fontFeatures

Jul 17, 2023

Contents:

CHAPTER 1

The fontFeatures library

The FontFeatures class is a way of representing the transformations - substitutions and positionings - going on inside a font at a semantically high level. It aims to be independent of and unconstrained by the OpenType representation, with the assumption that these high-level objects can be either "compiled down" into AFDKO feature code or directly written to the GSUB/GPOS tables of a font.

FontFeatures aims to marshal data between OTF binary representations, AFDKO feature code, FontDame, and can power other representations such as the FEZ language (see the 'fez' library).

A FontFeatures representation of a font will make use of two other top-level concepts: Features and Routines. Routines are collections of rules; they play a similar function to the AFDKO concept of a lookup, but unlike lookups, Routines do not need to be comprised of rules of the same type. You can think of them as functions that are called on a glyph string.

Here is an example of constructing a simple feature file using fontFeatures:

```
ff = FontFeatures()

liga_ffi = Substitution( [ ["f"], ["f"], ["i"] ], replacement=[["f_f_i"]] )
liga_ffl = Substitution( [ ["f"], ["f"], ["l"] ], replacement=[["f_f_l"]] )
liga_fi = Substitution( [ ["f"], ["i"] ], replacement=[["fi"]] )
liga_ff = Substitution( [ ["f"], ["f"] ], replacement=[["f_f"]] )
liga_routine = Routine(rules=[liga_ffi, liga_ffl, liga_fi, liga_ff])

ff.addFeature("liga", [liga_routine])

# Export Adobe syntax
print(ff.asFea())

font = TTFont("Test.ttf")
ff.buildBinaryFeatures(font)
font.save("Test-liga.ttf")
```

class fontFeatures.**FontFeatures**

An object representing the layout rules in a font.

The initializer has no parameters.

addFeature(*name*, *rs*)
Add Routines to a named feature.

Parameters

- **name** – The feature name.
- **rs** – A sequence of *Routine* or *RoutineReference* objects.

allRules(*ruletype=None*)
Return all rules in the font, optionally filtered by type

Parameters **ruletype** – A class (`Positioning`, `Substitution` etc) to filter the results.

Returns Routines stored in the preamble and within features.

anchors = None
A dictionary mapping glyph names to a dictionary of anchor names / positions.

asFeaAST(*do_gdef=True*)
Returns this font's features as a feaLib AST object, for later translation to AFDKO code.

buildBinaryFeatures(*font*, *axes=[]*)
Adds GDEF, GSUB and GPOS tables to a font object.

Parameters

- **font** – a fontTools `ttFont` object.
- **axes** – an optional list of objects conforming to the `fontTools.designspaceLib.AxisDescriptor` protocol.

ensureLookupsAreReferences(*lookuplist*)
Ensures that all references are lookups.

Naughty people might put *Routine* objects directly into `Chain` lookups. This tidies them up.

features = None
An ordered dictionary mapping feature tags to a list of routine references.

classmethod fromXML(*el*)
Creates a FontFeatures object from a lxml Element object.

gensym(*category*)
Generate a new unique symbol (used for labeling unlabeled data).

Parameters **category** (*str*) – The category for this symbol

Returns: a string representing a unique label.

getNamedClassFor(*glyphs*, *name*)
Find and optionally stores a named class of glyphs

Parameters

- **glyphs** – A sequence of glyph names.
- **name** – A name for this glyph class if it does not exist.

Returns The name of a glyph class. If the exact same set of glyphs was already stored as a glyph class, then the name of that class will be returned. If not, then the class will be stored and the name provided as the `name` argument will be returned.

glyphclasses = None
A dictionary mapping glyph names to their categories.

`hasScriptSupport`(*script*)
Check if the features object has support for a particular script.

Parameters **`script`** (`str`) – A four-character OpenType script code.

Returns: boolean

`hoist_languages`()
Sort routines into scripts and languages and resolve wildcards.

`markRoutineUseInChains`()
Annotate routines which are used in chaining rules.

Generally used when converting the fontFeatures object to another format; allows routines to know where they are being used by annotating them with the `.usedin` property for optimization purposes.

`namedClasses = None`
A mapping of named classes to a list of glyph names which make up the class.

`partitionRoutine`(*routine*, *factor*)
Splits a routine based on a predicate.

This method applies a function to each rule in the routine and creates distinct routines, each containing rules with the same return value from the function. This is useful, for example, when exporting to OpenType, to ensure that all rules in a routine must have the same type, same flags, etc.

Parameters

- **`routine`** – A `Routine` object.
- **`factor`** – A function applied to each of the `Rule` objects.

Returns: A list of `Routine` objects. Additionally, modifies the `.routines` list of the FontFeatures object.

`referenceRoutine`(*r*, *do_usecount=True*)
Store a routine and return a reference to it.

Parameters **`r`** – A `Routine` object.

`resolveAllRoutines`()
Resolve reference use in chains.

Checks that all routines referenced in chain rules can actually be found within the object, and adds pointers to match named routine references with the relevant `Routine` object.

`routineNamed`(*name*)
Finds a routine with the given name.

Parameters **`name`** (`str`) – The name to find

Returns: a `Routine` object if the named routine was found in the features object. Raises a `ValueError` if not.

`routines = None`
All of the layout routines used in this font.

`scratch = None`
Space for items to communicate context to each other.

`setGlyphClassesFromFont`(*font*)
Loads glyph classes from the font.

toXML()
Serializes a FontFeatures object to a lxml Element object.

1.1 Routines: representing collections of layout rules

class fontFeatures.**Routine**(*name=”*, *rules=None*, *address=None*, *inlined=False*, *languages=None*, *parent=None*, *flags=0*, *markFilteringSet=None*, *markAttachmentSet=None*)
Represent a Routine (similar to OT Lookup).

A routine is a set of rules, sometimes but not always with an explicit name. It can apply to a set of language/script pairs.

addComment(*comment*)
Adds a comment to a Routine.

Comments are emitted when the Routine is converted to text formats such as AFDKO.

Parameters **comment** – A string comment.

addRule(*rule*)
Adds a rule to a Routine.

Parameters **rule** – A `Substitution`, `Positioning`, etc. object.

dependencies
Returns a list of `Routine` objects called as lookups in this Routine.

classmethod fromXML(*el*)
Creates a Routine from a lxml Element object.

involved_glyphs
Returns the names of all of the glyphs involved in this Routine.

stage
Returns which shaping stage this routine is used in.

Returns: `sub` for substitution stage, `pos` for positioning stage.

toOTLookup(*font*, *ff*)
Converts a fontFeatures.Routine object to binary.

Parameters

- **font** – A `TTFont` object.
- **ff** – The parent `FontFeatures` object containing this routine.

Returns a list of `fontTools.otlLib.builder` Builder objects allowing this routine to be converted to binary layout format.

toXML()
Serializes a Routine to a lxml Element object.

class fontFeatures.**RoutineReference**(*name=None*, *routine=None*)
A reference to a Routine object, used in a lookup.

Routines can be referenced either by name (for example, when loaded from a textual representation), in which case they will be resolved at a later time, or by providing a pointer to the `Routine` object.

asFea()
Returns this Rule as a string of AFDKO feature text.

classmethod fromXML(*el*)
Creates a RoutineReference from a lxml Element object.

resolve(*ff*)
Resolves the reference in the context of a `FontFeatures` object.

Raises a `ValueError` if a named routine cannot be found.

stage
Returns which shaping stage this routine is used in.

Returns: `sub` for substitution stage, `pos` for positioning stage.

toXML()
Serializes a RoutineReference to a lxml Element object.

class fontFeatures.**ExtensionRoutine**(***kwargs*)
OpenType-specific concept: A routine which contains other routines.

apply_to_buffer(*buf*, *stage=None*, *feature=None*)
Applies shaping rules from this routine to a buffer.

Parameters

- **buf** – A `fontFeatures.shaperLib.Buffer` object.
- **stage** (*str*) – Shaping stage - `sub` or `pos`.
- **feature** (*str*) – The feature being processed. (For debugging.)

Modifies the `buf` object.

asFeaAST()
Returns this extension routine as `fontTools.feaLib.ast` objects.

rules
All rules under this extension.

Returns: A flattened list of `Rule` objects.

stage
Returns which shaping stage this routine is used in.

Returns: `sub` for substitution stage, `pos` for positioning stage.

1.2 Representing individual layout rules

class fontFeatures.**Rule**
A base class for all rules.

asFea()
Returns this Rule as a string of AFDKO feature text.

dependencies
Returns a list of `Routine` objects called as lookups in this Routine.

feaPreamble(*ff*)
Computes any text that needs to go in the feature file header.

classmethod fromXML(*el*)
Creates a Rule from a lxml Element object.

has_context
Does this rule have any pre- or post-context defined?

toXML()
Serializes a Rule to a lxml Element object.

class fontFeatures.**Substitution**(*input_*, *replacement*, *precontext=None*, *postcontext=None*, *address=None*, *languages=None*, *lookups=None*, *reverse=False*, *flags=0*, *force_alt=False*)
Represents a Substitution rule.

A substitution represents any kind of exchange of one set of glyphs for another: single substitutions, multiple substitutions and ligatures are all substitutions. Optionally, substitutions may be followed by precontext and postcontext.

Parameters

- **input** – A list of lists. The outer list represents the positions in the glyph stream to substitute, with the inner list representing the glyph names at each position.
- **replacement** – A list of glyph names.
- **precontext** – A list of list of glyphs which must appear before the input sequence.
- **postcontext** – A list of list of glyphs which must appear before the input sequence.
- **lookups** – A list of list of lookups to be applied to the glyph sequence. The outer list represents the positions in the input sequence, with the inner list containing Routines to apply.
- **reverse** – Boolean representing if the substitutions should take place from the end of the string.
- **force_alt** – Force this substitution to be interpreted as an alternate substitution.

Examples:

```
lig = Substitution(
    [ ["f"], ["i"] ],
    ["f_i"]
) # sub f i by f_i;

contextual = Substitution(
    [ ["dotbelow"] ],
    [ ["dotbelow.post"] ],
    precontext = [["ra-myanmar", "ra-myanmar.bt1", "ra-myanmar.bt2"]]
) # sub [ra-myanmar ra-myanmar.bt1 ra-myanmar.bt2] dotbelow-myanmar'
  # by dotbelow-myanmar.post;
```

classmethod fromXML(*el*)
Creates a rule from a lxml Element object.

involved_glyphs
Returns a set of all glyphs involved in this rule.

lookup_type(*forFea=False*)
Mixin to determine the GSUB lookup type of a fontFeatures.Substitution object

Returns: integer GSUB lookup type.

class fontFeatures.**Positioning**(*glyphs*, *valuerecords*, *precontext=None*, *postcontext=None*, *address=None*, *languages=None*, *flags=0*)
Represents a Positioning rule.

Parameters

- **input** – A list of lists. The outer list represents the positions in the glyph stream to position, with the inner list representing the glyph names at each glyph stream position.
- **valuerecords** – A list of `ValueRecord` objects to be applied at each glyph stream position.
- **precontext** – A list of list of glyphs which must appear before the input sequence.
- **postcontext** – A list of list of glyphs which must appear before the input sequence.

Example:

```
open_up_behs = Positioning(
    [
        ["BEi1", "BEi2"],
        ["sda", "sdb", "dda", "ddb"]
    ],
    [
        ValueRecord(xAdvance=200),
        ValueRecord(xPlacement=50),
    ]
    postcontext = [ medis_finas ]
)
# pos [BEi1 BEi2]' <0 0 200 0> [sda sdb dda ddb]' <0 50 0 0> @medis_finas;
```

classmethod fromXML(*el*)
Creates a rule from a lxml Element object.

involved_glyphs
Returns a set of all glyphs involved in this rule.

lookup_type()
Mixin to determine the GPOS lookup type of a fontFeatures.Positioning object

Returns: integer GPOS lookup type.

class fontFeatures.**Attachment**(*base_name*, *mark_name*, *bases=None*, *marks=None*, *fullname=None*, *flags=0*, *address=None*, *font=None*, *languages=None*, *force_markmark=False*)
Represents an Attachment rule.

Parameters

- **base_name** – Name of the base class.
- **mark_name** – Name of the mark class.
- **bases** – Dictionary. They keys are names of glyphs to act as bases to the attachment (this may be categorized as mark glyphs if the attachment is a mark-to-mark operation); the associated values are a two-element tuple with the coordinates of the anchor.
- **marks** – Dictionary. They keys are names of glyphs to act as marks; the associated values are a two-element tuple with the coordinates of the anchor.
- **force_markmark** – boolean. If true, force this to be interpreted as a mark-to-mark operation

Whether this is a mark-to-base or mark-to-mark operation will be determined by the glyph category of the glyphs involved in the *bases* dictionary and the value of the *force_markmark* argument.

Examples:

```
ff.anchors = {
    "a": { "top": (250, 603) },
    "acutecomb": { "_top": (56, 0) }
}

top_bases = {}
top_marks = {}
for glyphname, anchors in ff.anchors.items():
    for anchorname, position in anchors.items():
        if anchorname == "top":
            top_bases[glyphname] = position
        if anchorname == "_top":
            top_marks[glyphname] = position

# top_bases = { "a": (260,603) }
# top_marks = { "acutecomb": (56,0) }

tops = Attachment("top", "_top", top_bases, top_marks)
```

classmethod fromXML(*el*)
Creates a rule from a lxml Element object.

involved_glyphs
Returns a set of all glyphs involved in this rule.

is_cursive
Returns true if this is a cursive attachment rule.

lookup_type()
Mixin to determine the GPOS lookup type of a fontFeatures.Attachment object

Returns: integer GPOS lookup type.

shaper_inputs()
Returns a list of potential glyphs to determine whether to test if this rule applies at a given point.

would_apply_at_position(*buf*, *ix*, *namedclasses={}*)
Tests to see if this rule would apply at position `ix` of the buffer.

class fontFeatures.**Chaining**(*input_*, *precontext=None*, *postcontext=None*, *address=None*, *languages=None*, *lookups=None*, *flags=0*)
Represents a Chain rule.

A Chain rule represents the operation of calling another Routine when a particular input context is met.

Parameters

- **input** – A list of lists. The outer list represents the positions in the glyph stream to substitute, with the inner list representing the glyph names at each position.
- **precontext** – A list of list of glyphs which must appear before the input sequence.
- **postcontext** – A list of list of glyphs which must appear before the input sequence.
- **lookups** – A list of list of lookups to be applied to the glyph sequence. The outer list represents the positions in the input sequence, with the inner list containing Routines to apply.

Example:

```
sub_Qu = Routine(rules=[
    Substitute([["Q"]], [["Q.beforeu"]])
])

chain = Chain(
    [["Q"]],
    postcontext = [ ["u", "v", "u.sc", "v.sc"] ],
    lookups = [ [sub_Qu] ]
) # sub Q' lookup sub_Qu [u v u.sc v.sc];
```

dependencies
Returns a list of `Routine` objects called as lookups in this Routine.

classmethod fromXML(*el*)
Creates a rule from a lxml Element object.

involved_glyphs
Returns a set of all glyphs involved in this rule.

lookup_type()
Mixin to determine the GSUB/GPOS lookup type of a fontFeatures.Chaining object

Returns: integer GSUB/GPOS lookup type.

stage
Returns which shaping stage this routine is used in.

Returns: `sub` for substitution stage, `pos` for positioning stage.

1.3 Value Records

class `fontFeatures.`**ValueRecord**(*xPlacement=None*, *yPlacement=None*, *xAdvance=None*, *yAdvance=None*, *xPlaDevice=None*, *yPlaDevice=None*, *xAdvDevice=None*, *yAdvDevice=None*, *vertical=False*, *location=None*)
A value record for representing positional changes in advance and placement.

See `fontTools.feaLib.ValueRecord`, from which this inherits.

is_variable
Returns true if any of the elements of the value record are a `fontTools.feaLib.VariableScalar`.

toOTValueRecord(*ff*, *pairPosContext=False*)
Converts the ValueRecord to an `OTLValueRecord` object. If the value record contains any variable scalars, they are saved to the GDEF variation store.

CHAPTER 2

Converting features data between formats

The following modules help with converting font features information to and for different formats.

class `fontFeatures.feaLib.`**`FeaParser`**(*featurefile*, *font=None*, *glyphNames=None*, *includeDir=None*)

Turns a AFDKO feature file or string into a FontFeatures object.

Parameters

- **`featurefile`** – File object or string.
- **`font`** – Optionally, a TTFont object.
- **`glyphNames`** – Optionally, a list of glyph names in the font

`parse`()

Parse the feature code.

Returns A `FontFeatures` object containing the rules of this file.

ttLib: Interfacing with TrueType fonts.

This package contains routines for converting between fontTools objects (representing TrueType/OpenType fonts) and fontFeatures. This particular module is mainly concerned with getting information out of binary OTF/TTF fonts and into fontFeatures.

`fontFeatures.ttLib.`**`unparse`**(*font*, *do_gdef=False*, *doLookups=True*, *config={}*)

Convert a binary OpenType font into a fontFeatures object

Parameters

- **`font`** – A `TTFont` object.
- **`do_gdef`** – Boolean. Whether the GDEF table should also be read.
- **`doLookups`** – Whether the lookups should be read, or just the script/language information and top-level features.
- **`config`** – A dictionary of glyph class and routine names.

`fontFeatures.ttLib.`**`unparseLanguageSystems`**`(`*tables*`)`

Build a set of script / language pairs from a GSUB/GPOS table.

Parameters **`tables`** – A list of `fontTools.ttLib.tables.G_S_U_B_.table_G_S_U_B_` / `fontTools.ttLib.tables.G_P_O_S_.table_G_P_O_S_` objects.

Returns an ordered dictionary whose keys are four-character script tags and whose values are a set of four-character language tags.

`class` `fontFeatures.fontDameLib.`**`FontDameParser`**`(`*lines*, *config={}*, *glyphset=()*`)`

Convert layout files in Monotype's FontDame format to fontFeatures.

Parameters

- **`lines`** – An array of strings containing the FontDame file, one line per string.
- **`config`** – A dictionary of glyph class names.
- **`glyphset`** – A list of glyph names in the font.

`parse``()`

Parses the font file, creating a fontFeatures object.

Returns: A fontFeatures object containing the rules in the FontDame file.

CHAPTER 3

Supporting Modules

The following modules were written to support the creation of fontFeatures plugins.

OpenType fonts are “programmed” using features, which are normally authored in Adobe’s feature file format. This like source code to a computer program: it’s a user-friendly, but computer-unfriendly, way to represent the features.

Inside a font, the features are compiled in an efficient internal format. This is like the binary of a computer program: computers can use it, but they can’t do else anything with it, and people can’t read it.

The purpose of this library is to provide a middle ground for representing features in a machine-manipulable format, kind of like the abstract syntax tree of a computer programmer. This is so that:

- features can be represented in a structured human-readable *and* machine-readable way, analogous to the XML files of the Unified Font Object format.
- features can be more directly authored by programs (such as font editors), rather than them having to output AFDKO feature file format.
- features can be easily manipulated by programs - for example, features from two files merged together, or lookups moved between languages.

  How is this different from fontTool’s `feaLib`? I’m glad you asked. `feaLib` translates between the Adobe feature file format and a abstract syntax tree representing *elements of the feature file* - not representing the feature data. The AST is still “source equivalent”. For example, when you code an `aalt` feature in feature file format, you might include code like `feature salt` to include lookups from another feature. But what’s actually *meant* by that is a set of lookups. `fontFeatures` allows you to manipulate meaning, not description.

CHAPTER 4

Components

fontFeatures consists of the following components:

- *`fontFeatures`* itself, which is an abstract representation of the different layout operations inside a font.
- *`fontFeatures.feaLib`* (included as a mixin) which translates between Adobe feature syntax and fontFeatures representation.
- *`fontFeatures.ttLib`* which translates between OpenType binary fonts and fontFeatures representation. (Currently only OTF -> `fontFeatures` is partially implemented; there is no `fontFeatures` -> OTF compiler yet.)
- *`fontFeatures.fontDameLib`* which translate FontDame text files into fontFeatures objects.

And the following utilities:

- `otf2fea`: translates an OTF file into Adobe features syntax.
- `txt2fea`: translates a FontDame txt file into Adobe features syntax.

CHAPTER 5

Indices and tables

- genindex
- modindex
- search

Python Module Index

Index