defcon Documentation

Release 0.1

Tal Leming

November 12, 2013

Contents

defcon is a set of UFO based objects optimized for use in font editing applications. The objects are built to be lightweight, fast and flexible. The objects are very bare-bones and they are not meant to be end-all, be-all objects. Rather, they are meant to provide *base functionality* so that you can focus on your application's behavior, not *object observing* or *maintaining cached data*.

CHAPTER 1

Basic Usage

defcon is very easy to use:

```
from defcon import Font
font = Font()
# now do some stuff!
```

CHAPTER 2

Concepts

2.1 Notifications

defcon uses something similar to the Observer Pattern for inter-object communication and object observation. This abstraction allows you to cleanly listen for particular events happening in particular objects. You don't need to wire up lots of hooks into the various objects or establish complex circular relationships thoughout your interface code. Rather, you register to be notified when something happens in an object. In defcon, these are referred to as *notifications*. For example, I want to be notified when the my font changes:

```
class MyInterface(object):

    # random code here, blah, blah.

    def setGlyph(self, glyph):
        glyph.addObserver(self, "glyphChangedCallback", "Glyph.Changed")

    def glyphChangedCallback(self, notification):
        glyph = notification.object
        print "the glyph (%s) changed!" % glyph.name
```

When the glyph is changed in anyway by anyone, it posts a "Glyph.Changed" notification to all registered observers. My method above is called when this happens and I can react as needed.

The *NotificationCenter* object implements all of this. However, all objects derived from `dercon.BaseObject` have a simplified API for tapping into notifications. Each object posts its own unique notifications, so look at the relevant reference for information about the available notifications.

2.1.1 Don't Forget removeObserver

The only real gotcha in this is that you must remove the observer from the observed object when the observation is no longer needed. If you don't do this and the observed object is changed, it will try to post a notification to the object you have discarded. That could lead to trouble.

2.2 Subclassing

The defcon objects are built to have basic functionality. Your application can, and should, have its own functionality that is not part of the standard defcon repertoire. The objects are built with this in mind – they are built to be subclassed and extended. This is done easily:

```
from defcon import Glyph

class MyCustomGlyph(Glyph):

  def myCustomMethod(self):
    # do something to the glyph data
```

When it is time to load a font, you pass this custom class to the Font object:

```
from defcon import Font

font = Font(glyphClass=MyCustomGlyph)
```

When a glyph is loaded, the glyph class you provided will be used to create the glyph object.

2.3 External Changes

It may be advantagious for your application to notice changes to a UFO that were made outside of your application. the `Font` object can help you with this. This object has a `testForExternalChanges()` method. This method will compare the data that has been loaded into the font, glyphs, etc. with the data in the UFO on disk. It will report anything that is different from when the UFO was last loaded/saved.

To do this in a relatively effecient way, it stores the modification data and raw text of the UFO file inside the object. When the `testForExternalChanges()` method is called, the modification date of the UFO file and the stored modification date are compared. A mismatch between these two will trigger a comparison between the raw text in the UFO file and the stored raw text. This helps cut down on a significant number of false positives.

The `testForExternalChanges()` method will return a dictionary describing what could have changed. You can then reload the data as appropriate. The `Font` object has a number of *reload* methods specifically for doing this.

2.3.1 Scanning Scheduling

defcon does not automatically search for changes, it is up to the application to determine when the scanning should be performed. The scanning can be an expensive operation, so it is best done at key moments when the user *could* have done something outside of your application. A good way to do this is to catch the event in which your application/document has been selected after being inactive.

2.3.2 Caveats

There are a couple of caveats that you should keep in mind:

1. If the object has been modified and an external change has happened, the *object* is considered to be the most current data. External changes will be ignored. *This may change in the future. I'm still thinking this through.*
2. The font and glyph data is loaded only as needed by defcon. This means that the user could have opened a font in your application, looked at some things but not the "X" glyph, switched out of your application, edited the GLIF file for the "X" glyph and switched back into your application. At this point defcon will not notice that the "X" has changed because it has not yet been loaded. This probably doesn't matter as when the "X" is finally

loaded the new data will be used. If your application needs to know the exact state of all objects when the font is first created, preload all font and glyph data.

2.4 Representations

One of the painful parts of developing an app that modifies glyphs is managing the visual representation of the glyphs. When the glyph changes, all representations of it in cached data, the user interface, etc. need to change. There are several ways to handle this, but they are all cumbersome. defcon gives you a very simple way of dealing with this: *representations* and *representation factories*.

2.4.1 Representations and Representation Factories

A *representation* is an object that represents a glyph. As mentioned above, it can be a visual representation of a glyph, such as a NSBezierPath. Representations aren't just limited to visuals, they can be any type of data that describes a glyph or something about a glyph, for example a string of GLIF text, a tree of point location tuples or anything else you can imagine. A *representation factory* is a function that creates a representation. You don't manage the representations yourself. Rather, you register the factory and then ask the glyphs for the representations you need. When the glyphs change, the related representations are destroyed and recreated as needed.

2.4.2 Example

As an example, here is a representation factory that creates a NSBezierPath representation:

```
def NSBezierPathFactory(glyph, font):
    from fontTools.pens.cocoaPen import CocoaPen
    pen = CocoaPen(font)
    glyph.draw(pen)
    return pen.path
```

To register this factory, you do this:

```
from defcon import addRepresentationFactory
addRepresentationFactory("NSBezierPath", NSBezierPathFactory)
```

Now, when you need a representation, you simply do this:

```
path = glyph.getRepresentation("NSBezierFactory")
```

Not only do you only have to register this *once* to be able get the representation for *all* glyphs, the representation is always up to date. So, if you change the outline in the glyph, all you have to do to get the updated representation is:

```
path = glyph.getRepresentation("NSBezierFactory")
```

2.4.3 Implementation Details

Representation Factories

Representation factories should be functions that accept at least two arguments. The first argument is always a glyph and the second argument is always a font. After that, you are free to define any keyword arguments you need. You must register the factory with the `addRepresentationFactory` function. When doing this, you must define a unique name for your representation. The recommendation is that you follow the format of "applicationOrPackage-Name.representationName" to prevent conflicts. Some examples:

```
addRepresentationFactory("MetricsMachine.groupEditorGlyphCellImage", groupEditorGlyphCellImageFactory
addRepresentationFactory("Prepolator.previewGlyph", previewGlyphFactory)
```

Representations

Once the factory has been registered, glyphs will be able to serve the images. You can get the representation like this:

```
image = glyph.getRepresentation("MetricsMachine.groupEditorGlyphCellImage")
```

You can also pass keyword arguments when you request the representation. For example:

```
image = glyph.getRepresentation("MetricsMachine.groupEditorGlyphCellImage", cellSize=(40, 40))
```

These keyword arguments will be passed along to the representation factory. This makes it possible to have very dynamic factories.

All of this is highly optimized. The representation will be created the first time you request it and then it will be cached within the glyph. The next time you request it, the cached representation will be returned. If the glyph is changed, the representation will automatically be destroyed. When this happens, the representation will not be recreated automatically. It will be recreated the next time you ask for it.

CHAPTER 3

Objects

3.1 Font

See Also:

Notifications: The Font object uses notifications to notify observers of changes.

External Changes: The Font object can observe the files within the UFO for external modifications.

3.1.1 Tasks

File Operations

- `Font`
- `save()`
- `path`
- `ufoFormatVersion`
- `testForExternalChanges()`
- `reloadInfo()`
- `reloadKerning()`
- `reloadGroups()`
- `reloadFeatures()`
- `reloadLib()`

Sub-Objects

- `info`
- `kerning`
- `groups`
- `features`

- `lib`
- `unicodeData`

Glyphs

- `Font`
- `newGlyph()`
- `insertGlyph()`
- `keys()`

Reference Data

- `glyphsWithOutlines`
- `componentReferences`
- `bounds`
- `controlPointBounds`

Changed State

- `dirty`

Notifications

- `dispatcher`
- `addObserver()`
- `removeObserver()`
- `hasObserver()`

Font

3.2 Glyph

See Also:

Notifications**:** The Glyph object uses notifications to notify observers of changes.

Representations**:** The Glyph object can maintain representations of various arbitrary types.

3.2.1 Tasks

Name and Unicodes

- `name`
- `unicodes`

- `unicode`

Metrics

- `leftMargin`
- `rightMargin`
- `width`

Reference Data

- `bounds`
- `controlPointBounds`

General Editing

- `clear()`
- `move()`

Contours

- `Glyph`
- `clearContours()`
- `appendContour()`
- `insertContour()`
- `contourIndex()`
- `autoContourDirection()`

Components

- `components`
- `clearComponents()`
- `appendComponent()`
- `componentIndex()`
- `insertComponent()`

Anchors

- `anchors`
- `clearAnchors()`
- `appendAnchor()`
- `anchorIndex()`

- `insertAnchor()`

Hit Testing

- `pointInside()`

Pens and Drawing

- `getPen()`
- `getPointPen()`
- `draw()`
- `drawPoints()`

Representations

- `getRepresentation()`
- `hasCachedRepresentation()`
- `representationKeys()`
- `destroyRepresentation()`
- `destroyAllRepresentations()`

Changed State

- `dirty`

Notifications

- `dispatcher`
- `addObserver()`
- `removeObserver()`
- `hasObserver()`

Parent

- `getParent()`
- `setParent()`

Glyph

3.3 Contour

See Also:

Notifications: The Contour object uses notifications to notify observers of changes.

3.3.1 Tasks

Reference Data

- `bounds`
- `controlPointBounds`
- `open`

Direction

- `clockwise`
- `reverse()`

Points

- `Contour`
- `index()`
- `onCurvePoints`
- `setStartPoint()`

Segments

- `segments`
- `removeSegment()`
- `positionForProspectivePointInsertionAtSegmentAndT()`
- `splitAndInsertPointAtSegmentAndT()`

Hit Testing

- `pointInside()`

Drawing

- `draw()`
- `drawPoints()`

Changed State

- `dirty`

Notifications

- `dispatcher`
- `addObserver()`
- `removeObserver()`
- `hasObserver()`

Parent

- `getParent()`
- `setParent()`

Contour

3.4 Component

See Also:

Notifications: The Component object uses notifications to notify observers of changes.

3.4.1 Tasks

Reference Data

- `bounds`
- `bounds`

Properties

- `baseGlyph`
- `transformation`

Hit Testing

- `pointInside()`

Drawing

- `draw()`
- `drawPoints()`

Changed State

- `dirty`

Notifications

- `dispatcher`
- `addObserver()`
- `removeObserver()`
- `hasObserver()`

Parent

- `getParent()`
- `setParent()`

Component

3.5 Point

Note: This object is not a subclass of `BaseObject` and therefore it does not produce notifications or have any parent attributes. This may change in the future.

3.5.1 Tasks

Position

- `x`
- `y`

Type

- `segmentType`
- `smooth`

Move

- `move`

Point

3.6 Anchor

See Also:

Notifications: The Anchor object uses notifications to notify observers of changes.

3.6.1 Tasks

Position

- `x`
- `y`

Name

- `name`

Move

- `move`

Notifications

- `dispatcher`
- `addObserver()`
- `removeObserver()`
- `hasObserver()`

Parent

- `getParent()`
- `setParent()`

Anchor

3.7 Info

See Also:

Notifications: The Info object uses notifications to notify observers of changes.

3.7.1 Tasks

Generic Identification

- familyName
- styleName
- styleMapFamilyName
- styleMapStyleName
- versionMajor
- versionMinor
- year

Generic Legal

- copyright
- trademark

Generic Dimensions

- unitsPerEm
- descender
- xHeight
- capHeight
- ascender
- italicAngle

Generic Miscellaneous

- note

OpenType head Table

- openTypeHeadCreated
- openTypeHeadLowestRecPPEM
- openTypeHeadFlags

OpenType hhea Table

- openTypeHheaAscender
- openTypeHheaDescender
- openTypeHheaLineGap
- openTypeHheaCaretSlopeRise

- `openTypeHheaCaretSlopeRun`
- `openTypeHheaCaretOffset`

OpenType name Table

- `openTypeNameDesigner`
- `openTypeNameDesignerURL`
- `openTypeNameManufacturer`
- `openTypeNameManufacturerURL`
- `openTypeNameLicense`
- `openTypeNameLicenseURL`
- `openTypeNameVersion`
- `openTypeNameUniqueID`
- `openTypeNameDescription`
- `openTypeNamePreferredFamilyName`
- `openTypeNamePreferredSubfamilyName`
- `openTypeNameCompatibleFullName`
- `openTypeNameSampleText`
- `openTypeNameWWSFamilyName`
- `openTypeNameWWSSubfamilyName`

OpenType OS/2 Table

- `openTypeOS2WidthClass`
- `openTypeOS2WeightClass`
- `openTypeOS2Selection`
- `openTypeOS2VendorID`
- `openTypeOS2Panose`
- `openTypeOS2FamilyClass`
- `openTypeOS2UnicodeRanges`
- `openTypeOS2CodePageRanges`
- `openTypeOS2TypoAscender`
- `openTypeOS2TypoDescender`
- `openTypeOS2TypoLineGap`
- `openTypeOS2WinAscent`
- `openTypeOS2WinDescent`
- `openTypeOS2Type`
- `openTypeOS2SubscriptXSize`

- `openTypeOS2SubscriptYSize`
- `openTypeOS2SubscriptXOffset`
- `openTypeOS2SubscriptYOffset`
- `openTypeOS2SuperscriptXSize`
- `openTypeOS2SuperscriptYSize`
- `openTypeOS2SuperscriptXOffset`
- `openTypeOS2SuperscriptYOffset`
- `openTypeOS2StrikeoutSize`
- `openTypeOS2StrikeoutPosition`
- `openTypeVheaVertTypoAscender`
- `openTypeVheaVertTypoDescender`
- `openTypeVheaVertTypoLineGap`
- `openTypeVheaCaretSlopeRise`
- `openTypeVheaCaretSlopeRun`
- `openTypeVheaCaretOffset`

Postscript

- `postscriptFontName`
- `postscriptFullName`
- `postscriptSlantAngle`
- `postscriptUniqueID`
- `postscriptUnderlineThickness`
- `postscriptUnderlinePosition`
- `postscriptIsFixedPitch`
- `postscriptBlueValues`
- `postscriptOtherBlues`
- `postscriptFamilyBlues`
- `postscriptFamilyOtherBlues`
- `postscriptStemSnapH`
- `postscriptStemSnapV`
- `postscriptBlueFuzz`
- `postscriptBlueShift`
- `postscriptBlueScale`
- `postscriptForceBold`
- `postscriptDefaultWidthX`
- `postscriptNominalWidthX`

- `postscriptWeightName`
- `postscriptDefaultCharacter`
- `postscriptWindowsCharacterSet`

Macintosh FOND Resource

- `macintoshFONDFamilyID`
- `macintoshFONDName`

Info

3.8 Kerning

See Also:

Notifications: The Kerning object uses notifications to notify observers of changes.

3.8.1 Tasks

Notifications

- `dispatcher`
- `addObserver()`
- `removeObserver()`
- `hasObserver()`

Parent

- `getParent()`
- `setParent()`

Kerning

3.9 Groups

See Also:

Notifications: The Groups object uses notifications to notify observers of changes.

3.9.1 Tasks

Notifications

- `dispatcher`
- `addObserver()`
- `removeObserver()`
- `hasObserver()`

Parent

- `getParent()`
- `setParent()`

Groups

3.10 Features

See Also:

Notifications: The Features object uses notifications to notify observers of changes.

3.10.1 Tasks

Feature Text

- `text`

Notifications

- `dispatcher`
- `addObserver()`
- `removeObserver()`
- `hasObserver()`

Parent

- `getParent()`
- `setParent()`

Features

3.11 Lib

See Also:

Notifications: The Lib object uses notifications to notify observers of changes.

3.11.1 Tasks

Notifications

- `dispatcher`
- `addObserver()`
- `removeObserver()`
- `hasObserver()`

Parent

- `getParent()`
- `setParent()`

Lib

3.12 Unicode Data

See Also:

Notifications: The UnicodeData object uses notifications to notify observers of changes.

3.12.1 Types of Values

This object works with three types of Unicode values: *real*, *pseudo* and *forced*. A *real* Unicode value is the value assigned in the glyph object. A *pseudo*-Unicode value is an educated guess about what the Unicode value for the glyph could be. This guess is made by splitting the suffix, if one exists, off of the glyph name and then looking up the resulting base in the UnicodeData object. If something is found, the value is the pseudo-Unicode value. A *forced*-Unicode value is a Private Use Area value that is temporaryily mapped to a glyph in the font. These values are stored in the font object only as long as the font is active. They will not be saved into the font. **Note:** Forced-Unicode values are very experimental. They should not be relied upon.

3.12.2 Tasks

Value From Glyph Name

- `unicodeForGlyphName`
- `pseudoUnicodeForGlyphName`

- `forcedUnicodeForGlyphName`

Glyph Name from Value

- `glyphNameForForcedUnicode`
- `glyphNameForUnicode`

Glyph Descriptions

- `blockForGlyphName`
- `categoryForGlyphName`
- `scriptForGlyphName`

Open and Closed Relatives

- `closeRelativeForGlyphName`
- `openRelativeForGlyphName`

Decomposition

- `decompositionBaseForGlyphName`

Sorting Glyphs

- `sortGlyphNames()`

Notifications

- `dispatcher`
- `addObserver()`
- `removeObserver()`
- `hasObserver()`

Parent

- `getParent()`
- `setParent()`

UnicodeData

3.13 NotificationCenter

Direct creation of and interation with these objects will most likely be rare as they are automatically handled by `BaseObject`.

3.13.1 NotificationCenter

3.13.2 Notification

3.14 BaseObject

The main objects in defcon all subclass these objects.

See Also:

`NotificationCenter` The base object uses notifications to notify observers about changes. The API for subscribing/unsubscribing to notifications are detailed below. Some familiarity with the `NotificationCenter` might be helpful.

3.14.1 BaseObject

3.14.2 BaseDictObject

CHAPTER 4

Dependencies

- FontTools
- RoboFab

CHAPTER 5

Indices and tables

- *genindex*
- *modindex*
- *search*

Python Module Index