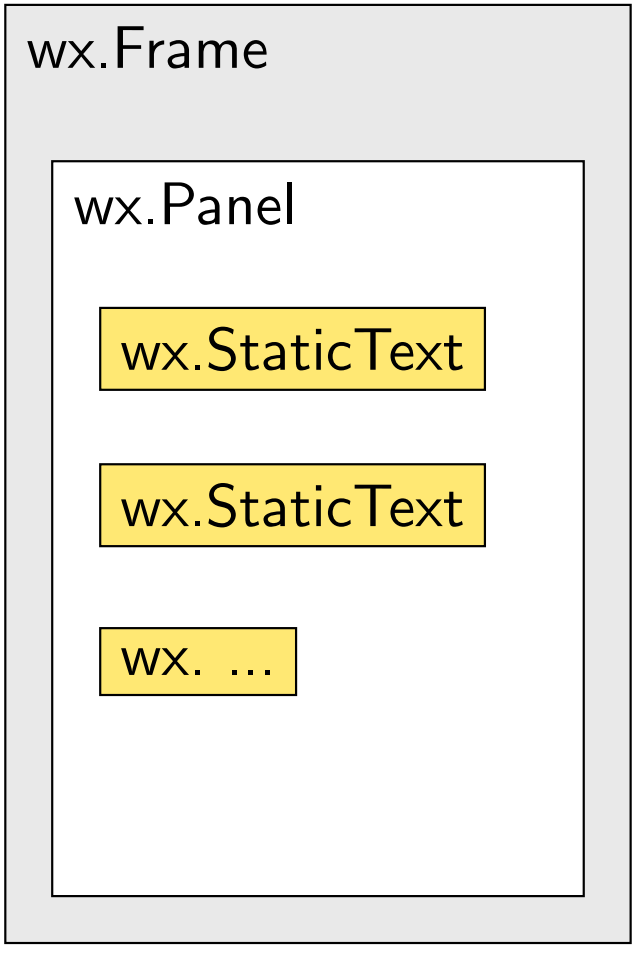
- parent: Ist `None` für Top Level Windows
- id: Integer; Automatische Generierung mit -1 (zu bevorzugen)
- title: Fenstertitel, wird in Titelleiste angezeigt
- pos: Integer-Tupel (x, y); (-1, -1) lässt das unterliegende System entscheiden
- size: Integer-Tupel (width, height); (-1, -1) lässt das unterliegende System entscheiden

Widgets in ein Frame einfügen

Etwas Text in unserem Fenster:

```
class MainFrame(wx.Frame):
    def __init__(self):
        wx.Frame.__init__(self, parent=None,
                          title="Hello World")
        panel = wx.Panel(parent=self)
        wx.StaticText(parent=panel,
                      label="How are you?")
        self.Show(True)
```

Widgets in ein Frame einfügen



- Panel: Container, welcher beliebig viele weitere Widgets enthalten kann.
- Parent-Beziehungen legen fest, welches Widget in welchem Widget dargestellt wird

Nicht-editierbarer Text: StaticText

```
wx.StaticText(parent, id=-1, label="",
              pos=(-1,-1), size=(-1,-1),
              style=0, ...)
```

- label: Der darzustellende Text
- pos bezieht sich auf die Position innerhalb des Parent-Widgets
- style: `wx.ALIGN_CENTER`, `wx.ALIGN_LEFT`, `wx.ALIGN_RIGHT`
- Auch mehrzeiliger Text möglich
- Einige Methoden:
  - `SetLabel`: Text nachträglich ändern
  - `SetForegroundColour`, `SetBackgroundColour`
  - `SetFont`

Auf Benutzeraktionen reagieren

```
class MainFrame(wx.Frame):
    def __init__(self):
        wx.Frame.__init__(parent=None)
        panel = wx.Panel(parent=self)
        button = wx.Button(parent=panel,
                           label="&Click me")
        self.Bind(wx.EVT_BUTTON, self.on_button,
                  button)
        self.Show(True)

    def on_button(self, evt):
        print "You pressed the button!"
```

Der Button kann mit Alt+C „geclickt“ werden (wg. &C...)

Ereignisgesteuerte Programmierung

- Herkömmliche Programme laufen linear ab
- GUI-Programme: Anwender kann Bedienelemente zu beliebiger Zeit in beliebiger Reihenfolge bedienen
- GUI-Programm *reagiert* auf den Anwender
- → Hauptschleife wartet auf Events und leitet diese an passende Event-Handler weiter

```
self.Bind(wx.EVT_BUTTON, self.on_button,
          button)
```

`MainFrame` soll alle Button-Events vom Widget `button` mit der Methode `onButton` behandeln.

Events und die Widget-Hierarchie

```
class MainFrame(wx.Frame):
    def __init__(self):
        ...
        self.Bind(wx.EVT_BUTTON,
                  self.on_buttonF, button)
        button.Bind(wx.EVT_BUTTON,
                    self.on_buttonB, button)

    def on_buttonF(self, evt):
        print "You pressed the button!"

    def on_buttonB(self, evt):
        print "You pressed me!"
```

Events und die Widget-Hierarchie

- Widget generiert Event
- Hat das Widget passenden Event-Handler?
  - ja: behandle Event
  - nein: Sende Event an das Parent-Widget
- Hat das Parent-Event passenden Event-Handler? ...

→ Nur `onButtonB` wird ausgeführt!

Behandeltes Event weiter propagieren mit Skip:

```
def on_buttonB(self, evt):
    print "You pressed me!"
    evt.Skip()
```

→ `onButtonB` und `onButtonF` werden ausgeführt

Widgets anordnen mit Sizern

Widgets per Hand anordnen hat Nachteile:

- Unpraktikabel für viele Widgets
- Widgets haben für unterschiedliche Standard-Schriften unterschiedlieche Ausmaße
- Muss angepasst werden, wenn die Fenstergröße verändert wird

→ Sizer

- Nehmen mehrere Widgets auf
- Ordnen sie nach vorgegebenen Regeln in einem Panel an
- Ordnen sie automatisch neu

Widgets anordnen mit Sizern

```
# Sizer erstellen:
panel = wx.Panel(parent=self)
box = wx.BoxSizer(wx.HORIZONTAL)
panel.SetSizer(box)

# Widgets einfuegen:
button = wx.Button(parent=panel, label="Spam")
box.Add(button, proportion=1, flag=wx.CENTER)
```

Es können beliebig viele Widgets mit Add in den Sizer eingefügt werden, jedes mit eigenen Platzierungs-Regeln.

Widgets anordnen mit Sizern

```
Add(widget, proportion=0, flag=0, border=0)
```

- proportion: Verhältnis, in dem der freie Platz zwischen Widgets aufgeteilt wird (nur bei BoxSizern.)
- flag: Bestimmt Ausrichtung dieses Wigets und seines Rahmens:
  - `wx.ALIGN_TOP`, `wx.ALIGN_BOTTOM`, `wx.ALIGN_LEFT`, `wx.ALIGN_RIGHT`, `wx.ALIGN_CENTER`: Aurichtung des Widgets
  - `wx.EXPAND`: Widget wird gestreckt
  - `wx.ALL`, `wx.TOP`, `wx.BOTTOM` `wx.LEFT`, `wx.RIGHT`: An welchen Seiten ein Rahmen eingefügt werden soll
  - Flags können mit | kombiniert werden: `flag=wx.ALIGN_CENTER|wx.ALL`
- border: Rahmen (Freiraum) um das Widget in Pixeln

BoxSizer und GridSizer

```
BoxSizer(wx.HORIZONTAL) # oder wx.VERTICAL
```

BoxSizer: Widgets werden in einer horizontalen oder vertikalen Reihe angeordnet.

```
GridSizer(rows, cols, hgap, vgap)
```

GridSizer: Widgets werden in einem regelmäßigen Gitter angeordnet.

- rows, cols: Anzahl der Zeilen und Spalten an Widgets
- hgap, vgap: Horizontaler/Vertikaler Abstand zwischen Widgets in Pixeln

FlexGridSizer und GridBagSizer

```
grid = wx.FlexGridSizer(3, 3, 5, 5)
grid.AddGrowableRow(idx=2, proportion=1)
grid.AddGrowableCol(idx=2, proportion=1)
```

FlexGridSizer: Wie GridSizer, aber:

- Zeilen/Spalten mit unterschiedlichen Höhen/Breiten möglich
- Zeilen/Spalten können flexibel in Höhen/Breiten wachsen, ähnlich BoxSizer

GridBagSizer:

- Bei Add kann die Zelle angeben werden, in welche das Widget einfegügt wird
- Widgets können über mehrere Zellen gehen

Texteingaben mit TextCtrl

```
TextCtrl(parent, id=-1, value="", pos=(-1,-1),
         size=(-1,-1), style=0, ...)
```

- automatisch Standard-Tastaturkürzel: Ctrl-x, Ctrl-v, ...
- value: Der anfängliche Inhalt des Textfeldes
- style:
  - `wx.TE_CENTER`, `wx.TE_LEFT`, `wx.TE_RIGHT`: Ausrichtung des Textes
  - `wx.TE_MULTILINE`: Mehrzeilige Texteingabe zulassen
  - `wx.TE_PASSWORD`: Text wird durch Sternchen verborgen
  - ...

Texteingaben mit TextCtrl

Einige Methoden von TextCtrl:

- GetValue, SetValue: Textinhalt lesen/setzen
- GetStringSelection: Den markierten Textbereich lesen
- Clear: Textinhalt löschen

```
txt = wx.TextCtrl(panel, value="Default",
          style=wx.TE_MULTILINE|wx.TE_CENTER)

txt.SetValue("Neuer Default")
print txt.GetStringSelection()
```

Auswahl mit Checkboxen

```
check = wx.CheckBox(parent=panel,
                    label="Check &me")
self.Bind(wx.EVT_CHECKBOX, self.on_checkbox,
          check)
print check.IsChecked()
```

- Statusabfrage mit der Methode `IsChecked`
- Betätigung der Checkbox löst `wx.EVT_CHECKBOX` aus
- Liste von Checkboxen: Voneinander unabhängige Checkboxen, es können beliebig viele Boxen ausgewählt werden

Einzel-Auswahl mit RadioBox

Aus einer Liste von Optionen kann nur eine ausgewählt werden.

```
items = ["One", "Two", "Three"]
radio = wx.RadioBox(parent=panel,
                    choices=items,
                    label="Your choice")
```

- Statusabfrage mit der Methode `GetStringSelection`
- Betätigung der Checkbox löst `wx.EVT_RADIOBOX` aus
- Mit zusätzlichen Parametern des Konstruktors kann Anzahl Zeilen/Spalten bestimmt werden:
  - majorDimension: Anzahl Zeilen oder Spalten
  - style: `wx.RA_SPECIFY_COLS` oder `wx.RA_SPECIFY_ROWS`

Auswahl mit ListBox

```
items = ["One", "Two", "Three"]
list = wx.RadioBox(parent=panel,
                   choices=items,
                   style=wx.SINGLE)
```

- Statusabfrage mit der Methode `GetStringSelection` oder `GetSelections`
- Betätigung der Listbox löst `wx.EVT_LISTBOX` aus
- Verschiedene Styles:
  - `wx.LB_SINGLE`: Anwender kann nur eine Option auf einmal auswählen
  - `wx.EXTENDED`: Anwender kann einen Bereich auswählen
  - `wx.MULTIPLE`: Anwender kann beliebig viele Optionen auswählen

Modale Dialoge

Modaler Dialog: Kleines Popup-Fenster, welches die restliche Anwendung blockiert.

```
msg = wx.MessageDialog(parent=panel,
                       message="Are you ok?",
                       caption="Question",
                       style=wx.YES_NO|wx.ICON_QUESTION)

value = msg.ShowModal()
if value == wx.ID_YES:
    print "That's fine!"
else:
    print "I'm sorry."
```

MessageDialog

Stellt ein (optionales) Icon, einen Text und Buttons dar.

```
wx.MessageDialog(parent, message,
                 caption="Message box",
                 style=wx.OK|wx.CANCEL, pos=(-1,-1))
```

Style-Optionen:

- `wx.YES_NO`, `wx.OK`, `wx.CANCEL`: Dargestellte Buttons
- `wx.ICON_ERROR`, `wx.ICON_INFORMATION`, `wx.ICON_QUESTION`: Dargestelltes Icon

TextEntryDialog

Für kurze Eingaben vom Anwender.

```
dlg = wx.TextEntryDialog(parent=panel,
                         message="Your name?",
                         caption="Question", style=wx.OK)

value = dlg.ShowModal()
if value == wx.ID_OK:
    print dlg.GetValue()
```

Weitere Dialoge:

- `wx.PasswordEntryDialog`
- `wx.SingleChoiceDialog` (Stellt eine ListBox dar)

FileDialog

```
dlg = wx.FileDialog(parent=panel,
                    message="Choose a file",
                    wildcard="Python|*.py|All|*",
                    style=wx.OPEN)

value = dlg.ShowModal()
if value == wx.ID_OK:
    print dlg.GetPath()
```

- Wichtigste Style-Optionen: `wx.OPEN` oder `wx.SAVE`
- Ähnlich: DirDialog für Verzeichnisse

Menüs und Menüleiste: MenuBar

Vorgehensweise für eine vollständige Menüleiste:

- MenuBar erstellen und dem Frame zuordnen
- Einzelne Menüs erstellen und der MenuBar hinzufügen
- Items zu den einzelnen Menüs hinzufügen
- Event Handler erstellen und den Items zuordnen

```
class MainFrame(wx.Frame):
    def __init__(self):
        wx.Frame.__init__(self, parent=None)
        menubar = wx.MenuBar()
        self.SetMenuBar(menubar)
```

Menüs in die Menüleiste einfügen

```
menu = wx.Menu()
menubar.Append(menu, "&Menu")

blah = menu.Append(-1, "&Blah\tCtrl+B",
                   "Some help text")
self.Bind(wx.EVT_MENU, self.on_blah, blah)
```

- Mnemonic Shortcuts mit & im Item-Namen
- Accelerator Shortcuts mit \t im Item-Namen
- Hilfetext wird in der Statuszeile angezeigt
- AppendSeparator() zum Unterteilen der Items mit einer Linie

Statuszeile: StatusBar

```
class MainFrame(wx.Frame):
    def __init__(self):
        wx.Frame.__init__(self, parent=None)
        self.CreateStatusBar()
        self.SetStatusText("Hallo Welt")
```

- Hilfetext der Menü-Items wird automatisch angezeigt
- Setzen des angezeigten Textes mit SetStatusText

Weitere Möglichkeiten

- Toolbars mit wx.ToolBar
- Tabs und gesplittete Fenster: wx.NoteBook, wx.SplitterWindow
- Flexible Listen und Tabellen: wx.ListCtrl, wx.grid.Grid
- Baumdarstellungen: TreeCtrl
- Schriften und Schrift-Auswahldialoge: wx.Font, wx.FontDialog
- Farben und Farb-Auswahldialoge: wx.Colour, wx.ColourDialog
- Umgang mit Bildern und Grafik; Zeichnen
- . . . → wxPython-Demo

Dokumentation:

- http://www.wxpython.org/onlinedocs.php
- Buch: *wxPython in Action*

Bekannte wxPython-Anwendungen

- wxGlade: GUI-Designer für wxWidgets
- Boa Constructor: Python-IDE und GUI-Designer für wxWidgets
- SPE: Python-IDE und GUI-Designer für wxWidgets
- DrPython: Python-IDE
- BitTorrent: Bittorrent-Client
- wxRemind: Graphisches Frontend für den Linux-Kalender Remind

Zusammenfassung und Ausblick

Zusammenfassung

Wir haben kennengelernt:

- verschiedene Datentypen (tw. „High Level“)
- die wichtigsten Statements
- Funktionsdeklaration und -Benutzung
- Module und Pakete
- Fehler und Ausnahmen, Behandlung selbiger
- objektorientierte Programmierung
- einige häufig verwendete Standardmodule

Offene Punkte

Nicht behandelte, tw. fortgeschrittene Themen:

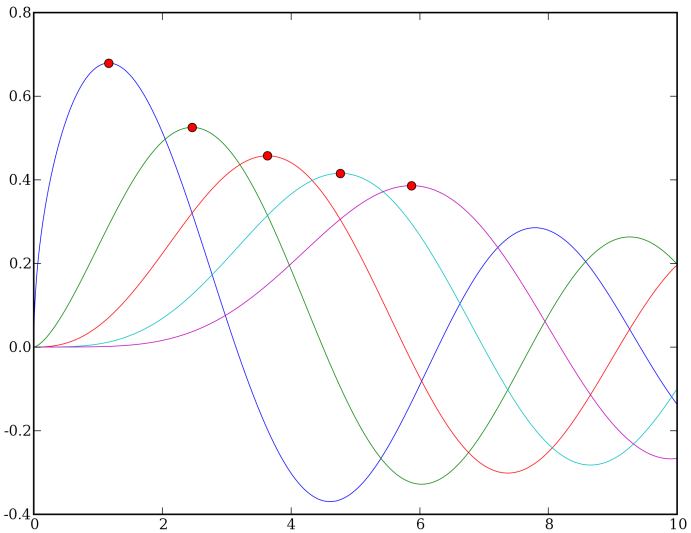
- Closures, Dekoratoren (Funktionswrapper)
- Metaklassen
- Weitere Standardmodule: Mail, WWW, XML, Zeit&Datum, ... → http://docs.python.org/lib/modindex.html
- Profiling, Debugging, Unittesting
- Extending und Embedding: Python & C/C++ → http://docs.python.org/ext/ext.html
- Third Party-Module: Grafik, Webprogrammierung, Datenbanken, ... → http://pypi.python.org/pypi

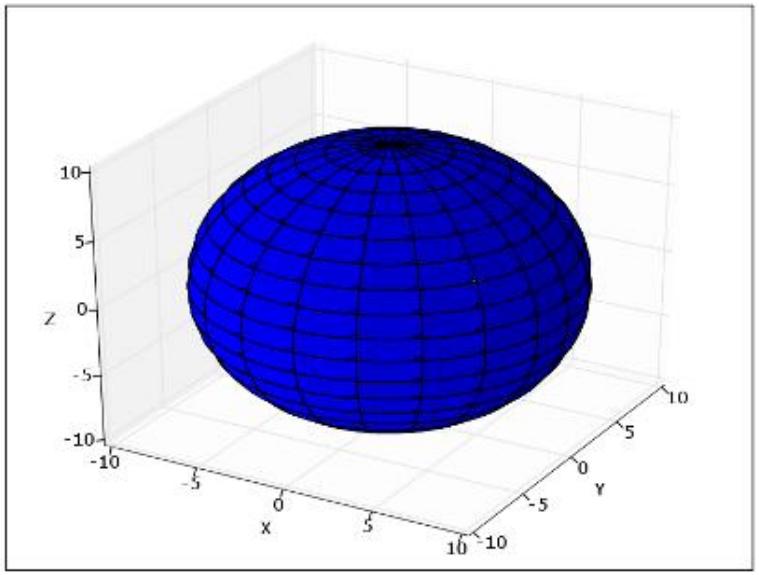
Web-Programmierung

- CGI-Scripte: Modul `cgi` aus Standardbibliothek
- Webframeworks: Django, TurboGears, Pylons, ...
- Templatesysteme: Cheetah, Genshi, Jinja, ...
- Content Management Systeme (CMS): Zope, Plone, Skeletonz, ...
- Wikis: MoinMoin, ...

NumPy + SciPy + Matplotlib = Pylab

Ein Ersatz für MatLab: Matritzenrechnung, numerische Funktionen, Plotten, ...





→ Kurs *Scientific Python* im JSC, voraussichtl. Oktober 2011

Und mehr...

- ipython: Eine komfortablere Python-Shell
- Python und andere Programmiersprachen:
  - Jython: Python-Code in der Java VM ausführen
  - Ctypes: C-Libraries mit Python ansprechen (ab 2.5 in der stdlib)
  - SWIG: C- und C++ -Libraries mit Python ansprechen
- PIL: Python Imaging Library für Bildmanipulation
- SQLAlchemy: ORM-Framework
  - Abstraktion: Objektorientierter Zugriff auf DB-Daten

PyCologne

PyCologne: Python User Group Köln

- Trifft sich jeden zweiten Mittwoch im Monat am Rechenzentrum der Uni Köln
- URL: http://wiki.python.de/pyCologne

Viel Spaß mit