Umstieg von Basic auf Python mit AOO unter Linux

Ein Erfahrungsbericht mit Makros, die in einem Calc-Dokument eingebettet sind

von Volker Lenhardt

Inhaltsverzeichnis

1. Vorbereitungen

Meine ersten Versuche mit Python bewegten sich natürlich innerhalb der OO-Umgebung (Apache OpenOffice 4.1.2). Die Makrodateien sind im Verzeichnis „~/.openoffice/4/user/Scripts/python“ untergebracht, damit sie im Pythonpfad sind und von OO erkannt werden. Dateien mit Funktionen, die von verschiedenen Dateien aus importiert werden sollen, müssen im Unterverzeichnis „pythonpath“ liegen, denn auch dieses Verzeichnis liegt im Pythonpfad.

Das funktioniert so nicht mehr, wenn die Makrodateien im Dokument selbst liegen, denn dort befinden sie sich nicht mehr im Pythonpfad. Alles muss nun auf Dokumentebene erledigt werden.

Ich bin von einem Dokument ausgegangen, das ein funktionierendes System von Basic-Skripten und in der Basic-IDE definierten Dialogen enthält. Natürlich ist es etwas einfacher, neue Makros in der OO-Umgebung zu entwickeln, denn nach Änderungen am Code muss die OO-Komponente nicht immer wieder neu gestartet werden. Doch spätestens wenn man die Codedateien in das Dokument verschoben hat, wird man bemerken, welche Fehler man übersehen hat und welche Erweiterungen unbedingt noch notwendig sind. Dann ist man doch an dieser hier beschriebenen Stelle. Und es ist gar nicht so kompliziert. Wenn man sich richtig eingerichtet hat, geht es auch ganz bequem.

Der erste Schritt bestand darin, die ods-Datei in ein Testarbeitsverzeichnis auszupacken. Ich nutze unter Linux (openSUSE 13.2) am liebsten Ark. Die Datei mit Namen „Kostenabrechnung.ods“ wird also in ein Verzeichnis mit dem Namen „Kostenabrechnung“ ausgepackt. Darin befinden sich folgende Verzeichnisse und Dateien

```
Basic
   Standard (Bibliothek)
      diverse Makrodateien (Module) als xml und die Metadatei script-lb.xml
   script-lc.xml
Configurations2 (alles Verzeichnisse für die vom Benutzer vorgenommenen Anpassungen)
   accelerator
   floater
   images
   menubar
      menubar.xml
   popupmenu
   progressbar
   statusbar
   toolbar
   toolpanel
Dialogs
   Standard (Bibliothek)
      diverse Dialogdefinitionen als xml und die Metadatei dialog-lb.xml
   dialog-lc.xml
META-INF
   manifest.xml
Thumbnails
   thumbnail.png
content.xml
manifest.rdf
meta.xml
mimetype
settings.xml
styles.xml
```

Meine neuen Python-Makros müssen in das Verzeichnis „Scripts“ mit dem Unterverzeichnis „python“. Also füge ich in den Verzeichnisbaum neue Verzeichnisse und eine Makrodatei „macros.py“ ein:

```
Scripts
   python
      macros.py
```

Damit diese auch erkannt werden, benötige ich Einträge in der Datei „META-INF/manifest.xml“.

Dort stehen bisher Einträge, die auf die einzelnen Basic-Skripte und Metadateien („script-lc.xml“ und „script-lb.xml“) verweisen, z.B.

```
<manifest:file-entry manifest:media-type="text/xml" manifest:
full-path="Basic/Standard/Tools.xml"/>
```

Ich muss für das Python-Skript folgende Einträge einfügen:

```
<manifest:file-entry manifest:media-type="" manifest:
full-path="Scripts/python/macros.py"/>
<manifest:file-entry manifest:media-type="application/binary" manifest:
full-path="Scripts/python/"/>
<manifest:file-entry manifest:media-type="application/binary" manifest:
full-path="Scripts/"/>
```

Diese Änderungen sind zwar nun im ausgepackten Verzeichnis „Kostenabrechnung“ enthalten, doch noch nicht in der Datei „Kostenabrechnung.ods“. Dazu muss ich das Verzeichnis in die von OO akzeptierte zip-Datei verpacken. Ich mache das mit „zip“ aus dem aktuellen Verzeichnis „Kostenabrechnung“ heraus (mit anderen Programmen sieht die Kommandozeile natürlich anders aus):

```
zip -r -Z store Pfad/Kostenabrechnung.ods *
```

2. Arbeitsumgebung

Im Laufe der Arbeit werden nach jeder Änderung an einer Skript- oder Administrationsdatei folgende Arbeitsgänge nötig, um die Änderungen zu testen:

- mit zip packen
- OO mit der neuen Datei starten
- testen
- OO schließen

Außerdem fühle ich mich sicherer, wenn ich auf vorherige Versionen zurückgreifen kann, falls das Durcheinander im Code zu groß geworden sein sollte. Ich mache vor dem Zippen noch eine Sicherungskopie.

Es kommt erschwerend hinzu, dass man auf die Fehlermeldungen angewiesen ist, die OO in einer Box anzeigt. Mir ist es nicht gelungen, diesen Text zu kopieren. Wenn ich also meinen Code überprüfe, muss die Meldung so lange sichtbar bleiben, bis ich den Fehler geortet habe. OO darf also nicht versehentlich beendet werden. Schlimmer noch: nicht alle Fehler werden auch angezeigt.

2.1. Debug-Informationen

Die Ausgabe von Laufzeitinformationen kann über zwei Umgebungsvariablen gesteuert werden, „PYSCRIPT_LOG_LEVEL“ und „PYSCRIPT_LOG_STDOUT“. In der Datei „pythonscript.py“ im OO-Installationsverzeichnis (bei mir „/opt/openoffice4/program/“) sind sie beschrieben.

PYSCRIPT_LOG_STDOUT=0	: Ausgabe in die Datei „user/Scripts/python/log.txt“
PYSCRIPT_LOG_STDOUT=1 (oder leer):	: Ausgabe nach stdout (= Standard)

Bei den Debug-Modi geht es nur um die Ausgabe von Fehlermeldungen und Laufzeitprotokollen. Die Ausgabe von print-Anweisungen wird in keinem Fall unterdrückt.

PYSCRIPT_LOG_LEVEL=DEBUG	: alle Informationen (wirkt etwas unübersichtlich)
PYSCRIPT_LOG_LEVEL=ERROR	: nur Fehlerausgaben (leider nicht alle Fehler)
PYSCRIPT_LOG_LEVEL=	: keine Ausgaben (= Standard)

All diese Dinge habe ich in einer Skriptdatei „updateKAR.sh“ zusammengefügt:

```
#!/bin/bash
#
# Genutzt nach Änderung von Python-Skripten.
# Erstellt die Datei $1.ods neu aus den internen Dateien.
# Kopiert die aktuelle Dateiversion als Sicherung.
# Setzt die Python-Systemvariablen zur Ausgabe von Debug-Informationen.
# Startet die neue Dateiversion.
#
# Argument 1: der Dateiname ohne Endung
# Argument 2: der Debug-Level für die Systemvariable PYSCRIPT_LOG_STDOUT:
# - kein Argument: keine Debug-Ausgabe
# - ERROR/error  : nur Fehlermeldungen
# - DEBUG/debug  : alle Infos

odsName="$1"
debug="$2"
odsPath="/home/volker/Dokumente/EDV/OpenOffice/Makros/Python/"

# Aktuelles Datum und Uhrzeit für den Dateinamen der Sicherungskopie
dateTime=$(date +"_%F_%T")

# Wechsel in das Verzeichnis mit den ausgepackten Dateien
cd $odsPath$odsName
```

```
# Umbenennung der aktuellen Datei in eine Datei gleichen Namens,
# aber mit Datums- und Zeitergänzung
if [ -f $odsPath$odsName.ods ]; then
  mv $odsPath$odsName.ods $odsPath$odsName$dateTime.ods
fi

# Debug-Ausgabe:
#export PYSCRIPT_LOG_STDOUT=      : keine Debug-Ausgaben (= Standard)
#export PYSCRIPT_LOG_STDOUT=ERROR : normale Ausgaben und Fehlermeldungen
#export PYSCRIPT_LOG_STDOUT=DEBUG : alle Infos

# Argument 2 umwandeln in Großbuchstaben
if [ "$debug" != "" ]; then
  debug=$(echo "$debug" | tr '[:lower:]' '[:upper:]')
fi

for s in "DEBUG" "ERROR"
  do
    if [ $s == "$debug" ]; then
      export PYSCRIPT_LOG_LEVEL=$debug
    fi
  done

# Debug-Ausgabeort:
#export PYSCRIPT_LOG_STDOUT=1 oder leer: Ausgabe nach stdout (= Standard)
#export PYSCRIPT_LOG_STDOUT=0          : Ausgabe nach
#                                        "~/.openoffice/4/user/Scripts/python/log.txt"

zip -r -Z store $odsPath$odsName.ods *
wait
apacheopenoffice4 $odsPath$odsName.ods
```

2.2. Wahl des Editors

Die nächste Frage stand an, mit welchem Editor ich arbeiten soll. Irgendwie liegt Spyder nahe, denn man bekommt Syntaxfehler schon beim Editieren angezeigt, und man kann Codeteile in der integrierten Pythonkonsole testen.

Dennoch war ich nicht zufrieden. Gegenüber einem wirklich guten Editor sind die Spyder-Möglichkeiten doch begrenzt. So fehlt, um nur eines zu nennen, eine Funktion zum Einklappen von Codeteilen. Doch was mich am meisten genervt hat, war der Umstand, dass ich mit der Maus markierten Text nicht so einfach mit Mittel-Klick anderswo einfügen kann. Alle anderen Programme erlauben das als Kopie, Spyder aber löscht dabei den markierten Text. Ich habe nicht herausfinden können, wie man das ändern kann.

Unübertroffen ist sicher VIM, bzw. Gvim als GUI-Version (die EMACS-Freunde werden protestieren). Wenn ich Profi wäre, käme nichts anderes in Frage. Aber da ich manchmal über Wochen und Monate nicht an Programmen werkele, muss ich mich immer wieder neu mit den seltener benötigten Tastenkombinationen vertraut machen. Irgendwann habe ich resigniert.

Seitdem nutze ich Kate. Nützlich ist auch die Möglichkeit, ein Projekt zu definieren, d.h. einen kompletten Verzeichnisbaum zur Bearbeitung bereitzustellen. Im Verzeichnis „/home/volker/Dokumente/EDV/OpenOffice/Makros/Python/“ habe ich eine Datei namens „.kateproject“ mit folgendem Inhalt (s. http://kate-editor.org/2012/11/02/using-the-projects-plugin-in-kate/):

```
{
  "name": "Kostenabrechnung"
, "files": [ { "directory": "Kostenabrechnung", "recursive": 1 } ]
}
```

2.3. Die praktische Arbeit

Meine Arbeitsumgebung sieht also so aus:

Ich starte 3 Programme:

Kate mit der Datei „/home/volker/Dokumente/EDV/OpenOffice/Makros/Python/Kostenabrechnung/Scripts/python/macros.py“ (über die „Projekte“-Anzeige kann ich ganz schnell jede Datei im Verzeichnis „Kostenabrechnung“ öffnen, praktisch auch in geteilter Ansicht.)

Spyder, um vor dem Zippen noch auf Syntaxfehler zu kontrollieren und um in der Pythonkonsole Codeschnipsel zu testen.

Ein GUI-Terminal (ich nutze die KDE-Konsole), in dem ich meine Skriptdatei „updateKAR.sh“ (s.o.) starte.

Nachdem ich in Kate irgendwelche Änderungen vorgenommen und gespeichert habe, starte ich im Terminal das Skript mit:

```
updateKAR.sh Kostenabrechnung debug
```

Aber nur beim ersten Mal. Beim nächsten Aufruf genügt Pfeiltaste-NachOben zur Anzeige der zuvor genutzten Kommandozeile. Bei Bedarf kann ich „debug“ durch „error“ ersetzen oder ganz weglassen.

OO startet mit der Datei „Kostenabrechnung.ods“ und schreibt die Debug-Ausgaben in das Terminal. Ich schließe OO, korrigiere das Skript und drücke im Terminal die Pfeiltaste-NachOben. Das ist alles. Geht blitzschnell. Zur meiner Beruhigung hat das Skript eine Sicherungskopie mit Datum und Uhrzeit angelegt, die ich mit Ark auspacken kann.

Achtung: Wenn beim Starten des Skripts „updateKAR.sh“ ein OO-Dokument geöffnet ist, werden keine Ausgaben im Terminal erscheinen, denn das neue Dokument wird in der Programmumgebung der schon geöffneten Komponente ausgeführt.

3. Bibliotheken und Module

Bei einem größeren Projekt mit mehreren Dialogen kann es unübersichtlich werden, wenn der gesamte Code in einer einzigen Datei steckt. Die OO-Basic-Strukturen kennen die Aufteilung in Bibliotheken und Module. Die Module sind Dateien, in denen der Code steckt. Bibliotheken sind die Verzeichnisse, in denen Module zusammengefasst sind, in meinem Beispiel ist das „Standard"-Verzeichnis in „Basic" eine Bibliothek.

Für Python ist diese Bibliotheksebene nicht vorgesehen. Die Codedateien, auch in Python als Module bezeichnet, liegen also direkt im „Scripts"-Verzeichnis.

Um die Organisation meines Python-Codes an die aus der Basic-Umgebung anzugleichen, habe ich die Aufteilung der Basic-Module auf die Python-Module abgebildet. Damit sie im Dokument bekannt werden, benötige ich Einträge für alle diese Dateien in „META-INF/manifest.xml":

```
<manifest:file-entry manifest:media-type="" manifest:full-path=
"Scripts/python/macros01.py"/>
<manifest:file-entry manifest:media-type="" manifest:full-path=
"Scripts/python/macros02.py"/>
<manifest:file-entry manifest:media-type="" manifest:full-path=
"Scripts/python/toolbox.py"/>
```

Mindestens eins dieser Module (sagen wir „toolbox.py") enthält Variablen, Klassen und Funktionen, die in den anderen Modulen importiert werden. Ein solches Modul wird als Library bezeichnet. Man sieht, dass obwohl der Name an eine Verwandtschaft mit einer Basic-Bibliothek denken lässt, in Wirklichkeit doch ein fundamentaler Unterschied besteht. In der OO-Python-Umgebung ist für Libraries das Unterverzeichnis „pythonpath" vorgesehen, das automatisch im Pythonpfad liegt.

Bei eingebettetem Code funktioniert dieser „pythonpath" leider nicht, wie ich nach frustrierenden Tests erkennen musste. Mir gelang es aber, Libraries über eine Ergänzung des Pythonpfads einzubinden. An den Anfang jeder betreffenden Datei setze ich daher folgende Anweisungen, um die Klassen und Funktionen aus der Datei „toolbox.py" zu importieren:

```
import uno
import sys

doc = XSCRIPTCONTEXT.getDocument()

pythonPath = uno.fileUrlToSystemPath(doc.URL) + '/Scripts/python'
if pythonPath not in sys.path: sys.path.append(pythonPath)

from toolbox import *
```

Das funktioniert ausgezeichnet. Doch als ich meine Projektdatei als Dokumentvorlage gespeichert hatte und daraus dann ein neues Dokument startete, bemerkte ich, dass die Pfadergänzung nur funktionieren kann, wenn die Datei gespeichert ist. Und das Dokument „Unbekannt1" ist noch nicht gespeichert. Es gibt keinen URL. Auch wenn ich die Fehlermeldung abfange, ist doch die Library „toolbox" unbekannt. Ich habe keinen Weg gefunden, dieses Problem zu lösen, ohne doch wieder den gesamten Code in einer einzigen Datei zusammenzufassen. Man kann eben nicht alles haben, sauber getrennten Code **und** Dokumentvorlage.

4. Unterschiede zwischen Basic und Python

Eine wichtige Information vorweg: Mit Apache OpenOffice 4.1.2 wird Python 2.7.6 installiert, mit neueren Versionen von LibreOffice jedoch Python 3.x, das nicht hundertprozentig kompatibel mit der 2.x-Version ist (zu den Unterschieden s. die Python-Dokumentation). Ich habe das bemerkt, als ich meine Projektdatei zum Testen in LO öffnete und auf einen Fehler stieß, der mich auf eine „print“-Anweisung hinwies. In 3.x ist das nämlich eine Funktion!

Meine wichtigsten Quellen sind

http://www.openoffice.org/de/doc/entwicklung/python_bruecke.html und
https://wiki.openoffice.org/wiki/Python/Transfer_from_Basic_to_Python

Darüber hinaus sind mir weitere Besonderheiten aufgefallen.

4.1. Öffentliche Variablen und Konstanten

In Basic sind alle Konstanten, Variablen und Prozeduren, die außerhalb von Prozeduren definiert sind, in der gesamten Bibliothek öffentlich, das heißt in allen Modulen dieser Bibliothek. Da es in Python keine Bibliotheken gibt, sondern nur Module (Libraries), sind Variablen, Klassen und Funktionen, die in einem Modul außerhalb von Klassen und Funktionen definiert sind, nur für dieses Modul öffentlich – „global“ im Python-Sprachgebrauch.

In Python werden Variablen, Klassen und Funktionen, die in mehreren Modulen benötigt werden, in einer Library definiert und bei Bedarf in anderen Modulen importiert.

In Basic werden Konstanten mit dem Schlüsselwort „Const“ als unveränderlich definiert. So etwas gibt es in Python nicht. Der Programmierer ist selbst dafür verantwortlich, dass als Konstanten vorgesehene Variablen nicht verändert werden. Man kann zum Beispiel Namen für Konstanten in Großbuchstaben schreiben. Da in Python-Namen Groß- und Kleinschreibung unterschieden werden, hat man schon einmal mehr Sicherheit als in Basic. Man kann auch für alle Konstanten eine Klasse bilden, die nur Klassenattribute hat, also gar nicht dafür vorgesehen ist, instanziiert zu werden – und dann auch noch in Großbuchstaben. Ein Beispiel in Ausschnitten:

```
class Const:
    """
    Als konstant geltende Werte für Belegblätter und Dialoge.
    """
    RECCNT_ROW = 1       # Zeilenindex der Zelle mit der Gesamtzahl der Belege
    RECCNT_COL = 23      # Spaltenindex der Zelle mit der Gesamtzahl der Belege
    START_ROW = 3        # Zeilenindex des ersten Belegs in einem Blatt
    END_COL = 22         # Spaltenindex des letzten Belegelements
    # u.
    # s.
    # w.
    # Standardverzeichnis für den Filepicker
    SCANDIR = '/home/volker/Dokumente/Kosten/Belege/'
    # Gültige Dateinamenerweiterungen für Grafikdateien
    GRAPHIC_EXTS = ['.bmp', '.jpg', '.jpeg', '.jfif', '.jif', '.png',
                    '.tif', '.tiff']
    # u.
    # s.
    # w.
```

Verwendung:

```
cell = sheet.getCellByPosition(Const.RECCNT_COL, Const.RECCNT_ROW)
```

4.2. Locale

Man kann sich nicht darauf verlassen, dass automatisch das benötigte System-Locale zur Verfügung steht. Zum Formatieren von locale-abhängigen Strings ist es angebracht, das Locale gezielt zu setzen. Alles Notwendige bietet die Library „locale“.

Ich benötige zum Beispiel Strings für die deutsche Währungsdarstellung von Double-Werten aus Calc und umgekehrt.

```
def currency_de_val2str(value):
    """
    Konvertiert einen Double-Wert in einen deutschen Währungsstring (d.h. mit
    Dezimalkomma und Tausenderpunkt).
    """
    from locale import getlocale, setlocale, LC_ALL, format as locformat
    oldLoc = getlocale()       # Sichert das aktuelle Locale.
    setlocale(LC_ALL, ('de_DE', 'UTF-8'))
    try:
        # Formatiert nach deutschen Regeln.
        retStr = locformat('%0.2f', value, grouping=True)
    except:
        retStr = ''
    setlocale(LC_ALL, oldLoc)  # Stellt das ursprüngliche Locale wieder her.
    return retStr


def currency_de_str2val(valStr):
    """
    Konvertiert einen deutschen Währungsstring in einen Double-Wert.
    """
    from locale import getlocale, setlocale, LC_ALL, atof
    oldLoc = getlocale()
    setlocale(LC_ALL, ('de_DE', 'UTF-8'))
    try:
        retVal = atof(valStr)      # Konvertiert zum Typ Float
    except:
        retVal = 0
    setlocale(LC_ALL, oldLoc)
    return retVal
```

4.3. Calc-Funktionen aufrufen

Argumente von Funktionen müssen als Tuple angegeben werden, auch wenn es nur ein einziger Wert ist. Funktionsnamen müssen in Englisch sein. Zur Erinnerung: ein Tuple ist eine durch Komma getrennte Sequenz von Werten in runden Klammern. Nach dem letzten Element darf ein Komma stehen. Man kann also ein Tuple mit nur einem Element durch beispielsweise „(2,)“ erzeugen, alternativ mit „tuple([2])“.

```
fAccess = uno.getComponentContext().ServiceManager.createInstance(
                                                    'com.sun.star.sheet.FunctionAccess')
result = fAccess.callFunction('Min', (24, 6, 18, 234))  # 6
result = fAccess.callFunction('Abs', (-2,))             # 2
result = fAccess.callFunction('Upper', ('klein',))      # KLEIN
```

4.4. Zellbereich sortieren

Das einfache Zuweisen eines Tuples von TableSortField-Structs zur SortDescriptor-Property „SortFields“ funktioniert nicht. Ist das ein allgemeines Phänomen in Structs? Ich habe keine weiteren Tests gemacht.

Zum Beispiel: Gegeben sind zwei definierte TableSortField-Structs in einem Tuple:

```
sortFields = (sortField1, sortField2)
```

Das Tuple lässt sich zuweisen, ohne einen Fehler zu produzieren, aber auch ohne zu sortieren:

```
sortDesc1.Name = 'SortFields'
sortDesc1.Value = sortFields   # Das Sortieren funktioniert einfach nicht.
```

Stattdessen funktioniert die folgende Zuweisung:

```
sortDesc1.Name = 'SortFields'
sortDesc1.Value = uno.Any('[]com.sun.star.table.TableSortField', sortFields)
```

4.5. Datum in Zellen

Die Datums- und Uhrzeitformate in OO unterscheiden sich in wesentlichen Punkten von denen in der Python-Library „datetime“ definierten Klassen. Ein als Datum-/Uhrzeit verwendeter Calc-Zellwert und ein Basic-Date-Typ sind Double-Werte, deren ganzzahliger Anteil das Datum als Anzahl der Tage vom 30.12.1899 an darstellt, und deren Bruchanteil die Uhrzeit als Bruchteil des Tages enthält.

Die Python-Library „datetime“ kennt die Klassen datetime.date, datetime.time und die beides umfassende Klasse datetime.datetime, die alle einen Punkt in der Zeit beschreiben. Dazu gesellt sich die Klasse datetime.timedelta, die einen Zeitraum definiert. Man kann mit Objekten dieser Klassen aber auch rechnen, zum Beispiel addieren oder subtrahieren. Ein timedelta-Objekt kann auch ohne Arithmetik aus Zahlenwerten definiert werden (zu den Einzelheiten s. die Python-Dokumentation).

Diese Zusammenhänge werden dann wichtig, wenn ein OO-Datum analysiert werden muss, wie zum Beispiel wenn ein Dialogelement ein Datum aus einer Calc-Zelle aufnehmen soll.

Die **Eingabe** eines Datums in eine Zelle erfolgt am besten als String, z.B. als ISO-String oder formatiert:

```
dateString = '2015-12-28'
```

Oder als Beispiel für das aktuelle Datum in der Form „09.02.2016“:

```
from datetime import date
today = date.today()
dateString = '{:02}.{:02}.{:04}'.format(today.day, today.month, today.year)
```

Je nach Datumsformat erfolgt die Eingabe als Formula oder FormulaLocal:

```
sheet.getCellByPosition(0, 0).FormulaLocal = dateString
```

Das **Lesen** eines Datumswerts aus einer Zelle kann auch als String erfolgen. Für die Übertragung in ein Dialog-Datumselement ist ein String allerdings nicht geeignet. Er müsste geparst werden, angesichts der Vielfalt von Anzeigeformaten etwas umständlich. Stattdessen lesen wir den Double-Wert aus und nutzen die Möglichkeiten der Library „datetime“:

```
from datetime import date, timedelta
ooDate = cell.Value
```

Das folgende Datum benötigen wir zur Konvertierung vom OO- zum Python-Datum.

```
ooZeroDate = date(1899, 12, 30)
```

Da der Double-Wert in ooDate die Anzahl der Tage darstellt, nutzen wir ihn als days-Parameter zur Initialisierung eines timedelta-Objekts:

```
daysToOODate = timedelta(days=ooDate)
```

Die Konvertierung in ein Python-Datum ist dann eine simple Addition:

```
pyDate = ooZeroDate + daysToOODate
```

Die Gesetze der Arithmetik gelten auch für den umgekehrten Fall:

```
ooDate = (pyDate – ooZeroDate).days # Die timedelta-Eigenschaft days gibt den Zeitraum
                                    # als Anzahl der Tage zurück
```

Das folgende Beispiel, das ein Dialog-Datumsfeld mit einem Datum aus einer Calc-Zelle füllt, funktioniert sowohl mit AOO als auch mit neueren Versionen von LO.

```
from datetime import date, timedelta
dateVal = cell.Value
pyDate = date(1899, 12, 30) + timedelta(days=dateVal)

try:
    # LO verwendet ein Struct für ein Datumsfeld.
    from com.sun.star.util import Date
    ctrlDate = Date()
    ctrlDate.Year = pyDate.year
    ctrlDate.Month = pyDate.month
    ctrlDate.Day = pyDate.day
    ctrl.Model.Date = ctrlDate
except:
    # AOO verwendet einen Long-Wert in der Form YYYYMMDD für ein Datumsfeld.
    ctrl.Model.Date = pyDate.year * 10000\
                      + pyDate.month * 100\
                      + pyDate.day
```

4.6. MsgBox und InputBox

Die aus Basic bekannten und beliebten Anweisungen MsgBox und InputBox findet man in Python nicht. Da in der OO-Installation die für die Gestaltung von grafischen Widgets vorgesehene Library Tkinter zwar mitgeliefert ist, aber nicht funktioniert, muss man die in der OO-API verfügbaren Instrumente nutzen, um unkomplizierte Ein- und Ausgaben zu ermöglichen. Man muss sie in eigenen Funktionen definieren.

Als Ersatz für die **MsgBox** bietet sich com.sun.star.awt.XMessageBoxFactory an mit der Methode createMessageBox. Im Internet finden sich leicht unterschiedliche Versionen für Message Boxes. Meine Variante sieht so aus:

```
def msg_box(doc, msgText, msgTitle='', msgType=MESSAGEBOX, msgButtons=BUTTONS_OK):
    """
    Öffnet einen Dialog mit einem Hinweistext
    im Kontext des Frame-Container-Fensters des Dokuments.
    Parameter: msgText   : der in der Box angezeigte Text
               msgTitel  : der in der Titelleiste angezeigte Text
               msgType   : ein Wert aus der Enumeration com.sun.star.awt.MessageBoxType
               msgButtons: ein Wert oder eine Summe von Werten
                           aus der Konstantengruppe com.sun.star.awt.MessageBoxButtons
    Rückgabe:  aus der Konstantengruppe com.sun.star.awt.MessageBoxResults:
               CANCEL = 0
               OK     = 1
               YES    = 2
               NO     = 3
               RETRY  = 4
               IGNORE = 5
    """
    ctx = uno.getComponentContext()
    sm = ctx.ServiceManager
    sv = sm.createInstanceWithContext('com.sun.star.awt.Toolkit', ctx)
    parentWin = doc.CurrentController.Frame.ContainerWindow
    mBox = sv.createMessageBox(parentWin, msgType, msgButtons, msgTitle, msgText)
    return mBox.execute()
```

Die Funktion benötigt das Objekt des Dokuments für dessen Frame-Container-Fenster. Da ich in meinem Projekt die Box nur für das aktuelle Dokument brauche, dachte ich, auf die Übergabe der Dokumentreferenz verzichten zu können, denn die Variable „doc“ ist als globale Variable initialisiert. Um den Aufruf der Funktion kompakter zu gestalten, habe ich sie ohne den Parameter „doc“ definiert:

```
def msg_box(msgText, msgTitle='', msgType=MESSAGEBOX, msgButtons=BUTTONS_OK):
```

Wenn die Funktion in derselben Datei liegt wie die sie aufrufenden Funktionen, ist alles in Ordnung. Wenn „msg_box“ jedoch aus toolbox.py importiert wird, so ist „doc“ plötzlich nicht bekannt, auch wenn „doc“ in toolbox.py global initialisiert sein sollte. Reumütig bin zu der ersten Version zurückgekehrt, die zudem noch flexibler ist, da sie auch für andere vom Projekt geöffnete Dokumente nutzbar sein kann.

Die **InputBox** ist nicht so einfach zu ersetzen. Man benötigt schon einen ausgewachsenen Dialog, den man natürlich in der Basic-IDE bauen kann. Da er jedoch nicht sehr komplex ist, kann man ihn zur Laufzeit erstellen. (Für Basic hat Andrew Pitonyak in seinem Buch OOME ein ausführliches Beispiel zur Erstellung von Dialogen zur Laufzeit vorgestellt, s. https://www.uni-due.de/~abi070/ooo.html, Abschnitt 18.5.2).

Für den Ersatz der InputBox in Python findet man diverse Beispiele im Internet. Ich habe das Beispiel aus der oben erwähnten Quelle „Transfer_from_Basic_to_Python“ leicht abgewandelt, so dass die Fensterbreite einstellbar ist. Außerdem habe ich das in meinen Augen größte Manko der Basic-InputBox beseitigt, nämlich dass nicht unterschieden werden kann, ob der Nutzer einen leeren String eingegeben oder die Eingabe abgebrochen hat. In meiner Version gibt die Funktion im „Abbrechen“-Fall „None“ zurück.

```
def input_box(message, title="", default="", width=350, x=None, y=None):
    """
    Dialog mit einem Texteingabefeld.
    Parameter:  message: Hinweistext
                title:   Fenstertitel
                default: Vorgabetext
                width:   Fensterbreite in Pixel
                x:       Dialogposition horizontal in Twips, nicht ohne y
                y:       Dialogposition vertikal in Twips, nicht ohne x
    Rückgabe:   String   nach Auslösen der OK-Schaltfläche
                None     nach Auslösen der Abbrechen-Schaltfläche
    https://wiki.openoffice.org/wiki/Python/Transfer_from_Basic_to_Python
    Diese Funktion kann wie folgt aufgerufen werden:
    inputbox("Bitte geben Sie einen Text ein", "Eingabe", "Vorgabetext")
    """
    WIDTH = min(350, width)
    HORI_MARGIN = VERT_MARGIN = 8
    BUTTON_WIDTH = 100
    BUTTON_HEIGHT = 26
    HORI_SEP = VERT_SEP = 8
    LABEL_HEIGHT = BUTTON_HEIGHT * 2 + 5
    EDIT_HEIGHT = 24
    HEIGHT = VERT_MARGIN * 2 + LABEL_HEIGHT + VERT_SEP + EDIT_HEIGHT
    import uno
    from com.sun.star.awt.PosSize import POS, SIZE, POSSIZE
    from com.sun.star.awt.PushButtonType import OK, CANCEL
    from com.sun.star.util.MeasureUnit import TWIP
    ctx = uno.getComponentContext()
    def create(name):
        return ctx.getServiceManager().createInstanceWithContext(name, ctx)
    dialog = create("com.sun.star.awt.UnoControlDialog")
    dialog_model = create("com.sun.star.awt.UnoControlDialogModel")
    dialog.setModel(dialog_model)
    dialog.setVisible(False)
    dialog.setTitle(title)
    dialog.setPosSize(0, 0, WIDTH, HEIGHT, SIZE)
    def add(name, type, x_, y_, width_, height_, props):
        model = dialog_model.createInstance("com.sun.star.awt.UnoControl"
                                            + type + "Model")
        dialog_model.insertByName(name, model)
        control = dialog.getControl(name)
        control.setPosSize(x_, y_, width_, height_, POSSIZE)
        for key, value in props.items():
            setattr(model, key, value)
    label_width = WIDTH - BUTTON_WIDTH - HORI_SEP - HORI_MARGIN * 2
```

```
    add("label", "FixedText", HORI_MARGIN, VERT_MARGIN, label_width,
        LABEL_HEIGHT, {"Label": str(message), "NoLabel": True})
    add("btn_ok", "Button", HORI_MARGIN + label_width + HORI_SEP, VERT_MARGIN,
        BUTTON_WIDTH, BUTTON_HEIGHT,
        {"PushButtonType": OK, "DefaultButton": True})
    add("btn_cancel", "Button", HORI_MARGIN + label_width + HORI_SEP,
        VERT_MARGIN + BUTTON_HEIGHT + 5, BUTTON_WIDTH, BUTTON_HEIGHT,
        {"PushButtonType": CANCEL})
    add("edit", "Edit", HORI_MARGIN, LABEL_HEIGHT + VERT_MARGIN + VERT_SEP,
            WIDTH - HORI_MARGIN * 2, EDIT_HEIGHT, {"Text": str(default)})
    #frame = create("com.sun.star.frame.Desktop").getCurrentFrame()
    frame = create("com.sun.star.frame.Desktop").CurrentFrame
    window = frame.getContainerWindow() if frame else None
    dialog.createPeer(create("com.sun.star.awt.Toolkit"), window)
    if not x is None and not y is None:
        ps = dialog.convertSizeToPixel(uno.createUnoStruct
                                       ("com.sun.star.awt.Size", x, y), TWIP)
        _x, _y = ps.Width, ps.Height
    elif window:
        ps = window.getPosSize()
        _x = ps.Width / 2 - WIDTH / 2
        _y = ps.Height / 2 - HEIGHT / 2
    dialog.setPosSize(_x, _y, 0, 0, POS)
    edit = dialog.getControl("edit")
    edit.setSelection(uno.createUnoStruct("com.sun.star.awt.Selection", 0,
                                          len(str(default))))
    edit.setFocus()
    ret = edit.Model.Text if dialog.execute() else None
    dialog.dispose()
    return ret
```

5. Python-Funktionen als Ereignisbehandlung

5.1. Als Menüereignis

Wenn man einem Menüeintrag eine Funktion zuweist, so steht diese Einstellung in der Datei

`Configurations2/menubar/menubar.xml`

Für Basic-Skripte sieht das zum Beispiel so aus:

```
<menu:menuitem menu:id="vnd.sun.star.script:Standard.EntryForm.EntryForm?
language=Basic&amp;location=document" menu:label="Formular"/>
```

Für ein eingebettetes Python-Skript muss man den Eintrag in die folgende Form bringen:

```
<menu:menuitem menu:id="vnd.sun.star.script:macros.py$start_record_dialog?
language=Python&amp;location=document" menu:label="Formular"/>
```

Man kann die Zuweisung auch im geöffneten Dokument vornehmen, über **Extras|Anpassen|Menüs**. Das ist etwas aufwendiger, denn man darf nicht vergessen, dass die Änderung nur im aktuellen Dokument vorgenommen wird. Anschließend muss man die Datei wieder auspacken, wenn man mit den geänderten Daten weiterarbeiten will.

Die aus Menüs oder über Tastaturereignisse aufgerufenen Python-Funktionen benötigen einen Parameter für eine Sequenz von Argumenten, der mit „*args“ definiert wird. Dabei ist die Bezeichnung „args“ beliebig. Wichtig ist der Asterisk davor, der bedeutet, dass die Funktion/Methode mit einer undefinierten Anzahl (einschliesslich null) von Argumenten aufgerufen werden darf. Die oben zitierte Funktion ist also so definiert:

```
def start_record_dialog(*args):
```

5.2. Als Ereignis eines Dialogelements

Wenn Python-Funktionen als Dialogereignis ausgeführt werden, so sind sie in der entsprechenden Dialogdefinition eingetragen. Als Beispiel diene ein Dialog mit dem Namen „RecordDialog“. Seine Definition findet man in „Dialogs/Standard/RecordDialog.xml“. Der Name des Unterverzeichnisses „Standard“ ist der bei der Definition in der Basic-IDE festgelegte Bibliotheksname.

In diesem Dialog gibt es eine Schaltfläche mit dem Namen „EntryButton“. Sie ist wie folgt definiert, mit einem Basic-Makro als Behandlung des Ereignisses „on-performaction“:

```
<dlg:button dlg:id="EntryButton" dlg:tab-index="18" dlg:left="112" dlg:top="319"
dlg:width="53" dlg:height="13" dlg:value="Eintragen">
  <script:event script:event-name="on-performaction"
  script:macro-name="vnd.sun.star.script:Standard.EntryForm.WriteRecToTable?
  language=Basic&amp;location=document" script:language="Script"/>
</dlg:button>
```

Für ein eingebettetes Python-Skript muss man den Eintrag in folgende Form bringen:

```
<dlg:button dlg:id="EntryButton" dlg:tab-index="18" dlg:left="112" dlg:top="319"
dlg:width="53" dlg:height="13" dlg:value="Eintragen">
  <script:event script:event-name="on-performaction"
  script:macro-name="vnd.sun.star.script:macros.py$on_write_record?language=Python&amp;
  location=document" script:language="Script"/>
</dlg:button>
```

Man kann die Zuweisung auch im geöffneten Dokument vornehmen, über die Eigenschaften der Dialogelemente in der Basic-IDE. Das ist etwas aufwendiger, denn man darf nicht vergessen, dass die Änderung nur im aktuellen Dokument vorgenommen wird. Anschließend muss man die Datei wieder auspacken, wenn man mit den geänderten Daten weiterarbeiten will.

Die als Dialogereignis aufgerufenen Python-Funktionen benötigen einen Parameter für das Ereignis-Objekt (ein Struct). In Basic kann man darauf verzichten, wenn man keine Informationen aus dem Objekt benötigt. In Python darf der Parameter jedoch nicht fehlen. Die oben zitierte Funktion ist also so definiert:

```
def on_write_record(event):
```

5.3. Als Dokumentereignis

Wenn Ereignisse des Dokuments behandelt werden sollen (**Extras|Anpassen|Ereignisse**), so finden sie sich in der Datei „content.xml“ wieder. In meinem Fall ist es das Ereignis „office:new“, das eintritt, wenn über die Dokumentvorlage ein neues Dokument erzeugt wird. Das Beispiel zeigt den Eintrag für eine Basic-Funktion:

```
<office:scripts>
  <office:event-listeners>
    <script:event-listener script:language="ooo:script" script:event-name="office:new"
    xlink:href="vnd.sun.star.script:Standard.Document.OnOpenDoc?language=Basic&amp;
    location=document" xlink:type="simple"/>
  </office:event-listeners>
</office:scripts>
```

Auch eine an einer Zelle verankerte Schaltfläche in einem Tabellenblatt ist als Ereignisdefinition in „content.xml“ eingetragen. Es handelt sich um das Ereignis „dom:click“. Auch dieses Beispiel zeigt eine Basic-Funktion:

```
<table:table-cell>
  <draw:rect table:end-cell-address="Muster.X8" table:end-x="2.105cm"
  table:end-y="0.074cm" draw:z-index="1" draw:style-name="gr1" draw:text-style-name="P3"
  svg:width="1.94cm" svg:height="0.647cm" svg:x="0.165cm" svg:y="0.323cm">
    <office:event-listeners>
      <script:event-listener script:language="ooo:script" script:event-name="dom:click"
      xlink:href="vnd.sun.star.script:Standard.Document.NewLine?language=Basic&amp;
      location=document"/>
    </office:event-listeners>
    <text:p text:style-name="P2">
      <text:span text:style-name="T1">Neue Zeile</text:span>
    </text:p>
  </draw:rect>
</table:table-cell>
```

Ich versuchte, die Einträge auf die Python-Funktionen abzuändern. Mit den mir verfügbaren Editoren ist es mir aber nicht gelungen, diese Einträge sauber zu editieren, auch nicht im Hex-Modus. Anschließend lief das Zippen in einen Fehler. Die Zuweisungen zu Python-Funktionen konnte ich also nur im geöffneten Dokument vornehmen. Danach sehen sie so aus:

```
....
    <script:event-listener script:language="ooo:script" script:event-name="office:new"
    xlink:href="vnd.sun.star.script:macros.py$init_doc?language=Python&amp;
    location=document" xlink:type="simple"/>
....

......
      <script:event-listener script:language="ooo:script" script:event-name="dom:click"
      xlink:href="vnd.sun.star.script:macros.py$new_row?language=Python&amp;
      location=document"/>
......
```

Die oben zitierten Funktionen benötigen ebenfalls den Parameter für eine übergebene Argumentsequenz, sind also so definiert:

```
def init_doc(*args):
```

```
def new_row(*args):
```

6. Nach dem Projektabschluss

Ich hatte angefangen mit einem Dokument, in dem Basic-Makros die benutzerdefinierten Funktionen bereitstellten. Nach dem Ende des Projekts sind alle Makros durch Python-Funktionen ersetzt. Die Struktur des ausgepackten Dokuments „Kostenabrechnung.ods“ sieht nun so aus:

```
Configurations2
   accelerator
   floater
   images
   menubar
      menubar.xml
   popupmenu
   progressbar
   statusbar
   toolbar
   toolpanel
Dialogs
   Standard
      diverse Dialogdefinitionen als xml und die Metadatei dialog-lb.xml
   dialog-lc.xml
META-INF
   manifest.xml
Scripts
   python
      macros.py
Thumbnails
   thumbnail.png
content.xml
manifest.rdf
meta.xml
mimetype
settings.xml
styles.xml
```

Die Basic-Makros sind entfernt. In den Administrationsdateien sind alle Verknüpfungen auf die Python-Funktionen umgestellt. Ich kann das Dokument nun als Dokumentvorlage einsetzen.

Im Februar 2016