Hartmut Ring, Bernd Jungmann,
Martin Sturm

capella Skript-Programmierung

Handbuch für die Entwicklung von Python-Skripten zur Erweiterung von *capella* und *capella Abo*

capella-software

Dokument-Version	Datum	Änderung zur Vorversion
0.9.1	18.02.25	Kapitel zu mehrsprachigen Plugins ergänzt, Stichwortverzeichnis erweitert, kleine Fehlerkorrekturen
0.9.0	17.02.25	Initialversion

Dieses Handbuch zur Skript-Entwicklung bezieht sich auf folgende Programmversionen

capella Abo, Version 10.0 - 02

capella, Version 10.0

Begründet von Hartmut Ring

Weitergeführt von Bernd Jungmann, Christian Schauß und Markus Hübenthal

Copyright © 1992 – 2009
Hartmut Ring

Copyright © 2010 – 2025
capella-software AG

Grafik-Design
Joscha Ilge
www.joscha-ilge.de

Herausgeber
capella-software AG
Hauptstraße 47
D-34253 Lohfelden
info@capella-software.com
www.capella-software.com

capella ist ein eingetragenes Warenzeichen der capella-software AG

Inhaltsverzeichnis

Einstieg und Überblick....5
Hallo Welt....5
Interne und externe Skriptprogrammierung....5
die capXML-Datenstruktur und das Dateiformat capx....9
Struktur einer capella-Partitur anzeigen....9
Aufbau der XML-Struktur des capXML-Formats....10
interne Skripte....12
Zugriff auf eine geöffnete Partitur....12
Cursorposition und Markierung auswerten....15
Markierung und Cursorposition auswerten mit curSelection()....15
Weitere Auswertungsmöglichkeiten der Cursorposition....17
die Cursorposition setzen mit setSelection()....18
externe Skripte....19
Die Bibliothek caplib....21
capDOM....22
ScoreChange....23
ScoreExport....24
Hilfsfunktionen....25
capSAX....25
CapSaxHandler....26
SaxScoreChange....26
Nutzerinteraktion....27
Mitteilungsfenster / Messageboxen....27
Standarddialoge....27
Individuelle Dialoge mit dem Dialogbaukasten....27
Ausführung eines Skripts unterstützen....31
die Rückgängig-Funktion ermöglichen....31
ein Icon für das Plugin-Menü mitgeben....32
mehrsprachige Plugins....32
Sprach-Kennungen....35
Lokalisierte Hilfedateien für Skripte....36
Parameter speichern und auslesen....36
Grafikobjekte und Grafikkennungen....37

Referenz der internen Programmierschnittstelle....................39
Globale Funktionen....................39
Verzeichnis- und Dateimanagement....................39
capella-spezifische Funktionen....................40
Nutzerkommunikation....................43
allgemeine Hilfsklassen....................43
FileDialog....................43
Rational....................45
Color....................46
RelDiatonicNote....................47
ScriptOptions....................48
Clipboard....................49
MidiOut....................50
Die Klassen des Dialogbaukastens....................51
Dialog....................51
Label....................52
Edit....................52
Radio....................53
ComboBox....................54
CheckBox....................55
HBox....................55
VBox....................56
Die Klasse CapXNode zum Auswerten der XML-Stuktur....................56
Klassen der Partiturelemente....................58
Score....................58
System....................61
Staff....................64
Voice....................67
NoteObj....................67
Head....................72
Grafikobjekte....................72
Gemeinsam für alle Grafikobjekte....................73
Platzierung eines Grafikobjekts....................76
Gruppen von Grafikobjekten....................79
Grafikelemente der Notenschrift....................80
elementare geometrische Formen....................86
importierte Grafiken....................89
Textobjekte....................90
Technische Ergänzungen....................94

Einstieg und Überblick

Mit Skriptprogrammen lassen sich neue Funktionen in *capella* integrieren. Dazu stellt *capella* eine Programmierschnittstelle für die Programmiersprache Python zur Verfügung. Diese Programmierschnittstelle wird in diesem Handbuch beschrieben. Darüber hinaus enthält es Informationen zu der internen Darstellung einer capella-Partitur und dem XML-basierten Datenformat, in dem *capella* Partituren speichert (siehe S. 9).

Hallo Welt

Hier ein minimales Beispiel für ein internes capella-Skript: Es färbt alle Pausen in der Partitur rot:

```
01: # -*- coding: UTF-8 -*-
02: activeScore().registerUndo('Pausen rot färben')
03: for obj in activeScore().noteObjs():
04:     if obj.isRest():
05:         obj.setColor(Color.red)
```

Zeile 1 ist immer nötig, wenn Zeichen außerhalb des ASCII-Bereichs vorkommen (hier das „ä“ in „färben“).

Zeile 2 bewirkt, dass nach dem Ausführen des Skripts im Menü unter Bearbeiten → Rückgängig der Eintrag „Pausen rot färben“ erscheint. Gleichzeitig „merkt“ sich *capella* die Änderungen, die das Skript an der Partitur vorgenommen hat, so dass man mit der Auswahl des Menüeintrags oder der Tastenkombination `Strg`+`Z` alle Anpassungen zurücknehmen kann.

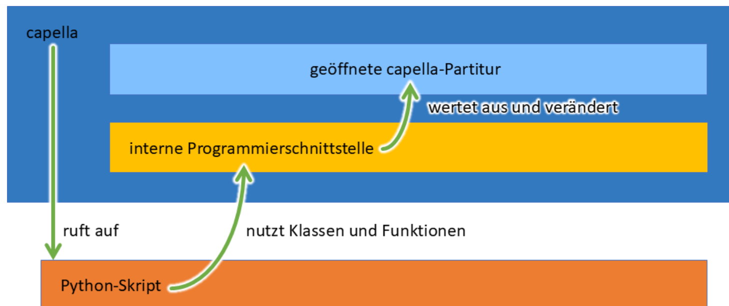
Die eigentliche Funktion des Programms ist in den Zeilen 3-5 kodiert. Details dazu siehe S. 12

Interne und externe Skriptprogrammierung

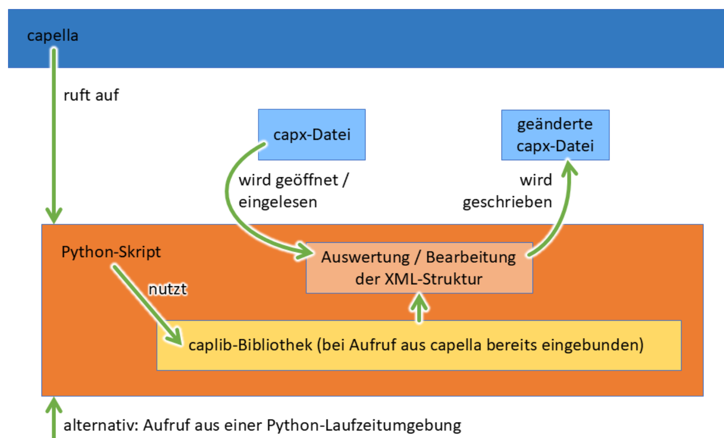
Es gibt zwei Methoden, um eine Partitur mit Hilfe eines Skripts zu manipulieren: durch einen internen Zugriff auf eine geöffnete Partitur oder einen externen Zugriff auf eine abgespeicherte *capella*-Datei.

Im Entwickler-Sprachgebrauch haben sich die Kurzformen „internes Skript“ und „externes Skript“ eingebürgert. Es ist allerdings möglich, dass ein Skript sowohl einen internen als auch externen Zugriff auf eine Partitur vornimmt.

- **interner Zugriff**: Das Skript greift direkt auf die in *capella* geöffnete Partitur zu. Dazu hat man Zugriff auf eine interne Programmierschnittstelle, die auch Programmfunktionen von *capella* zugänglich macht (z.B. den Export von Grafiken). Es können zudem Attribute geändert und Grafikobjekte eingefügt werden. Allerdings erlaubt die Programmierschnittstelle keine Eingriffe in die Struktur einer Partitur (Einfügen bzw. Löschen von Notenobjekten, Stimmen, Zeilen, Systemen usw.). Zum Aufruf eines Skripts mit internem Dateizugriff siehe S. 31. Zu den Funktionen der internen Programmierschnittstelle siehe S. 39.



- **externer Zugriff**: Das Skript operiert auf der Datenstruktur einer auf Festplatte gespeicherten capx-Datei. Ein Skript mit externem Zugriff kann im Prinzip auch ohne *capella* ausgeführt werden. Um die Arbeit mit der Datenstruktur einer capx-Datei zu vereinfachen, stellt die *capella*-Installation die Python-Programmbibliothek caplib mit den Klassen capDOM und capSAX zur Verfügung. Diese basieren auf den verbreiteten Programmierschnittstellen DOM und SAX und enthalten einige *capella*-spezifische Erweiterungen. Siehe S. 21.



Hier ein tabellarischer Vergleich der beiden Zugriffsarten:

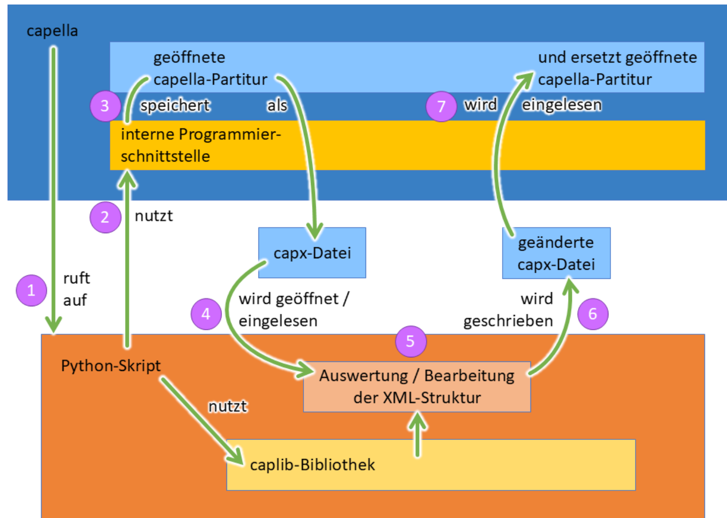
	Skript mit internem Zugriff	**Skript mit externem Zugriff (gestartet aus *capella* heraus)**	**Skript mit externem Zugriff (Start in anderer Laufzeitumgebung)**
Nutzung der internen Programmierschnittstelle	ja	möglich	nein
Änderung der Dokumentenstruktur möglich	nur Ergänzung von Grafikobjekten / Anpassung von Attributen	ja	ja
Nutzung von caplib-Bibliothek zur	nicht nötig, da direkte Bearbeitung des geöffneten Dokuments	ja, caplib-Bibliothek ist automatisch eingebunden	möglich, caplib-Bibliothek muss manuell eingebunden werden
Zugriff auf Dialogbaukasten	ja	ja	nein
Aufruf aus capella heraus	ja	ja	möglich über Aufruf durch ein internes Skript
mögliche Programmiersprache	Python 2.7.13	Python 2.7.13	beliebig (Nutzung von caplib-Bibliothek nur mit Python 2.7.13)

Für alle Skripte, die aus *capella* heraus gestartet werden, stellt die interne Programmierschnittstelle einige Funktionen zur Verfügung, die das Skript für einen Nutzer wie eine interne Programmfunktion erscheinen lassen. Dazu gehören:

- die Rückgängig-Funktion ermöglichen für die Änderungen, die das Skript an der Partitur vornimmt, Details dazu siehe S. 31
- das Abfragen von Optionen über einen Optionsdialog, Details siehe S. 27
- die Möglichkeit, Optionen und Parameter des Plugins zu speichern und bei einer erneuten Ausführung wieder aufzurufen, Details siehe S. 48

- das Abspeichern der aktuell geöffneten Partitur als Datei und das Öffnen einer *capella*-Datei als neue aktuelle Partitur

Mit den letztgenannten Möglichkeiten können auch Skripte mit externem Zugriff so gestaltet werden, dass sie die aktuell geöffnete Partitur bearbeiten und gegenüber dem Nutzer nicht von internen Skripten zu unterscheiden sind:



Mehr zu externen Skripts siehe S. 19

Falls eine andere Programmiersprache als Python genutzt werden soll, ist auch das möglich. Aus einem internen Skript kann man externe Programme aufrufen. Allerdings muss man sich in diesem Fall „um alles selbst kümmern".

Ein Wort zur Python-Version: *capella* nutzt aus Gründen der Abwärtskompatibilität Python in Version 2.7.13. Der Python-Interpreter wird bei der Installation von capella automatisch im Programmverzeichnis installiert und bei jedem Aufruf eines Skripts aus capella heraus aufgerufen. Eine separate Installation von Python ist nicht erforderlich.

die capXML-Datenstruktur und das Dateiformat capx

Frühere Versionen von *capella* nutzten zum Speichern einer Partitur die Dateiformate *.cap und *.all. Beide Binärformate sind mittlerweile nicht mehr gebräuchlich und werden hier nicht näher beschrieben.

Das aktuelle Dateiformat **CapXML** mit der Erweiterung *.capx ist seit Version capella 5 das Standardformat von capella. Es ist ein XML-Format, das von Zeit zu Zeit Erweiterungen erfährt.

Eine CapXML-Datei (Erweiterung .capx) ist ein ZIP-Archiv. Indem man die Dateiendung umbenennt (z.B. von „Partitur.capx“ in „Partitur.zip“), kann man das ZIP-Archiv öffnen und entpacken.

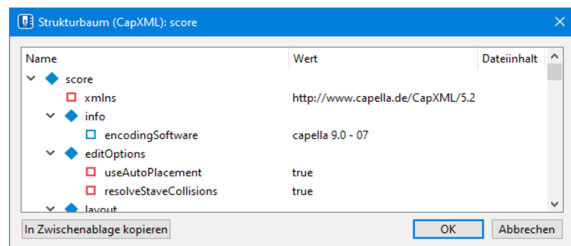
In jedem Fall ist darin die Datei `score.xml` enthalten. Sie enthält die eigentliche Partitur einschließlich aller Format-Parameter.

Daneben können weitere Dateien enthalten sein, auf die in `score.xml` verwiesen wird. Das sind z.B. Grafikdateien für Grafikobjekte und `rtf`-Dateien (Rich Text Format) oder html-Dateien (Hypertext Markup Language) für Textobjekte.

Struktur einer capella-Partitur anzeigen

Mit ANSICHT → STRUKTURBAUM (CAPXML)... können Sie sich die logische Struktur Ihrer aktuellen Partitur anzeigen lassen. Es erscheint eine Baumansicht, die alle CapXML-Informationen in einer übersichtlichen Form zeigt.

Im ersten Eintrag des Strukturbaums kann man die Dateiversion ablesen (hier 5.2). Alternativ können Sie die Datei außerhalb von *capella* öffnen und so die XML-Struktur direkt einsehen (und verändern):



1. Speichern Sie die Partitur im CapXML-Format.

2. Benennen Sie die Erweiterung der gespeicherten Datei von `.capx` in `.zip` um.

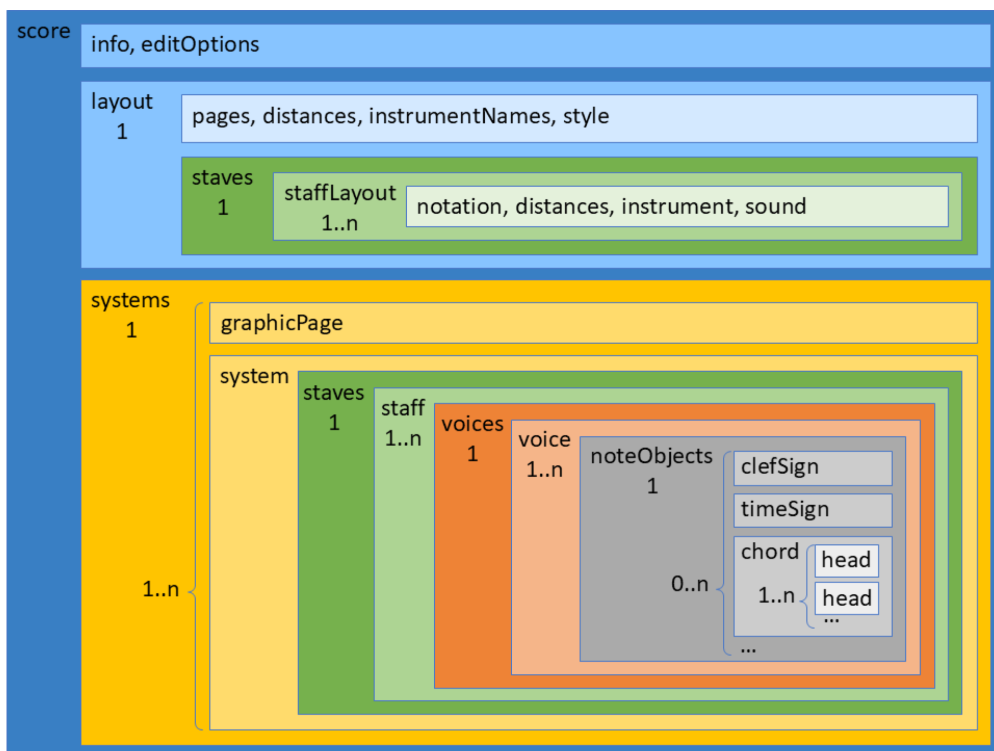
3. Entpacken Sie diese Datei und öffnen Sie die darin enthaltene Datei `score.xml` in einem Text- oder XML-Editor.

4. Wenn Sie Änderungen vornehmen, packen Sie die Datei `score.xml` anschließend zurück in die Zip-Datei, benennen diese wieder um und öffnen sie erneut mit *capella*.

Eine andere Möglichkeit, die `score.xml` außerhalb von capella in einem Texteditor anzuschauen, bietet das Schaltfeld IN ZWISCHENABLAGE KOPIEREN. Aus der Zwischenablage können Sie den Dateiinhalt dann in jeden Texteditor einfügen.

Aufbau der XML-Struktur des capXML-Formats

Die XML-Datei `score.xml` einer capx-Datei hat folgende Struktur (vereinfacht):



In gleicher Weise wird eine Partitur im internen Speicher gehalten, wenn sie in *capella* geöffnet ist. Bei internen und externen unterscheiden sich jedoch die Art, in der auf diese Struktur zugegriffen wird:

- Interne Skripte können die capella-Klassen der internen Programmierschnittstelle nutzen. Siehe S. 12

- Externe Skripte können mit der Bibliothek `capLib` auf die XML-Struktur einer (temporär) gespeicherten capx-Datei zugriefen. Siehe S. 21

In beiden Fällen ist die Kenntnis der Struktur sehr hilfreich. Die Details studiert man am besten im Web: unter der Webadresse

http://www.capella.de/CapXML/

ist die XML-Struktur für alle bisherigen Versionen von CapXML abrufbar: sowohl das XML-Schema (`*.xsd`) als auch eine besser lesbare tabellarische Form (`*.html`).

Einige Hinweise zu den :

- Das Hauptelement ist `<score>`. Neben Datei-Informationen und allgemeinen Parametern hat es den Unterknoten `<layout>` für das Mustersystem. Der Notentext selbst wird im Element `<systems>` abgelegt.
- Jede Partitur folgt der hierarchischen Ordnung: System (`systems/system`) → Zeile (`staves/staff`) → Stimme (`voices/voice`) → Notenobjekt.
- Alle `<system>`-Elemente sind Teil *eines* `<systems>`-Elements. Da jede Partitur mindestens ein System enthält, enthält `<systems>` immer mindestens ein Element `<system>`.

  Analog ist die Beziehung zwischen `<staff>` und `<staves>`, `<voice>` und `<voices>` sowie `<head>` und `<chord>`.
- Eine Ausnahme ist das Element `<noteObjects>`. Dessen Unterelemente können verschiedene Ausprägungen haben:
  - `<clefSign>` – ein Notenschlüssel
  - `<keySign>` – eine Tonartvorzeichnung
  - `<timeSign>` – eine Taktangabe
  - `<barline>` – ein fester Taktstrich
  - `<chord>` – eine Note bzw. ein Akkord
  - `<rest>` – eine Pause

  Zudem kann das Element `<noteObjects>` auch leer sein (im Gegensatz zu `<systems>`, `<voices>` etc.).
- Grafik-Elemente (z.B. Artikulationszeichen) werden an Noten/Akkorde, Pausen, expliziten Taktstrichen oder die Seite als Ganzes gehängt. Details siehe S. 37
- Liedtexte werden für jede einzelne Note/jeden Akkord separat in den Unterelementen `<lyric>` und dessen Unterelement `<verse>` abgelegt.

interne Skripte

Wenn Sie aus *capella* heraus ein Skript starten, stehen Ihnen neben allen Möglichkeiten der Sprache Python zusätzliche Klassen und Funktionen zur Verfügung. Damit können Sie in capella geöffnete Partituren direkt bearbeiten.

In diesem Kapitel werden typische Problemstellungen behandelt. Eine vollständige Darstellung aller Funktionen und Klassen der internen Programmierschnittstelle siehe S. 39

Ein Skript mit internem Dateizugriff, das die aktuell geöffnete Partitur analysieren oder verändern soll, folgt typischerweise diesem Programmablauf:

Programmschritt
Parameter laden (sofern das Skript persistente Parameter verwendet, die von Ausführung zu Ausführung übernommen werden), siehe S. 48
bei Bedarf mit Optionen mit einem Dialog erfragen, siehe S. 27
sofern die Partitur geändert werden soll: Bearbeitung in der Rückgängig-Funktion registrieren, siehe S. 31
Partitur auswerten bzw. manipulieren (unter Nutzung der Klassen und Funktionen der internen Programmierschnittstelle)
ggf. Parameter speichern und/oder Rückmeldung an den Nutzer

Zugriff auf eine geöffnete Partitur

Auf die XML-Struktur einer geöffneten capx-Datei kann auf zwei verschiedene Arten zugegriffen werden:

Variante 1) ausschließlich lesend mit der Klasse CapXNode

Diese Klasse erlaubt es, sämtliche Elemente des XML-Baums auszuwerten, auch die, die nicht über die sonstigen Klassen und Funktionen der internen Programmierschnittstelle zugänglich sind. Dazu zählen z.B. das Mustersystem oder Grafikseiten. Sie können allerdings nicht verändert werden. Näheres siehe S. 56

Variante 2) lesend und z.T. schreibend mittels der Objekthierarchie der internen Programmierschnittstelle

Eine geöffnete Partitur wird dabei von der Klasse `Score` repräsentiert. Ein Objekt dieser Klasse kann mit der globalen Funktion `activeScore()` gewonnen werden.

Für die Gliederungsebenen darunter stehen spezialisierte Klassen zur Verfügung. Ihre Namen orientieren sich an der XML-Struktur: `System`, `Staff`, `Voice`, `NoteObj`, `Head`.

Jede dieser Klassen bietet Methoden, um auf die darunterliegende Objektebene zuzugreifen:

- Man kann die **Anzahl der direkten Unterobjekte** ermitteln. Z.B. kann man in der Klasse `System` mit der Methode `nStaves()` herausfinden, wie viele Notenzeilen das vorliegende System hat.
- Es gibt jeweils eine Methode zum **wahlfreien Zugriff auf direkte Unterobjekte**, mit der man zu einem Index das zugehörige Unterobjekt übergeben bekommt. Dabei steht der Index `0` für das erste Unterelement. Z.B. gibt in der Klasse `Voice` die Methode `noteObj(0)` das erste Notenobjekt der Stimme zurück.
- Umgekehrt kann man mit `index()` den **Index ermitteln**, unter dem das aktuelle Objekt in der darüberlegenden Objektebene angesprochen werden kann. Ruft man `index()` für ein `Voice`-Objekt, erhält man eine Integerzahl `i`. Ruft man im übergeordneten `Staff`-Objekt die Methode `voice(i)` auf, erhält man die ursprüngliche `Voice`-Objekt.
- Will man nicht auf die direkten Unterobjekte sondern auf alle Unterobjekte eines bestimmten Typs zugreifen, gibt es dafür spezielle Methoden, die einen passenden **Iterator** zurückgeben. Z.B. liefert die Methode `heads()` immer einen Iterator über alle Notenköpfe zurück. Ruft man die Methode `heads()` für ein `Staff`-Objekt, dann erhält man einen Iterator über alle Notenköpfe in der entsprechenden Notenzeile. Ruft man `heads()` dagegen für ein NoteObj-Objekt, erhält man einen Iterator über alle Notenköpfe des Notenobjekts (sofern es sich dabei um einen Akkord handelt; für alle anderen Notenobjekte erhält man in diesem Fall einen leeren Iterator).

Hier ein Überblick über die verschiedenen Zugriffsfunktionen:

Partitur-element	Partitur	System	Notenzeile	Stimme	Noten-objekt	Noten-kopf
Klasse	`Score`	`System`	`Staff`	`Voice`	`NoteObj`	`Head`
Anzahl der direkten Unter-elemente ermitteln	`nSystems()`	`nStaves()`	`nVoices()`	`nNoteObjs()`	`nHeads()`	-
Wahlfreier Zugriff mit Index i	`system(i)`	`staff(i)`	`voice(i)`	`noteObj()`	`head(i)`	-
Index innerhalb der über-geordneten Struktur	-	`index()`	`index()`	`index()`	`index()`	`index()`

Iterator für Unter-objekte	`Systems()`
	`staves()`
	`voices()`
	`noteObjs()`
	`heads()`
		`staves()`
		`voices()`
		`noteObjs()`
		`heads()`
			`voices()`
			`noteObjs()`
			`heads()`
				`noteObjs()`
				`heads()`
					`heads()`	-

Referenz der weiteren Methoden	S. 58	S. 61	S. 64	S. 67	S. 67	S. 72

Bei vielen dieser Partitur-Klassen gibt es eine Reihe weiterer Methoden, die für Standard-Aufgaben elegante Lösungen ermöglichen. Daher ist ein Blick in die Klassenreferenz in jedem Fall ratsam, siehe Seite S. 58

Cursorposition und Markierung auswerten

Häufig möchte man mit einem Skript nur an einer bestimmten Stelle der Partitur Veränderungen durchführen. Dazu wäre es hilfreich, auf die Cursorposition zugreifen zu können, bzw. auf den Abschnitt, der beim Aufruf des Skripts markiert war. Für interne Skripte stellt die Programmierschnittstelle mehrere Funktionen zur Verfügung:

Markierung und Cursorposition auswerten mit `curSelection()`

Mit der Funktion `curSelection()` lässt sich die Position des Cursors bzw. des markierten Bereichs detailliert auswerten. Sie gibt ein Tupel aus zwei Cursorpositionen zurück:

```
(<Cursorposition_1>, <Cursorposition_2>)
```

Eine Cursorposition ist ein Tupel aus vier Integer-Zahlen:

```
(indexSystem, indexStaff, indexVoice, indexNoteObject)
```

Die einzelnen Zahlen sind die Indices des Notenobjekts, vor dem der Cursor steht. Die Cursorposition `(0, 1, 2, 3)` steht für das erste System, die zweite Zeile, die dritte Stimme und dort das vierte Notenobjekt. Siehe auch S. 12

Achtung: Beim letzten Index (für das Notenobjekt) muss man beachten, dass der Cursor auch hinter dem letzten Notenobjekt stehen kann. Es wird trotzdem ein Integerwert als Index für das Notenobjekt geliefert, und zwar um 1 höher als der Index des letzten tatsächlich vorhandenen Notenobjekts.

Im Zusammenspiel der beiden zurückgegebenen Cursorpositionen gibt es mehrere Konstellationen, die man im Blick behalten muss:

```
<Cursorposition_1> == <Cursorposition_2>
```

In diesem Fall steht der Cursor in einer einzelnen Stimme vor einem Notenobjekt (oder am Ende der Stimme), ohne dass ein Bereich markiert ist oder sich der Cursor über mehrere Stimmen/Zeilen erstreckt.

Im Beispiel rechts liefert `curSelection()` den Rückgabewert `((0, 1, 0, 6), (0, 1, 0, 6))` In diesem Fall kann man die Funktionen `cursorObj()`, `curKey()` und `pitches(...)` nutzen. Siehe folgender Abschnitt.

`<Cursorposition_1> != <Cursorposition_2>`

In diesem Fall sind zwei Konstellationen denkbar:

1. Der Cursor steht senkrecht in mehreren Stimmen bzw. Zeilen.

Im Beispiel rechts liefert `curSelection()` den Rückgabewert

```
((0, 0, 0, 3),
 (0, 1, 0, 6))
```

Ohne eine genauere Betrachtung ist diese Konstellation nicht von der nächsten zu unterscheiden:

2. Es ist ein Bereich ausgewählt.

Im Beispiel rechts liefert `curSelection()` den Rückgabewert

```
((0, 0, 0, 3),
 (0, 1, 0, 8))
```

Auch hier bedarf es einer genauen Betrachtung, z.B. um herauszufinden, welche Notenobjekte in der Markierung liegen. In den folgenden Beispielen sind immer die gleichen Notenobjekte ausgewählt, allerdings unterscheiden sich die Cursorpositionen, mit denen die Markierung aufgezogen wurde – und deshalb auch die Rückgabewerte:

Rückgabewert:

```
((0, 1, 0, 6),
 (0, 0, 0, 4))
```

Rückgabewert:

```
((0, 1, 0, 8),
 (0, 0, 0, 3))
```

Welche weitere Auswahl-Konstellation ist möglich? Und wie sieht dann der Rückgabewert aus?

Weitere Auswertungsmöglichkeiten der Cursorposition

Die folgenden Funktionen und Methoden funktionieren nur dann, wenn der Cursor vor einem einzelnen Notenobjekt steht und keinen Bereich markiert.

- `cursorObj()` gibt das Notenobjekt rechts vom Cursor zurück. Siehe S. 41
- `curKey()` ist eine Methode der Klasse `NoteObj` und gibt die zu Beginn des Notenobjekts geltende Tonart an. Mit dem Aufruf

  ```
  cursorObj().curKey()
  ```

  erhält man die an der Cursorposition gültige Tonartenvorzeichnung (sofern sichergestellt ist, dass der Cursor nicht am Ende einer Zeile steht. Siehe S. 70
- `pitches(...)` ist eine Methode der Klasse `System`. Sie liefert eine Liste aller Tonhöhen des Systems (über alle Stimmen) an der angegebenen Zeit-Position. Siehe S. 63

Beispiel für die Funktionen:

```
sel = curSelection()
if sel[0] != sel[1]:
    messageBox('Fehler', 'Markierung ist nicht leer.')
else:
    cur = sel[0]
    sys = activeScore().system(cur[0])
    staff = sys.staff(cur[1])
    voice = staff.voice(cur[2])
    if cur[3] == voice.nNoteObjs():
        messageBox('Fehler',
                   'Cursor steht hinter letztem Notenobjekt.')
    else:
        obj = cursorObj() #identisch: obj=voice.noteObj(cur[3])
        key = obj.curKey()
        ausgabe = 'An der Cursorposition gelten '
        if key < 0:
            ausgabe += str(-key) + ' b als Vorzeichen.\n'
        elif key > 0:
            ausgabe += str(key) + ' # als Vorzeichen.\n'
        else:
            ausgabe += 'keine Vorzeichen.\n'
        pitches = sys.pitches(obj.time(), True)
        ausgabe +=  'Klingende Töne: ' + str(pitches)
        messageBox('Info', ausgabe)
```

die Cursorposition setzen mit `setSelection()`

Das Gegenstück zu `curSelection()` ist `setSelection(set)`. Damit kann man den markierten Abschnitt innerhalb der Partitur setzen. Als Parameter erwartet es ein Tupel aus zwei Cursorpositionen in der gleichen Struktur, wie sie `curSelection()` zurückliefert.

externe Skripte

Ein Skript mit externem Dateizugriff, das die aktuell geöffnete Partitur verändern soll, folgt typischerweise diesem Programmablauf:

Programmschritt	Zugriffsart
Parameter laden (sofern das Skript persistente Parameter verwendet, die von Ausführung zu Ausführung übernommen werden), siehe S. 48	interne Programmierschnittstelle
bei Bedarf mit Optionen mit einem Dialog erfragen, siehe S. 27	
Bearbeitung in der Rückgängig-Funktion registrieren, siehe S. 31	
Partitur in eine temporäre capx-Datei speichern, siehe S. 58	
Temporäre Datei einlesen und manipulieren	externer Zugriff mittels caplib-Bibliothek, siehe S. 21
Ergebnis in temporäre capx-Datei speichern	
temporäre Datei wieder in *capella* einlesen, siehe S. 59	interne Programmierschnittstelle
ggf. Parameter speichern und/oder Rückmeldung an den Nutzer	

Da dieser Ablauf auf sehr viele Anwendungsfälle anwendbar ist, hier ein Skript-Grundgerüst, das alle diese Elemente enthält, allerdings ohne jede Funktion ist. Es kann als Ausgangspunkt für eigene Skripte dienen.

Hinweis: Dieses Skript verstößt in dieser inhaltsleeren Form gegen die Empfehlung, die Skriptausführung nur dann für die Rückgängig-Funktion zu registrieren, wenn tatsächlich eine Änderung der Partitur erfolgt.

```
from caplib.capDOM import ScoreChange
import tempfile, os

# Definition der externen Verarbeitung
class noChange(ScoreChange):
    def changeElement(self, el):
        pass

if activeScore():
    # Parameter auslesen
    optFile = ScriptOptions()
    opt = optFile.get()

    # Optionen mit Dialog erfragen
    label = Label('Alles bleibt, wie es ist.')
    dlg = Dialog('nichts tun', label)

    if dlg.run():

        # Rückgängig-Funktion registrieren
        activeScore().registerUndo('tue nichts')

        # aktive Partitur in temporäre Datei speichern
        tempFile1 = tempfile.mktemp('.capx')
        tempFile2 = tempfile.mktemp('.capx')
        activeScore().write(tempFile1)

        # die externe Verarbeitung anstoßen
        noChange(tempFile1, tempFile2)

        # Ergebnis in aktuelle Partitur laden und aufräumen
        activeScore().read(tempFile2)
        os.remove(tempFile1)
        os.remove(tempFile2)

        # Rückmeldung und Parameter sichern
        messageBox('nichts getan', 'Es ist alles beim alten.')
        optFile.set(opt)
```

Die Bibliothek caplib

Die Bibliothek caplib unterstützt das Öffnen, Bearbeiten und Speichern von capx-Dateien und bietet sich damit für die Erstellung von Skripten mit externem Dateizugriff an.

> Hinweis zum Einbinden der caplib-Bibliothek: Beim Aufruf eines Skripts aus *capella* heraus ist der Pfad der caplib-Bibliothek bereits in der Laufzeitumgebung bekannt gemacht. Andernfalls muss man zunächst den Dateispeicherort der caplib-Bibliothek einbinden mit:
>
> ```
> import sys
> sys.path.append(r'c:\Programme\...\capella\py-ext')
> ```
>
> Der Pfad muss dabei entsprechend der vorhandenen *capella*-Installtion angepasst werden.

Zunächst ein Beispiel (das alle Bindebögen in der Partitur `beispiel.capx` in gestrichelte Bindebögen umwandelt und das Ergebnis in der Datei `beispiel-neu.capx` speichert):

```
from caplib.capDOM import ScoreChange

class SlurChange (ScoreChange):
    def changeElement(self, el):
        if el.tagName == 'slur':
            form = self.getFirstChildElement(el, 'form')
            if form == None:
                form = self.doc.createElement('form')
                el.appendChild(form)
            else: form = form[0]
            form.setAttribute('endWidth', '0.0625')
            form.setAttribute('midWidth', '0.0625')
            form.setAttribute('dotDist', '1')
            form.setAttribute('dotWidth', '0.75')

SlurChange(r'c:\...\beispiel.capx',
           r'c:\...\beispiel-neu.capx')
```

Die Klasse `SlurChange` wird von `ScoreChange` abgeleitet, wobei eine einzige Methode überschrieben wird: `changeElement(...)`.

Die Hauptarbeit wird durch den abschließenden Konstrutor-Aufruf von `SlurChange` angestoßen. Dabei wird als erster Parameter eine Ausgangsdatei und als zweiter Parameter eine Zieldatei übergeben. Die Ausgangsdatei muss existieren, sonst wird ein Feh-

ler geworfen; die Zieldatei wird entweder neu erzeugt oder ggf. ohne weitere Rückfrage überschrieben.

Alles weitere bringt die Basisklasse `ScoreChange` bereits mit:

Der Konstruktor…

- …öffnet die übergebene capx-Datei
- …entpackt das enthaltene ZIP-Archiv
- …liest die enthaltene Datei `score.xml` ein und baut die XML-Struktur im Hauptspeicher auf.
- …durchläuft den gesamten XML-Baum und ruft für jedes Element die Methode `changeElement(...)`
- …schreibt anschließend den (ggf. veränderten) XML-Baum in eine (neue) `score.xml`-Datei, packt sie und legt sie unter dem Namen der Zieldatei auf der Festplatte ab.

Die überschriebene Methode `changeElement(...)` muss lediglich prüfen, ob das Element, für das sie aufgerufen wurde, ein Bindebogen ist und ggf. die Attribute so anpassen, dass er gestrichelt erscheint.

Im Folgenden werden die relevanten Klassen und Funktionen der caplib-Bibliothek kurz beschrieben. Sie gliedern sich in zwei Gruppen, die den beiden Grundmustern im Umgang mit XML-strukturierten Daten entsprechen.

- SAX (Simple API for XML): Hier wird die XML-Struktur lediglich durchlaufen und direkt geschrieben. D.h. Änderungen an der Struktur können nur in diesem einen Durchlauf durch alle Elemente vorgenommen werden. Der Vorteil gegenüber DOM ist ein geringerer Speicherbedarf und eine höhere Verarbeitungsgeschwindigkeit.

Details zu diesen beiden De-facto-Standards sind in der Python-Dokumentation zu finden. Davon sind die beiden Module `capDOM` und `capSAX` abgeleitet, die Teil der caplib-Bibliothek sind. Sie sind in den beiden Dateien `capDOM.py` bzw. `capSAX.py` definiert, die im capella-Programmverzeichnis im Unterordner `\bin\py-ext\caplib` zu finden sind. Es empfiehlt sich, ihre Funktion und ihren Ablauf nachzuvollziehen.

capDOM

Das Document Object Model (DOM) ist eine Zugriffsart auf XML-Daten, bei dem zunächst die komplette XML-Struktur (in diesem Fall die Partitur) eingelesen und als XML-

Baum im Hauptspeicher abgelegt wird. Dadurch ist es möglich, sich vom Root-Knoten (score) durch die verschiedenen Ebenen der Baumstruktur zu bewegen und Änderungen/Ergänzungen dieser Struktur vorzunehmen.

ScoreChange

Klasse, die den externen Dateizugriff auf eine capx-Datei auf Basis des minidom-Frameworks kapselt.

`ScoreChange``(inputFile, outputFile='')`

Im Konstruktor wird der Pfad zu einer capx-Datei übergeben. Wird ein zweiter Dateipfad mitgegeben, wird das Ergebnis der Verarbeitung in dieser Datei gespeichert. Andernfalls wird die Eingabedatei überschrieben.

Der Konstruktor übernimmt die komplette Ablaufsteuerung. D.h., sobald man die Klasse initialisiert, wird eine Kette von Methodenaufrufen angestoßen:

1. **`changeScore`**`(self, score)`

   Diese Methode wird gerufen, wenn der Aufbau des DOM-Baums abgeschlossen ist. Die komplette XML-Struktur ist im Parameter score enthalten (im Format xml.dom.minidom.Document). Durch Überschreiben der Methode kann diese Strutur mit den Mitteln des minidom-Frameworks verändert werden. Details dazu sind in der Pyhton-Domumentation zu finden.

   Wird die Methode nicht überschrieben, bleibt die XML-Struktur unangetastet.

2. Eine Veränderung einzelner Partiturelemente ist im Zuge des anschließenden rekursiven Durchlaufs aller Elemente des XML-Baums möglich. Für jedes Element (Knoten genauso wie Blätter) wird die folgende Methode aufgerufen:

   `changeElement``(self, el)`

   Dabei enthält `el` den XML-Strukturbaum des Elements inkl. aller Kinder-Knoten. Durch die Abfrage von `el.tagName` kann die Methode feststellen, für welchen Elementtyp sie aufgerufen wurde und entsprechend reagieren. Wird diese Methode nicht überschrieben, erfolgt keine Änderung an der XML-Struktur.

Die Klasse `ScoreChange` bringt noch zwei Hilfsmethoden mit, die beim Durchsuchen der XML-Struktur und beim Ändern von Attributen nützlich sein können:

`getFirstChildElement``(el, childTag)`

liefert das erste Kind-Element von `el` mit dem Namen `childTag`

`removeAtt``(el, att)`

entfernt aus dem Element `el` das Attribut mit den Namen `att`

Alle weiteren Methoden des `minidom`-Frameworks stehen natürlich ebenfalls zur Verfügung.

ScoreExport

Die Klasse ScoreExport kapselt einen externen Dateizugriff auf eine capx-Datei auf Basis des minidom-Frameworks. Sie bietet im Gegensatz zu ScoreChange etwas mehr Komfort, beschränkt sich aber allein auf eine Auswahl an Partitur-Elementen:

Es gibt keine changeScore-Methode. Statt einer unspezifischen changeElement-Methode gibt es für jeden inneren Knoten der capXML-Struktur (score, system, staff, voice, noteObj, chord) jeweils zwei Methoden, die beim Eintreten bzw. Verlassen dieses Knotens aufgerufen werden.

Zusätzlich gibt es für jedes Blatt des XML-Baumes (clefSign, keySign, timeSign, rest, head) eine (funktionslose) Handle-Funktion, die aufgerufen wird, wenn dieses Blatt erreicht ist.

D.h. es werden in dieser Reihenfolge beispielhaft aufgerufen:

```
startScore(score)
  startSystem(system1)
    startStaff(staff1)
      startVoice(voice1)
        startNoteObj(clefSign1)
          handleClefSign(clefSign1)
        finishNoteObj(clefSign1)
        startNoteObj(keySign1)
          handleKeySign(keySign1)
        finishNoteObj(keySign1)
        startNoteObj(timeSign1)
          handleTimeSign(timeSign1)
        finishNoteObj(timeSign1)
        startNoteObj(chord1)
          startChord(chord1)
            handleHead(head1)
          finishChord(chord1)
        finishNoteObj(chord1)
      finishVoice(voice1)
    finishStaff(staff1)
  finishSystem(system1)
finishScore(score)
```

Dieser Ablauf ist dann hilfreich, wenn es lediglich um die Auswertung bzw. den Export der Partiturdaten geht. Dazu bringt ScoreExport noch zwei Hilfsmethoden mit:

midiPitch(head)

erwartet als Parameter eine minidom-Struktur, die einen Notenkopf repräsentiert und liefert die zugehörige MIDI-Tonhöhe als Integer-Zahl zurück.

duration(note)

erwartet die minidom-Struktur einer Pause (rest) oder eines Akkords (chord) und liefert die Dauer des Objekts als Rational-Objekt zurück.

Hilfsfunktionen

Neben den beiden Klassen ScoreChange und ScoreExport liefert capDOM noch zwei klassenunabhängige Funktionen:

firstChildElement(el, childTag)

ist funktionsgleich zur Methode getFirstChildElement der Klasse ScoreChange, funktioniert aber auch außerhalb dieser Klasse. Es erwartet als Parameter el eine minidom-Struktur und liefert die minidom-Struktur des ersten Kind-Elements mit dem Namen childTag zurück.

childElements(el, childTag)

liefert für ein übergebenes Element el ein Dictionary mit allen Kind-Elementen von el mit dem Namen childTag.

capSAX

Das SAX-Framework verfolgt die Zielsetzung, dass zu keinem Zeitpunkt die vollständige XML-Struktur geladen wird. Vielmehr wird eine XML-Datei wie eine Text-Datei sukzessive gelesen und verarbeitet.

Hier ein Beispielskript, das exakt die gleiche Aufgabe vollzieht wie das Eingangsbeispiel zur caplib-Bibliothek allgemein, siehe S. 21

```
from caplib.capSAX import CapSaxHandler, SaxScoreChange

class SlurSaxHandler(CapSaxHandler):
    def enhanceElement(self, name):
        if name == 'slur':
            self.outFile.write('><form')
            self.outFile.write(' endWidth="0.0625"')
            self.outFile.write(' midWidth="0.0625"')
            self.outFile.write(' dotDist="1"')
            self.outFile.write(' dotWidth="0.75"')
            self.outFile.write('/>')
```

```
            self.openStartTag = False

SaxScoreChange(SlurSaxHandler,
               r'c:\...\beispiel.capx',
               r'c:\...\beispiel-neu.capx')
```

Die Pfadangaben im Aufruf von `SaxScoreChange` müssen entsprechend agepasst werden. Die beiden relevanten Klassen des capSAX-Moduls sind:

CapSaxHandler

Basisklasse, die nichts weiter tut, als ein outFile zu schreiben – eine Textdatei, die wohlgeordnet alle XML-Tags enthält, die eine XML-Struktur beschreiben. Einzige (sinnvolle) Eingriffsmöglichkeit ist die Methode

`enhanceElement``(self, name)`

Sie wird aufgerufen, wenn für ein Element des XML-Baumes das öffnende Tag bis auf das schließende '>' bzw. '/>' geschrieben wurde. Beispiel: Wird die Methode für das info-Tag einer capx-Datei aufgerufen, steht zu diesem Zeipunkt in `outFile` bereits:

```
<score xmlns="http://www.capella.de/CapXML/5.3">
<info
```

Wird `enhanceElement(...)` nicht überschrieben, stellen die übrigen Methoden von `capSaxHandler` sicher, dass das `info`-Tag noch geschlossen wird und alle weiteren Tags sauber nach `outFile` geschrieben werden.

Greift man ein, kann man durch direktes Schreiben nach `outFile` die XML-Struktur verändern, muss sich dabei aber an das XML-Format halten und selbst dafür sorgen, dass die XML-Struktur wohlgeformt bleibt.

Im Beispiel oben geschieht das immer dann, wenn es sich um ein `slur`-Element handelt: Zunächst wird das schließende '>' für das Start-Tag geschrieben. Anschließend werden das Kinder-Element `<form.../>` ergänzt. Abschließend muss die Variable `openStartTag` auf gesetzt werden, falls das Element, das man gerade bearbeitet hat, Kinder-Elemente hat. Daran erkennt CapSaxHandler, ob es das begonnene öffnende Tag (im Beispiel '`<slur`') mit einem '/>' abschließen muss.

SaxScoreChange

Dem Konstruktor dieser Klasse ist lediglich der von `CapSaxHandler` abgeleitete individuelle Handler zu übergeben, sowie die Eingabe- und ggf. Ausgabe-Datei (siehe Beispiel oben). Alles weitere übernimmt die Klasse selbst. Wird keine Ausgabe-Datei übergeben, wird die Eingabedatei überschrieben.

Nutzerinteraktion

Mitteilungsfenster / Messageboxen

Die einfachste Art, aus einem Skript heraus mit dem Nutzer zu kommunizieren ist eine Messagebox. Sie erlaubt es, einen kurzen Text auszugeben. Das ist praktisch, um Ergebnisse oder Fehlermeldungen auszugeben und hilft beim Debuggen eines Skripts. Die Abfrage von Parametern ist damit aber nicht möglich. Details siehe S. 43

Standarddialoge

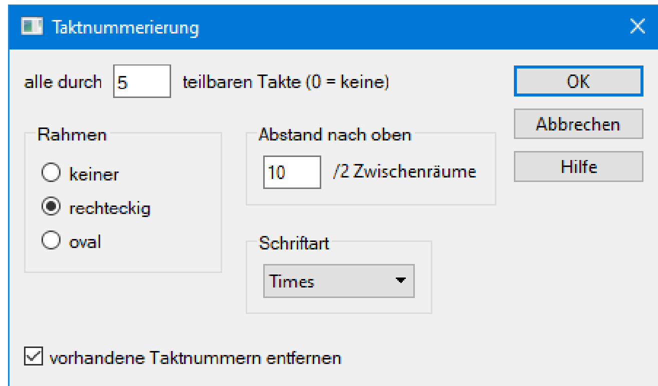
Einige häufig benötigte Dialoge stellt das Betriebssystem in standardisierter Form bereit. Das spart Programmierarbeit und bietet Nutzern eine vertraute Bedienung. Die interne Programmierschnittstelle erlaubt die Nutzung des Datei-Dialogs des jeweiligen Betriebssystems, der zudem noch angepasst werden kann (z.B. um den Filter für bestimmte Dateiendungen zu setzen. Details siehe S. 43

Individuelle Dialoge mit dem Dialogbaukasten

Zur Abfrage von spezifischen Optionen und Parametern ist ein Dialogfenster das Mittel der Wahl. Zu Python gehört die Bibliothek Tkinter, mit der sich u. a. Dialoge programmieren lassen. Abgesehen vom nicht ganz einfachen Einstieg in Tkinter gibt es ein Problem, das Anwender leicht verwirren kann: Ein Tkinter-Dialog ist ein eigener Prozess: Er hat einen eigenen Eintrag in der Taskleiste und kann unabhängig vom capella-Fenster in den Hintergrund befördert werden. Außerdem ist er durch sein Erscheinungsbild als Fremdkörper erkennbar.

Deshalb gibt es einen direkt in *capella* eingebetteten Dialogbaukasten. Beim Gestalten eines Dialogs brauchen Sie sich nicht um Maße und Positionen der einzelnen Dialogelemente zu kümmern. Stattdessen gruppieren Sie diese einfach in (beliebig verschachtelbare) horizontale oder vertikale Rahmen (die auch unsichtbar sein können) und der Dialogbaukasten berechnet automatisch die passende Anordnung.

Zunächst ein Beispieldialog:



Dieser Dialog lässt sich mit folgendem Code erstellen und auswerten:

```
#--- Dialogelemente
erstellen ---
label1 = Label ('alle durch')
edit1 = Edit ('5', min=0, max=50, width=4)
label2 = Label ('teilbaren Takte (0 = keine)')
radio = Radio (['keiner','rechteckig','oval'],
               text='Rahmen', value=1)
edit2 = Edit ('10', min=-20, max=20, width=4)
label3 = Label ('/2 Zwischenräume')
combo = ComboBox(['Arial', 'Times', 'Courier'],
                 width=12, value=1)
check = CheckBox ('vorhandene Taktnummern entfernen', value=1)

#--- Anordnung festlegen ---
hBox1 = HBox ([label1, edit1, label2], padding=8)
hBox2 = HBox ([edit2, label3],
              padding=8, text='Abstand nach oben')
hBox3 = HBox ([combo], padding = 8, text='Schriftart')
vBox1 = VBox ([hBox2, hBox3], padding = 16)
hBox = HBox ([radio, vBox1], padding=16)
vBox = VBox ([hBox1, hBox, check], padding=16)

#--- Dialog erstellen, ausführen und auswerten ---
dlg = Dialog('Taktnummerierung', vBox, 'noch kein Hilfetext!')
if dlg.run():
    step = int(edit1.value())
    y = int(edit2.value())
    form = radio.value()
    clear = check.value()
    font = combo.value()

# ... und jetzt etwas mit den Werten anfangen...
```

Im Abschnitt „Dialogelemente erstellen“ wird eine Reihe von Objekten instanziiert. Jedes dieser Objekte steht für ein Element des Dialogs:

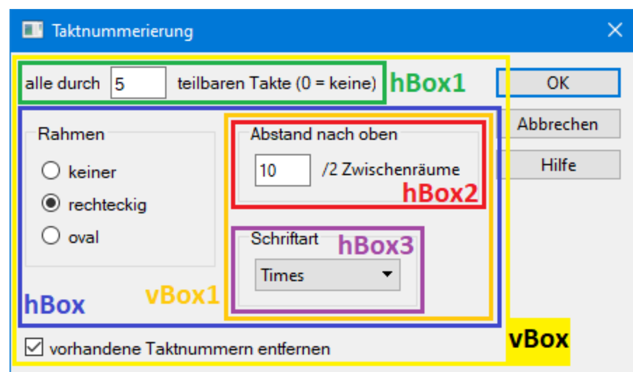
- Texte (`Label`)
- Eingabefelder (`Edit`)
- Radiobuttons (`Radio`)
- Aufklappfelder (Comboboxen)
- Checkboxen (`Checkbox`)

Zur Referenz der einzelnen Dialogelemente siehe S. 52

Im folgenden Abschnitt „Anordnung festlegen“ werden die Lagebeziehungen unter diesen Elementen festgelegt. Dazu nutzt man Rahmen: Darin lassen sich eine beliebige Menge von Elementen anordnen. Es stehen zwei Typen von Rahmen zur Verfügung:

- VBox: Die Elemente innerhalb des Rahmens stehen untereinander.
- Hbox: Die Elemente innerhalb des Rahmens stehen nebeneinander.

Mit dem Parameter `padding` (der auch weggelassen werden kann) lässt sich festlegen, wie groß der Abstand der Elemente innerhalb des Rahmens untereinander ist.



Wird ein Text mitgegeben (Parameter `text`), dann wird ein sichtbarer Rahmen um die Elemente gezeichnet, auf dem der übergebene Text angezeigt wird (z.B. bei `hBox2` und `hBox3`).

Wird kein Text mitgegeben, ist der Rahmen unsichtbar (z.B. bei `hBox1`, `hBox` und `vBox`).

Rahmen können auch um einzelne Elemente gezeichnet werden (z.B. `hBox3`).

Umgekehrt lassen sich Rahmen auch als Element in einen weiteren Rahmen einbinden. So sind im alles umschließenden Rahmen `vBox` drei Elemente angeordnet: `hBox1`, `hBox` und eine `Checkbox`. Durch eine solche Verschachtelung lassen sich auch komplexe Dialoge konstruieren.

Im letzten Abschnitt „Dialog erstellen“ wird ein Objekt der Klasse `Dialog` instanziiert. Dabei ermittelt *capella* ohne weiteres Zutun die Maße und das Aussehen des vollständigen Dialogs. Mit der Methode `run()` wird der Dialog angezeigt. Der Rückgabewert ist `True`, sofern der Dialog mit OK verlassen wurde (und `False`, falls er abgebrochen wird).

Anschließend können die einzelnen Elemente mit der Methode `value()` ausgelesen werden. Tipp: Wenn Parameter abgefragt werden, die bei einem erneuten Aufruf des Skripts vom Nutzer wahrscheinlich gleich oder ähnlich eingegeben werden, dann empfiehlt es sich, die Parameter zwischenzuspeichern und vor einem erneuten Anzeigen des Dialogs zu laden. Wie das geht? Siehe S. 36

Ausführung eines Skripts unterstützen

Mit dem Menüeintrag PLUGINS → PLUGIN... oder der Tastenkombination `Strg`+`F3` kann der Plugin-Auswahl-Dialog gestartet werden. Hat man ein neues Skript erstellt, wählt man auf „Anderes Plugin…“ und sucht im Dateibrowser die zugehörige *.py-Datei. Mit einem Klick auf Ausführen wird das Skript gestartet.

Will man das Skript wiederholen, steht dafür die Tastenkombination `Strg`+`R` und der Menüeintrag PLUGINS → PLUGIN WIEDERHOLEN zur Verfügung.

Darüber hinaus tauchen einmal ausgeführte Skripte im Plugin-Auswahl-Dialog im Reiter „Eigene“ auf und können zur Favoritenliste hinzugefügt werden.

die Rückgängig-Funktion ermöglichen

In *capella* lassen sich alle Veränderungen an einer Partitur rückgängig machen.

Sofern ein Plugin Änderungen an der Partitur vornimmt, ist es zwingend notwendig, das Plugin bei *capella* zu registrieren.

Dazu fügt man zu Beginn des (internen) Skripts die Zeile

```
activeScore().registerUndo('tue irgendwas')
```

ein, wobei man `'tue irgendwas'` durch eine sinnvolle Beschreibung des Plugins ersetzt. Alles weitere übernimmt *capella* im Hintergrund:

- capella merkt sich den Bearbeitungsstand der Partitur vor Ausführung des Plugins.
- Der Beschreibungstext erscheint nach Ausführung im BEARBEITEN-Menü.
- Wird nun die RÜCKGÄNGIG-Funktion aufgerufen, wird die Partitur auf den Stand vor Ausführung des Skripts zurückgesetzt.

Wenn ein Plugin keine Änderungen an der Partitur vornimmt, ist es nicht sinnvoll, das Plugin zu registrieren. Zum Beispiel, wenn ein Plugin lediglich Informationen aus der Partitur gewinnt, anzeigt oder exportiert. In solchen Fällen würde der Anwender nur verwirrt, wenn das Rückgängigmachen trotzdem angeboten würde.

Intern ersetzt `registerUndo()` die Partitur durch ein Duplikat und hebt das Original zum Rückgängigmachen auf. Original und Duplikat sind unter verschiedenen Adressen im Hauptspeicher abgelegt. Objekte, die man vor dem Aufruf von registerUndo() aus der Partitur abgeleitet hat, beziehen sich nach dem Aufruf also auf die Sicherung. Im folgenden Codefragment würde jede Änderung an den Objekten im Iterator doc die Rückgängig-Funktion durcheinander bringen.

```
doc = activeScore()
doc.registerUndo("Demo")
for obj in doc.noteObjs():
    ...
```

Deshalb ist es erforderlich, nach dem Aufruf von `registerUndo()` erneut ein Score-Objekt mit `activeScore()` abzurufen, bevor Änderungen an der Partitur vorgenommen werden. Ein möglicher Ablauf könnte folgender sein:

```
doc = activeScore()
# Analyse der Partitur
# Anzeige der Ergebnisse in einem Dialog
# Abfrage, was das Skript tun soll
activeScore().registerUndo('tue was der Nutzer will')
doc = activeScore()
# Anpassungen am doc-Objekt vornehmen
```

ein Icon für das Plugin-Menü mitgeben

Wenn ein Plugin häufig ausgeführt wird oder weitergegeben werden soll, ist es sinnvoll, ein Icon mitzugeben. Dieses Symbolbild wird im Plugin-Auswahl-Dialog nach der erstmaligen Ausführung neben dem Plugin-Namen angezeigt. Wenn man das Plugin in die Favoriten aufnimmt, erscheint das Icon zusätzlich im Plugins-Menü.

Um das Icon in *capella* anzeigen zu lassen, muss es unter demselben Namen der Plugin-Datei im Format `.png` oder `.svg` im gleichen Verzeichnis gefunden wird. Falls kein ladbares Icon gefunden wird, benutzt *capella* ein Ersatz-Icon (schwarzes Puzzleteil).

Als Icon sind einfache quadratische Zeichnungen auf transparentem Untergrund am besten geeignet. Das Vektorformat `.svg` hat gegenüber dem Rastergrafikformat `.png` den Vorteil, dass es beliebig skaliert werden kann, was bei modernen Monitoren mit hoher Pixeldichte zu einem schärferen Erscheinungsbild führt. Alle mitgelieferten Skripte von capella nutzen Icons im `.svg`-Format.

mehrsprachige Plugins

Da *capella* in mehreren Sprachen erscheint, ist es möglich, Skripte zu lokalisieren. Die unterschiedlichen Sprachinformationen werden dabei unabhängig von der eigentlichen Python-Datei gepflegt.

Das folgende Beispiel-Skript läuft in allen *capella*-Sprachvarianten, zeigt aber überall die deutschen Dialogtitel und -texte:

```
dlg = FileDialog()
dlg.setTitle("Test des Dateiauswahldialogs")
dlg.addFilter("capella-Dateien", "*.cap;*.capx")
if dlg.run():
    messageBox ("gewählte Datei", dlg.filePath())
```

Um dieses Skript so zu gestalten, dass es in jeder Sprachversion korrekt erscheint, sind folgende Anpassungen erforderlich:

- im Quelltext: Ersetzen der angezeigten Texte (z.B. „Test des Dateiauswahldialogs“) durch eindeutige Referenzen
- Erstellung eines „Verzeichnisses“, in dem zu jeder Referenz für alle gewünschten Sprachen die passenden Texte aufgeführt sind.
- Ergänzung von Plugin-Titel und -Beschreibung in allen Sprachen, damit das Plugin im Plugin-Auswahldialog korrekt angezeigt wird.

Durch die Trennung von Quelltext und Sprachinformationen kann man ohne Eingriff in das Skript später weitere Sprachen ergänzen bzw. muss zum Übersetzen der Dialogtexte den Quelltext nicht weitergeben. Es empfiehlt sich jedoch, zumindest eine Sprachversion innerhalb der eigentlichen Quelltextdatei (wenn auch separiert von den Programmanweisungen) mitzugeben, damit das Skript auch ohne weitere Sprachversionen lauffähig ist. Die Inhalte sind wie folgt auf drei Dateien aufzuteilen, die in einem gemeinsamen Verzeichnis liegen müssen:

- `[Skriptname].py`

  das eigentliche Skript mit den (separierten) Texten einer Sprachversion – dieses Skript ist damit auch ohne die beiden anderen Dateien ausführbar

- `[Skriptname].info`

  die Bezeichnungen und Beschreibungen des Plugins in verschiedenen Sprachvarianten – hier wird festgelegt, wie das Skript im Plugin-Menü erscheint. Die Info-Datei trägt dabei den gleichen Dateinamen wie das eigentliche Skript, allerdings mit der Dateiendung `info`. Daran ordnet *capella* die beiden Dateien einander zu. Der Dateiname erscheint nicht im Plugin-Menü; dort wird der in der Info-Datei festgelegte Titel in der passenden Sprache angezeigt (s.u.).

- `[Skriptname]_tr.py`

  die Dialog-Texte in weiteren Sprachversionen. Die Ergänzung des Skriptnames um die Erweiterung „`_tr`“ bewirkt, dass diese Python-Datei von capella nicht als eigenständiges Plugin angezeigt wird, obwohl es die Endung `py` hat.

Das Eingangsbeispiel hat dann folgende Gestalt:

demo.py
```
german = ("de", {
    "title"       : "Test des Dateiauswahldialogs",
    "capFiles"    : "capella-Dateien",
    "chosenFile"  : "gewählte Datei"})
try:
    from demo_tr import translations
    translations.append(german)
    setLanguages(translations)
except:
    def tr(s):
        return german[1].get(s, "???")

dlg = FileDialog()
dlg.setTitle(tr("title"))
dlg.addFilter(tr("capFiles"), "*.cap;*.capx")
if dlg.run():
    messageBox (tr("chosenFile"), dlg.filePath())
```

demo.info
```
<info>
  <lang id="en">
    <title>Test of the localization</title>
    <descr>
      <p>This is a demo of the
        localization possibilities in capella.</p>
    </descr>
  </lang>
  <lang id="de">
    <title>Test der Lokalisierung</title>
    <descr>
      <p>Dies ist eine Demo für die
        Lokalisierungsmöglichkeiten in capella.</p>
    </descr>
  </lang>
</info>
```

demo_tr.py
```
translations = [
("en", {
    "title"       : "Test of the file dialog",
    "capFiles"    : "capella files",
    "chosenFile"  : "chosen file"}) ]
```

Erläuterung:

Am Beginn der eigentlichen Skript-Datei wird die deutsche Sprachinformation als eine Liste [...] mit einem Element in Form eines Tupels (...) aus Sprachkennung („de“) und einem Dictionary definiert. Das Dictionary {...} enthält zu jeder Textreferenz den Text in deutscher Sprache.
Die weiteren Sprachversionen (hier „en“ für (amerikanisches) Englisch) werden anschließend importiert und ergänzt. Sinnvollerweise geschieht das in einem `try`-Block.
Die Anweisung

```
setLanguages(translations)
```

erwartet als Parameter eine Liste mit Tupel aus Sprachkennung und Dictionary, wie sie im Beispiel definiert ist. Sie weist capella an, die lokal gültige Sprachversion zu laden und in der Folge für alle Aufrufe von

```
tr(ref)
```

den passenden Eintrag zur Referenz ref im passenden Dictionary zurückzugeben.

In der Info-Datei sind für alle unterstützten Sprachversionen der Titel und eine kurze Beschreibung des Plugins angegeben. Diese erscheinen im Plugin-Dialog von *capella*.

Sprach-Kennungen

Die internationale Norm ISO 639-1 definiert Kürzel aus zwei Kleinbuchstaben für alle möglichen Sprachen der Welt, z. B. „de“ für Deutsch, „en“ für Englisch. Die Norm ISO 3166 definiert Kürzel aus zwei Großbuchstaben für geografische Einheiten, z. B. „DE“ für Deutschland, „US“ für die USA.
In den `<lang>`-Elementen der Info-Datei kann für das Attribut `id` entweder eine Sprachkennung (z. B. de) oder eine Kombination aus Sprachkennung und geografischer Einheit, getrennt durch ein Minuszeichen verwendet werden (z. B. en-GB):
Für die aktuellen Sprachversionen von capella kommen folgende Kennungen in Frage:

Code	**Bedeutung**
de	Deutsch
en	(Amerikanisches) Englisch – Wenn zwei englische Versionen mitgegeben werden sollen, sollte eine davon einfach mit „en“ bezeichnet werden. Bei „en“ für amerikanisches Englisch und „en-GB“ für britisches Englisch wird das amerikanische Englisch als Standard gewählt und die britische Version nur für das Vereinigte Königreich. Wird dagegen die amerikanische Version mit „en-US“ und die britische „en“ benannt, ist Britisch der Standard und Amerikanisch wird nur für USA gewählt.

`en-GB`	Britisches Englisch
`nl`	Niederländisch
`cs`	Tschechisch
`pl`	Polnisch
`fi`	Finnisch
`sv`	Schwedisch

Jede lokalisierte *capella*-Version kennt (aus ihren Ressourcen) ein vollständiges Sprachkürzel (z. B. `de-DE`).
capella sucht zunächst nach einer Lokalisierung mit dem vollständigen Sprachkürzel. So wird die deutsche Version zunächst nach `de-DE` suchen. Wird dies nicht gefunden (was in der Regel der Fall ist), schraubt *capella* seine Ansprüche zurück und sucht nach einer Version mit dem Sprachkürzel (`de`). Hier würde z. B. auch `de-AT` akzeptiert. Wenn auch dies erfolglos ist, sucht *capella* nach einem Kürzel, das mit `en` beginnt. Gibt es auch das nicht, wird die als erste angegebene Sprache genommen.

Lokalisierte Hilfedateien für Skripte

capella sucht zunächst nach `.html`-Dateien mit dem um ein Minuszeichen und eine Sprachkennung verlängerten Namen.

Beispiel: Wenn neben dem Skript `demo.py` auch die Dateien `demo-en.html`, `demoen-GB.html` und `demo-de.html` existieren, wählt *capella* (wie oben angegeben) die passendste Version. Nur wenn gar keine lokalisierte Version vorhanden ist, greift *capella* auf die Hilfedateien mit neutralem Namen (`demo.html`) zurück.

Parameter speichern und auslesen

Bei komplexeren Aufgaben ist es für die Nutzung komfortabel, wenn ein Skript sich Parameter „merken“ kann, so dass man nicht bei jedem Aufruf alle Optionen neu auswählen muss. Das lässt sich mit der Hilfsklasse ScriptOptions bewerkstelligen, sogar über den Neustart von *capella* hinaus. Details siehe S. 48

Grafikobjekte und Grafikkennungen

Neben Notenobjekten gibt es im CapXML-Datenmodell auch Grafikobjekte. Hier ein Vergleich zwischen diesen beiden Objekttypen:

	Notenobjekt	Grafikobjekt
Beispiele	Noten/Akkorde, Pausen, Notenschlüssel, Tonartvorzeichnungen, Taktangaben, feste Taktstriche	z.B. Musiksymbole (Fermate, Akzente, Triller), Textfelder, Grafikelemente (Linien, Rechtecke, Ellipsen), Bindebögen, Voltenklammern, importierte Grafiken
wie verankert	als `NoteObj` Teil einer Stimme	verankert an einer Seite oder an einem Akkord bzw. Pause
Zugriff in externen Skripten	durch Einfügen / Ändern / Löschen der zugehörigen XML-Elemente	durch Einfügen / Ändern / Löschen der zugehörigen XML-Elemente
Einfügen bzw. Löschen in internen Skripten	nicht möglich	mit `addDrawObj` und `deleteDrawObj` (Methoden von `NoteObj`, siehe S. 71)
Änderungen in internen Skripten	mit Methoden der Klasse `NoteObj`, siehe S. 67	durch Änderung der Schlüssel-Wert-Paare in Python-Dictionaries, siehe S. 72

Beim Arbeiten mit Grafikobjekten kann es passieren, dass Grafikobjekte, die ein Skript erstellt hat, gezielt wieder gelöscht oder verändert werden sollen.

> Beispiel: Ein Skript setzt individuelle Taktmarkierungen in einem bestimmten Abstand. Nachträglich wird ein Takt eingefügt: Alle Markierungen verschieben sich und müssen neu gesetzt werden. Vor dem Neusetzen sollten aber alle ungültig gewordenen Markierungen entfernt werden. Aber wie identifiziert man sie?

Zu diesem Zweck kann man jedem Grafikobjekt eine (unsichtbare) Kennung mitgeben, die am Grafikobjekt gespeichert wird. Damit lassen sich später Grafikobjekte eines Skripts identifizieren und gezielt modifizieren.

Mit dem Methode `deleteTaggedGraphics` aus der Klasse `Score` können alternativ alle Grafikobjekte mit einer bestimmten Kennung gelöscht werden. Siehe dazu S. 60

Als Konvention wird folgendes Format genutzt: Eine Kennung ist ein String, bestehend aus zwei durch ein Minuszeichen getrennten Dezimalzahlen (führende Nullen können

weggelassen werden). Die erste Zahl ist die Kundennummer des Skript-Entwicklers bei capella-software, die zweite Zahl kann vom Entwickler im Bereich von 1 bis 4096 frei vergeben werden. Die Kundennummer ist der zweite (sechsstellige) Teil Ihrer Seriennummer. Sie finden diese z. B. im Info-Dialog von *capella*. Wenn Sie z. B. die Kundennummer 012345 haben, können Sie die Kennungen `'012345-1'`, bis `'012345-4096'` verwenden.

Beispiel für ein internes Skript, das über allen Akkorden in der Partitur eine Fermate einfügt:

```
activeScore().registerUndo('Fermaten einfügen')
fermate = dict(type="text",
               placement = "auto",
               placementHint ="musicSymbol",
               content = "u",
               tag = "123456-1",
               font = dict (face = "capella3",
                            charSet = 2,
                            height = 18.0,
                            pitchAndFamily = 2)

               )
for n in activeScore().noteObjs():
    if n.isChord():
        n.addDrawObj(fermate)
```

Dabei wird den eingefügten Fermaten das Tag `123456-1` mitgegeben. Dadurch lassen sich diese Fermaten von anderen Fermaten unterscheiden, die z.B. manuell eingefügt wurden. Das folgende Skript löscht alle Grafikobjekte mit dieser Kennung.

```
activeScore().registerUndo('Fermaten einfügen')
activeScore().deleteTaggedGraphics("123456-1")
```

Referenz der internen Programmierschnittstelle

In diesem Kapitel werden alle Klassen und Funktionen erläutert, die über die interne Programmierschnittstelle angesprochen werden können. Falls eine Aufgabe mit diesen Problemen nicht gelöst werden kann, ist ein externes Skript erforderlich.

Globale Funktionen

Diese Funktionen können in einem internen Skript direkt (d.h. ohne vorherige Instanzierung eines Objekts) aufgerufen werden.

Verzeichnis- und Dateimanagement

getPersonalDataDir()

Gibt das Unterverzeichnis `capella` im Verzeichnis `Eigene Dateien` (oder `Documents` o.ä.) des aktuellen Benutzers zurück.

getProgramDir()

Gibt das *capella*-Programmverzeichnis (in dem sich `capella.exe` befindet) zurück. Das Programmverzeichnis ist nicht das Verzeichnis, in dem das aktuelle Skript liegt. Dieses können Sie so ermitteln:

```
dir, pyFile = os.path.split(sys.argv[0])
```

Im Rückgabewert `dir` ist das Verzeichnis gespeichert, in `pyFile` der Name des Python-Skripts, das gerade ausgeführt wird.

getUserDataDir()

Gibt das Unterverzeichnis `capella-software\capella` im Verzeichnis Anwendungsdaten (oder AppData o.ä.) des aktuellen Benutzers zurück.

allFiles(dir, ext='', recursive=False)

Gibt einen Iterator über alle Dateien mit der Erweiterung `ext` im Verzeichnis `dir` zurück. Falls der Parameter `ext` weggelassen wird oder der leere String ist, werden alle Dateien gesucht. Falls der Parameter `recursive=True` übergeben wird, schließt die Iteration alle Unterverzeichnisse ein. Beispiel:

```
for f in allFiles(r'C:/Benutzer/JamesBond', 'capx'):
    messageBox('Datei:', f)
```

Dieser Aufruf gibt alle Dateien mit der Endung „`*.capx`“ im Nutzerverzeichnis des Nutzers „JamesBond“ zurück.

openScore(path)

Öffnet die Partitur mit dem Dateipfad `path`.

closeActiveScore()

Schließt die derzeit aktive Partitur ohne Rückfrage.

capella-spezifische Funktionen

capVersion()

Liefert ein Python-Tupel der Form (Hauptversion, Unterversion, Service-Release). Beispiel:

```
ver, sub, sr = capVersion()
messageBox('capella',
           'Version %d.%02d-%02d' % (ver, sub, sr))
```

checkCapVersion(ver, sub, sr)

Püft, ob mindestens die capella-Version <ver>.<sub>-<sr> installiert ist. Falls das nicht der Fall ist, wird das Skript abgebrochen. Beispiel:

```
checkCapVersion(8, 0, 0)
```

activeScore()

Diese Funktion ist der Einstiegspunkt in die aktuell geöffnete Partitur. Zum generellen Vorgehen zum Navigieren in einer Partitur in einem internen Skript siehe S. 12

Die Funktion `activeScore` Gibt die aktive Partitur als Objekt der Klasse `Score` zurück. Da *capella* in der aktuellen Version die Ausführung von Skripten nur dann zulässt, wenn eine Partitur geöffnet ist, kann man beim Aufruf davon ausgehen, dass ein `Score`-Objekt zurückgegeben wird.

In früheren *capella*-Versionen konnte man Skripte auch dann ausführen, wenn keine Partitur geöffnet war. In diesem Fall ist der Rückgabewert `None`. Möchte man sicherstellen, dass ein Plugin auch bei älteren *capella*-Versionen funktioniert, ist es ratsam, diese Möglichkeit abzufangen:

```
if activeScore():
    # das eigentliche Skript
```

curSelection()

Mit dieser Funktion lässt sich die Cursorposition bzw. der markierte Bereich auswerten. Der Rückgabewert ist ein Tupel aus zwei Vierer-Tupeln:

`((sys1, sta1, voi1, nOb1), (sys2, sta2, voi2, nOb2))`

Die einzelnen Werte sind Integer-Zahlen und stehen für Indices:

`sys1, sys2` – Indices der Systeme, in dem der Beginn bzw. das Ende der Markierung stehen, wobei das erste System der Partitur den Index 0 hat

`sta1, sta2` – Indices der Notenzeilen, in dem der Beginn bzw. das Ende der Markierung innerhalb der jeweiligen Systemen stehen, wobei die erste Notenzeile eines Systems den Index 0 hat

`voi1, voi2` – Indices der Stimmen, in dem der Beginn bzw. das Ende der Markierung innerhalb der jeweiligen Zeilen stehen, wobei die erste Stimme einer Notenzeile den Index 0 hat

`nOb1, nOb2` – Indices der Notenobjekte innerhalb der jeweiligen Stimme, die nach den in dem Beginn bzw. dem Ende der Markierung stehen bzw. stehen würden, wobei das erste Notenobjekt einer Stimme den Index 0 hat

Weitere Details dazu siehe S. 15

`cursorObj``()`

Gibt das Notenobjekt (ein Objekt der Klasse `NoteObj`) rechts vom Cursor zurück. Falls der Cursor am Ende einer Stimme steht, wird 0 zurückgegeben. Wenn ein Bereich markiert ist, dann wird trotzdem die Cursorposition ausgewertet, an der der Cursor steht.

`displayPage``(iPage)`

Setzt den Cursor an den Beginn des ersten Systems auf der übergebenen Seite (gezählt ab 1). Dadurch wird auch der angezeigte Ausschnitt der Partitur im *capella*-Programmfenster beeinflusst.

`displaySystem``(iSystem)`

Setzt den Cursor an den Beginn des übergebenen Systems (gezählt ab 1). Wie bei `displayPage()` wird dadurch auch der angezeigte Ausschnitt beeinflusst.

`intervals``(l)`

Mit dieser Funktion kann man die Intervalle zwischen 2 oder mehr Tönen bestimmen.

Der Parameter `l` ist eine Liste von Tonhöhen im Format (diatonische Stufe, Alteration). Z.B. steht (35, 0) für das eingestrichene c und (44, -1) für das zweigestrichene es.

Zurückgegeben wird ein Tupel mit den Intervallen zwischen dem ersten und den übrigen Tönen der übergebenen Liste aus. Dabei werden die Intervalle als Paare (Stufe, Alteration) übergeben. Die Stufe ist die Zahl der diatonischen Tonschritte (0 = Prime, 1 = Sekunde, 3 = Terz, …). Die Alteration ist wie folgt codiert:

`-2` = vermindert
`-1` = klein
`0` = rein
`1` = groß
`2` = übermäßig

Die Intervalle werden modulo einer Oktave (z.B. Sekunde statt None) angegeben. Beispiel:

```
pitches = [(35,0), (37,0), (39,1), (43,-1)]
messageBox('Intervalle', str(intervals(pitches)))
```

Die übergebenen Töne stehen für c, e, gis (eingestrichen) und das zweigestrichene des. Es wird ((2,1),(4,2),(1,-1)) ausgegeben, also große Terz, übermäßige Quinte und kleine Sekunde (statt kleiner None).

`rootNode``()`

Liefert ein Objekt der Klasse `CapXNode` zurück. Es enthält den Root-Knoten (vom Typ Score) der aktuellen Partitur. Details siehe S. 56

`selectedNode``(sys, staff, voice, note)`

Liefert ein Objekt der Klasse `CapXNode` zurück. Es enthält den Knoten für das Notenobjekt an der spezifizierten Position in der Partitur. Die vier Parameter sind vom Typ Integer:

`sys` – der Index des Systems in der Partitur (gezählt ab `0`)

`staff` – der Index der Notenzeile innerhalb des Systems (gezählt ab `0`)

`voice` – der Index der Stimme innerhalb der Notenzeile (gezählt ab `0`)

`note` – der Index des Notenobjekts innerhalb der Stimme (gezählt ab `0`)

Wird ein Parametersatz übergeben, zu dem es keinen Knoten im Strukturbaum gibt, wird ein leeres `CapXNode`-Objekt zurückgegeben.

Details siehe S. 56

`setSelection``(sel)`

Setzt den markierten Bereich innerhalb der Partitur. Der Parameter `sel` hat die gleiche Struktur wie der Rückgabewert der Funktion `curSelection()`. Details siehe auch S. 15

`shiftDrawObj``(d, dx, dy)`

Verschiebt das Grafikobjekt `d` um `dx`, `dy`. Das Grafikobjekt `d` wird als Dictionary übergeben, wie es von den Methoden `drawObj` und `drawObjs` der Klasse `NoteObj` geliefert wird. Die Parameter `dx` und `dy` sind Abstandsangaben gemessen in Zwischenräumen.

Die Verschiebung hat zur Folge, dass das Grafikobjekt anschließend als „manuell ausgerichtet“ registriert ist.

```
activeScore().registerUndo("Grafikobjekte verschieben")
for note in activeScore().noteObjs():
    for i in range(note.nDrawObjs()):
        d = note.drawObj(i)
        shiftDrawObj(d, 2, -2)
        note.replaceDrawObj(i, d)
```

Nutzerkommunikation

`messageBox``(title, text, img=0, buttons=0, defBtn=1)`

Zeigt eine Standard-Meldungsbox. Parameter:

`title` Text, der im Fensterkopf angezeigt wird.

`text` Text, der im Inneren angezeigt wird.

`img` Anzuzeigendes Symbol:
0 = keines (Standard)
1 = Fragezeichen
2 = Info
3 = Ausrufezeichen
4 = Stop. (Standard: 0)

`buttons` – Buttons des Dialogs:
0 = OK (Standard)
1 = OK/Abbrechen
2 = Ja/Nein
3 = Ja/Nein/Abbrechen.

`DefBtn` – Button, der beim Anzeigen der Messagebox den Auswahlfokus hat (Zählung ab 1). Wird kein Parameter übergeben, ist der Fokus auf dem OK- bzw. Ja-Button.

Der Rückgabewert dieser Funktion gibt an, welcher Button gedrückt wurde:
0 = Abbrechen
1 = OK
2 = Ja
3 = Nein

allgemeine Hilfsklassen

FileDialog

Ein Dialog zum Erfragen eines Dateinamens. Die Klasse `FileDialog` hat folgende Methoden:

`FileDialog``(bOpen=True)`

Konstruktor: Erzeugt ein Objekt, das einen Dateinamen erfragen kann (mit Methode `run`). Parameter:

`bOpen`: `True` zum Erfragen einer existierenden Datei (zum Öffnen)
`False` zum Erfragen eines beliebigen Dateinamens (zum Speichern).

`setDefaultExt``(ext)`

Setzt mit dem Parameter `ext` die Standard-Dateierweiterung. Falls im Konstruktor `False` für `bOpen` und kein Dateifilter (per `addFilter`) übergeben wurde, wird diese Erweiterung (und ein Punkt davor) automatisch an einen ohne Erweiterung eingegebenen Dateinamen angehängt.

`setStartFile``(name)`

Setzt mit dem Parameter `name` den Dateiname, der zu Anfang im Feld "Dateiname" erscheint (Standard: keiner).

`setStartPath``(path)`

Setzt mit dem Parameter `path` das Verzeichnis, mit dem die Auswahl beginnt (Standard: aktuelles Verzeichnis)

`setTitle``(title)`

Mit dem Parameter title kann der Titel der Dialogbox gegenüber dem Standard geändert werden. Wenn diese Methode nicht aufgerufen wird, erscheint der Titel "Speichern unter" bzw. "Öffnen" je nach dem im Konstruktor übergebenen Parameter.

`addFilter``(text, filter)`

Angabe eines Musters für Dateierweiterungen im Aufklappfeld „Dateityp“. Parameter:

`text`: Im Aufklappfeld „Dateityp“ angezeigter Text (z. B. 'html-Dateien (*.htm)')

`filter`: Für die Auswahl benutztes Muster (z. B. '*.cap;*.capx')

`run``()`

Hiermit wird der Dialog ausgeführt. Rückgabewert `True`, falls ein gültiger Dateiname eingegeben und mit OK bestätigt wurde, sonst `False`.

`filePath``()`

Gibt den absoluten Pfad des vom Anwender eingegebenen Dateinamens zurück.

Beispiel:

```
dlg = FileDialog()
dlg.setTitle('capella-Datei öffnen')
dlg.addFilter('capella-Dateien', '*.cap;*.capx')
dlg.addFilter('alle Dateien', '*.*')
dlg.setStartPath(getPersonalDataDir())
if dlg.run():
    messageBox ('gewählte Datei', dlg.filePath())
```

Rational

Diese Klasse steht für rationale Zahlen (Brüche aus ganzen Zahlen). In der Python-Standardbibliothek gibt es seit der Python-Verion 2.6 auch das Modul `fractions` mit der Klasse `Fraction`. Da capella aber wesentlich früher Python als Skriptsprache angebunden hat, wurde zu diesem Zweck die eigene Klasse `Rational` implementiert, die aus Kompatibilitätsgründen auch weiterhin genutzt werden sollte.
`Rational` wird für die Repräsentation der Notenwerte (Ganze, Halbe, Viertel, … aber auch punktierte oder triolische Notenwerte) verwendet. Der Konstruktor akzeptiert zum Initialisieren einer rationalen Zahl folgende Konstellationen:

- Zähler als Integer (wobei der Nenner = 1 gesetzt wird), z.B. führt
  `a = Rational(3)`
  zu einer Initialisierung der rationalen Zahl mit Zähler = 3 und Nenner = 1.
- Zähler und Nenner als Integer, z.B. wird
  `b = Rational(3,4)`
  als der Bruch ¾ interpretiert.
- String im Format 'Zähler/Nenner', z.B. liefert `c = Rational('1/2')`
  ein Rational-Objekt mit Zähler = 1 und Nenner = 2

Dabei werden automatisch folgende Konventionen sichergestellt:

- Der Nenner ist immer positiv (bei negativen Zahlen ist der Zähler negativ).
  Die Initialisierung `Rational(3,-5)` liefert den Bruch -3/5 zurück.
- Der Bruch wird immer in gekürzter Form gespeichert. `Rational(2/6)` liefert also den Bruch 1/3.

Die arithmetischen Operatoren (+, -, *, / sowie +=, -=, *=, /=) sind überladen. Die Konvertierung in einen String ist ebenso möglich, wie in eine Fließkommazahl. Bei der Konvertierung in einen Integer wird nach den kaufmännischen Regeln gerundet. Beispiel:

```
a = Rational(4)
b = Rational(3,-5)
c = Rational('6/12')
ausgabe = 'a = ' + str(a) + '\n'
ausgabe += 'b = ' + str(b) + '\n'
ausgabe += 'c = ' + str(c) + '\n'
ausgabe += 'b + c = ' + str(b+c) + '\n'
ausgabe += 'b - c = ' + str(b-c) + '\n'
ausgabe += 'b * c = ' + str(b*c) + '\n'
ausgabe += 'b / c = ' + str(b/c) + '\n'
```

```
ausgabe += '-b/a = ' + str(-b/a) + '\n'
b += 3
ausgabe += 'b += 3  ->  b = ' + str(b) + '\n'
b -= 4
ausgabe += 'b -= 4  ->  b = ' + str(b) + '\n'
b *= c
ausgabe += 'b *= c  ->  b = ' + str(b) + '\n'
b /= a
ausgabe += 'b /= a  ->  b = ' + str(b) + '\n'
ausgabe += 'abs(a*b) = ' + str(abs(a*b)) + '\n'
ausgabe += 'float(b) = ' + str(float(b)) + '\n'
ausgabe += 'b+c = ' + str(b+c) + '\n'
ausgabe += 'int(b+c) = ' + str(int(b+c)) + '\n'
messageBox('Rational-Demo', ausgabe)
```

Color

Diese Klasse enthält Hilfsfunktionen für als Zahl dargestellte Farben, wie sie in *capella*-Objekten verwendet werden. Die Klasse `Color` enthält eine Aufzählung mit folgenden Farbkonstanten:

Konstante	rot	grün	blau
`black`	0	0	0
`blue`	0	0	255
`cyan`	0	255	255
`darkGray`	64	64	64
`gray`	128	128	128
`green`	0	255	0
`lightGray`	192	192	192
`magenta`	255	0	255
`orange`	255	200	0
`pink`	255	175	175
`red`	255	0	0
`white`	255	255	255
`yellow`	255	255	0

Die Klasse `Color` hat folgende Methoden:

`Color.RGB``(r,g,b)`

Statische Methode. Gibt eine als Zahl codierte RGB-Farbe mit den Komponenten r für rot, g für grün und b für blau (jeweils im Bereich von 0 bis 255) zurück.

_Color.red(c), **_Color.green**(c), **_Color.blue**(c)

Statische Methoden der Basisklasse _Color. Geben die Rot-, Grün- bzw. Blau-Komponente der als Zahl c codierten RGB-Farbe (im Bereich von 0 bis 255) zurück. Beispiel:

```
newColor = Color.RGB(0, 88,176)
messageBox("Färben mit:","rot "+
           str(_Color.red(newColor)) + " grün "+
           str(_Color.green(newColor)) + " blau "+
           str(_Color.blue(newColor)))
if activeScore():
    for obj in activeScore().noteObjs():
        if obj.isChord() and obj.duration() == Rational(1,8):
            obj.setColor(newColor)
```

RelDiatonicNote

Hilfsklasse zur Darstellung einer diatonischen Tonhöhe ohne Berücksichtigung der Oktavlage. Objekte diese Klasse können mit den Operatoren + und - addiert und subtrahiert werden sowie mit str() in lesbare Form verwandelt werden. Die Klasse RelDiatonicNote hat folgende Methoden:

RelDiatonicNote(diatonic, chromatic)

Konstruktor mit den folgenden Parametern:

diatonic: Die diatonische Tonhöhe. Mögliche Werte 0 für C usw. bis 6 für H. Andere Zahlenwerte werden modulo 7 übernommen (z. B. stehen -5 und 9 für 2).

chromatic: Die chromatische Tonhöhe. Mögliche Werte 0 für C usw. bis 11 für H. Andere Zahlenwerte werden modulo 12 übernommen (z. B. stehen -10 und 14 für 2).

getStep()

Gibt die diatonische Grundnote (0 bis 6) zurück.

getChromatic()

Gibt die relative chromatische Tonhöhe (0 bis 11) zurück.

getAlteration()

Gibt die Alteration des Tons in Halbtonschritten zurück. Für Stammtöne wird 0 zurückgeliefert. Bei tiefalterierten Tönen werden negative Werte, bei hochalterierten Tönen positive Werte zurückgeliefert.

`RelDiatonicNote.fromCircleOfFifth``(n)`

Diese statische Methode gibt ein `RelDiatonicNote`-Objekt entsprechend der Stufe im Quintenzirkel zurück. Der Parameter n gibt den Schritt im Quintenzirkel an:

- n = 0 liefert den Ton C
- Positive Werte für n liefert den Grundton der entsprechenden Kreuz-Tonarten, z.B. liefert n = 4 den Ton E (diatonic=2, chromatic=4).
- Negative Werte für n liefert den Grundton der zugehörigen B-Tonart, z.B. liefert n = -2 den Ton B (diatonic=6, chromatic=10).

`RelDiatonicNote.fromStepAlter``(step, alt)`

Diese statische Methode gibt ein `RelDiatonicNote`-Objekt entsprechend der übergebenen diatonischen Stufe (Parameter `step`) und Alteration (Parameter `alt`) zurück. Beispiel:

```
RelDiatonicNote.fromStepAlter(2, -1)
```

liefert den Ton Es (diatonic=2, chromatic=3)

`RelDiatonicNote.fromSymbolic``(s)`

Die statische Methode gibt ein `RelDiatonicNote`-Objekt entsprechend dem Beschreibungstext zurück. Der Parameter s ist ein String, bestehend aus Notenbuchstabe (a..g oder A..G) und optional # oder b). Beispiel:

```
RelDiatonicNote.fromSymbolic('Gb')
```

liefert den Ton Ges (diatonic=4, chromatic=6).

ScriptOptions

Hilfsklasse zum Lesen und Speichern von Benutzer-Optionen zu einem Skript. Diese werden in einer Optionen-Datei im Anwendungsdaten-Verzeichnis des Nutzers abgelegt. Die Klasse `ScriptOptions` hat folgende Methoden:

`ScriptOptions``()`

Der Konstruktor erzeugt ein Objekt, das Optionen für das aktuelle Skript lesen und speichern kann. Der Name der Optionen-Datei entspricht dem Namen der Python-Datei (mit Erweiterung .opt).

`get``()`

Liefert die gespeicherten Optionen zum laufenden Skript, sofern dafür im Anwendungsdaten-Verzeichnis eine Optionendatei abgelegt ist. Daraus wird ein Dictionary erzeugt und als Funktionswert zurückgegeben.

Falls die Optionsdatei nicht existiert, wird ein leeres Dictionary zurückgegeben.

set(opt)

Schreibt die Optionen aus dem im Parameter opt übergebenen Dictionary-Objekts in die Optionsdatei des Skripts (siehe dazu `get()`).

Beispiel:

```
optFile = ScriptOptions()
opt = optFile.get()
s =  opt.get('secret', 'noch leer')
label = Label('Geheimnis: ')
edit = Edit(s, width=50)
hBox = HBox ([label, edit], padding=8)
dlg = Dialog('Tresor', hBox, 'für ein Geheimnis')
if dlg.run():
    optFile.set(dict(secret=edit.value()))
```

Das Beispiel nutzt die gespeicherten Optionen für einen kleinen „Tresor“. Beim ersten Aufruf wird eine (leere) Optionsdatei initiiert. Das Eingabefenster wird mit Hilfe des Dialogbaukastens erstellt, siehe S. 27
Nach Ausführung des Dialogs wird der eingegebene Text in der Optionsdatei des Dialogs gespeichert. Bei erneuter Ausführung des Skripts wird der Text ausgelesen und angezeigt. Das funktioniert auch nach einem Neustart von *capella*.

Clipboard

Hilfsklasse zum Ablegen und Entnehmen von Text in der Zwischenablage.

getText()

Liefert den Textinhalt der Windows-Zwischenablage zurück. Wenn die Zwischenablage leer ist oder etwas anderes als Text enthält, wird ein Leerstring `""` zurückgegeben.

setText(t)

Schreibt den Text `t` in die Windows-Zwischenablage.

Beispiel:

```
elements = NoteObj.__dict__.items()
s=''
for e in elements:
    s += str (e) + '\n'
Clipboard.setText(s)
```

```
messageBox('Info', s)
```

Das Skript ermittelt für alle Notenobjekte der Partitur alle Konstanten und Methoden und kopiert diese Aufstellung als Text in die Zwischenablage (und zeigt sie zusätzlich in einer MessageBox).

MidiOut

Enthält statische Methoden zur Soundausgabe über die MIDI-Schnittstelle.

setMainVolume(nChannel, nVolume)

Setzt die Grundlautstärke für einen Kanal. Parameter:

nChannel: Kanal (0..15)
nVolume: Lautstärke (0..127)

programChange(nChannel, nInstr)

Wählt die Stimme für einen Kanal. Parameter:

nChannel: Kanal (0..15)
nInstr: Stimme (0..127)

noteOn(nChannel, nNote, nVolume)

Schaltet einen Ton auf einem Kanal ein. Parameter:

nChannel: Kanal (0..15)
nNote: MIDI-Note (0..127, eingestrichenes C = 60)
nVolume: Laustärke (0..127)

Die Lautstärke, mit dem der Ton letztlich wiedergegeben wird, ergibt sich als Produkt aus

- dem hier als Parameter nVolume übergebenen Wert
- dem mit setMainVolume für den Kanal eingestellten Wert
- und der in der Systemsteuerung eingestellten Lautstärke.

noteOff(nChannel, nNote)

Schaltet einen Ton auf einem Kanal aus. Parameter:

nChannel: Kanal (0..15)
nNote: auszuschaltende MIDI-Note (0..127)

allNotesOff()

Die womöglich wichtigste MIDI-Funktion überhaupt: Schaltet alle Noten auf allen Kanälen aus.

Beispiel:

```
MidiOut.setMainVolume(0, 127)
MidiOut.programChange(0, 0)
MidiOut.noteOn(0, 60, 127)
messageBox ('MIDI-Tontest' ,'Hören Sie etwas?')
MidiOut.allNotesOff()
```

Die Klassen des Dialogbaukastens

Mit dem Dialogbaukasten können beliebig komplexe Dialoge aufgebaut werden. Für eine Einführung in das Grundprinzip des Dialogbaukastens siehe S. 27

Daneben gibt es fertig verwendbare Dialoge:

- messageBox – siehe S. 43
- FileDialog – siehe S. 43

Dialog

Mit dieser Klasse kann in internen Skripten ein Dialog ausgeführt und abgefragt werden. Die Klasse Dialog hat folgende Methoden:

`Dialog``(caption, content, help='', timeout=0)`

Der Konstruktor erzeugt ein Dialog-Objekt mit folgenden Parametern:

`caption`: Fenstertitel des Dialogs

`content`: Das Wurzelelement des Dialogs. Dies muss eines der im Folgenden beschriebenen Objekte sein. Mit den Containerklassen HBox und VBox können beliebig verschachtelte Layouts erzeugt werden.

`help`: Ein Text, der in einer Meldungsbox gezeigt wird, wenn im Dialog der Hilfe-Button angeklickt wird. Wenn der Parameter weggelassen wird, wird kein Hilfetext angezeigt.

`timeout`: Mit diesem Parameter ist es möglich, Skriptdialoge ohne Benutzereingabe von selber zum Verschwinden zu bringen. Er übergibt die Zeitdauer in Millisekunden, die das Dialogfeld angezeigt wird, bevor es von allein wieder verschwindet. Sobald die Maus über dem Dialogfenster bewegt wird oder eine Taste gedrückt wird, ist der Timeout beendet und das Dialogfeld verhält sich wie gewohnt.

Wird ein Wert von 0 übergeben oder der Parameter weggelassen, wird das Dialogfeld dauerhaft angezeigt.

`run``()`

Startet den Dialog. Anhand des Rückgabewertes kann man ermitteln, wie der Dialog beendet wurde:

```
dlg = Dialog(caption, content)
dlg['timeout'] = 2000      # Zwei Sekunden
if dlg.run():              # OK wurde gewählt
```

```
    doAction()
elif dlg['timeout'] == 0: # die Zeit war abgelaufen
    doDefaultAction()
else:                     # es wurde manuell abgebrochen
    pass
```

Jeder Dialog kann aus den im Folgenden beschriebenen Elementen zusammengesetzt werden. Die Klassen HBox und VBox dienen dabei zum Anordnen mehrerer Elemente.

Die Konstruktoren der Dialogelemente haben neben normalen Parametern auch so genannte Schlüsselwortparameter. Die können auch weggelassen werden. Auf sie wird in den folgenden Beschreibungen durch drei Punkte am Ende der Parameterliste hingewiesen.

Label

Ein statisches (nicht bearbeitbares) einzeiliges Textfeld. Die Klasse `Label` hat folgende Methoden:

`Label``(text, ...)`

Der Konstruktor hat einen Pflichtparameter

> `text`: Der Text des Textfelds.

Wahlfreier Schlüsselwortparameter:

> `width`: Breite in mittleren Buchstabenbreiten. Ohne diesen Parameter oder bei zu kleiner Angabe wird die Breite passend gewählt. Bei Angabe einer Breite größer als benötigt, wird der Text ggf. rechts mit Leerraum aufgefüllt.

Beispiel:

```
label = Label('ein verbreiterter Text', width=50)
dlg = Dialog('Demo', label)
dlg.run()
```

Edit

Ein bearbeitbares Textfeld. Die Klasse Edit hat folgende Methoden:

`Edit``(value, ...)`

Der Konstruktor hat als einzigen Pflichtparameter:

> `value`: Der anfängliche Inhalt des Textfelds.

Wahlfreie Schlüsselwortparameter:

> `width`: Breite des Feldes in mittleren Buchstabenbreiten

`sel`: Wird ein `(int, int)`-Tupel übergeben, lässt sich damit die Cursorposition bei Aufruf des Dialogs steuern. Sind beide übergebenen Zahlen gleich, wird der Cursor an die entsprechende Position gesetzt. Sind die Zahlen verschieden (und die zweite größer als die erste), wird ein Bereich markiert. Dabei steht `0` für die Cursorposition am Anfang des Eingabefeldes.

Das funktioniert für genau ein Edit-Feld in einem Dialog, das dann auch den Fokus bekommt.

`min`: Vor dem Schließen des Dialogs wird überprüft, ob eine Zahl eingegeben wurde, die mindestens diesen Wert hat.

`max`: Vor dem Schließen des Dialogs wird überprüft, ob eine Zahl eingegeben wurde, die höchstens diesen Wert hat.

Nur wenn weder `min` noch `max` angegeben werden, werden beliebige Texte akzeptiert.

`value``()`

Gibt den (vom Anwender eingegebenen) Text zurück.

`sel``()`

Gibt die aktuelle Cursorposition des Textcursors bzw. die Markierung in Form eines Tupels `(int, int)` zurück. Dabei steht `0` für die Cursorposition am Anfang des Eingabefeldes.

Hinweis: Aktuell (Stand Februar 2025) ist diese Funktion inkonsistent / fehlerhaft implementiert, wenn im Textfeld ein Bereich markiert war: Unter MacOS wird in dem Tupel der Bereich der Selektion zurückgegeben. Unter Windows liefert die Funktion zeimal die Cursorposition in Form eines Tupels aus zwei gleichen Integer-Zahlen. In Mantis erfasst als #9801.

Beispiel:

```
edit = Edit('Bitte diesen Text ändern!', width=40, sel=(6,12))
dlg = Dialog('Demo', edit)
if (dlg.run()):
    messageBox('geänderter Text', edit.value()
        + '\nCursorposition / Markierung: ' + str(edit.sel()))
```

Radio

Eine Gruppe von Radioknöpfen. Die Klasse Radio hat folgende Methoden:

`Radio(items, ...)`

Der Konstruktor hat als Pflichtparameter

`items`: Eine Liste mit den Texten der einzelnen Radioknöpfe.

Wahlfreie Schlüsselwortparameter:

`text`: Wenn dieser Parameter übergeben wird, bekommt die Gruppe einen Rahmen mit diesem Text im oberen Rand

`value`: Index des vorzuwählenden Knopfs (Zählung ab 0)

`value``()`

Gibt den Index des (vom Anwender) markierten Knopfs zurück.

Beispiel:

```
options = ['Frühling', 'Sommer', 'Herbst', 'Winter']
radio = Radio(options, text='Jahreszeit', value=1)
dlg = Dialog('Demo', radio)
if dlg.run():
    messageBox('gewählte Jahreszeit', options[radio.value()])
```

ComboBox

Ein Aufklappfeld mit vorgegebenen Texten. Die Klasse ComboBox hat folgende Methoden:

`ComboBox``(items, ...)`

Der Konstruktur hat den Pflichtparameter

`items`: Eine Liste mit den Texten der ausklappenden Zeilen.

Wahlfreie Schlüsselwortparameter:

`width`: Breite in mittleren Buchstabenbreiten

`value`: Index der vorzuwählenden Zeile (Zählung ab 0)

`value``()`

Gibt den Index der (vom Anwender) gewählten Zeile zurück.

Beispiel:

```
options = ['Violine', 'Viola', 'Violoncello', 'Kontrabass']
combo = ComboBox(options, width=12, value=1)
dlg = Dialog('Instrument', combo)
if dlg.run():
    messageBox('gewähltes Instrument', options[combo.value()])
```

CheckBox

Ein Optionsfeld, in dem ein Häkchen gesetzt werden kann. Die Klasse CheckBox hat folgende Methoden:

CheckBox(text, ...)

Der Konstruktor hat den Pflichtparameter

> `text`: Der Text der neben der CheckBox angezeigt wird

Wahlfreie Schlüsselwortparameter:

> `width`: Breite in mittleren Buchstabenbreiten
>
> `value`: ein Bool-Wert, der angibt, ob die ComboBox beim Aufruf des Dialogs vorausgewählt sein soll oder nicht

value()

> Gibt True zurück, falls das Häkchen (vom Anwender) gesetzt wurde, sonst False.

Beispiel:

```
check = CheckBox('&Zugabe')
dlg = Dialog('CheckBox', check)
if dlg.run() and check.value():
    messageBox('Zugabe', 'wurde gewählt')
```

HBox

Ein (normalerweise unsichtbarer) Rahmen in den mehrerer Dialogelemente in einer horizontalen Reihe von links nach rechts angeordnet werden. Die Klasse HBox hat folgende Methoden:

HBox(content, ...)

Der Konstruktor hat den Pflichtparameter

> `content`: Eine Liste mit den aufzureihenden Elementen.

Wahlfreie Schlüsselwortparameter:

> `padding`: Abstand zwischen den aufgereihten Elementen (Standard: 0)
>
> `text`: Wenn dieser Parameter übergeben wird, bekommt die Box einen Rahmen mit diesem Text im oberen Rand

Beispiel:

```
edit = Edit('', width=4, min=1, max=130)
hBox = HBox([Label('Ich bin'), edit, Label('Jahre alt')],
```

```
            padding=8, text='Ihr Alter')
dlg = Dialog('CheckBox', hBox)
if dlg.run():
    messageBox('Eingegebenes Alter',
               '%d Jahre' % int(edit.value()))
```

VBox

Ein (normalerweise unsichtbarer) Rahmen in den mehrerer Dialogelemente in einer vertikalen Reihe von oben nach unten angeordnet werden. Die Klasse VBox hat folgende Methoden:

`VBox(content, ...)`

Der Konstruktor hat den Pflichtparameter

`content`: Eine Liste mit den aufzureihenden Elementen.

Wahlfreie Schlüsselwortparameter:

`padding`: Abstand zwischen den aufgereihten Elementen (Standard: 0)

`text`: Wenn dieser Parameter übergeben wird, bekommt die Box einen Rahmen mit diesem Text im oberen Rand

Beispiel:

```
edit1 = Edit('', width=4, min=1, max=100)
edit2 = Edit('', width=4, min=1, max=100)
hBox1 = HBox([Label('1. Faktor', width=8), edit1], padding=8)
hBox2 = HBox([Label('2. Faktor', width=8), edit2], padding=8)
vBox = VBox([hBox1, hBox2], padding=8)
dlg = Dialog('Multiplikation', vBox)
if dlg.run():
    a = int(edit1.value())
    b = int(edit2.value())
    messageBox('Produkt:', '%d * %d = %d' % (a, b, a*b))
```

Die Klasse CapXNode zum Auswerten der XML-Stuktur

Die Klasse CapXNode repräsentiert einen Knoten des XML-Strukturbaums, der die Partitur strukturiert speichert. Mit den Methoden dieser Klasse kann man in diesem Strukturbaum navigieren und Attribute auslesen. Eine Änderung von Attributen oder das Hinzufügen von Knoten ist allerdings nicht möglich.

Die globalen Funktion `rootNode()` und `selectedNode(...)` liefern Objekte dieser Klasse zurück. Ihre Methoden:

isEmpty()

Liefert True zurück, wenn das CapXNode-Objekt leer ist, d.h. keinen Knoten im Strukturbaum repräsentiert.

getName()

Liefert einen String mit dem Namen des Knotens zurück.

getAttributes()

Liefert ein Dictionary zurück, dessen Einträge die Attribute des jeweiligen Knotens sind.

getChildNodeCount()

Gibt die Anzahl der Kinder-Knoten als Integer zurück.

getChildNodeI(index)

Gibt den Kind-Knoten mit dem übergebenen Index zurück. Die Indices sind entsprechend der Reihenfolge in der Partitur sortiert, beginnend bei 0. Gibt es unter dem übergebenen Index keinen Kind-Knoten, wird ein leeres CapXNode-Objekt zurückgegeben.

getChildNode(name, index)

Gibt den Kind-Knoten vom Typ name mit dem übergebenen Index zurück. Die Indices sind entsprechend der Reihenfolge in der Partitur sortiert, beginnend bei 0. Im Unterschied zu getChildNodeI werden dabei nur die Knoten vom übergebenen Typ

getParentNode()

Liefert den Elternknoten in Form eines Objekts der Klasse CapXNode zurück. Ruft man die Methode für den Root-Knoten auf, wird ein leeres CapXNode-Objekt zurückgegeben.

Beispiel (das Skript gibt einige Infos über die geöffnete Partitur aus):

```
root = rootNode()
xmlVersion = root.getAttributes()["xmlns"]
text = "Knoten und Attribute: \n"
for i in range(root.getChildNodeCount()):
    node = root.getChildNodeI(i)
    text += node.getName() + ": "
         + str(node.getChildNodeCount()) + " Kinde(r)\n"
    attr = node.getAttributes()
    for a in attr:
        text += a + ": " + attr[a] + "\n"
systems = root.getChildNode("systems", 0)
graphicPage = systems.getChildNode("graphicPage", 0)
if graphicPage.isEmpty():
```

```
    text += "Partitur enthält keine Grafikseiten."
else:
    text += "Partitur enthält mindestens eine Grafikseite."
messageBox(xmlVersion, text)
```

Klassen der Partiturelemente

Diese Klassen repräsentieren die Elemente einer Partitur. Sie stehen in einer hierarchischen Ordnung zueinander. In diesem Abschnitt werden die einzelnen Klassen vorgestellt. Zum generellen Aufbau einer Partitur und zum Zugriff auf die Elemente einer Partitur in einem internen Skript, siehe S. 12

Score

Diese Klasse steht für eine in capella geöffnete Partitur. Ein Objekt dieser Klasse kann mit der globalen Funktion `activeScore()` gewonnen werden. Die Klasse Score hat folgende Methoden:

`pathName``()`

Gibt den absoluten Pfad an (einschließlich des Dateinamens), unter dem die Partitur zuletzt gespeichert wurde. Falls die Partitur noch nicht gespeichert wurde, wird der Fenstertitel (z. B. „Partitur 1“) zurückgegeben.

`title``()`

Seit Version *capella* 8 identisch zu `pathName()`

`voiceList``()`

Liefert eine Liste der Bezeichnungen aller Stimmen des Mustersystems der Partitur. Diese kann z. B. zum Füllen einer Combobox im *capella*-Dialogbaukasten verwendet werden.

`registerUndo``(text)`

Diese Methode muss von einem Skript vor der ersten Änderung einer Partitur aufgerufen werden, damit die Skriptausführung rückgängig gemacht werden kann. Der im Parameter text übergebene String erscheint nach Ausführung des Skripts im Rückgängig-Menü. Zur korrekten Verwendung in einem Skript siehe S. 31

`flushUndoBuffer``()`

Hiermit wird das gesamte Protokoll zum Rückgängig machen gelöscht.

`write``(fileName, xml=1)`

Speichert die Partitur unter `fileName` und zwar im CapXML-Format, falls der Parameter xml den Wert 1 hat (oder weggelassen wird) bzw. im binären Format (bis *capella* Version 2002), für xml=0. Beispiel:

```
dlg = FileDialog(False)
dlg.setTitle('capella-Datei im alten Binärformat abspeichern')
dlg.addFilter('capella-Dateien', '*.cap')
dlg.setStartPath(getPersonalDataDir())
if dlg.run():
    activeScore().write(dlg.filePath(), 0)
```

read(fileName)

Die aktive Partitur wird durch die in der Datei `fileName` gespeicherte Partitur ersetzt.

Mit `write()` und `read()` lassen sich externe Skripte realisieren, die sich nahtlos in capella integriefen und sich aus Nutzersicht nicht von internen Skripten unterscheiden. Siehe S. 19

isUsingAutoPlacement()

Gibt zurück (als `bool`), ob bei dieser Partitur neue Grafikobjekte / Liedtexte als Auto-Elemente eingefügt werden. Entspricht dem „`useAutoPlacement`“-Attribut des „`editOptions`“-Knoten im CapXML, bzw. dem Schalter „Neue Objekte automatisch platzieren“ in der *capella*-GUI.

Python-Skripte, die neue Objekte einfügen, sollten diese Eigenschaft abfragen, und die '`placement`'-Eigenschaft der neuen Objekte entsprechend setzen.

nPages()

Gibt die Anzahl der Druckseiten der Partitur zurück.

nSystems()

Gibt die Anzahl der Systeme der Partitur zurück.

system(i)

Gibt das i-te System der Partitur zurück, wobei das erste System der Partitur den Index 0 hat. Dieses ist vom Typ System. Wenn alle Systeme ausgewertet werden sollen, ist es einfacher, die Methode systems zu verwenden:

systems()

Gibt einen Iterator über alle Systeme der Partitur zurück. Diese sind jeweils vom Typ `System`. Man kann z. B. mit

```
for s in activeScore().systems():
    ...
```

über die Systeme der Partitur iterieren. Wenn die komplette Liste der Systeme benötigt wird, können man diese mit `list(activeScore().systems())` erzeugen. Diese Anmerkung gilt sinngemäß für alle weiteren im Folgenden beschriebenen Iterator-Funktionen.

`staves``()`

Gibt einen Iterator über alle Notenzeilen (vom Typ `Staff`) der Partitur zurück. Wenn man direkt alle Notenzeilen einer Partitur anpassen möchte, erspart das die explizite Iteration über Systeme. Beispiel:

```
for s in activeScore().staves():
    ...
```

ist gleichbedeutend mit:

```
for s in activeScore().systems():
    for z in s.staves():
        ...
```

`voices``()`

Gibt einen Iterator über alle Stimmen der Partitur zurück. Diese sind vom Typ `Voice`.

`noteObjs``()`

Gibt einen Iterator über alle Notenobjekte (vom Typ `NoteObj`) der Partitur zurück.

`heads``()`

Gibt einen Iterator über alle Notenköpfe (vom Typ Head) der Partitur zurück.

`deleteTaggedGraphics``(tag)`

Löscht in der gesamten Partitur alle Grafikobjekte mit der Kennung `tag`. Siehe S. 37

`exportGraphics``(filePath, part, n, dpi, format, quality)`

Exportiert einen Teil der Partitur als Rastergrafik, Vektorgrafik oder PDF. Parameter:

`filePath`: Absoluter Pfadname der Zieldatei inkl. Dateiendung. Anhand dieser Dateiendung wird das Exportformat festgelegt. Folgende Formate werden unterstützt: png, bmp, tif, jpg, svg und pdf.

`part`: wird seit *capella 8* nicht mehr genutzt und wird ignoriert

`n`: Bei Rastergrafikformaten und svg ist `n` der Index der Seite gezählt ab 0. Beim Export als PDF wird immer die gesamte Partitur exportiert.

`dpi`: Auflösung in dpi (bei Rastergrafikformaten)

`format`: wird seit Version 8 von *capella* nicht mehr genutzt und wird ignoriert

`quality`: Qualität (nur bei Export im jpg-Format – Bereich 0 bis 100)

Beispiel (exportiert alle Seiten der aktuellen Partitur als jpg-Dateien mit 150 dpi)

```
if activeScore():
    origName = activeScore().pathName().split('.')
    for i in range(activeScore().nPages()):
        name = origName[0] + '-%d.jpg' % (i+1)
        activeScore().exportGraphics(name, 0, i, 150, 0, 80)
```

System

Diese Klasse steht für ein System einer in capella geöffneten Partitur. Objekte dieser Klasse können mit den Methoden `system()` und `systems()` der Klasse `Score` gewonnen werden (siehe dort). Die Klasse `System` hat folgende Methoden:

`index``()`

Gibt den Index des Systems innerhalb der Partitur zurück (gezählt ab 0 für das erste System). Mit diesem Index kann man das System mit der Methode system() der Klasse `Score` ansprechen (sofern man die Methode für dasjenige Objekt aufruft, das die Partitur repräsentiert, zu der das System gehört).

`get``(key)`

Als Parameter `key` kann einer der folgenden Strings übergeben werden: `'tempo'`, `'beamGrouping'`, `'pageBreak'`, `'leftIndent'`, `'leftIndentSingle'`, `'rightIndent'`, `'rightIndentSingle'`, `'bracketColor'`, `'justified'`, `'instrNotation'`, `'pageBreakSingle'`, `'justifiedSingle'`. Der Rückgabewert gibt den aktuellen Wert des Attributs an. Zur Bedeutung der einzelnen keys hilft ein Blick in die Dokumentation der XML-Struktur einer capella-Datei. Diese ist identisch aufgebaut wie die Klassenstruktur und enthält die gleichen Attribute. Siehe S. 9

`getAttributes``()`

Diese Methode gibt alle Attribute in einem Dictionary zurück. Vgl. Methode `get`.

`set``(key, value)`

Hier kann als Parameter `key` einer der Strings angegeben werden wie auch bei `get(key)`. Der Parameter `value` muss einen dazu passenden Wert bekommen.

Zum Format der einzelnen keys ist die Dokumentation der XML-Struktur einer capella-Datei eine gute Quelle. Siehe S. 9

Zudem ist es hilfreich vorher mit der Methode `get` zu experimentieren. Ein Beispiel:

```
p = activeScore()
if p and p.nSystems() > 2:
    p.registerUndo('Attribut-Test')
    s = ''
    att = p.system(0).getAttributes()
    for key in att.keys():
        s += '%s : %s\n' % (key, str(att[key]))
    messageBox('System-Attribute', s, 1)
    att = dict(tempo=99, justified=1, leftIndent=30)
    p.system(0).setAttributes(att)
    p.system(1).setAttributes(dict(tempo=99,
                                   justified=1,
                                   leftIndent=20))
    p.system(1).set('bracketColor', Color.RGB(255,0,64))
    p.system(2).set('leftIndent', 10)
```

Dieses Skript demonstriert mehrere Techniken: System-Attribute ermitteln und anzeigen, mehrere System-Attribute mit einem Dictionary im Verbund ändern sowie einzelne System-Attribute ändern. Es funktioniert bei Partituren mit mindestens 3 Systemen.

setAttributes(a)

Vgl. Methode `set`. Diese Methode setzt alle Attribute, die in einem Dictionary übergeben werden. Die Struktur muss dem Dictionary entsprechen, wie es von `getAttributes` geliefert wird.

nStaves()

Gibt die Anzahl der Notenzeilen des Systems zurück.

staff(i)

Gibt die i-te Notenzeile (Zählung ab Null!) des Systems zurück. Dieses ist vom Typ `Staff`. Wenn alle Notenzeilen ausgewertet werden sollen, ist die Methode `staves` einfacher zu verwenden.

staves()

Gibt einen Iterator über alle Notenzeilen des Systems zurück. Diese sind jeweils vom Typ `Staff`.

voices()

Gibt einen Iterator über alle Stimmen des Systems zurück. Diese sind jeweils vom Typ `Voice`. Erspart die explizite Iteration über Notenzeilen.

noteObjs()

Gibt einen Iterator über alle Notenobjekte des Systems zurück (vom Typ `NoteObj`). Erspart die explizite Iteration über Notenzeilen und Stimmen.

heads()

Gibt einen Iterator über alle Notenköpfe des Systems zurück (vom Typ Head). Erspart die explizite Iteration über Notenzeilen, Stimmen und Notenobjekte.

getBarCount()

Gibt ein Dictionary zurück, dass mit den Schlüsseln 'reset' und 'nAdd' zurück. Die (Integer-)Werte hinter den Schlüsseln haben folgenden Inhalt:

reset:

0 – die Taktnummerierung wird gegenüber dem vorherigen System fortgesetzt.

1 – die Taktnummerierung wird zurückgesetzt

nAdd – gibt die Anzahl Takte an, die bei der Taktnummerierung in diesem System hinzuaddiert oder abgezogen werden

setBarCount(d)

Übergibt man ein Dictionary d in der gleichen Struktur, wie unter getBarCount erläutert, lassen sich damit die Parameter der Taktnummerierung für das System setzen.

Beispiel:

```
p = activeScore()
p.registerUndo('BarCount-Test')
for i in p.systems():
    d = i.getBarCount()
    messageBox('getBarCount vor Anpassung', str(d))
    if d["reset"]==0:
        d["reset"]=1
    else:
        d["reset"]=0
    d["nAdd"]+=1
    i.setBarCount(d)
    d = i.getBarCount()
     messageBox('getBarCount nach Anpassung', str(d))
```

Das Beispiel invertiert den Wert von reset und Erhöht den Wert von nAdd.

pitches(time, continued=False, diatonic=True)

Liefert eine Liste aller Tonhöhen des Systems (über alle Stimmen) an der angegebenen Zeit-Position. Die Methode hat folgende Parameter:

time: Zeit-Position, gemessen in Notenwerten ab Beginn des Systems als Rational-Objekt, siehe S. 45

`continued`: Wenn hier True übergeben wird, werden auch Noten berücksichtigt, die vor der Zeit time beginnen bzw. länger als bis time dauern. Bei False werden nur Noten geliefert, die zu diesem Zeitpunkt beginnen.

`diatonic`: Wenn hier True übergeben wird, werden die Töne als diatonische Noten ausgegeben (Paare aus Schritt und Alteration), sonst (Vorgabe) enharmonisch).

Beispiel:

```
diatonic = str(activeScore().system(0).pitches(0))
enharmonic = str(activeScore().system(0).pitches(0,
                     True, False))
messageBox('Töne', 'Diatonisch: ' + diatonic
                 + '\nEnharmonisch: ' + enharmonic)
```

Wenn am Anfang des ersten Systems die Noten c, e, gis (eingestrichen) und das zweigestrichene des stehen, wird ausgegeben:

Diatonisch: [(35,0),(37,0),(39,1),(43,-1)]
Enharmonisch: [60,64,68,73]

`staffIndexFromDescr``(descr)`

Gibt den Index der Notenzeile an, die für die die Beschreibung im Mustersystem dem übergebenen String `descr` entspricht. Falls keine entsprechende Zeile vorhanden ist, wird -1 zurückgegeben.

Staff

`index``()`

Gibt den Index an, unter dem die Notenzeile mit der Methode `staff()` des Systems, in der sie sich befindet, angesprochen wird.

`defaultMeter``()`

Gibt die Dauer des Standardtakts der Notenzeile als rationale Zahl (Typ: `Rational`) wieder. Das ist die Taktart, die in einer Zeile gilt, wenn am Zeilenbeginn keine explizite Taktangabe in der Partitur verzeichnet wird. Diese Angabe verwendet *capella*, um die Taktstrich-Automatik auch über Systemwechsel hinweg zu gewährleisten.

`setColor``(c)`

Setzt die Farbe der Notenlinien. Als Parameter muss eine Instanz der Klasse `Color` übergeben werden.

```
activeScore().registerUndo('Zeilen regenbogenbunt färben')
c = 0
```

```
for s in activeScore().staves():
    s.setColor((Color.red, Color.orange, Color.yellow,
                Color.green, Color.cyan, Color.blue,
                Color.magenta)[c])
    c = (c + 1) % 7
```

transpose(step)

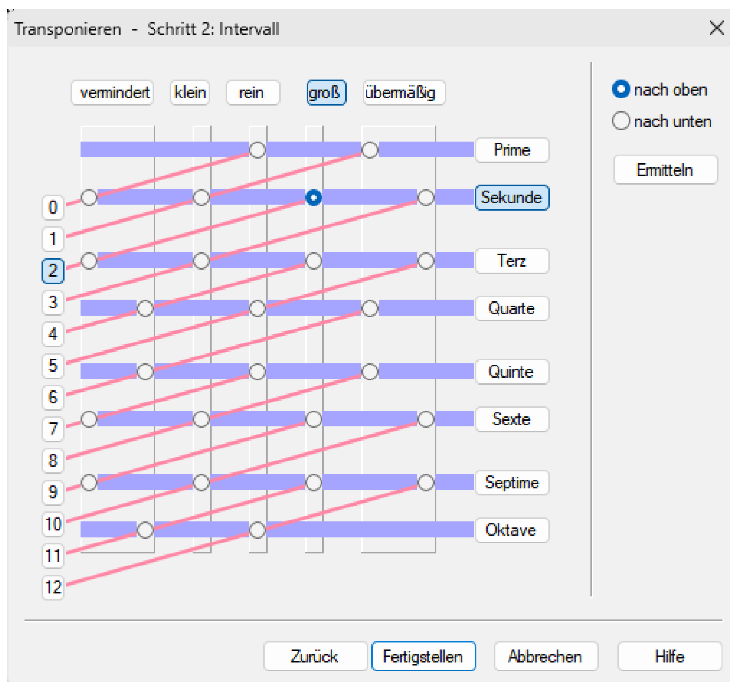
Die Funktion `transpose` führt eine Transponierung aller Notenobjekte der Notenzeile aus (mit Tonartwechsel). Die Methode erwartet als Parameter ein Tubel aus zwei Integerzahlen. Diese spezifizieren das Intervall, um das die Notenzeile transponiert werden soll.

Die erste Zahl gibt das diatonische Intervall an: Prime (0), Sekunde (1), Terz (2), Quarte (3)... Negative Werte haben zur Folge, dass die Zeile abwärts transponiert wird.

Die zweite Zahl des Tupels steht für die Ausprägung. Mögliche Werte sind vermindert (0), klein (1), rein (2), groß (3) und übermäßig (4).

Wird ein Intervall spezifiziert, das es nicht gibt (etwa eine "reine Sexte" mit `transpose([5,2])`), wird immer um eine übermäßige Sekunde mit drei Halbtonschritten nach oben bzw. unten transponiert. Sollte sich beim Transponieren eine Tonart ergeben, die es nicht gibt, wird automatisch enharmonisch verwechselt. Transponiert man etwa von Cis-Dur um eine reine Quinte nach oben, käme eigentlich die Tonart Gis-Dur heraus, für die es keine Vorbezeichnung gibt. Statt dessen erhält man als Zieltonart As-Dur.

Diese Definitionen haben ihren Ursprung in dem Dialog "Transponieren - Schritt 2: Intervall" aus capella 7 und früheren Versionen.



`nVoices()`

Gibt die Anzahl der Stimmen der Notenzeile zurück.

`voice(i)`

Gibt die i-te Stimme der Notenzeile zurück, wobei ab 0 gezählt wird. Der Rückgabewert ist vom Typ `Voice`. Wenn alle Stimmen ausgewertet werden sollen, ist die Methode `voices()` einfacher zu verwenden.

`voices()`

Gibt einen Iterator über alle Stimmen der Notenzeile zurück. Diese sind jeweils vom Typ `Voice`.

`noteObjs()`

Gibt einen Iterator über alle Notenobjekte der Notenzeile zurück. Erspart die explizite Iteration über Stimmen.

`heads()`

Gibt einen Iterator über alle Notenköpfe der Notenzeile zurück. Erspart die explizite Iteration über Stimmen und Notenobjekte.

Voice

`index()`

Gibt den Index an, unter dem die Stimme mit der Methode `voice()` der Notenzeile, in der sie sich befindet, angesprochen wird.

`nNoteObjs()`

Gibt die Anzahl der Notenobjekte der Stimme zurück.

`noteObj(i)`

Gibt das i-te Notenobjekt der Stimme zurück. Die Zählung beginnt bei 0. Dieses ist vom Typ `NoteObj`. Wenn alle Notenobjekte ausgewertet werden sollen, ist die Methode `noteObjs()` einfacher zu verwenden.

`noteObjs()`

Gibt einen Iterator über alle Notenobjekte der Stimme zurück. Diese sind jeweils vom (polymorphen) Typ `NoteObj`.

`heads()`

Gibt einen Iterator über alle Notenköpfe der Stimme zurück. Erspart die explizite Iteration über Notenobjekte.

NoteObj

Die Klasse NoteObj steht für ein Notenobjekt. Das können nicht nur Noten sein, sondern auch alle anderen Elemente in einer Notenzeile, die mit einem Cursor angesteuert werden können. Zur Unterscheidung sind die folgenden Konstanten definiert:

Konstante	Bedeutung
`REST`	Pause
`CHORD`	Akkord/Note
`CLEF`	Schlüssel
`KEY`	Tonart
`METER`	Taktangabe
`EXPL_BARLINE`	expliziter Taktstrich
`IMPL_BARLINE`	impliziter Taktstrich
`PAGE_BKGR`	Seitenhintergrund-Objekt

Ein impliziter Taktstrich ist ein Taktstrich, der in der Arbeitsansicht (Farbansicht) grün gezeichnet wird. Er wird automatisch aufgrund der aktuellen Taktangabe und der bis-

lang vorgekommenen Notenwerte gesetzt und ist kein eigenständiges Notenobjekt, wird also bei der Zählung und bei der Iteration über die Notenobjekte einer Stimme nicht berücksichtigt. Man kann ihn aber mit der Methode `NoteObj::implBarline()` ansprechen.

Das Seitenhintergrund-Objekt ist eine unsichtbares Notenobjekt, das keinem der sichtbaren Systeme zugeordnet ist, sondern einem unsichtbaren System, das in jeder Partitur genau einmal existiert. An das Seitenhintergrund kann man alle Grafikobjekte anhängen (z.B. Werktitel, Seitenzahlen etc.). Dabei gelten einige besondere Regeln, z.B. die Sichtbarkeitsregeln „auf der ersten Seite", „auf allen Seiten", „auf ungeraden Seiten", „auf geraden Seiten". Dieses System kann man (seit *capella* 10) mit `activeScore().system(-1)` ansprechen. Das Seitenhintergrundobjekt erhält man dann mit `activeScore().system(-1).staff(0).voice(0).noteObj(0)`. Noch einfacher ist die Methode `activeScore().pageBackgroundObj()`.

`subType`()

Gibt den Untertyp des Notenobjekts zurück (einer der unter Konstanten dieser Klasse angegebenen Werte).

Beispiel:

```
for n in activeScore().noteObjs():
    s = 'subType: '+ str(n.subType())
    messageBox(s, '\nzum Vergleich:\n'
            + 'NoteObj.REST = ' + str(NoteObj.REST) + '\n'
            + 'NoteObj.CHORD = ' + str(NoteObj.CHORD) + '\n'
            + 'NoteObj.CLEF = ' + str(NoteObj.CLEF) + '\n'
            + 'NoteObj.KEY = ' + str(NoteObj.KEY) + '\n'
            + 'NoteObj.METER = ' + str(NoteObj.METER) + '\n'
            + 'NoteObj.EXPL_BARLINE = '
            + str(NoteObj.EXPL_BARLINE) + '\n'
            + 'NoteObj.IMPL_BARLINE = '
            + str(NoteObj.IMPL_BARLINE) + '\n'
            + 'NoteObj.PAGE_BKGR = '
            + str(NoteObj.PAGE_BKGR))
```

Eleganter als die Abfrage über subType() und den Vergleich mit Konstanten ist die Nutzung der Methoden `isChord()`, `isRest()`, `isBarline()` etc. Die Beschreibungen dazu finden sich weiter hinten in diesem Abschnitt.

Die Position und Dauer von `NoteObj`-Objekten:

`index`()

Gibt den Index an, unter dem das Notenobjekt mit der Methode `noteObj()` der Stimme, in der es sich befindet, angesprochen wird.

duration()

Liefert die Spieldauer als rationale Zahl (Typ: `Rational`) des Notenobjekts (Dauer in ganzen Noten).

time()

Gibt die Zeit (in ganzen Noten) vom Anfang der Stimme bis zu diesem Objekt als Gleitkommazahl zurück. Achtung: Da hierbei Gleitkomma-Rundungsfehler auftreten können, sollten Sie nicht auf exakte Übereinstimmung testen. Sie erhalten die genaue Zeit ohne Rundungsfehler als rationale Zahl, wenn Sie die einzelnen Notenwerte der Stimme addieren (vgl. Methode `duration()`).

Methoden für Pausen:

isRest()

Gibt True zurück, falls das Notenobjekt eine Pause ist, sonst False.

fullMeasures()

Gibt True zurück, falls das Notenobjekt eine Mehrtaktpause ist, sonst False.

Methoden für Noten und Akkorde:

isChord()

Gibt True zurück, falls das Notenobjekt ein Akkord (oder eine Note) ist, sonst False.

nHeads()

Gibt die Anzahl der Notenköpfe des Notenobjekts zurück. Falls das Notenobjekt kein Akkord (bzw. Note) ist, ist der Rückgabewert Null.

head(i)

Gibt den i-ten Notenkopf (Zählung ab Null!) des Notenobjekts zurück. Dieses ist vom Typ Head. Wenn alle Notenköpfe ausgewertet werden sollen, ist die Methode `heads()` einfacher zu verwenden.

heads()

Gibt einen Iterator über alle Notenköpfe des Notenobjekts zurück. Diese sind jeweils vom Typ Head. Falls das Notenobjekt kein Akkord ist, ist die Liste leer.

diatonicPitch(i)

Gibt die diatonische Höhe des i-ten Notenkopfs des Akkords zurück. Siehe S. 72

chromaticPitch(i)

Liefert die chromatische Tonhöhe des i-ten Notenkopfs in MIDI-Einheiten (60 = eingestrichenes C usw.).

Methoden für Taktstriche, Taktangaben und Tonart

`isBarline``()`

Gibt True zurück, falls das Notenobjekt ein expliziter Taktstrich ist, sonst False.

`implBarline``()`

Gibt an, ob an der Stelle dieses Notenobjekts ein impliziter Taktstrich steht (Hinweis: der implizite Taktstrich ist kein eigenes Objekt).

`meter``()`

Gibt den Wert des Takts in ganzen Noten als rationale Zahl (Typ: `Rational`) zurück, falls das Objekt eine Taktangabe ist, sonst Null. Da `Rational`-Objekte rationale Zahlen in gekürzter Form enthalten, lassen sich damit z.B. 3/4-Takte nicht von 6/8-Takten unterscheiden.

`curKey``()`

Gibt die zu Beginn des Objektes geltende Tonartenvorzeichnung an. Falls das Objekt selbst eine Tonartvorzeichnung ist, wird die vorher geltende Tonart zurückgegeben.

Der Rückgabewert hat den Typ `int`. Der Wert steht für die Position der Tonart im Quintenzirkel bzw. die Anzahl der Vorzeichen:

- 0 steht für „keine Vorzeichen“, also C-Dur bzw. a-Moll
- positive Zahlenwerte stehen für Kreuztonarten. Z.B. wird D-Dur/h-Moll mit 2 kodiert, H-Dur/gis-Moll mit 5 etc.
- negative Zahlenwerte stehen für B-Tonarten. -1 steht für F-Dur/d-Moll, -3 für Es-Dur/c-Moll etc.

Methoden zu Position und Erscheinungsbild

`posX``(shifted=True)`

Liefert die horizontale Position des Notenobjekts in Notenlinienabständen als Gleitkommazahl. Der Parameter shifted entscheidet darüber, ob eine Verschiebung (mit Strg+J) des Notenobjekts mit berücksichtigt werden soll.

`setColor``(c)`

Setzt die Farbe des Notenobjekts auf c. c ist dabei ein RGB-Wert, der mit der Funktion RGB() definiert oder aus den vorfefinierten Farben der Klasse Color gewählt werden kann.

Methoden zu angehängten Grafikobjekten

Eine Beschreibung der allgemeinen Vorgehensweise mit Grafikobjekten siehe S. 72

`nDrawObjs``()`

Gibt die Anzahl der Grafikobjekte des Notenobjekts an.

`drawObj``(i)`

Gibt das i-te Grafikobjekt (Zählung ab Null) des Notenobjekts zurück. Voraussetzung ist, dass das Notenobjekt vom Typ CHORD oder REST ist und mindestens i+1 Grafikobjekte besitzt. Der Rückgabewert ist ein Python-Dictionary. Einzelheiten dazu siehe S. 72

`drawObjs``()`

Gibt eine Liste der Grafikobjekte des Notenobjekts zurück. Die Elemente dieser Liste sind Python-Dictionaries. Einzelheiten dazu siehe S. 72

`addDrawObj``(obj)`

Erzeugt ein neues Grafikobjekt und hängt es an den Akkord. Voraussetzung ist, dass das Notenobjekt vom Typ CHORD oder REST ist. Der Parameter `obj` ist Dictionary, das das Grafikobjekt definiert. Einzelheiten dazu siehe S. 72

`deleteDrawObj``(i)`

Löscht das i-te Grafikobjekt (Zählung ab Null). Voraussetzung ist, dass das Notenobjekt vom Typ CHORD oder REST ist und mindestens i+1 Grafikobjekte besitzt. Wenn mehrere Grafikobjekte am gleichen Notenobjekt löschen werden sollen, muss bedacht werden, dass sich die Indizes verschieben. Wenn z.B. die ersten zwei Grafikobjekte gelöscht werden sollen, können entweder (Zählung ab 0!) erst `deleteDrawObj(1)` und dann `deleteDrawObj(0)` aufgerufen werden oder zweimal hintereinander `deleteDrawObj(0)`.

Beispiel:

```
for note in activeScore().noteObjs():
    i = note.nDrawObjs()-1
    while i >= 0:
        d = note.drawObj(i)
        if d['type'] == 'bracket':
            note.deleteDrawObj(i)
            i -= 1
```

Es werden alle Triolenklammern in der ganzen Partitur gelöscht, wobei die Zählvariable `i` rückwärts läuft, um eine Indexverschiebung zu vermeiden.

`replaceDrawObj``(i, obj)`

Ersetzt das i-te Grafikobjekt (Zählung ab Null) durch ein Grafikobjekt auf Basis des übergebenen Dictionarx `obj`. Voraussetzung ist, dass das Notenobjekt vom Typ CHORD oder REST ist und mindestens i+1 Grafikobjekte besitzt. Einzelheiten dazu siehe S. 72

`textSize``(text)`

Gibt die Größe (als Tupel aus Breite und Höhe) zurück, die ein Einfachtext einnimmt, der an das Notenobjekt gehängt wird. Als Parameter `text` wird ein Dictionary übergeben, das das Textobjekt beschreibt. Damit können mehrere Einfachtexte zueinander ausgerichtet werden, auch wenn sie unterschiedliche Fonts benutzen.

Ein Skript, das von dieser Methode Gebrauch macht, ist das von *capella* mitgelieferte Skript `chordSymbols.py`.

Head

`index``()`

Gibt den Index an, unter dem der Notenkopfs mit der Methode head() des Akkordes, in dem er sich befindet, angesprochen wird.

`chromaticPitch``()`

Gibt die MIDI-Tonhöhe des Notenkopfs zurück.

`diatonicPitch``()`

Gibt die diatonische Tonhöhe des Notenkopfs als Tupel aus Tonstufe und Alteration zurück. Das eingestrichene C entspricht der diatonischen Tonhöhe (35, 0). Beispiel:

```
for head in activeScore().heads():
    messageBox('diatonicPitch', str(head.diatonicPitch()))
    break
```

Das Skript gibt die diatonische Höhe des ersten Notenkopfs der Partitur aus.

Grafikobjekte

In capella können Grafikobjekte entweder an einem Notenobjekt (Akkord, Pause, fester Taktstrich) oder am Seitenhintergrund-Objekt (das ebenfalls als Klasseninstanz von `NoteObj` erscheint) verankert werden. Weiteres zur Klasse `NoteObj` siehe S. 67

Für Grafikobjekte bietet die Klasse `NoteObj` mehrere Methoden an: Den Methoden `addDrawObj()` und `replaceDrawObj()` werden Grafikobjekte übergeben, die Methoden `drawObj()` und `drawObjs()` liefern Grafikobjekte zurück. Details siehe Siehe S. 71

Grafikobjekte werden in internen Skripts durch Python-Dictionaries repräsentiert. Ihr Aufbau orientiert sich an der capXML-Struktur. Deshalb finden Sie in der capXML-Sprachbeschreibung im Zweifelsfall weitere Hinweise. Siehe auch S. 9

Die wichtigsten Unterschiede zwischen capXML und den Python-Dictionaries sind:

- In den Dictionaries wird nicht zwischen Attributen und Elementen unterschieden.
- Der Inhalt des Unterelements basic aus capXML wird ins Dictionary des Grafikobjekts integriert.
- In einigen Fällen werden Attribute durch Zahlen statt durch benannte Werte angegeben.

In den folgenden Abschnitten wird die Struktur der Python-Dictionaries für die einzelnen Grafik-Objekt-Typen tabellarisch dargestellt. Bei jedem Schlüssel-Wert-Paar ist dabei der **Datentyp** des Wertes angegeben. Wenn der Typ `float` ist, bedeutet eine Angabe wie `(Zw.32)` oder `(Zw.10)` dahinter, dass der Wert in Notenlinien-Zwischenräumen angegeben wird und in *capella* in ein Raster von 1/32 bzw. 1/10 Zwischenräumen gerundet wird. Beim Typ `float (Zw.10)` macht es demnach keinen Unterschied, ob der Wert 0.4 oder 0.36 übergegeben wird.

Wenn als Standardwert „keiner“ angegeben ist, muss für das Attribut beim Erzeugen eines Dictionarys unbedingt ein Wert angegeben werden.

Wenn ein Standardwert angegeben ist, muss man bei der Abfrage des Dictionaries den Fall berücksichtigen, dass der Wert nicht vorhanden ist. Statt

```
wert = d['key']
```

sollte man also

```
wert = d.get('key', Standardwert)
```

schreiben.

Gemeinsam für alle Grafikobjekte

Die Schlüssel in dem folgenden Abschnitt tauchen bei allen bzw. mehreren Typen Grafikobjekten auf.

Typ, Gültigkeit und Zuordnung eines Grafikobjekts

Schlüssel	Typ	Bedeutung	Standard-wert
`'type'`	`str`	Typ des Grafikobjekts. Gültige Werte: `'group'` – Gruppe von Grafikobjekten	keiner, muss immer

		Grafikelemente der Notenschrift: `'bracket'` – Triolen-Klammer `'guitar'` – Gitarrengriffsymbol `'notelines'` – Notenlinien `'octaveClef'` – Oktavierungsklammer `'transposable'` – transponierb. Obj. `'slur'` – Binde-/Legatobogen `'trill'` – Trillerschlange `'volta'` – Voltenklammer `'wavyLine'` – Schlangenlinie `'wedge'` – (De)Crescendo-Keil **elementare Formen:** `'ellipse'` – Ellipse `'line'` – gerade Linie `'polygon'` – Polygon `'rectangle'` – Rechteck **importierte Grafiken:** `'metafile'` – Windows-Metafile-Grafik `'pixmap'` – Rastergrafik `'svg'` – Vektorgrafik **Textobjekte:** `'richText'` – Textfeld im RTF-Format `'text'` – Einfachtext	mitgegeben werden
`'scope'`	`str`	bestimmt die Gültigkeit des Grafikobjekts, sofern es an einem regulären Notenobjekt (Akkord, Pause, expliziter Taktstrich) verankert ist. Details (insbes. bzgl. dem Unterschied zur Sichtbarkeit siehe *capella*-Handbuch). Gültige Werte: `'voice'` – Objekt gilt für die Stimme `'staff'` – Objekt gilt für die Notenzeilen `'brace'` – Objekt gilt für alle Notenzeilen einer geschweiften Klammer `'bracket'` – Objekt gilt für alle Notenzeilen einer eckigen Klammer `'system'` – Objekt gilt für das System	`'voice'`
`'horizPageAlign'`	`int`	Anker-Bezug bei seitenverankerten Objekten: 0 = links, 1 = mittig, 2 = rechts	`0`
`'vertPageAlign'`	`int`	Anker-Bezug bei seitenverankerten Objekten: 0 = oben, 1 = mittig, 2 = unten	`0`
`'tag'`	`str`	Eine gültige Grafikkennung, siehe S. 37	`'0-0'`

Verankerung und Sichtbarkeit eines Grafikobjekts

Schlüssel	Typ	Bedeutung	Standard-wert
'noteRange'	int	Bei doppelt verankerten Objekten (z.B. Bindebögen, Crescendogabeln,...): Anzahl der Notenobjekte vom Start- zum Endanker.	0
'continued'	bool	gibt an, ob ein Grafikobjekt die Fortsetzung eines anderen Grafikobjekts im vorhergehenden System ist (z.B. Bindebögen und Crescendo-Gabeln, die über einen Systemumbruch reichen.)	False
'drawNormal'	bool	wird nur gebraucht, wenn das Attribut 'continued' auf True gesetzt ist. 'drawNormal' gibt dann an, dass das Grafikobjekt trotzdem normal gezeichnet werden soll (wird typischerweise nur für Volta-Objekte genutzt). Gilt nur für Objekte, die an Akkorden/Pausen oder expliziten Taktstrichen verankert sind.	False
'visibilityFlags'	int	Übergeben wird eine Integer-Zahl im Bereich 0-31, die als Summe von Zweierpotenzen zu lesen ist. Je nachdem, ob einer der folgenden Zahlenwerte als Summand verwendet wird, ist die jeweilige Option aktiviert oder nicht aktiviert. 1 – sichtbar, wenn Hals nach oben 2 – sichtbar, wenn Hals nach unten Hierbei kommt es bei Grafikobjekten, die an mehreren Notenobjekten verankert sind auf die Halsrichtung der Note am ersten Anker an. 4 – sichtbar in Oberstimme 8 – sichtbar in Hauptstimme 16 – sichtbar in Unterstimme 32 – sichtbar in Gesamtpartitur 64 – sichtbar in Einzelstimmenauszug Beispiel: Ist dieser Wert im Dictionary mit 109 abgelegt, ist das Grafikobjekt sichtbar, sofern die Note, an der es verankert ist, nach oben gestielt ist und in der Ober- oder Hauptstimme der Notenzeile steht und unabhängig davon, ob die Gesamtpartitur	127, d.h. das Objekt ist immer sichtbar

Schlüssel	Typ	Bedeutung	Standard-wert
		oder ein Einzelstimmenauszug angezeigt wird. Es sollte nur in Ausnahmefällen vom Standardwert abgewichen werden.	
'onlySingle'	bool	gibt an, ob das Objekt ausschließlich dann sichtbar ist, wenn der Stimmauszug aktiviert ist	False

Erscheinungsbild

Schlüssel	Typ	Bedeutung	Standard-wert
'behindNotes'	bool	gibt an, ob das Objekt hinter den Noten gezeichnet wird	False
'color'	int	Wenn 'type' einen Wert aus den Gruppen • Grafikelemente der Notenschrift • Elementare Formen • Textobjekte hat, kann für das Objekt eine Farbe festgelegt werden. Es handelt sich um eine Integer-Zahl in folgendem Format: color = 65536 * blau + 256 * grün + rot wobei rot, grün und blau für die Werte der RGB-Grundfarben im Wertebereich 0..255 handelt.	0 = schwarz

Platzierung eines Grafikobjekts

Folgende Schlüssel steuern die Platzierung eines Grafikobjekts:

Schlüssel	Typ	Bedeutung	Standard-wert
'placement'	str	Legt fest, ob das Objekt automatisch oder manuell platziert wird. Sofern die Auto-Platzierung aktiviert ist, werden mitunter Parameter ignoriert, die die Platzierung der Objekte beschreiben, Details dazu siehe	'manually'

Schlüssel	Typ	Bedeutung	Standard-wert
		capella-Handbuch. Werte: `'auto'` – das Objekt wird horizontal und vertikal automatisch platziert `'semiAuto'` – die horizontale Position des Objekts wird manuell festgelegt, die vertikale Position automatisch `'manually'` – manuelle Platzierung	
`'yPlacement'`	`str`	Bei automatischer oder halbautomatischer Platzierung legt dieser Parameter fest, ob das Objekt über oder unter der Notenzeile erscheinen soll. Werte: `'auto'` – es wird automatisch entschieden, ob das Objekt oberhalb oder unterhalb platziert wird `'above'` – oberhalb der Notenzeile `'below'` – unterhalb der Notenzeile	`'auto'`
`'placementHint'`	`str`	gibt einen semantischen Hinweis, um welche Art Objekt es sich handelt. Z.B. können ganz verschiedene Grafiksymbole Dynamikanweisungen beinhalten (u.a. Textobjekte). Umgekehrt können Textobjekte ganz verschiedene Bedeutungen haben. Der Schlüssel hilft der Platzierungsautomatik, das Objekt passend zu semantisch verwandten Grafikobjekten auszurichten. Mögliche Ausprägungen: `''` ohne Charakterisierung `'downBow'` Abstrich (Streicher) `'downStroke'` Abschlag (Gitarre) `'upBow'` Aufstrich (Streicher) `'upStroke'` Aufschlag (Gitarre) `'dynamics'` Dynamikanweisung `'pedal'` Pedalsymbol `'jump'` Sprunganweisung `'breathMark'` Atemzeichen `'musicSymbol'` Musiksymbol	`''`

Schlüssel	Typ	Bedeutung	Standard-wert
	'systemTitle' 'movementTitle' 'chordSymbol' 'tempoMarking' 'rehearsalMark' 'performanceInstruction' 'fingering' 'chordDiagram'	Systemüberschrift Satzbezeichnung Akkordsymbol Tempoanweisung Abschnittsbezeichnung Spielanweisung Fingersatz Griffsymbol	
'placementFlags'	int	Wenn der Schlüssel auf den Wert 1 gesetzt ist, wird das Grafikobjekt isoliert platziert und nicht an den anderen Grafikobjekten mit verwandter Bedeutung ausgerichtet (vgl. 'placementHint')	0
'noAdjust'	bool	Wenn dieser Schlüssel auf True gesetzt ist, wird das Objekt (z.B. Bindebogen) beim Transponieren nicht automatisch an die Notenlage angepasst.	False, das Objekt wird automatisch angepasst
'horizAlign'	int	legt fest, worauf sich die horizontale Lage des Grafikobjekts bei manueller bzw. „halbautomatischer“ Platzierung bezieht: 0 – auf den Notenkopf 1 – auf den Hals	0
'vertAlign'	int	legt fest, worauf sich bei manueller Platzierung die vertikale Lage des Grafikobjekts bezieht: 0 – auf die Notenlinien 1 – auf die äußerste Note 2 – auf die innerste Note 3 – auf das Halsende 4 – oberes Ende des Notenobjekts 5 – unteres Ende des Notenobjekts	0
'minDistY'	float (Zw.2)	Bei manuell platzierten Grafikobjekten lässt sich mit diesem Schlüssel ein Mindestabstand von der mittleren Notenlinie festlegen (zwischen 0 und 8)	0

Wenn ein Objekt an bestimmte feste Koordinaten verschoben werden soll, muss sichergestellt sein, dass das 'placement' des Objekts auf 'manually' gesetzt ist. Falls 'placement' == 'auto' wird *capella* das Objekt sonst eigenmächtig gemäß der Auto-Platzierung platzieren, und die gewünschten Koordinaten gehen verloren).

Wenn ein Objekt, dessen 'placement' von 'manually' auf 'auto' bzw. 'semiAuto' verändert wird, muss ein geeigneter Wert für 'placementHint' gesetzt werden. Dieser hilft *capella*, das Auto-Objekt korrekt zu platzieren.

Wenn neue Objekte eingefügt werden, sollte die Methode 'Score.isUsingAutoPlacement()' genutzt werden, um das 'placement' des neuen Objekts vorzugeben. Der Benutzer wird erwarten, dass neue Objekte als 'auto' eingefügt werden, wenn isUsingAutoPlacement den Wert True zurückliefert, ansonsten als 'manually'.

Wenn ein Objekte mit 'placement' == 'auto' eingefügt wird, müssen keine Koordinaten angegeben werden. Sie werden von *capella* automatisch ermittelt. Bei 'semiAuto' muss nur die x-Koordinate angegeben werden.

Bei Objekten mit 'placement' == 'auto' können weiterhin Koordinaten ausgelesen werden. Allerdings können sich diese später ändern (z.B. wenn es sich um ein Auto-Objekt handelt, das gerade erst eingefügt wurde und das in *capella* noch nicht angezeigt wurde).

Falls das Skript auch mit älteren Versionen von *capella* kompatibel sein soll, ist es ratsam, neue Funktionen wie 'isUsingAutoPlacement()' und die neuen Einträge in den Objekt-Dictionaries nur, nachdem mit 'capVersion() >= (9, 0, 0)' geprüft wurde, dass sie unterstützt werden.

In den folgenden Abschnitten werden die individuellen Parameter (= Einträge im Dictionary) der verschiedenen Grafikobjekte aufgeführt.

Gruppen von Grafikobjekten

'group'

Damit wird eine Gruppe von Grafikelementen zusammengefasst.

Schlüssel	Typ	Bedeutung	Standardwert
'items'	list	Eine Python-Liste von Grafikobjekten	keiner

Beispiel:

```
activeScore().registerUndo("Demo")
for note in activeScore().noteObjs():
    for d in note.drawObjs():
        if d['type'] == 'group':
```

```
            t=dict(type = 'text', x=0, y=6,
                   content='Gruppe:%d Objekte'%len(d['items']),
                   font = dict(height=10))
            note.addDrawObj(t)
            dx = 0
            for item in d['items']:
                shiftDrawObj(item, dx, -2.0)
                dx += 1
                note.addDrawObj(item)
```

Das Skript schreibt unter jedes Gruppenobjekt die Anzahl der Elemente und reiht Kopien der einzelnen Elemente über dem Objekt auf.

Grafikelemente der Notenschrift

`'bracket'` – Klammer für Triolen und andere irreguläre Teilungen

Schlüssel	Typ	Bedeutung	Standard-wert
`'x1'`	`float (Zw.32)`	x-Koordinate des Anfangspunkts	
`'y1'`	`float (Zw.32)`	y-Koordinate des Anfangspunkts	
`'x2'`	`float (Zw.32)`	x-Koordinate des Endpunkts	
`'y2'`	`float (Zw.32)`	y-Koordinate des Endpunkts	
`'lineWidth'`	`float (Zw.10)`	Linienstärke	0.1
`'orientation'`	`str`	`'none'`, `'up'` oder `'down'`	`'none'`
`'number'`	`int`	Erlaubte Werte: 0 (für keine Ziffer) und 2 bis 15	0

Beispiel:

```
activeScore().registerUndo("Duolenklammern rot färben")
for note in activeScore().noteObjs():
    for i in range(note.nDrawObjs()):
        d = note.drawObj(i)
        if d['type'] == 'bracket':
            if d.get('number', 0) == 2:
                d['color'] = Color.red
                note.replaceDrawObj(i, d)
```

Das Skript färbt in einer Partitur alle Duolenklammern rot. Alle anderen irregulären Teilungen bleiben unverändert.

`'guitar'` – frei konfigurierbares Griffsymbol für Zupfinstrumente

Schlüssel	Typ	Bedeutung	Standard-wert
`'x'`	`float (Zw.32)`	relative x-Koordinate	
`'y'`	`float (Zw.32)`	relative y-Koordinate	
`'frets'`	`int`	Anzahl der Bünde (3 bis 8)	4
`'thickNut'`	`bool`	gibt an, ob zur Orientierung der Sattel verstärktdargestellt werden soll	`False`
`'open-Strings'`	`bool`	gibt an, ob die Saiten mit offenen Enden dargestellt werden sollen	`False`
`'vertical'`	`bool`	gibt an, ob das Griffsymbol vertikal oder horizontal angezeigt werden soll	`True`
`'stringDist'`	`float (Zw.4)`	Abstand zwischen den Saiten (mögliche Werte: 0.5, 0.75 oder 1)	0.75
`'fretDist'`	`float (Zw.2)`	Abstand zwischen den Bünden (mögliche Werte: 1, 1.5 oder 2)	2
`'strings'`	`str`	Zeichenkette der Länge 3 bis 8 (entsprechend dem Schlüssel `'frets'`): Für jede Saite ist die Fingerposition als Ziffer angegeben (bzw. "/")?	

Beispiel:

```
activeScore().registerUndo("Gitarrensymbole drehen")
for note in activeScore().noteObjs():
    for i in range(note.nDrawObjs()):
        d = note.drawObj(i)
        if d['type'] == 'guitar':
            d['vertical'] = not d.get('vertical', True)
            note.replaceDrawObj(i, d)
```

Das folgende Skript dreht alle Gitarrensymbole um 90° (Wechsel zwischen Vertikal- und Horizontaldarstellung). Das Attribut `'vertical'` hat den Standardwert `True`. Deshalb darf in der vorletzten Zeile nicht `d['vertical'] = not d['vertical']` stehen, da bei vertikalen Gitarrensymbolen das Attribut fehlt. Hier hilft die Dictionary-Methode `get()`, der man (im zweiten Parameter) einen Vorgabewert übergeben kann für den Fall, dass der Schlüssel fehlt.

`'notelines'` – Notenlinien

Schlüssel	Typ	Bedeutung	Standard-wert

'x1'	float (Zw.32)	relative x-Koordinate des Anfangs	
'x2'	float (Zw.32)	relative x-Koordinate des Endes	
'y'	float (Zw.32)	relative y-Koordinate der mittleren Linie	
'lines'	str	Beschreibung der Notenlinien: '5' – reguläres Fünf-Linien-System '1' – Ein-Linien-System für Schlagzeug ODER ein String mit 11 Zeichen (je möglicher Notenlinie eines), bestehend aus: | für eine durchgezogene Notenlinie : für eine gepunktete Notenlinie _ für 'keine Notenlinie'	'5'

'octaveClef' – Oktavierungsklammer

Schlüssel	Typ	Bedeutung	Standardwert
'x1'	float (Zw.32)	relative x-Koordinate des Anfangs	
'x2'	float (Zw.32)	relative x-Koordinate des Endes	
'y'	float (Zw.32)	relative y-Koordinate waagerechten Linke	
octave	int	Zahl der Oktaven, um die oktaviert werden soll (negative Zahlen stehen für Oktavierungen nach unten)	1
onlyNumber	bool	Wenn octave = 1 oder -1 ist, bewirkt der Wert True die alleinige Anzeige der Oktavierungszahl „8“ statt „8va“	False
courtesyRepeat		Wenn die Oktavierungsklammer ein Fortsetzungsobjekt ist (das continued-Attribut ist True), wird als Default die Oktavierungszahl eingeklammert am Anfang der Fortsetzung wiederholt. Wenn courtesyRepeat aber False ist, wird nur die gestrichelte Linie	True

		gezeichnet.	

'transposable' – transponierbares Objekt, z.B. Akkordsymbol

Zum Verständnis der Datenstruktur ist es sehr hilfreich, zunächst im *capella*-Handbuch den Abschnitt zu transponierbaren Symbolen durchzuarbeiten und probeweise ein transponierbares Symbol selbst zu entwerfen.

Anschließend kann man es in einer Partitur einsetzen und die konkrete Struktur anhand des Strukturbaum-Anzeige nachvollziehen, siehe S. 10

Schlüssel	Typ	Bedeutung	Standard-wert
'nRefNote'	int	Index der Bezugsnote im Quintenzirkel, Das gleiche Format wird von curKey() zurückgeliefert, siehe **S. 70** Der Wert von nRefNote entscheidet darüber, welches Objekt aus der Liste 'items' angezeigt wird. Er ändert sich beim Transponieren.	
'items'	list	Eine Python-Liste mit 12 oder 21 Grafikobjekten, die je nach dem Wert von nRefNote angezeigt werden. Details hierzu finden Sie im Benutzerhandbuch bei den transponierbaren Grafikobjekten.	

Beispiel:

```
activeScore().registerUndo("transponierbares Symbol auflösen")
for note in activeScore().noteObjs():
    for d in note.drawObjs():
        if d['type'] == 'transposable':
            dx = 0
            for item in d['items']:
                shiftDrawObj(item, dx, -2.0)
                dx += 2
                note.addDrawObj(item)
            break
```

'slur' – Binde-/Legatobogen

Bindebögen werden in *capella* durch kubische Bézierkurven beschrieben. Der erste und vierte Kontrollpunkt sind die Endpunkte des Bindebogens. Mit den beiden anderen Kontrollpunkten wird die Form der Kurve beeinflusst.

Bindebögen sind in der Regel an zwei Notenobjekten verankert (dadurch ist der Parameter `'noteRange'` > 0). Die Koordinaten sind relativ zu den Notenobjekten zu verstehen, an denen der Bindebogen verankert ist, wobei der 1. und 2. Kontrollpunkt relativ zum ersten, der 3. und 4. Kontollpunkt relativ zum zweiten Notenobjekt zu interpretieren sind.

Schlüssel	Typ	Bedeutung	Standardwert
`'x1'`	`float (Zw.32)`	x-Koordinate des 1. Kontrollpunkts	
`'y1'`	`float (Zw.32)`	y-Koordinate des 1. Kontrollpunkts	
`'x2'`	`float (Zw.32)`	x-Koordinate des 2. Kontrollpunkts	
`'y2'`	`float (Zw.32)`	y-Koordinate des 2. Kontrollpunkts	
`'x3'`	`float (Zw.32)`	x-Koordinate des 3. Kontrollpunkts	
`'y3'`	`float (Zw.32)`	y-Koordinate des 3. Kontrollpunkts	
`'x4'`	`float (Zw.32)`	x-Koordinate des 4. Kontrollpunkts	
`'y4'`	`float (Zw.32)`	y-Koordinate des 4. Kontrollpunkts	
`'endWidth'`	`float (Zw.32)`	Stärke an den Enden	`0`
`'midWidth'`	`float (Zw.32)`	Stärke in der Mitte	`0.1875`
`'dotDist'`	`float (Zw.32)`	Bei gestrichelten Bindebögen: Abstand von Strichanfang zu Strichanfang	`0` = durchgezogen
`'dotWidth'`	`float`	Relative Strichlänge (z.B. 0,5 = 50% von dotDist – die Striche und die Lücken dazwischen sind gleich lang).	`1`

`'trill'` – Trillerschlange

Trillerschlangen sind in der Regel an zwei Notenobjekten verankert (dadurch ist der Parameter `'noteRange'` > 0). Die Koordinaten sind relativ zu den Notenobjekten zu verstehen, an denen der Bindebogen verankert ist, wobei x1 zum ersten und x2 zum zweiten Notenobjekt zu interpretieren sind.

Schlüssel	Typ	Bedeutung	Standardwert
`'x1'`	`float (Zw.32)`	x-Koordinate des Anfangspunkts	
`'x2'`	`float (Zw.32)`	x-Koordinate des Endpunkts	
`'y'`	`float (Zw.32)`	y-Koordinate	
`'tr'`	`bool`	gibt an, ob das tr-Symbol angezeigt wird	`True`

'volta' – Voltenklammer

Voltenklammern sind in der Regel an zwei Notenobjekten verankert (dadurch ist der Parameter `'noteRange'` > 0). Die Koordinaten sind relativ zu den Notenobjekten zu verstehen, an denen die Voltenklammer verankert ist, wobei x1 zum ersten und x2 zum zweiten Notenobjekt zu interpretieren sind.

Schlüssel	Typ	Bedeutung	Standard-wert
'x1'	float (Zw.32)	x-Koordinate des Anfangspunkts	1
'x2'	float (Zw.32)	x-Koordinate des Endpunkts	
'y'	float (Zw.32)	y-Koordinate	
'leftBent'	bool	links geschlossen	True
'rightBent'	bool	rechts geschlossen	True
'firstNumber'	int	Anfangsziffen (0 = keine Ziffer)	0
'lastNumber'	int	Endziffer (0 = keine Ziffer)	0
'allNumbers'	bool	alle Ziffern einzeln ausgeschrieben	False

'wavyLine' – Schlangenlinie (Glissando)

Wenn die Schlangenlinie an mehreren Notenobjekten verankert ist (d.h. `'noteRange'` > 0), gelten die Koordianten x1 und y1 relativ zum ersten Notenobjekt und x2 und y2 relativ zum zweiten Notenobjekt.

Schlüssel	Typ	Bedeutung	Standard-wert
'x1'	float (Zw.32)	x-Koordinate des Anfangspunkts	
'y1'	float (Zw.32)	y-Koordinate des Anfangspunkts	
'x2'	float (Zw.32)	x-Koordinate des Endpunkts	
'y2'	float (Zw.32)	y-Koordinate des Endpunkts	
'waveLen'	float (Zw.32)	Wellenlänge	1
'fullWaves'	bool	bestimmt, ob die Länge der Wellenlinie auf Vielfaches ganzer Wellen gerundet wird	True

'wedge' – (De)Crescendo-Keil

Wenn der (De)Crescendo-Keil an mehreren Notenobjekten verankert ist, (d.h. `'noteRange'` > 0), gelten die Koordianten x1 und y1 relativ zum ersten Notenobjekt und x2 und y2 relativ zum zweiten Notenobjekt.

Schlüssel	Typ	Bedeutung	Standard-wert
`'x1'`	`float (Zw.32)`	x-Koordinate des Anfangspunkts	
`'y1'`	`float (Zw.32)`	y-Koordinate des Anfangspunkts	
`'x2'`	`float (Zw.32)`	x-Koordinate des Endpunkts	
`'y2'`	`float (Zw.32)`	y-Koordinate des Endpunkts	
`'lineWidth'`	`float (Zw.10)`	Linienstärke	
`'span'`	`float (Zw.32)`	Weite der Gabelöffnung	
`'decrescendo'`	`bool`	`False` – Keilspitze links (crescendo) `True` – Spitze rechts (decrescendo)	`False`

elementare geometrische Formen

`'line'` – gerade Linie

Wenn die Linie an mehreren Notenobjekten verankert ist, (d.h. `'noteRange'` > 0), gelten die Koordianten x1 und y1 relativ zum ersten Notenobjekt und x2 und y2 relativ zum zweiten Notenobjekt.

Schlüssel	Typ	Bedeutung	Stan-dardwert
`'x1'`	`float (Zw.32)`	x-Koordinate des Anfangspunkts	
`'y1'`	`float (Zw.32)`	y-Koordinate des Anfangspunkts	
`'x2'`	`float (Zw.32)`	x-Koordinate des Endpunkts	
`'y2'`	`float (Zw.32)`	y-Koordinate des Endpunkts	
`'lineWidth'`	`float (Zw.10)`	Linienstärke	`0.1`

Beispiel:

Das folgende Skript macht alle Linien, deren Steigung um weniger als 10% von der Waagerechten abweichen, exakt waagerecht.

```
activeScore().registerUndo("fast waagerechte Linien einrasten")
for voice in activeScore().voices():
    for i in range(voice.nNoteObjs()):
        note = voice.noteObj(i)
        for k in range(note.nDrawObjs()):
            d = note.drawObj(k)
```

```
        if d['type'] == 'line':
            x1,y1,x2,y2 = d['x1'],d['y1'],d['x2'],d['y2']
            r = d.get('noteRange', 0)
            if r > 0 and i+r < voice.noteObjs():
                x1 += note.posX()
                x2 += voice.noteObj(i+r).posX()
            dx = abs(x2 - x1)
            dy = abs(y2 - y1)
            if dx > dy and dy/dx < 0.1:
                d['y2'] = y1
                note.replaceDrawObj(k, d)
```

Bei Linien, deren Endpunkte an verschiedenen Notenobjekten verankert sind, beziehen sich die beiden (relativen) x-Koordinaten auf zwei verschiedene Noten. Deshalb werden in diesem Fall zunächst die absoluten Notenpositionen mit der Methode `posx` ermittelt, bevor die Steigung der Linie berechnet wird.

`'ellipse'`, `'polygon'`, `'rectangle'`

Diese drei elementaren geometrischen Formen haben alle einen Rand und ein Inneres. Deshalb besitzen sie alle die folgenden Schlüssel:

Schlüssel	Typ	Bedeutung	Standardwert
`'lineWidth'`	`float (Zw.10)`	Linienstärke	`0.1`
`'filled'`	`bool`	gibt an, ob das Objekt gefüllt dargestellt werden soll	`False` = ungefüllt
`'fillColor'`	`int`	Füllfarbe, kodiert wie die Farbe aller Grafikobjekte, siehe S. 76	`0` = schwarz
`'lineColor'`	`int`	Linienfarbe	`0` = schwarz

Im Folgenden werden lediglich die weiteren, individuellen Schlüssel beschrieben.

`'ellipse'` – Ellipse

Wenn die Ellipse an mehreren Notenobjekten verankert ist, (d.h. `'noteRange'` > 0), gelten die Koordianten x1 und y1 relativ zum ersten Notenobjekt und x2 und y2 relativ zum zweiten Notenobjekt.

Schlüssel	Typ	Bedeutung	Standardwert
`'x1'`	`float (Zw.32)`	x-Koordinate der linken Begrenzung	
`'y1'`	`float (Zw.32)`	y-Koordinate der oberen Begrenzung	

'x2'	float (Zw.32)	x-Koordinate der rechten Begrenzung	
'y2'	float (Zw.32)	y-Koordinate der unteren Begrenzung	

'polygon' – Polygon

Ein Polygon (Vieleck) ist an einem einzigen Notenobjekt verankert, besteht selbst aber aus beliebig vielen Punkten. Die Koordinaten (relativ zum Notenobjekt) werden in zwei Python-Listen gespeichert.

Schlüssel	Typ	Bedeutung	Standard-wert
'x'	list, Elemente vom Typ float (Zw.32)	x-Koordinaten der Polygonpunkte	
'y'	list, Elemente vom Typ float (Zw.32)	y-Koordinaten der Polygonpunkte	

Beispiel: Das folgende Skript zeichnet über der Note rechts vom Cursor einen gelben Pfeil nach unten.

```
note = cursorObj()
if note and note.isChord():
    activeScore().registerUndo("gelber Pfeil")
    note = cursorObj()
    note.addDrawObj({'type':'polygon',
                     'filled':True,
                     'fillColor':Color.yellow,
                     'x':[-1,-1,-2, 0, 2, 1, 1],
                     'y':[-7,-5,-5,-3,-5,-5,-7]})
```

Das Beispiel zeigt die typische Problematik des Rückgängigmachens (siehe S. 31): Stünde der Aufruf von registerUndo gleich in der ersten Zeile, so würde (falls der Cursor nicht vor einer Note steht) eine wirkungslose Aktion in die Kette zum Rückgängigmachen aufgenommen (was den Anwender verwirren könnte). Deshalb sollte man registerUndo erst dann aufrufen, wenn wirklich klar ist, dass die Partitur verändert wird. registerUndo ersetzt aber die Partitur durch eine Kopie. Deshalb müssen danach alle partiturbezogenen Objekte neu ermittelt werden, weil die internen Verweise in den alten Objekten (hier cursorObj()) ungültig geworden sind.

'rectangle' – Rechteck

Wenn das Rechteck an mehreren Notenobjekten verankert ist, (d.h. 'noteRange' > 0), gelten die Koordianten x1 und y1 relativ zum ersten Notenobjekt und x2 und y2 relativ zum zweiten Notenobjekt.

Schlüssel	Typ	Bedeutung	Standard-

			wert
'x1'	float (Zw.32)	x-Koordinate der linken unteren Ecke	
'y1'	float (Zw.32)	y-Koordinate der linken unteren Ecke	
'x2'	float (Zw.32)	x-Koordinate der rechten oberen Ecke	
'y2'	float (Zw.32)	y-Koordinate der rechten oberen Ecke	
'radius'	float (Zw.32)	Krümmungsradius der abgerundeten Ecken	0

importierte Grafiken

Bei importierten Grafiken wird die Grafik als separate Datei im ZIP-Archiv der capx-Datei abgespeichert. Diese enthält dann neben `score.xml` weitere Dateien. Siehe S. 9

Drei importierte Grafik-Typen sind in *capella* berücksichtigt:

'pixmap' – Rastergrafik

Unterstützt werden aktuell die Grafikformate PNG, JPG, JPEG, GIF, BMP, TIF und TIFF.

'svg' – Vektorgrafik

Unterstützt werden sowohl SVG als auch SVGZ.

'metafile' – Grafik im Windows-Metafile-Format (WMF oder EMF)

Dieser Typ von Grafikobjekten wird aus Kompatibilitätsgründen von *capella* weiterhin unterstützt. Windows-Metafile war ein gängiges Austauschformat, das z.B. von der Windows-Zwischenablage genutzt wurde.

Für die Neuerstellung von Grafikobjekten auf Basis importierter Grafiken ist es ratsam, das Format SVG für Vektorgrafiken bzw. ein `pixmap`-Objekt für Rastergrafiken zu nutzen.

Alle Grafikobjekte für importierte Grafiken haben die gleichen Parameter als Schlüssel:

Schlüssel	Typ	Bedeutung	Standard-wert
'x'	float (Zw.32)	x-Koordinate (relativ zum Notenobjekt)	
'y'	float (Zw.32)	y-Koordinate (relativ zum Notenobjekt)	
'width'	float (Zw.32)	Breite	
'height'	float (Zw.32)	Höhe	
'file'	str	Dateiname der Grafikdatei – ist nur dann relevant, wenn ein solches Grafikobjekt neu in die Partitur eingefügt wird. Dann	

		wird mit diesem Parameter die zu importierende Grafikdatei übergeben.	
`'data'`	`str`	Binärdaten des jeweiligen Bildformats	

Achtung: Falls als `'file'` ein relativer Dateipfad übergeben wird, ist es vom Dateityp abhängig, wie der Pfad interpretiert wird: Bei `metafile`-Dateien wird er vom Speicherort des Skriptes aus gesehen. Für `pixmap` und `svg` wird er vom Pfad des ausführbaren *capella*-Programms aus gesehen. Zum Experimentieren: Bild einfügen.py
Hinweis: Der Name der Grafikdatei im Zip-Archiv, der auch im CapXML-Strukturbaum angezeigt wird, wird automatisch vergeben und ist völlig unabhängig von dem Dateinamen, der als file eingegeben werden kann!

Beispiel zum Experimentieren: Das folgende Skript zeigt zunächst den Pfad der *capella*-Programmdatei an. Das Skript erwartet, dass unter diesem Pfad eine Datei namens „frosch.png" liegt.

```
import os
messageBox("Bild einfügen","cwd: " + os.getcwd())
note = cursorObj()
if note and note.isChord():
    activeScore().registerUndo("Bild einfügen")
    note.addDrawObj({'type':'pixmap',
                     'x':0.0,
                     'y':-5.0,
                     'width':20.0,
                     'height':20.0,
                     'file':'frosch.png'
                     })
```

Textobjekte

`'text'` – Einfachtext

Mit Einfachtexten lassen sich neben Text auch Spielanweisungen etc. kodieren. In diesem Fall ist der Schlüssel `placementHint` auf einen passenden Wert zu setzen. Siehe S. 77

Sofern bei Einfachtexten ein Rahmen um den Text gezeichnet werden soll (Schlüssel frame), werden die Größe und die Position des Rahmens von *capella* automatisch angepasst. Die Schlüssel `'frameDistance'` und `'radius'` dienen der Steuerung dieser Automatik.

Schlüssel	Typ	Bedeutung	Standardwert
`'x'`	`float (Zw.32)`	x-Koordinate (relativ zum No-	

		tenobjekt)	
'y'	float (Zw.32)	y-Koordinate (relativ zum Notenobjekt)	
'align'	str	gültige Werte sind: 'left' – linksbündig 'center' – zentriert 'right' – rechtsbündig	'left'
'content'	str	der Textinhalt	
'frame'	str	legt Form des Rahmens fest: 'none' – kein Rahmen 'rectangle' – rechteckig 'ellipse' – oval 'circle' – kreisrund	'none'
'frameDistance'	float (Zw.32)	Abstand zwischen Text und Rahmen	0
'radius'	float (Zw.32)	nur beim Rahmen-Typ 'rectangle': Radius der abgerundeten Ecken	0 = keine Abrundung
'font'	dict	die Parameter der Schriftart	

Der Wert zum Schlüssel 'font' ist selbst ein Python-Dictionary mit folgendem Aufbau:

Schlüssel	**Typ**	**Bedeutung**	**Standardwert**
'face'	str	Schriftart	
'height'	int	Schriftgröße	
'width'	int	Schriftbreite (0 bedeutet den Standardwert des Fonts zu verwenden)	0
'weight'	int	Schriftstärke (Bereich 0 bis 1000) 400 = normal 700 = fett	400
'italic'	bool	kursiv	False
'underline'	bool	unterstrichen	False
'strikeOut'	bool	durchgestrichen	False
'charSet'	int	Bestimmt den Zeichensatz. Mögliche Werte siehe CapXML-Dokumentation, eine kleine Auswahl:	0

		`0` = ANSI_CHARSET 1 = DEFAULT_CHARSET 2 = SYMBOL_CHARSET 77 = MAC_CHARSET	
`'pitchAndFamily'`	`int`	In diesem Integer-Attribut sind einige weitere Eigenschaften des Fonts codiert. Mehr Details sind in der Beschreibung der CapXML-Strukturbaum enthalten. Siehe S. 10	

`'richText'` – RichText-Textfeld

Ein RichText-Textfeld hat wesentlich mehr Möglichkeiten als ein Einfachtext. Zum Beispiel kann man verschiedene Textformatierungen innerhalb des Textes mischen.

Es gibt zwei Varianten eines RichText-Textfeldes, die über den Schlüssel 'textFormat' unterschieden werden:

- `'html'` – Textfeld im HTML-Format: In diesem Fall ist der formatierte Text im HTML-Format im Schlüssel `'dataHtml'` enthalten und kann in einem Skript ausgewertet und verändert werden werden. Legt man in *capella* in einer aktuellen Version ein RichText-Textfeld an, wird dieses Format verwendet.
- `'rtf'` – Textfeld im RTF-Format: Dieses Format wird von *capella* aus Kompatibilitätsgründen weiter unterstützt.

Schlüssel	Typ	Bedeutung	Standard-wert
`'x'`	`float (Zw.32)`	relative x-Koordinate	
`'y'`	`float (Zw.32)`	relative y-Koordinate	
`'width'`	`float (Zw.32)`	Breite des Textfeldes	
`'textFormat'`	`str`	`'rtf'` – Textfeld im RTF-Format `'html'` – Textfeld im HTML-Format	
`'file'`	`str`	Falls eine neue Textdatei eingefügt werden soll, wird in diesem Parameter der Dateipfad einer einzufügenden Text-Datei übergeben.	
`'dataHTML'`	`str`	Wenn `'textFormat' == 'html'`, dann enthält dieser Schlüssel den HTML-Code des Textfeldes als String.	
`'data'`	`str`	ggf. weitere formatspezifische Daten	

Technische Ergänzungen

In diesem Kapitel werden Aspekte beleuchtet, die zwar nichts mit Programmierung zu tun haben, die aber ein tieferes technisches Verständnis und Interesse erfordern:

- Übernehmen von Partituren im capella-2-Format
- Partitur-Assistent-Vorlagen anpassen
- Partitur-Stile und eigene Schriftarten entwerfen
- transponierbare Symbole selbst entwerfen

Genaueres zu diesen Themen wird in einer späteren Version dieses Entwicklerhand-buchs ergänzt.

Stichwortverzeichnis

A

activeScore....40
allFiles....39

C

caplib....21
capDOM....22
childElements....25
firstChildElement....25
ScoreChange....23
ScoreExport....24
capSAX....25
CapSaxHandler....26
SaxScoreChange....26
capVersion....40
CapXNode....56
checkCapVersion....40
Clipboard....49
closeActiveScore....40
Color....46
curSelection....40
cursorObj....41
Cursorposition....15

D

Datei-Dialog....43
Dialogbaukasten....27, 51
CheckBox....55
ComboBox....54
Dialog....51
Edit....52
HBox....55
Label....52
Radio....53
VBox....56
displayPage....41
displaySystem....41

E

externe Skripte....6, 19

G

getPersonalDataDir....39
getProgramDir....39
getUserDataDir....39
Grafikobjekte....72
Erscheinungsbild....76
geometrische Formen....86
Ellipse....87
Linie....86
Polygon....88
Rechteck....88
Grafikkennungen....37
Gruppen....79
Gültigkeit....73
importierte Grafiken....89
Methoden von NoteObj....71
Notensymbole....80
Bindebogen....83
Crescendo/Decrescendo....85
Gitarrengriff....80
Glissando....85
Notenlinien....81
Oktavierungsklammer....82
transponierbares Symbol....83
Trillerschlange....84
Triolenklammer....80
Voltenklammer....85
Platzierung....76
Sichtbarkeit....75
Textobjekte....90
Einfachtext....90
RichText....92
Typ....73

I

Icon....32

interne Skripte 6, 12
intervals 41

M

Markierung 15
mehrsprachige Plugins 32
messageBox 43
MidiOut 50

O

openScore 40
Optionen speichern 36

P

Parameter speichern 36
Partitur ....
    Klassenreferenz 58
        Head 72
        NoteObj 67
        Score 58
        Staff 64
        System 61
        Voice 67
    XML-Struktur 9
    Zugriff 12

R

Rational 45
RelDiatonicNote 47
rootNode 42
Rückgängig-Funktion 31

S

ScriptOptions 48
selectedNode 42
setLanguages 35
setSelection 42
shiftDrawObj 42
Strukturbaum-Ansicht 9

T

tr 35