FontParts Documentation

Release 0.1

Dr. Rob O. Fab

Dec 09, 2022

Type Designers

CHAPTER 1

Getting Started

These need to be ported and updated from RoboFab's documentation.

For a quick start, here's the sample code from the introduction ported to fontparts:

```python
from fontParts.world import OpenFont

font = OpenFont("/path/to/my/font.ufo")

for glyph in font:
   glyph.leftMargin = glyph.leftMargin + 10
   glyph.rightMargin = glyph.rightMargin + 10
```

Find more of the original samples at https://github.com/robotools/robofab/tree/master/Docs/Examples

CHAPTER 2

Object Reference

FontParts scripts are built on with objects that represent fonts, glyphs, contours and so on. The objects are obtained through fontparts-world.

2.1 Objects

FontParts scripts are built on with objects that represent fonts, glyphs, contours and so on. The objects are obtained through fontparts-world.

2.1.1 Font

Note: This section needs to contain the following:

- description of what this is ✓
- sub-object with basic usage ✓
- bridge to default layer for glyphs for backwards compatibility
- glyph interaction with basic usage

Description

The *Font* object is the central part that connects all glyphs with font information like names, key dimensions etc.

Font objects behave like dictionaries: the glyph name is the key and the returned value is a *Glyph* object for that glyph. If the glyph does not exist, *Font* will raise an `IndexError`.

Font has a couple of important sub-objects which are worth checking out. The font's kerning is stored in a *Kerning* object and can be reached as an attribute at `Font.kerning`. Fontnames, key dimensions, flags etc are stored in a *Info* object which is available through `Font.info`. The `Font.lib` is a *Lib* object which behaves as a dictionary.

Overview

Copy

BaseFont.copy	Copy the font into a new font.

File Operations

BaseFont.path	The path to the file this object represents.
BaseFont.save	Save the font to **path**.
BaseFont.generate	Generate the font to another format.

Sub-Objects

BaseFont.info	The font's `BaseInfo` object.
BaseFont.groups	The font's `BaseGroups` object.
BaseFont.kerning	The font's `BaseKerning` object.
BaseFont.features	The font's `BaseFeatures` object.
BaseFont.lib	The font's `BaseLib` object.
`BaseFont.tempLib`	The font's `BaseLib` object.

Layers

BaseFont.layers	The font's `BaseLayer` objects.
BaseFont.layerOrder	A list of layer names indicating order of the layers in the font.
BaseFont.defaultLayer	The font's default layer.
BaseFont.getLayer	Get the `BaseLayer` with **name**.
BaseFont.newLayer	Make a new layer with **name** and **color**.
BaseFont.removeLayer	Remove the layer with **name** from the font.
BaseFont.insertLayer	Insert **layer** into the font.

Glyphs

BaseFont.__len__	An `int` representing number of glyphs in the layer.
BaseFont.keys	Get a list of all glyphs in the layer.
BaseFont.glyphOrder	The preferred order of the glyphs in the font.
BaseFont.__iter__	Iterate through the `BaseGlyph` objects in the layer.
BaseFont.__contains__	Test if the layer contains a glyph with **name**.
BaseFont.__getitem__	Get the `BaseGlyph` with name from the layer.
BaseFont.newGlyph	Make a new glyph with **name** in the layer.
BaseFont.insertGlyph	Insert **glyph** into the layer.
BaseFont.removeGlyph	Remove the glyph with name from the layer.

Reference

class fontParts.base.**BaseFont**(*pathOrObject=None*, *showInterface=True*)

A font object. This object is almost always created with one of the font functions in fontparts-world.

When constructing a font, the object can be created in a new file, from an existing file or from a native object. This is defined with the **pathOrObjectArgument**. If **pathOrObject** is a string, the string must represent an existing file. If **pathOrObject** is an instance of the environment's unwrapped native font object, wrap it with FontParts. If **pathOrObject** is None, create a new, empty font. If **showInterface** is `False`, the font should be created without graphical interface. The default for **showInterface** is `True`.

Copy

BaseFont.**copy**()

Copy the font into a new font.

```
>>> copiedFont = font.copy()
```

This will copy:

- info
- groups
- kerning
- features
- lib
- layers
- layerOrder
- defaultLayerName
- glyphOrder
- guidelines

File Operations

BaseFont.**path**

The path to the file this object represents.

```
>>> print font.path
"/path/to/my/font.ufo"
```

BaseFont.**save**(*path=None*, *showProgress=False*, *formatVersion=None*, *fileStructure=None*)

Save the font to **path**.

```
>>> font.save()
>>> font.save("/path/to/my/font-2.ufo")
```

If **path** is None, use the font's original location. The file type must be inferred from the file extension of the given path. If no file extension is given, the environment may fall back to the format of its choice. **showProgress** indicates if a progress indicator should be displayed during the operation. Environments may or may not implement this behavior. **formatVersion** indicates the format version that should be used for writing the given file type. For example, if 2 is given for formatVersion and the file type being written if UFO, the file is to be written in UFO 2 format. This value is not limited to UFO format versions. If no format version is given, the original format version of the file should be preserved. If there is no original format version it is implied that the format version is the latest version for the file type as supported by the environment. **fileStructure** indicates the file structure of the written ufo. The **fileStructure** can either be None, 'zip' or 'package', None will use the existing file strucure or the default one for unsaved font. 'package' is the default file structure and 'zip' will save the font to .ufoz.

Note: Environments may define their own rules governing when a file should be saved into its original location and when it should not. For example, a font opened from a compiled OpenType font may not be written back into the original OpenType font.

`BaseFont.`**`close`**(*save=False*)

Close the font.

```
>>> font.close()
```

save is a boolean indicating if the font should be saved prior to closing. If **save** is `True`, the *`BaseFont.save`* method will be called. The default is `False`.

`BaseFont.`**`generate`**(*format*, *path=None*, ***environmentOptions*)

Generate the font to another format.

```
>>> font.generate("otfcff")
>>> font.generate("otfcff", "/path/to/my/font.otf")
```

format defines the file format to output. Standard format identifiers can be found in `BaseFont.generateFormatToExtension`:

Environments are not required to support all of these and environments may define their own format types. **path** defines the location where the new file should be created. If a file already exists at that location, it will be overwritten by the new file. If **path** defines a directory, the file will be output as the current file name, with the appropriate suffix for the format, into the given directory. If no **path** is given, the file will be output into the same directory as the source font with the file named with the current file name, with the appropriate suffix for the format.

Environments may allow unique keyword arguments in this method. For example, if a tool allows decomposing components during a generate routine it may allow this:

```
>>> font.generate("otfcff", "/p/f.otf", decompose=True)
```

Sub-Objects

`BaseFont.`**`info`**

The font's *`BaseInfo`* object.

```
>>> font.info.familyName
"My Family"
```

`BaseFont.`**`groups`**

The font's *`BaseGroups`* object.

```
>>> font.groups["myGroup"]
["A", "B", "C"]
```

`BaseFont.`**`kerning`**
The font's *`BaseKerning`* object.

```
>>> font.kerning["A", "B"]
-100
```

`BaseFont.`**`features`**
The font's *`BaseFeatures`* object.

```
>>> font.features.text
"include(features/substitutions.fea);"
```

`BaseFont.`**`lib`**
The font's *`BaseLib`* object.

```
>>> font.lib["org.robofab.hello"]
"world"
```

Layers

`BaseFont.`**`layers`**
The font's *`BaseLayer`* objects.

```
>>> for layer in font.layers:
...     layer.name
"My Layer 1"
"My Layer 2"
```

`BaseFont.`**`layerOrder`**
A list of layer names indicating order of the layers in the font.

```
>>> font.layerOrder = ["My Layer 2", "My Layer 1"]
>>> font.layerOrder
["My Layer 2", "My Layer 1"]
```

`BaseFont.`**`defaultLayer`**
The font's default layer.

```
>>> layer = font.defaultLayer
>>> font.defaultLayer = otherLayer
```

`BaseFont.`**`getLayer`**`(`*name*`)`
Get the *`BaseLayer`* with **name**.

```
>>> layer = font.getLayer("My Layer 2")
```

`BaseFont.`**`newLayer`**`(`*name*, *color=None*`)`
Make a new layer with **name** and **color**. **name** must be a *String* and **color** must be a *Color* or `None`.

```
>>> layer = font.newLayer("My Layer 3")
```

The will return the newly created *`BaseLayer`*.

`BaseFont.`**`removeLayer`**`(`*name*`)`
Remove the layer with **name** from the font.

```
>>> font.removeLayer("My Layer 3")
```

`BaseFont.`**`insertLayer`**`(`*layer*, *name=None*`)`

Insert **layer** into the font.

```
>>> layer = font.insertLayer(otherLayer, name="layer 2")
```

This will not insert the layer directly. Rather, a new layer will be created and the data from **layer** will be copied to to the new layer. **name** indicates the name that should be assigned to the layer after insertion. If **name** is not given, the layer's original name must be used. If the layer does not have a name, an error must be raised. The data that will be inserted from **layer** is the same data as documented in *`BaseLayer.copy`*.

Glyphs

Interacting with glyphs at the font level is a shortcut for interacting with glyphs in the default layer.

```
>>> glyph = font.newGlyph("A")
```

Does the same thing as:

```
>>> glyph = font.getLayer(font.defaultLayerName).newGlyph("A")
```

`BaseFont.`**`__len__`**`()`

An `int` representing number of glyphs in the layer.

```
>>> len(layer)
256
```

`BaseFont.`**`keys`**`()`

Get a list of all glyphs in the layer.

```
>>> layer.keys()
["B", "C", "A"]
```

The order of the glyphs is undefined.

`BaseFont.`**`glyphOrder`**

The preferred order of the glyphs in the font.

```
>>> font.glyphOrder
["C", "B", "A"]
>>> font.glyphOrder = ["A", "B", "C"]
```

`BaseFont.`**`__iter__`**`()`

Iterate through the *`BaseGlyph`* objects in the layer.

```
>>> for glyph in layer:
...     glyph.name
"A"
"B"
"C"
```

`BaseFont.`**`__contains__`**`(`*name*`)`

Test if the layer contains a glyph with **name**.

```
>>> "A" in layer
True
```

BaseFont.**__getitem__**(*name*)

Get the `BaseGlyph` with name from the layer.

```
>>> glyph = layer["A"]
```

BaseFont.**newGlyph**(*name*, *clear=True*)

Make a new glyph with **name** in the layer.

```
>>> glyph = layer.newGlyph("A")
```

The newly created `BaseGlyph` will be returned.

If the glyph exists in the layer and clear is set to `False`, the existing glyph will be returned, otherwise the default behavior is to clear the exisiting glyph.

BaseFont.**insertGlyph**(*glyph*, *name=None*)

Insert **glyph** into the layer.

```
>>> glyph = layer.insertGlyph(otherGlyph, name="A")
```

This method is deprecated. `BaseFont.__setitem__` instead.

BaseFont.**removeGlyph**(*name*)

Remove the glyph with name from the layer.

```
>>> layer.removeGlyph("A")
```

This method is deprecated. `BaseFont.__delitem__` instead.

Guidelines

BaseFont.**guidelines**

An *Immutable List* of font-level `BaseGuideline` objects.

```
>>> for guideline in font.guidelines:
...     guideline.angle
0
45
90
```

BaseFont.**appendGuideline**(*position=None*, *angle=None*, *name=None*, *color=None*, *guideline=None*)

Append a new guideline to the font.

```
>>> guideline = font.appendGuideline((50, 0), 90)
>>> guideline = font.appendGuideline((0, 540), 0, name="overshoot",
>>> color=(0, 0, 0, 0.2))
```

position must be a *Coordinate* indicating the position of the guideline. **angle** indicates the *Angle* of the guideline. **name** indicates the name for the guideline. This must be a *String* or `None`. **color** indicates the color for the guideline. This must be a *Color* or `None`. This will return the newly created `BaseGuidline` object.

`guideline` may be a *BaseGuideline* object from which attribute values will be copied. If `position`, `angle`, `name` or `color` are specified as arguments, those values will be used instead of the values in the given guideline object.

`BaseFont.`**`removeGuideline`**`(`*guideline*`)`

Remove **guideline** from the font.

```
>>> font.removeGuideline(guideline)
>>> font.removeGuideline(2)
```

guideline can be a guideline object or an integer representing the guideline index.

`BaseFont.`**`clearGuidelines`**`()`

Clear all guidelines.

```
>>> font.clearGuidelines()
```

Interpolation

`BaseFont.`**`isCompatible`**`(`*other*`)`

Evaluate interpolation compatibility with **other**.

```
>>> compatible, report = self.isCompatible(otherFont)
>>> compatible
False
>>> report
[Fatal] Glyph: "test1" + "test2"
[Fatal] Glyph: "test1" contains 1 contours | "test2" contains 2 contours
```

This will return a `bool` indicating if the font is compatible for interpolation with **other** and a *String* of compatibility notes.

`BaseFont.`**`interpolate`**`(`*factor*, *minFont*, *maxFont*, *round=True*, *suppressError=True*`)`

Interpolate all possible data in the font.

```
>>> font.interpolate(0.5, otherFont1, otherFont2)
>>> font.interpolate((0.5, 2.0), otherFont1, otherFont2, round=False)
```

The interpolation occurs on a 0 to 1.0 range where **minFont** is located at 0 and **maxFont** is located at 1.0. **factor** is the interpolation value. It may be less than 0 and greater than 1.0. It may be a *Integer/Float* or a tuple of two *Integer/Float*. If it is a tuple, the first number indicates the x factor and the second number indicates the y factor. **round** indicates if the result should be rounded to integers. **suppressError** indicates if incompatible data should be ignored or if an error should be raised when such incompatibilities are found.

Normalization

`BaseFont.`**`round`**`()`

Round all approriate data to integers.

```
>>> font.round()
```

This is the equivalent of calling the round method on:

- info
- kerning

- the default layer
- font-level guidelines

This applies only to the default layer.

BaseFont.**autoUnicodes**()

Use heuristics to set Unicode values in all glyphs.

```
>>> font.autoUnicodes()
```

Environments will define their own heuristics for automatically determining values.

This applies only to the default layer.

Environment

BaseFont.**naked**()

Return the environment's native object that has been wrapped by this object.

```
>>> loweLevelObj = obj.naked()
```

BaseFont.**changed**(**args*, ***kwargs*)

Tell the environment that something has changed in the object. The behavior of this method will vary from environment to environment.

```
>>> obj.changed()
```

2.1.2 Info

Description

The *Info* object contains all names, numbers, URLs, dimensions, values, etc. that would otherwise clutter up the font object. You don't have to create a *Info* object yourself, *Font* makes one when it is created.

Info validates any value set for a *Info <BaseInfo>* item, but does not check if the data is sane (i.e., you can set valid but incorrect data).

Overview

BaseInfo.copy	Copy this object into a new object of the same type.
BaseInfo.font	The info's parent font.
BaseInfo.interpolate	Interpolate all pairs between minInfo and maxInfo.
BaseInfo.round	Round the following attributes to integers:
BaseInfo.update	Update this object with the values from **otherInfo**.
BaseInfo.naked	Return the environment's native object that has been wrapped by this object.
BaseInfo.changed	Tell the environment that something has changed in the object.

Reference

class fontParts.base.**BaseInfo**(**args*, ***kwargs*)

Copy

`BaseInfo.`**`copy`**`()`

Copy this object into a new object of the same type. The returned object will not have a parent object.

Parents

`BaseInfo.`**`font`**

The info's parent font.

Interpolation

`BaseInfo.`**`interpolate`**`(`*factor*, *minInfo*, *maxInfo*, *round=True*, *suppressError=True*`)`

Interpolate all pairs between minInfo and maxInfo. The interpolation occurs on a 0 to 1.0 range where minInfo is located at 0 and maxInfo is located at 1.0.

factor is the interpolation value. It may be less than 0 and greater than 1.0. It may be a number (integer, float) or a tuple of two numbers. If it is a tuple, the first number indicates the x factor and the second number indicates the y factor.

round indicates if the result should be rounded to integers.

suppressError indicates if incompatible data should be ignored or if an error should be raised when such incompatibilities are found.

Normalization

`BaseInfo.`**`round`**`()`

Round the following attributes to integers:

- unitsPerEm
- descender
- xHeight
- capHeight
- ascender
- openTypeHeadLowestRecPPEM
- openTypeHheaAscender
- openTypeHheaDescender
- openTypeHheaLineGap
- openTypeHheaCaretSlopeRise
- openTypeHheaCaretSlopeRun
- openTypeHheaCaretOffset
- openTypeOS2WidthClass
- openTypeOS2WeightClass
- openTypeOS2TypoAscender

- openTypeOS2TypoDescender
- openTypeOS2TypoLineGap
- openTypeOS2WinAscent
- openTypeOS2WinDescent
- openTypeOS2SubscriptXSize
- openTypeOS2SubscriptYSize
- openTypeOS2SubscriptXOffset
- openTypeOS2SubscriptYOffset
- openTypeOS2SuperscriptXSize
- openTypeOS2SuperscriptYSize
- openTypeOS2SuperscriptXOffset
- openTypeOS2SuperscriptYOffset
- openTypeOS2StrikeoutSize
- openTypeOS2StrikeoutPosition
- openTypeVheaVertTypoAscender
- openTypeVheaVertTypoDescender
- openTypeVheaVertTypoLineGap
- openTypeVheaCaretSlopeRise
- openTypeVheaCaretSlopeRun
- openTypeVheaCaretOffset
- postscriptSlantAngle
- postscriptUnderlineThickness
- postscriptUnderlinePosition
- postscriptBlueValues
- postscriptOtherBlues
- postscriptFamilyBlues
- postscriptFamilyOtherBlues
- postscriptStemSnapH
- postscriptStemSnapV
- postscriptBlueFuzz
- postscriptBlueShift
- postscriptDefaultWidthX
- postscriptNominalWidthX

Update

`BaseInfo.update(other)`
Update this object with the values from **otherInfo**.

Environment

`BaseInfo.naked()`
Return the environment's native object that has been wrapped by this object.

```
>>> loweLevelObj = obj.naked()
```

`BaseInfo.changed(*args, **kwargs)`
Tell the environment that something has changed in the object. The behavior of this method will vary from environment to environment.

```
>>> obj.changed()
```

2.1.3 Groups

Description

Groups are collections of glyphs. Groups are used for many things, from OpenType features, kerning, or just keeping track of a collection of related glyphs. The name of the group must be at least one character, with no limit to the maximum length for the name, nor any limit on the characters used in a name. With the exception of the kerning groups defined below, glyphs may be in more than one group and they may appear within the same group more than once. Glyphs in the groups are not required to be in the font.

Groups behave like a Python dictionary. Anything you can do with a dictionary in Python, you can do with Groups.

```
font = CurrentFont()
for name, members in font.groups.items():
    print(name)
    print(members)
```

It is important to understand that any changes to the returned group contents will not be reflected in the groups object. This means that the following will not update the font's groups:

```
group = list(font.groups["myGroup"])
group.remove("A")
```

If one wants to make a change to the group contents, one should do the following instead:

```
group = list(font.groups["myGroup"])
group.remove("A")
font.groups["myGroup"] = group
```

Kerning Groups

Groups may be used as members of kerning pairs in *`BaseKerning`*. These groups are divided into two types: groups that appear on the first side of a kerning pair and groups that appear on the second side of a kerning pair.

Kerning groups must begin with standard prefixes. The prefix for groups intended for use in the first side of a kerning pair is `public.kern1.`. The prefix for groups intended for use in the second side of a kerning pair is `public.kern2.`. One or more characters must follow the prefix.

Kerning groups must strictly adhere to the following rules:

1. Kerning group names must begin with the appropriate prefix.
2. Only kerning groups are allowed to use the kerning group prefixes in their names.
3. Kerning groups are not required to appear in the kerning pairs.
4. Glyphs must not appear in more than one kerning group per side.

These rules come from the Unified Font Object, more information on implementation details for application developers can be found there.

Overview

BaseGroups.copy	Copy this object into a new object of the same type.
BaseGroups.font	The Groups' parent *BaseFont*.
BaseGroups.__contains__	Tests to see if a group name is in the Groups.
BaseGroups.__delitem__	Removes **groupName** from the Groups.
BaseGroups.__getitem__	Returns the contents of the named group.
BaseGroups.__iter__	Iterates through the Groups, giving the key for each iteration.
BaseGroups.__len__	Returns the number of groups in Groups as an `int`..
BaseGroups.__setitem__	Sets the **groupName** to the list of **glyphNames**.
BaseGroups.clear	Removes all group information from Groups, resetting the Groups to an empty dictionary.
BaseGroups.get	Returns the contents of the named group.
BaseGroups.items	Returns a list of `tuple` of each group name and group members.
BaseGroups.keys	Returns a `list` of all the group names in Groups.
BaseGroups.pop	Removes the **groupName** from the Groups and returns the list of group members.
BaseGroups.update	Updates the Groups based on **otherGroups**.
BaseGroups.values	Returns a `list` of each named group's members.
BaseGroups.findGlyph	Returns a `list` of the group or groups associated with **glyphName**.
BaseGroups.naked	Return the environment's native object that has been wrapped by this object.
BaseGroups.changed	Tell the environment that something has changed in the object.

Reference

class `fontParts.base.`**BaseGroups**(**args*, ***kwargs*)

A Groups object. This object normally created as part of a *BaseFont*. An orphan Groups object can be created like this:

```
>>> groups = RGroups()
```

This object behaves like a Python dictionary. Most of the dictionary functionality comes from `BaseDict`, look at that object for the required environment implementation details.

Groups uses `normalizers.normalizeGroupKey` to normalize the key of the `dict`, and `normalizers.normalizeGroupValue` to normalize the value of the `dict`.

Copy

`BaseGroups.`**`copy`**`()`

Copy this object into a new object of the same type. The returned object will not have a parent object.

Parents

- `font` The groups' parent *BaseFont*.

Dictionary

`BaseGroups.`**`__contains__`**`(`*groupName*`)`

Tests to see if a group name is in the Groups. **groupName** will be a *String*. This returns a `bool` indicating if the **groupName** is in the Groups.

```
>>> "myGroup" in font.groups
True
```

`BaseGroups.`**`__delitem__`**`(`*groupName*`)`

Removes **groupName** from the Groups. **groupName** is a *String*.:

```
>>> del font.groups["myGroup"]
```

`BaseGroups.`**`__getitem__`**`(`*groupName*`)`

Returns the contents of the named group. **groupName** is a *String*. The returned value will be a *Immutable List* of the group contents.:

```
>>> font.groups["myGroup"]
("A", "B", "C")
```

It is important to understand that any changes to the returned group contents will not be reflected in the Groups object. If one wants to make a change to the group contents, one should do the following:

```
>>> group = font.groups["myGroup"]
>>> group.remove("A")
>>> font.groups["myGroup"] = group
```

`BaseGroups.`**`__iter__`**`()`

Iterates through the Groups, giving the key for each iteration. The order that the Groups will iterate though is not fixed nor is it ordered.:

```
>>> for groupName in font.groups:
>>>     print groupName
"myGroup"
"myGroup3"
"myGroup2"
```

`BaseGroups.`**`__len__`**`()`

Returns the number of groups in Groups as an `int`.:

```
>>> len(font.groups)
5
```

BaseGroups.**__setitem__**(*groupName*, *glyphNames*)

Sets the **groupName** to the list of **glyphNames**. **groupName** is the group name as a *String* and **glyphNames** is a `list` of glyph names as *String*.

```
>>> font.groups["myGroup"] = ["A", "B", "C"]
```

BaseGroups.**clear**()

Removes all group information from Groups, resetting the Groups to an empty dictionary.

```
>>> font.groups.clear()
```

BaseGroups.**get**(*groupName*, *default=None*)

Returns the contents of the named group. **groupName** is a *String*, and the returned values will either be *Immutable List* of group contents or `None` if no group was found.

```
>>> font.groups["myGroup"]
("A", "B", "C")
```

It is important to understand that any changes to the returned group contents will not be reflected in the Groups object. If one wants to make a change to the group contents, one should do the following:

```
>>> group = font.groups["myGroup"]
>>> group.remove("A")
>>> font.groups["myGroup"] = group
```

BaseGroups.**items**()

Returns a list of `tuple` of each group name and group members. Group names are *String* and group members are a *Immutable List* of *String*. The initial list will be unordered.

```
>>> font.groups.items()
[("myGroup", ("A", "B", "C"), ("myGroup2", ("D", "E", "F"))]
```

BaseGroups.**keys**()

Returns a `list` of all the group names in Groups. This list will be unordered.:

```
>>> font.groups.keys()
["myGroup4", "myGroup1", "myGroup5"]
```

BaseGroups.**pop**(*groupName*, *default=None*)

Removes the **groupName** from the Groups and returns the list of group members. If no group is found, **default** is returned. **groupName** is a *String*. This must return either **default** or a *Immutable List* of glyph names as *String*.

```
>>> font.groups.pop("myGroup")
("A", "B", "C")
```

BaseGroups.**update**(*otherGroups*)

Updates the Groups based on **otherGroups**. *otherGroups** is a `dict` of groups information. If a group from **otherGroups** is in Groups, the group members will be replaced by the group members from **otherGroups**. If a group from **otherGroups** is not in the Groups, it is added to the Groups. If Groups contain a group name that is not in *otherGroups**, it is not changed.

```
>>> font.groups.update(newGroups)
```

`BaseGroups.`**`values`**`()`

Returns a `list` of each named group's members. This will be a list of lists, the group members will be a *Immutable List* of *String*. The initial list will be unordered.

```
>>> font.groups.items()
[("A", "B", "C"), ("D", "E", "F")]
```

Queries

`BaseGroups.`**`findGlyph`**`(`*glyphName*`)`

Returns a `list` of the group or groups associated with **glyphName**. **glyphName** will be an *String*. If no group is found to contain **glyphName** an empty `list` will be returned.

```
>>> font.groups.findGlyph("A")
["A_accented"]
```

Environment

`BaseGroups.`**`naked`**`()`

Return the environment's native object that has been wrapped by this object.

```
>>> loweLevelObj = obj.naked()
```

`BaseGroups.`**`changed`**`(`**args*, ***kwargs*`)`

Tell the environment that something has changed in the object. The behavior of this method will vary from environment to environment.

```
>>> obj.changed()
```

2.1.4 Kerning

Description

Kerning groups must begin with standard prefixes. The prefix for groups intended for use in the first side of a kerning pair is `public.kern1.`. The prefix for groups intended for use in the second side of a kerning pair is `public.kern2.`. One or more characters must follow the prefix.

Kerning groups must strictly adhere to the following rules:

1. Kerning group names must begin with the appropriate prefix.
2. Only kerning groups are allowed to use the kerning group prefixes in their names.
3. Kerning groups are not required to appear in the kerning pairs.
4. Glyphs must not appear in more than one kerning group per side.

These rules come from the Unified Font Object, more information on implementation details for application developers can be found there.

Overview

Copy

BaseKerning.copy	Copy this object into a new object of the same type.

Parents

BaseKerning.font	The Kerning's parent *BaseFont*.

Dictionary

BaseKerning.__len__	Returns the number of pairs in Kerning as an `int`..
BaseKerning.keys	Returns a `list` of all the pairs in Kerning.
BaseKerning.items	Returns a list of `tuples` of each pair and value.
BaseKerning.values	Returns a `list` of each pair's values, the values will be *Integer/Float*s.
BaseKerning.__contains__	Tests to see if a pair is in the Kerning.
BaseKerning.__setitem__	Sets the **pair** to the list of **value**.
BaseKerning.__getitem__	Returns the kerning value of the pair.
BaseKerning.get	Returns the value for the kerning pair.
BaseKerning.find	Returns the value for the kerning pair - even if the pair only exists implicitly (one or both sides may be members of a kerning group).
BaseKerning.__delitem__	Removes **pair** from the Kerning.
BaseKerning.pop	Removes the **pair** from the Kerning and returns the value as an `int`.
BaseKerning.__iter__	Iterates through the Kerning, giving the pair for each iteration.
BaseKerning.update	Updates the Kerning based on **otherKerning**.
BaseKerning.clear	Removes all information from Kerning, resetting the Kerning to an empty dictionary.

Transformations

BaseKerning.scaleBy	Scales all kerning values by **factor**.

Interpolation

BaseKerning.interpolate	Interpolates all pairs between two *BaseKerning* objects:

Normalization

BaseKerning.round	Rounds the kerning values to increments of **multiple**, which will be an `int`.

Environment

BaseKerning.naked	Return the environment's native object that has been wrapped by this object.
BaseKerning.changed	Tell the environment that something has changed in the object.

Reference

class `fontParts.base.`**BaseKerning**(**args*, ***kwargs*)

A Kerning object. This object normally created as part of a *BaseFont*. An orphan Kerning object can be created like this:

```
>>> groups = RKerning()
```

This object behaves like a Python dictionary. Most of the dictionary functionality comes from `BaseDict`, look at that object for the required environment implementation details.

Kerning uses `normalizers.normalizeKerningKey` to normalize the key of the `dict`, and `normalizers.normalizeKerningValue` to normalize the the value of the `dict`.

Copy

`BaseKerning.`**copy**()

Copy this object into a new object of the same type. The returned object will not have a parent object.

Parents

`BaseKerning.`**font**

The Kerning's parent *BaseFont*.

Dictionary

`BaseKerning.`**__len__**()

Returns the number of pairs in Kerning as an `int`.:

```
>>> len(font.kerning)
5
```

`BaseKerning.`**keys**()

Returns a `list` of all the pairs in Kerning. This list will be unordered.:

```
>>> font.kerning.keys()
[("A", "Y"), ("A", "V"), ("A", "W")]
```

`BaseKerning.`**items**()

Returns a list of `tuple`s of each pair and value. Pairs are a `tuple` of two *String*s and values are *Integer/Float*.

The initial list will be unordered.

```
>>> font.kerning.items()
[(("A", "V"), -30), (("A", "W"), -10)]
```

BaseKerning.**values**()

Returns a `list` of each pair's values, the values will be *Integer/Float*s.

The list will be unordered.

```
>>> font.kerning.items()
[-20, -15, 5, 3.5]
```

BaseKerning.**__contains__**(*pair*)

Tests to see if a pair is in the Kerning. **pair** will be a `tuple` of two *String*s.

This returns a `bool` indicating if the **pair** is in the Kerning.

```
>>> ("A", "V") in font.kerning
True
```

BaseKerning.**__setitem__**(*pair*, *value*)

Sets the **pair** to the list of **value**. **pair** is the pair as a `tuple` of two *String*s and **value**

is a *Integer/Float*.

```
>>> font.kerning[("A", "V")] = -20
>>> font.kerning[("A", "W")] = -10.5
```

BaseKerning.**__getitem__**(*pair*)

Returns the kerning value of the pair. **pair** is a `tuple` of two *String*s.

The returned value will be a *Integer/Float*.:

```
>>> font.kerning[("A", "V")]
-15
```

It is important to understand that any changes to the returned value will not be reflected in the Kerning object. If one wants to make a change to the value, one should do the following:

```
>>> value = font.kerning[("A", "V")]
>>> value += 10
>>> font.kerning[("A", "V")] = value
```

BaseKerning.**get**(*pair*, *default=None*)

Returns the value for the kerning pair. **pair** is a `tuple` of two *String*s, and the returned values will either be *Integer/Float* or `None` if no pair was found.

```
>>> font.kerning[("A", "V")]
-25
```

It is important to understand that any changes to the returned value will not be reflected in the Kerning object. If one wants to make a change to the value, one should do the following:

```
>>> value = font.kerning[("A", "V")]
>>> value += 10
>>> font.kerning[("A", "V")] = value
```

`BaseKerning.`**`find`**`(`*pair*, *default=None*`)`

Returns the value for the kerning pair - even if the pair only exists implicitly (one or both sides may be members of a kerning group).

pair is a `tuple` of two *String*s, and the returned values will either be *Integer/Float* or `None` if no pair was found.

```
>>> font.kerning[("A", "V")]
-25
```

`BaseKerning.`**`__delitem__`**`(`*pair*`)`

Removes **pair** from the Kerning. **pair** is a `tuple` of two *String*s.:

```
>>> del font.kerning[("A","V")]
```

`BaseKerning.`**`pop`**`(`*pair*, *default=None*`)`

Removes the **pair** from the Kerning and returns the value as an `int`. If no pair is found, **default** is returned. **pair** is a `tuple` of two *String*s. This must return either

default or a *Integer/Float*.

```
>>> font.kerning.pop(("A", "V"))
-20
>>> font.kerning.pop(("A", "W"))
-10.5
```

`BaseKerning.`**`__iter__`**`()`

Iterates through the Kerning, giving the pair for each iteration. The order that the Kerning will iterate though is not fixed nor is it ordered.:

```
>>> for pair in font.kerning:
>>>     print pair
("A", "Y")
("A", "V")
("A", "W")
```

`BaseKerning.`**`update`**`(`*otherKerning*`)`

Updates the Kerning based on **otherKerning**. **otherKerning** is a `dict` of kerning information. If a pair from **otherKerning** is in Kerning, the pair value will be replaced by the value from **otherKerning**. If a pair from **otherKerning** is not in the Kerning, it is added to the pairs. If Kerning contains a pair that is not in **otherKerning**, it is not changed.

```
>>> font.kerning.update(newKerning)
```

`BaseKerning.`**`clear`**`()`

Removes all information from Kerning, resetting the Kerning to an empty dictionary.

```
>>> font.kerning.clear()
```

Transformations

`BaseKerning.`**`scaleBy`**`(`*factor*`)`

Scales all kerning values by **factor**. **factor** will be an *Integer/Float*, `tuple` or `list`. The first value of the **factor** will be used to scale the kerning values.

```
>>> myKerning.scaleBy(2)
>>> myKerning.scaleBy((2,3))
```

Interpolation

`BaseKerning.`**`interpolate`**(*factor*, *minKerning*, *maxKerning*, *round=True*, *suppressError=True*)

Interpolates all pairs between two *`BaseKerning`* objects:

```
>>> myKerning.interpolate(kerningOne, kerningTwo)
```

minKerning and **maxKerning**. The interpolation occurs on a 0 to 1.0 range where **minKerning** is located at 0 and **maxKerning** is located at 1.0. The kerning data is replaced by the interpolated kerning.

- **factor** is the interpolation value. It may be less than 0 and greater than 1.0. It may be an *Integer/Float*, `tuple` or `list`. If it is a `tuple` or `list`, the first number indicates the x factor and the second number indicates the y factor.
- **round** is a `bool` indicating if the result should be rounded to `ints`. The default behavior is to round interpolated kerning.
- **suppressError** is a `bool` indicating if incompatible data should be ignored or if an error should be raised when such incompatibilities are found. The default behavior is to ignore incompatible data.

Normalization

`BaseKerning.`**`round`**(*multiple=1*)

Rounds the kerning values to increments of **multiple**, which will be an `int`.

The default behavior is to round to increments of 1.

Environment

`BaseKerning.`**`naked`**()

Return the environment's native object that has been wrapped by this object.

```
>>> loweLevelObj = obj.naked()
```

`BaseKerning.`**`changed`**(**args*, ***kwargs*)

Tell the environment that something has changed in the object. The behavior of this method will vary from environment to environment.

```
>>> obj.changed()
```

2.1.5 Features

Description

Features is text in the Adobe Font Development Kit for OpenType .fea syntax that describes the OpenType features of your font. The OpenType Cookbook is a great place to start learning how to write features. Your features must be self-contained; for example, any glyph or mark classes must be defined within the file. No assumption should be made about the validity of the syntax, and FontParts does not check the validity of the syntax.

Note: It is important to note that the features file may contain data that is a duplicate of or data that is in conflict with the data in *BaseKerning*, *BaseGroups*, and *BaseInfo*. Synchronization is up to the user and application developers.

```
font = CurrentFont()
print(font.features)
```

Overview

BaseFeatures.copy	Copy this object into a new object of the same type.
BaseFeatures.font	The features' parent *BaseFont*.
BaseFeatures.text	The .fea formated text representing the features.It must be a *String*..

Reference

class fontParts.base.**BaseFeatures**(**args*, ***kwargs*)

Copy

BaseFeatures.**copy**()

Copy this object into a new object of the same type. The returned object will not have a parent object.

Parents

BaseFeatures.**font**

The features' parent *BaseFont*.

Attributes

BaseFeatures.**text**

The .fea formated text representing the features. It must be a *String*.

2.1.6 Lib

Overview

BaseLib.copy	Copy this object into a new object of the same type.
BaseLib.glyph	The lib's parent glyph.
BaseLib.font	The lib's parent font.
BaseLib.__len__	Returns the number of keys in Lib as an `int`..
BaseLib.keys	Returns a `list` of all the key names in Lib.
BaseLib.items	Returns a list of `tuple` of each key name and key items.

Continued on next page

Table 16 – continued from previous page

BaseLib.values	Returns a `list` of each named key's members.
BaseLib.__contains__	Tests to see if a lib name is in the Lib.
BaseLib.__setitem__	Sets the **key** to the list of **items**.
BaseLib.__getitem__	Returns the contents of the named lib.
BaseLib.get	Returns the contents of the named key.
BaseLib.__delitem__	Removes **key** from the Lib.
BaseLib.pop	Removes the **key** from the Lib and returns the `list` of key members.
BaseLib.__iter__	Iterates through the Lib, giving the key for each iteration.
BaseLib.update	Updates the Lib based on **otherLib**.
BaseLib.clear	Removes all keys from Lib, resetting the Lib to an empty dictionary.
BaseLib.naked	Return the environment's native object that has been wrapped by this object.
BaseLib.changed	Tell the environment that something has changed in the object.

Reference

class fontParts.base.**BaseLib**(**args*, ***kwargs*)

A Lib object. This object normally created as part of a *BaseFont*. An orphan Lib object can be created like this:

```
>>> lib = RLib()
```

This object behaves like a Python dictionary. Most of the dictionary functionality comes from `BaseDict`, look at that object for the required environment implementation details.

Lib uses `normalizers.normalizeLibKey` to normalize the key of the `dict`, and `normalizers.normalizeLibValue` to normalize the value of the `dict`.

Copy

BaseLib.**copy**()

Copy this object into a new object of the same type. The returned object will not have a parent object.

Parents

BaseLib.**glyph**

The lib's parent glyph.

BaseLib.**font**

The lib's parent font.

Dictionary

BaseLib.**__len__**()

Returns the number of keys in Lib as an `int`.:

```
>>> len(font.lib)
5
```

`BaseLib.`**`keys`**`()`

Returns a `list` of all the key names in Lib. This list will be unordered.:

```
>>> font.lib.keys()
["public.glyphOrder", "org.robofab.scripts.SomeData",
 "public.postscriptNames"]
```

`BaseLib.`**`items`**`()`

Returns a list of `tuple` of each key name and key items. Keys are *String* and key members are a `list` of *String*. The initial list will be unordered.

```
>>> font.lib.items()
[("public.glyphOrder", ["A", "B", "C"]),
 ("public.postscriptNames", {'be': 'uni0431', 'ze': 'uni0437'})]
```

`BaseLib.`**`values`**`()`

Returns a `list` of each named key's members. This will be a list of lists, the key members will be a `list` of *String*. The initial list will be unordered.

```
>>> font.lib.items()
[["A", "B", "C"], {'be': 'uni0431', 'ze': 'uni0437'}]
```

`BaseLib.`**`__contains__`**`(`*key*`)`

Tests to see if a lib name is in the Lib. **key** will be a *String*. This returns a `bool` indicating if the **key** is in the Lib.

```
>>> "public.glyphOrder" in font.lib
True
```

`BaseLib.`**`__setitem__`**`(`*key*, *items*`)`

Sets the **key** to the list of **items**. **key** is the lib name as a *String* and **items** is a `list` of items as *String*.

```
>>> font.lib["public.glyphOrder"] = ["A", "B", "C"]
```

`BaseLib.`**`__getitem__`**`(`*key*`)`

Returns the contents of the named lib. **key** is a *String*. The returned value will be a `list` of the lib contents.:

```
>>> font.lib["public.glyphOrder"]
["A", "B", "C"]
```

It is important to understand that any changes to the returned lib contents will not be reflected in the Lib object. If one wants to make a change to the lib contents, one should do the following:

```
>>> lib = font.lib["public.glyphOrder"]
>>> lib.remove("A")
>>> font.lib["public.glyphOrder"] = lib
```

`BaseLib.`**`get`**`(`*key*, *default=None*`)`

Returns the contents of the named key. **key** is a *String*, and the returned values will either be `list` of key contents or `None` if no key was found.

```
>>> font.lib["public.glyphOrder"]
["A", "B", "C"]
```

It is important to understand that any changes to the returned key contents will not be reflected in the Lib object. If one wants to make a change to the key contents, one should do the following:

```
>>> lib = font.lib["public.glyphOrder"]
>>> lib.remove("A")
>>> font.lib["public.glyphOrder"] = lib
```

BaseLib.**__delitem__**(*key*)
Removes **key** from the Lib. **key** is a *String*.:

```
>>> del font.lib["public.glyphOrder"]
```

BaseLib.**pop**(*key*, *default=None*)
Removes the **key** from the Lib and returns the `list` of key members. If no key is found, **default** is returned. **key** is a *String*. This must return either **default** or a `list` of items as *String*.

```
>>> font.lib.pop("public.glyphOrder")
["A", "B", "C"]
```

BaseLib.**__iter__**()
Iterates through the Lib, giving the key for each iteration. The order that the Lib will iterate though is not fixed nor is it ordered.:

```
>>> for key in font.lib:
>>>     print key
"public.glyphOrder"
"org.robofab.scripts.SomeData"
"public.postscriptNames"
```

BaseLib.**update**(*otherLib*)
Updates the Lib based on **otherLib**. *otherLib** is a `dict` of keys. If a key from **otherLib** is in Lib the key members will be replaced by the key members from **otherLib**. If a key from **otherLib** is not in the Lib, it is added to the Lib. If Lib contain a key name that is not in *otherLib**, it is not changed.

```
>>> font.lib.update(newLib)
```

BaseLib.**clear**()
Removes all keys from Lib, resetting the Lib to an empty dictionary.

```
>>> font.lib.clear()
```

Environment

BaseLib.**naked**()
Return the environment's native object that has been wrapped by this object.

```
>>> loweLevelObj = obj.naked()
```

BaseLib.**changed**(**args*, ***kwargs*)
Tell the environment that something has changed in the object. The behavior of this method will vary from environment to environment.

```
>>> obj.changed()
```

2.1.7 Layer

Note: This section needs to contain the following:

- description of what this is
- sub-object with basic usage
- glyph interaction with basic usage

Overview

Copy

BaseLayer.copy	Copy the layer into a new layer that does not belong to a font.

Parents

BaseLayer.font	The layer's parent *BaseFont*.

Attributes

BaseLayer.name	The name of the layer.
BaseLayer.color	The layer's color.

Sub-Objects

BaseLayer.lib	The layer's *BaseLib* object.
BaseLayer.tempLib	The layer's *BaseLib* object.

Glyphs

BaseLayer.__len__	An int representing number of glyphs in the layer.
BaseLayer.keys	Get a list of all glyphs in the layer.
BaseLayer.__iter__	Iterate through the *BaseGlyph* objects in the layer.
BaseLayer.__contains__	Test if the layer contains a glyph with **name**.
BaseLayer.__getitem__	Get the *BaseGlyph* with name from the layer.
BaseLayer.newGlyph	Make a new glyph with **name** in the layer.
BaseLayer.insertGlyph	Insert **glyph** into the layer.
BaseLayer.removeGlyph	Remove the glyph with name from the layer.

Interpolation

BaseLayer.isCompatible	Evaluate interpolation compatibility with **other**.
BaseLayer.interpolate	Interpolate all possible data in the layer.

Normalization

BaseLayer.round	Round all approriate data to integers.
BaseLayer.autoUnicodes	Use heuristics to set Unicode values in all glyphs.

Environment

BaseLayer.naked	Return the environment's native object that has been wrapped by this object.
BaseLayer.changed	Tell the environment that something has changed in the object.

Reference

class fontParts.base.**BaseLayer**(**args*, ***kwargs*)

Copy

BaseLayer.**copy**()

Copy the layer into a new layer that does not belong to a font.

```
>>> copiedLayer = layer.copy()
```

This will copy:

- name
- color
- lib
- glyphs

Parents

BaseLayer.**font**

The layer's parent *BaseFont*.

```
>>> font = layer.font
```

Attributes

BaseLayer.**name**

The name of the layer.

```
>>> layer.name
"foreground"
>>> layer.name = "top"
```

`BaseLayer.`**`color`**

The layer's color.

```
>>> layer.color
None
>>> layer.color = (1, 0, 0, 0.5)
```

Sub-Objects

`BaseLayer.`**`lib`**

The layer's *`BaseLib`* object.

```
>>> layer.lib["org.robofab.hello"]
"world"
```

Glyphs

`BaseLayer.`**`__len__`**`()`

An `int` representing number of glyphs in the layer.

```
>>> len(layer)
256
```

`BaseLayer.`**`keys`**`()`

Get a list of all glyphs in the layer.

```
>>> layer.keys()
["B", "C", "A"]
```

The order of the glyphs is undefined.

`BaseLayer.`**`__iter__`**`()`

Iterate through the *`BaseGlyph`* objects in the layer.

```
>>> for glyph in layer:
...     glyph.name
"A"
"B"
"C"
```

`BaseLayer.`**`__contains__`**`(`*name*`)`

Test if the layer contains a glyph with **name**.

```
>>> "A" in layer
True
```

`BaseLayer.`**`__getitem__`**`(`*name*`)`

Get the *`BaseGlyph`* with name from the layer.

```
>>> glyph = layer["A"]
```

BaseLayer.**newGlyph**(*name*, *clear=True*)

Make a new glyph with **name** in the layer.

```
>>> glyph = layer.newGlyph("A")
```

The newly created *BaseGlyph* will be returned.

If the glyph exists in the layer and clear is set to `False`, the existing glyph will be returned, otherwise the default behavior is to clear the exisiting glyph.

BaseLayer.**insertGlyph**(*glyph*, *name=None*)

Insert **glyph** into the layer.

```
>>> glyph = layer.insertGlyph(otherGlyph, name="A")
```

This method is deprecated. `BaseFont.__setitem__` instead.

BaseLayer.**removeGlyph**(*name*)

Remove the glyph with name from the layer.

```
>>> layer.removeGlyph("A")
```

This method is deprecated. `BaseFont.__delitem__` instead.

Interpolation

BaseLayer.**isCompatible**(*other*)

Evaluate interpolation compatibility with **other**.

```
>>> compat, report = self.isCompatible(otherLayer)
>>> compat
False
>>> report
A
-
[Fatal] The glyphs do not contain the same number of contours.
```

This will return a `bool` indicating if the layer is compatible for interpolation with **other** and a *String* of compatibility notes.

BaseLayer.**interpolate**(*factor*, *minLayer*, *maxLayer*, *round=True*, *suppressError=True*)

Interpolate all possible data in the layer.

```
>>> layer.interpolate(0.5, otherLayer1, otherLayer2)
>>> layer.interpolate((0.5, 2.0), otherLayer1, otherLayer2, round=False)
```

The interpolation occurs on a 0 to 1.0 range where **minLayer** is located at 0 and **maxLayer** is located at 1.0. **factor** is the interpolation value. It may be less than 0 and greater than 1.0. It may be a *Integer/Float* or a tuple of two *Integer/Float*. If it is a tuple, the first number indicates the x factor and the second number indicates the y factor. **round** indicates if the result should be rounded to integers. **suppressError** indicates if incompatible data should be ignored or if an error should be raised when such incompatibilities are found.

Normalization

`BaseLayer.`**`round`**`()`

Round all approriate data to integers.

```
>>> layer.round()
```

This is the equivalent of calling the round method on:

- all glyphs in the layer

`BaseLayer.`**`autoUnicodes`**`()`

Use heuristics to set Unicode values in all glyphs.

```
>>> layer.autoUnicodes()
```

Environments will define their own heuristics for automatically determining values.

Environment

`BaseLayer.`**`naked`**`()`

Return the environment's native object that has been wrapped by this object.

```
>>> loweLevelObj = obj.naked()
```

`BaseLayer.`**`changed`**`(*args, **kwargs)`

Tell the environment that something has changed in the object. The behavior of this method will vary from environment to environment.

```
>>> obj.changed()
```

2.1.8 Glyph

`BaseGlyph`(*args, **kwargs)	A glyph object.
`BaseGlyph.addImage`([path, data, scale, . . .])	Set the image in the glyph.
`BaseGlyph.anchors`	An *Immutable List* of all anchors in the glyph.
`BaseGlyph.appendAnchor`([name, position, . . .])	Append an anchor to this glyph.
`BaseGlyph.appendComponent`([baseGlyph, . . .])	Append a component to this glyph.
`BaseGlyph.appendContour`(contour[, offset])	Append a contour containing the same data as `contour` to this glyph.
`BaseGlyph.appendGlyph`(other[, offset])	Append the data from `other` to new objects in this glyph.
`BaseGlyph.appendGuideline`([position, angle, . . .])	Append a guideline to this glyph.
`BaseGlyph.area`	The area of the glyph as a *Integer/Float* or, in the case of empty glyphs `None`.
`BaseGlyph.autoContourOrder()`	Automatically order the contours based on heuristics.
`BaseGlyph.autoUnicodes`()	Use heuristics to set the Unicode values in the glyph.
`BaseGlyph.bottomMargin`	The glyph's bottom margin.
`BaseGlyph.bounds`	The bounds of the glyph in the form `(x minimum, y minimum, x maximum, y maximum)` or, in the case of empty glyphs `None`.

Continued on next page

Table 25 – continued from previous page

BaseGlyph.box	Deprecated Glyph.box
BaseGlyph.center([padding])	
BaseGlyph.changed(*args, **kwargs)	Tell the environment that something has changed in the object.
BaseGlyph.clear([contours, components, . . .])	Clear the glyph.
BaseGlyph.clearAnchors()	Clear all anchors in the glyph.
BaseGlyph.clearComponents()	Clear all components in the glyph.
BaseGlyph.clearContours()	Clear all contours in the glyph.
BaseGlyph.clearGuidelines()	Clear all guidelines in the glyph.
BaseGlyph.clearHGuides()	
BaseGlyph.clearImage()	Remove the image from the glyph.
BaseGlyph.clearVGuides()	
BaseGlyph.compatibilityReporterClass	alias of fontParts.base.compatibility.GlyphCompatibilityReporter
BaseGlyph.components	An *Immutable List* of all components in the glyph.
BaseGlyph.contours	An *Immutable List* of all contours in the glyph.
BaseGlyph.copy()	Copy this glyph's data into a new glyph object.
BaseGlyph.copyAttributes	
BaseGlyph.copyClass	
BaseGlyph.copyData(source)	Subclasses may override this method.
BaseGlyph.correctDirection([trueType])	Correct the winding direction of the contours following the PostScript recommendations.
BaseGlyph.decompose()	Decompose all components in the glyph to contours.
BaseGlyph.draw(pen[, contours, components])	Draw the glyph's outline data (contours and components) to the given type-pen.
BaseGlyph.drawPoints(pen[, contours, components])	Draw the glyph's outline data (contours and components) to the given type-pointpen.
BaseGlyph.dumpToGLIF([glyphFormatVersion])	This will return the glyph's contents as a string in GLIF format.
BaseGlyph.font	The glyph's parent font.
BaseGlyph.fromMathGlyph(mathGlyph[, . . .])	Replaces the contents of this glyph with the contents of mathGlyph.
BaseGlyph.getAnchors()	
BaseGlyph.getComponents()	
BaseGlyph.getLayer(name)	Get the type-glyph-layer with name in this glyph.
BaseGlyph.getParent()	
BaseGlyph.getPen()	Return a type-pen object for adding outline data to the glyph.
BaseGlyph.getPointPen()	Return a type-pointpen object for adding outline data to the glyph.
BaseGlyph.guidelines	An *Immutable List* of all guidelines in the glyph.
BaseGlyph.height	The glyph's height.
BaseGlyph.image	The *BaseImage* for the glyph.
BaseGlyph.interpolate(factor, minGlyph, maxGlyph)	Interpolate the contents of this glyph at location factor in a linear interpolation between minGlyph and maxGlyph.
BaseGlyph.isCompatible(other)	Evaluate the interpolation compatibility of this glyph and other.
BaseGlyph.isEmpty()	This will return type-bool indicating if there are contours and/or components in the glyph.
BaseGlyph.layer	The glyph's parent layer.

Continued on next page

Table 25 – continued from previous page

BaseGlyph.layers	Immutable tuple of the glyph's layers.
BaseGlyph.leftMargin	The glyph's left margin.
BaseGlyph.lib	The *BaseLib* for the glyph.
BaseGlyph.loadFromGLIF(glifData)	Reads glifData, in GLIF format, into this glyph.
BaseGlyph.mark	Deprecated Mark color
BaseGlyph.markColor	The glyph's mark color.
BaseGlyph.move(*args, **kwargs)	
BaseGlyph.moveBy(value)	Move the object.
BaseGlyph.naked()	Return the environment's native object that has been wrapped by this object.
BaseGlyph.name	The glyph's name.
BaseGlyph.newLayer(name)	Make a new layer with name in this glyph.
BaseGlyph.note	The glyph's note.
BaseGlyph.pointInside(point)	Determine if point is in the black or white of the glyph.
BaseGlyph.raiseNotImplementedError()	This exception needs to be raised frequently by the base classes.
BaseGlyph.readGlyphFromString(glifData)	
BaseGlyph.removeAnchor(anchor)	Remove anchor from the glyph.
BaseGlyph.removeComponent(component)	Remove component from the glyph.
BaseGlyph.removeContour(contour)	Remove contour from the glyph.
BaseGlyph.removeGuideline(guideline)	Remove guideline from the glyph.
BaseGlyph.removeLayer(layer)	Remove layer from this glyph.
BaseGlyph.removeOverlap()	Perform a remove overlap operation on the contours.
BaseGlyph.rightMargin	The glyph's right margin.
BaseGlyph.rotate(*args, **kwargs)	
BaseGlyph.rotateBy(value[, origin])	Rotate the object.
BaseGlyph.round()	Round coordinates to the nearest integer.
BaseGlyph.scale(*args, **kwargs)	
BaseGlyph.scaleBy(value[, origin, width, height])	Scale the object.
BaseGlyph.selected	The object's selection state.
BaseGlyph.selectedAnchors	An *Immutable List* of anchors selected in the glyph.
BaseGlyph.selectedComponents	An *Immutable List* of components selected in the glyph.
BaseGlyph.selectedContours	An *Immutable List* of contours selected in the glyph.
BaseGlyph.selectedGuidelines	An *Immutable List* of guidelines selected in the glyph.
BaseGlyph.setChanged()	
BaseGlyph.setParent(parent)	
BaseGlyph.skew(*args, **kwargs)	
BaseGlyph.skewBy(value[, origin])	Skew the object.
BaseGlyph.tempLib	The *BaseLib* for the glyph.
BaseGlyph.toMathGlyph([...])	Returns the glyph as an object that follows the MathGlyph protocol.
BaseGlyph.topMargin	The glyph's top margin.
BaseGlyph.transform(*args, **kwargs)	
BaseGlyph.transformBy(matrix[, origin])	Transform the object.
BaseGlyph.translate(*args, **kwargs)	
BaseGlyph.unicode	The glyph's primary unicode value.
BaseGlyph.unicodes	The glyph's unicode values in order from most to least important.
BaseGlyph.update()	

Continued on next page

Table 25 – continued from previous page

`BaseGlyph.width`	The glyph’s width.
`BaseGlyph.writeGlyphToString([...])`	

Description

The `Glyph` object represents a glyph, its parts and associated data.

`Glyph` can be used as a list of `Contour` objects.

When a `Glyph` is obtained from a `Font` object, the font is the parent object of the glyph.

Overview

Copy

`BaseGlyph.copy`	Copy this glyph’s data into a new glyph object.

Parents

`BaseGlyph.layer`	The glyph’s parent layer.
`BaseGlyph.font`	The glyph’s parent font.

Identification

`BaseGlyph.name`	The glyph’s name.
`BaseGlyph.unicodes`	The glyph’s unicode values in order from most to least important.
`BaseGlyph.unicode`	The glyph’s primary unicode value.

Metrics

`BaseGlyph.width`	The glyph’s width.
`BaseGlyph.leftMargin`	The glyph’s left margin.
`BaseGlyph.rightMargin`	The glyph’s right margin.
`BaseGlyph.height`	The glyph’s height.
`BaseGlyph.bottomMargin`	The glyph’s bottom margin.
`BaseGlyph.topMargin`	The glyph’s top margin.

Queries

`BaseGlyph.bounds`	The bounds of the glyph in the form `(x minimum, y minimum, x maximum, y maximum)` or, in the case of empty glyphs `None`.
`BaseGlyph.pointInside`	Determine if `point` is in the black or white of the glyph.

Pens and Drawing

BaseGlyph.getPen	Return a type-pen object for adding outline data to the glyph.
BaseGlyph.getPointPen	Return a type-pointpen object for adding outline data to the glyph.
BaseGlyph.draw	Draw the glyph's outline data (contours and components) to the given type-pen.
BaseGlyph.drawPoints	Draw the glyph's outline data (contours and components) to the given type-pointpen.

Layers

BaseGlyph.layers	Immutable tuple of the glyph's layers.
BaseGlyph.getLayer	Get the type-glyph-layer with `name` in this glyph.
BaseGlyph.newLayer	Make a new layer with `name` in this glyph.
BaseGlyph.removeLayer	Remove `layer` from this glyph.

Global

BaseGlyph.clear	Clear the glyph.
BaseGlyph.appendGlyph	Append the data from `other` to new objects in this glyph.

Contours

BaseGlyph.contours	An *Immutable List* of all contours in the glyph.
BaseGlyph.__len__	The number of contours in the glyph.
BaseGlyph.__iter__	Iterate through all contours in the glyph.
BaseGlyph.__getitem__	Get the contour located at `index` from the glyph.
BaseGlyph.appendContour	Append a contour containing the same data as `contour` to this glyph.
BaseGlyph.removeContour	Remove `contour` from the glyph.
BaseGlyph.clearContours	Clear all contours in the glyph.
BaseGlyph.removeOverlap	Perform a remove overlap operation on the contours.

Components

BaseGlyph.components	An *Immutable List* of all components in the glyph.
BaseGlyph.appendComponent	Append a component to this glyph.
BaseGlyph.removeComponent	Remove `component` from the glyph.
BaseGlyph.clearComponents	Clear all components in the glyph.
BaseGlyph.decompose	Decompose all components in the glyph to contours.

Anchors

BaseGlyph.anchors	An *Immutable List* of all anchors in the glyph.
BaseGlyph.appendAnchor	Append an anchor to this glyph.
BaseGlyph.removeAnchor	Remove `anchor` from the glyph.
BaseGlyph.clearAnchors	Clear all anchors in the glyph.

Guidelines

BaseGlyph.guidelines	An *Immutable List* of all guidelines in the glyph.
BaseGlyph.appendGuideline	Append a guideline to this glyph.
BaseGlyph.removeGuideline	Remove `guideline` from the glyph.
BaseGlyph.clearGuidelines	Clear all guidelines in the glyph.

Image

BaseGlyph.image	The *BaseImage* for the glyph.
BaseGlyph.addImage	Set the image in the glyph.
BaseGlyph.clearImage	Remove the image from the glyph.

Note

BaseGlyph.note	The glyph's note.
BaseGlyph.markColor	The glyph's mark color.

Sub-Objects

BaseGlyph.lib	The *BaseLib* for the glyph.
`BaseGlyph.tempLib`	The *BaseLib* for the glyph.

Transformations

BaseGlyph.transformBy	Transform the object.
BaseGlyph.moveBy	Move the object.
BaseGlyph.scaleBy	Scale the object.
BaseGlyph.rotateBy	Rotate the object.
BaseGlyph.skewBy	Skew the object.

Interpolation

BaseGlyph.isCompatible	Evaluate the interpolation compatibility of this glyph and `other`.

Continued on next page

Table 42 – continued from previous page

BaseGlyph.interpolate	Interpolate the contents of this glyph at location `factor` in a linear interpolation between `minGlyph` and `maxGlyph`.

Normalization

BaseGlyph.round	Round coordinates to the nearest integer.
BaseGlyph.autoUnicodes	Use heuristics to set the Unicode values in the glyph.

Environment

BaseGlyph.naked	Return the environment's native object that has been wrapped by this object.
BaseGlyph.changed	Tell the environment that something has changed in the object.

Reference

class `fontParts.base.`**`BaseGlyph`**(**args*, ***kwargs*)

A glyph object. This object will almost always be created by retrieving it from a font object.

Copy

`BaseGlyph.`**`copy`**()

Copy this glyph's data into a new glyph object. This new glyph object will not belong to a font.

```
>>> copiedGlyph = glyph.copy()
```

This will copy:

- name
- unicodes
- width
- height
- note
- markColor
- lib
- contours
- components
- anchors
- guidelines
- image

Parents

BaseGlyph.**layer**

The glyph's parent layer.

```
>>> layer = glyph.layer
```

BaseGlyph.**font**

The glyph's parent font.

```
>>> font = glyph.font
```

Identification

BaseGlyph.**name**

The glyph's name. This will be a *String*.

```
>>> glyph.name
"A"
>>> glyph.name = "A.alt"
```

BaseGlyph.**unicodes**

The glyph's unicode values in order from most to least important.

```
>>> glyph.unicodes
(65,)
>>> glyph.unicodes = [65, 66]
>>> glyph.unicodes = []
```

The values in the returned tuple will be type-int. When setting you may use a list of type-int or type-hex values.

BaseGlyph.**unicode**

The glyph's primary unicode value.

```
>>> glyph.unicode
65
>>> glyph.unicode = None
```

This is equivalent to `glyph.unicodes[0]`. Setting a `glyph.unicode` value will reset `glyph.unicodes` to a tuple containing that value or an empty tuple if `value` is `None`.

```
>>> glyph.unicodes
(65, 67)
>>> glyph.unicode = 65
>>> glyph.unicodes
(65,)
>>> glyph.unicode = None
>>> glyph.unicodes
()
```

The returned value will be an type-int or `None`. When setting you may send type-int or type-hex values or `None`.

Metrics

`BaseGlyph.`**`width`**

The glyph's width.

```
>>> glyph.width
500
>>> glyph.width = 200
```

The value will be a *Integer/Float*.

`BaseGlyph.`**`leftMargin`**

The glyph's left margin.

```
>>> glyph.leftMargin
35
>>> glyph.leftMargin = 45
```

The value will be a *Integer/Float* or *None* if the glyph has no outlines.

`BaseGlyph.`**`rightMargin`**

The glyph's right margin.

```
>>> glyph.rightMargin
35
>>> glyph.rightMargin = 45
```

The value will be a *Integer/Float* or *None* if the glyph has no outlines.

`BaseGlyph.`**`height`**

The glyph's height.

```
>>> glyph.height
500
>>> glyph.height = 200
```

The value will be a *Integer/Float*.

`BaseGlyph.`**`bottomMargin`**

The glyph's bottom margin.

```
>>> glyph.bottomMargin
35
>>> glyph.bottomMargin = 45
```

The value will be a *Integer/Float* or *None* if the glyph has no outlines.

`BaseGlyph.`**`topMargin`**

The glyph's top margin.

```
>>> glyph.topMargin
35
>>> glyph.topMargin = 45
```

The value will be a *Integer/Float* or *None* if the glyph has no outlines.

Queries

`BaseGlyph.`**`bounds`**

The bounds of the glyph in the form `(x minimum, y minimum, x maximum, y maximum)` or, in the case of empty glyphs `None`.

```
>>> glyph.bounds
(10, 30, 765, 643)
```

`BaseGlyph.`**`pointInside`**`(`*point*`)`

Determine if `point` is in the black or white of the glyph.

```
>>> glyph.pointInside((40, 65))
True
```

`point` must be a *Coordinate*.

Pens and Drawing

`BaseGlyph.`**`getPen`**`()`

Return a type-pen object for adding outline data to the glyph.

```
>>> pen = glyph.getPen()
```

`BaseGlyph.`**`getPointPen`**`()`

Return a type-pointpen object for adding outline data to the glyph.

```
>>> pointPen = glyph.getPointPen()
```

`BaseGlyph.`**`draw`**`(`*pen*, *contours=True*, *components=True*`)`

Draw the glyph's outline data (contours and components) to the given type-pen.

```
>>> glyph.draw(pen)
```

If `contours` is set to `False`, the glyph's contours will not be drawn.

```
>>> glyph.draw(pen, contours=False)
```

If `components` is set to `False`, the glyph's components will not be drawn.

```
>>> glyph.draw(pen, components=False)
```

`BaseGlyph.`**`drawPoints`**`(`*pen*, *contours=True*, *components=True*`)`

Draw the glyph's outline data (contours and components) to the given type-pointpen.

```
>>> glyph.drawPoints(pointPen)
```

If `contours` is set to `False`, the glyph's contours will not be drawn.

```
>>> glyph.drawPoints(pointPen, contours=False)
```

If `components` is set to `False`, the glyph's components will not be drawn.

```
>>> glyph.drawPoints(pointPen, components=False)
```

Layers

Layer interaction in glyphs is very similar to the layer interaction in fonts. When you ask a glyph for a layer, you get a *glyph layer* in return. A glyph layer lets you do anything that you can do to a glyph. In fact a glyph layer is really just a glyph.

```
>>> bgdGlyph = glyph.newLayer('background')
>>> bgdGlyph.appendGlyph(glyph)
>>> bgdGlyph.appendGuideline((10, 10), 45)
```

`BaseGlyph.`**`layers`**

Immutable tuple of the glyph's layers.

```
>>> glyphLayers = glyph.layers
```

This will return a tuple of all type-glyph-layer in the glyph.

`BaseGlyph.`**`getLayer`**(*name*)

Get the type-glyph-layer with `name` in this glyph.

```
>>> glyphLayer = glyph.getLayer("foreground")
```

`BaseGlyph.`**`newLayer`**(*name*)

Make a new layer with `name` in this glyph.

```
>>> glyphLayer = glyph.newLayer("background")
```

This will return the new type-glyph-layer. If the layer already exists in this glyph, it will be cleared.

`BaseGlyph.`**`removeLayer`**(*layer*)

Remove `layer` from this glyph.

```
>>> glyph.removeLayer("background")
```

Layer can be a type-glyph-layer or a *String* representing a layer name.

Global

`BaseGlyph.`**`clear`**(*contours=True*, *components=True*, *anchors=True*, *guidelines=True*, *image=True*)

Clear the glyph.

```
>>> glyph.clear()
```

This clears:

- contours
- components
- anchors
- guidelines
- image

It's possible to turn off the clearing of portions of the glyph with the listed arguments.

```
>>> glyph.clear(guidelines=False)
```

`BaseGlyph.`**`appendGlyph`**`(`*other*, *offset=None*`)`

Append the data from `other` to new objects in this glyph.

```
>>> glyph.appendGlyph(otherGlyph)
```

This will append:

- contours
- components
- anchors
- guidelines

`offset` indicates the x and y shift values that should be applied to the appended data. It must be a *Coordinate* value or `None`. If `None` is given, the offset will be `(0, 0)`.

```
>>> glyph.appendGlyph(otherGlyph, (100, 0))
```

Contours

`BaseGlyph.`**`contours`**

An *Immutable List* of all contours in the glyph.

```
>>> contours = glyph.contours
```

The list will contain `BaseContour` objects.

`BaseGlyph.`**`__len__`**`()`

The number of contours in the glyph.

```
>>> len(glyph)
2
```

`BaseGlyph.`**`__iter__`**`()`

Iterate through all contours in the glyph.

```
>>> for contour in glyph:
...     contour.reverse()
```

`BaseGlyph.`**`__getitem__`**`(`*index*`)`

Get the contour located at `index` from the glyph.

```
>>> contour = glyph[0]
```

The returned value will be a `BaseContour` object.

`BaseGlyph.`**`appendContour`**`(`*contour*, *offset=None*`)`

Append a contour containing the same data as `contour` to this glyph.

```
>>> contour = glyph.appendContour(contour)
```

This will return a `BaseContour` object representing the new contour in the glyph. `offset` indicates the x and y shift values that should be applied to the appended data. It must be a *Coordinate* value or `None`. If `None` is given, the offset will be `(0, 0)`.

```
>>> contour = glyph.appendContour(contour, (100, 0))
```

BaseGlyph.**removeContour**(*contour*)

Remove `contour` from the glyph.

```
>>> glyph.removeContour(contour)
```

`contour` may be a BaseContour or an type-int representing a contour index.

BaseGlyph.**clearContours**()

Clear all contours in the glyph.

```
>>> glyph.clearContours()
```

BaseGlyph.**removeOverlap**()

Perform a remove overlap operation on the contours.

```
>>> glyph.removeOverlap()
```

The behavior of this may vary across environments.

Components

BaseGlyph.**components**

An *Immutable List* of all components in the glyph.

```
>>> components = glyph.components
```

The list will contain *`BaseComponent`* objects.

BaseGlyph.**appendComponent**(*baseGlyph=None*, *offset=None*, *scale=None*, *component=None*)

Append a component to this glyph.

```
>>> component = glyph.appendComponent("A")
```

This will return a *`BaseComponent`* object representing the new component in the glyph. `offset` indicates the x and y shift values that should be applied to the appended component. It must be a *Coordinate* value or `None`. If `None` is given, the offset will be `(0, 0)`.

```
>>> component = glyph.appendComponent("A", offset=(10, 20))
```

`scale` indicates the x and y scale values that should be applied to the appended component. It must be a type-scale value or `None`. If `None` is given, the scale will be `(1.0, 1.0)`.

```
>>> component = glyph.appendComponent("A", scale=(1.0, 2.0))
```

`component` may be a *`BaseComponent`* object from which attribute values will be copied. If `baseGlyph`, `offset` or `scale` are specified as arguments, those values will be used instead of the values in the given component object.

BaseGlyph.**removeComponent**(*component*)

Remove `component` from the glyph.

```
>>> glyph.removeComponent(component)
```

`component` may be a BaseComponent or an type-int representing a component index.

`BaseGlyph.`**`clearComponents`**`()`

Clear all components in the glyph.

```
>>> glyph.clearComponents()
```

`BaseGlyph.`**`decompose`**`()`

Decompose all components in the glyph to contours.

```
>>> glyph.decompose()
```

Anchors

`BaseGlyph.`**`anchors`**

An *Immutable List* of all anchors in the glyph.

```
>>> anchors = glyph.anchors
```

The list will contain *`BaseAnchor`* objects.

`BaseGlyph.`**`appendAnchor`**`(`*name=None*, *position=None*, *color=None*, *anchor=None*`)`

Append an anchor to this glyph.

```
>>> anchor = glyph.appendAnchor("top", (10, 20))
```

This will return a *`BaseAnchor`* object representing the new anchor in the glyph. `name` indicated the name to be assigned to the anchor. It must be a *String* or `None`. `position` indicates the x and y location to be applied to the anchor. It must be a *Coordinate* value. `color` indicates the color to be applied to the anchor. It must be a *Color* or `None`.

```
>>> anchor = glyph.appendAnchor("top", (10, 20), color=(1, 0, 0, 1))
```

`anchor` may be a *`BaseAnchor`* object from which attribute values will be copied. If `name`, `position` or `color` are specified as arguments, those values will be used instead of the values in the given anchor object.

`BaseGlyph.`**`removeAnchor`**`(`*anchor*`)`

Remove `anchor` from the glyph.

```
>>> glyph.removeAnchor(anchor)
```

`anchor` may be an BaseAnchor or an type-int representing an anchor index.

`BaseGlyph.`**`clearAnchors`**`()`

Clear all anchors in the glyph.

```
>>> glyph.clearAnchors()
```

Guidelines

`BaseGlyph.`**`guidelines`**

An *Immutable List* of all guidelines in the glyph.

```
>>> guidelines = glyph.guidelines
```

The list will contain *`BaseGuideline`* objects.

BaseGlyph.**appendGuideline**(*position=None*, *angle=None*, *name=None*, *color=None*, *guideline=None*)

Append a guideline to this glyph.

```
>>> guideline = glyph.appendGuideline((100, 0), 90)
```

This will return a *BaseGuideline* object representing the new guideline in the glyph. `position` indicates the x and y location to be used as the center point of the anchor. It must be a *Coordinate* value. `angle` indicates the angle of the guideline, in degrees. This must be a *Integer/Float* between 0 and 360. `name` indicates an name to be assigned to the guideline. It must be a *String* or `None`.

```
>>> guideline = glyph.appendGuideline((100, 0), 90, name="left")
```

`color` indicates the color to be applied to the guideline. It must be a *Color* or `None`.

```
>>> guideline = glyph.appendGuideline((100, 0), 90, color=(1, 0, 0, 1))
```

`guideline` may be a *BaseGuideline* object from which attribute values will be copied. If `position`, `angle`, `name` or `color` are specified as arguments, those values will be used instead of the values in the given guideline object.

BaseGlyph.**removeGuideline**(*guideline*)

Remove `guideline` from the glyph.

```
>>> glyph.removeGuideline(guideline)
```

`guideline` may be a BaseGuideline or an type-int representing an guideline index.

BaseGlyph.**clearGuidelines**()

Clear all guidelines in the glyph.

```
>>> glyph.clearGuidelines()
```

Image

BaseGlyph.**image**

The *BaseImage* for the glyph.

BaseGlyph.**addImage**(*path=None*, *data=None*, *scale=None*, *position=None*, *color=None*)

Set the image in the glyph. This will return the assigned *BaseImage*. The image data can be defined via `path` to an image file:

```
>>> image = glyph.addImage(path="/path/to/my/image.png")
```

The image data can be defined with raw image data via `data`.

```
>>> image = glyph.addImage(data=someImageData)
```

If `path` and `data` are both provided, a `FontPartsError` will be raised. The supported image formats will vary across environments. Refer to *BaseImage* for complete details.

`scale` indicates the x and y scale values that should be applied to the image. It must be a type-scale value or `None`.

```
>>> image = glyph.addImage(path="/p/t/image.png", scale=(0.5, 1.0))
```

`position` indicates the x and y location of the lower left point of the image.

```
>>> image = glyph.addImage(path="/p/t/image.png", position=(10, 20))
```

`color` indicates the color to be applied to the image. It must be a *Color* or `None`.

```
>>> image = glyph.addImage(path="/p/t/image.png", color=(1, 0, 0, 0.5))
```

`BaseGlyph.`**`clearImage`**`()`

Remove the image from the glyph.

```
>>> glyph.clearImage()
```

Note

`BaseGlyph.`**`note`**

The glyph's note.

```
>>> glyph.note
"P.B. said this looks 'awesome.'"
>>> glyph.note = "P.B. said this looks 'AWESOME.'"
```

The value may be a *String* or `None`.

`BaseGlyph.`**`markColor`**

The glyph's mark color.

```
>>> glyph.markColor
(1, 0, 0, 0.5)
>>> glyph.markColor = None
```

The value may be a *Color* or `None`.

Sub-Objects

`BaseGlyph.`**`lib`**

The `BaseLib` for the glyph.

```
>>> lib = glyph.lib
```

Transformations

`BaseGlyph.`**`transformBy`**`(`*matrix*, *origin=None*`)`

Transform the object.

```
>>> obj.transformBy((0.5, 0, 0, 2.0, 10, 0))
>>> obj.transformBy((0.5, 0, 0, 2.0, 10, 0), origin=(500, 500))
```

matrix must be a *Transformation Matrix*. **origin** defines the point at with the transformation should originate. It must be a *Coordinate* or `None`. The default is `(0, 0)`.

`BaseGlyph.`**`moveBy`**`(`*value*`)`

Move the object.

```
>>> obj.moveBy((10, 0))
```

value must be an iterable containing two *Integer/Float* values defining the x and y values to move the object by.

`BaseGlyph.`**`scaleBy`**(*value*, *origin=None*, *width=False*, *height=False*)

Scale the object.

```
>>> obj.scaleBy(2.0)
>>> obj.scaleBy((0.5, 2.0), origin=(500, 500))
```

value must be an iterable containing two *Integer/Float* values defining the x and y values to scale the object by. **origin** defines the point at with the scale should originate. It must be a *Coordinate* or `None`. The default is `(0, 0)`.

width indicates if the glyph's width should be scaled. **height** indicates if the glyph's height should be scaled.

The origin must not be specified when scaling the width or height.

`BaseGlyph.`**`rotateBy`**(*value*, *origin=None*)

Rotate the object.

```
>>> obj.rotateBy(45)
>>> obj.rotateBy(45, origin=(500, 500))
```

value must be a *Integer/Float* values defining the angle to rotate the object by. **origin** defines the point at with the rotation should originate. It must be a *Coordinate* or `None`. The default is `(0, 0)`.

`BaseGlyph.`**`skewBy`**(*value*, *origin=None*)

Skew the object.

```
>>> obj.skewBy(11)
>>> obj.skewBy((25, 10), origin=(500, 500))
```

value must be rone of the following:

- single *Integer/Float* indicating the value to skew the x direction by.
- iterable cointaining type *Integer/Float* defining the values to skew the x and y directions by.

origin defines the point at with the skew should originate. It must be a *Coordinate* or `None`. The default is `(0, 0)`.

Interpolation

`BaseGlyph.`**`isCompatible`**(*other*)

Evaluate the interpolation compatibility of this glyph and `other`.

```
>>> compatible, report = self.isCompatible(otherGlyph)
>>> compatible
False
```

This will return a type-bool indicating if this glyph is compatible with `other` and a `GlyphCompatibilityReporter` containing a detailed report about compatibility errors.

`BaseGlyph.`**`interpolate`**(*factor*, *minGlyph*, *maxGlyph*, *round=True*, *suppressError=True*)

Interpolate the contents of this glyph at location `factor` in a linear interpolation between `minGlyph` and `maxGlyph`.

```
>>> glyph.interpolate(0.5, otherGlyph1, otherGlyph2)
```

`factor` may be a *Integer/Float* or a tuple containing two *Integer/Float* values representing x and y factors.

```
>>> glyph.interpolate((0.5, 1.0), otherGlyph1, otherGlyph2)
```

`minGlyph` must be a *`BaseGlyph`* and will be located at 0.0 in the interpolation range. `maxGlyph` must be a *`BaseGlyph`* and will be located at 1.0 in the interpolation range. If `round` is `True`, the contents of the glyph will be rounded to integers after the interpolation is performed.

```
>>> glyph.interpolate(0.5, otherGlyph1, otherGlyph2, round=True)
```

This method assumes that `minGlyph` and `maxGlyph` are completely compatible with each other for interpolation. If not, any errors encountered will raise a `FontPartsError`. If `suppressError` is `True`, no exception will be raised and errors will be silently ignored.

Normalization

`BaseGlyph.`**`round`**`()`

Round coordinates to the nearest integer.

```
>>> glyph.round()
```

This applies to the following:

- width
- height
- contours
- components
- anchors
- guidelines

`BaseGlyph.`**`autoUnicodes`**`()`

Use heuristics to set the Unicode values in the glyph.

```
>>> glyph.autoUnicodes()
```

Environments will define their own heuristics for automatically determining values.

Environment

`BaseGlyph.`**`naked`**`()`

Return the environment's native object that has been wrapped by this object.

```
>>> loweLevelObj = obj.naked()
```

`BaseGlyph.`**`changed`**`(`*`*args`*, *`**kwargs`*`)`

Tell the environment that something has changed in the object. The behavior of this method will vary from environment to environment.

```
>>> obj.changed()
```

2.1.9 Contour

Description

A Contour is a single path of any number of points. A Glyph usually consists of a couple of contours, and this is the object that represents each one. The `Contour` object offers access to the outline matter in various ways. The parent of `Contour` is usually `Glyph`.

Overview

Copy

BaseContour.copy	Copy this object into a new object of the same type.

Parents

BaseContour.glyph	The contour's parent *BaseGlyph*.
BaseContour.layer	The contour's parent layer.
BaseContour.font	The contour's parent font.

Identification

BaseContour.identifier	The unique identifier for the object.
BaseContour.index	The index of the contour within the parent glyph's contours.

Winding Direction

BaseContour.clockwise	Boolean indicating if the contour's winding direction is clockwise.
BaseContour.reverse	Reverse the direction of the contour.

Queries

BaseContour.bounds	The bounds of the contour: (xMin, yMin, xMax, yMax) or None.
BaseContour.pointInside	Determine if `point` is in the black or white of the contour.

Pens and Drawing

BaseContour.draw	Draw the contour's outline data to the given type-pen.
BaseContour.drawPoints	Draw the contour's outline data to the given type-point-pen.

Segments

BaseContour.segments	
BaseContour.__len__	
BaseContour.__iter__	
BaseContour.__getitem__	
BaseContour.appendSegment	Append a segment to the contour.
BaseContour.insertSegment	Insert a segment into the contour.
BaseContour.removeSegment	Remove segment from the contour.
BaseContour.setStartSegment	Set the first segment on the contour.
BaseContour.autoStartSegment	Automatically calculate and set the first segment in this contour.

bPoints

BaseContour.bPoints	
BaseContour.appendBPoint	Append a bPoint to the contour.
BaseContour.insertBPoint	Insert a bPoint at index in the contour.

Points

BaseContour.points	
BaseContour.appendPoint	Append a point to the contour.
BaseContour.insertPoint	Insert a point into the contour.
BaseContour.removePoint	Remove the point from the contour.

Transformations

BaseContour.transformBy	Transform the object.
BaseContour.moveBy	Move the object.
BaseContour.scaleBy	Scale the object.
BaseContour.rotateBy	Rotate the object.
BaseContour.skewBy	Skew the object.

Normalization

BaseContour.round	Round coordinates in all points to integers.

Environment

BaseContour.naked	Return the environment's native object that has been wrapped by this object.
BaseContour.changed	Tell the environment that something has changed in the object.

Reference

class fontParts.base.**BaseContour**(**args*, ***kwargs*)

Copy

BaseContour.**copy**()

Copy this object into a new object of the same type. The returned object will not have a parent object.

Parents

BaseContour.**glyph**

The contour's parent *BaseGlyph*.

BaseContour.**layer**

The contour's parent layer.

BaseContour.**font**

The contour's parent font.

Identification

BaseContour.**identifier**

The unique identifier for the object. This value will be an *Identifier* or a `None`. This attribute is read only.

```
>>> object.identifier
'ILHGJlygfds'
```

To request an identifier if it does not exist use *object.getIdentifier()*

BaseContour.**index**

The index of the contour within the parent glyph's contours.

```
>>> contour.index
1
>>> contour.index = 0
```

The value will always be a type-int.

Winding Direction

BaseContour.**clockwise**

Boolean indicating if the contour's winding direction is clockwise.

BaseContour.**reverse**()

Reverse the direction of the contour.

Queries

`BaseContour.`**`bounds`**
The bounds of the contour: (xMin, yMin, xMax, yMax) or None.

`BaseContour.`**`pointInside`**`(`*point*`)`
Determine if `point` is in the black or white of the contour.

```
>>> contour.pointInside((40, 65))
True
```

`point` must be a *Coordinate*.

Pens and Drawing

`BaseContour.`**`draw`**`(`*pen*`)`
Draw the contour's outline data to the given type-pen.

```
>>> contour.draw(pen)
```

`BaseContour.`**`drawPoints`**`(`*pen*`)`
Draw the contour's outline data to the given type-point-pen.

```
>>> contour.drawPoints(pointPen)
```

Segments

`BaseContour.`**`segments`**

`BaseContour.`**`__len__`**`()`

`BaseContour.`**`__iter__`**`()`

`BaseContour.`**`__getitem__`**`(`*index*`)`

`BaseContour.`**`appendSegment`**`(`*type=None*, *points=None*, *smooth=False*, *segment=None*`)`
Append a segment to the contour.

`BaseContour.`**`insertSegment`**`(`*index*, *type=None*, *points=None*, *smooth=False*, *segment=None*`)`
Insert a segment into the contour.

`BaseContour.`**`removeSegment`**`(`*segment*, *preserveCurve=False*`)`
Remove segment from the contour. If `preserveCurve` is set to `True` an attempt will be made to preserve the shape of the curve if the environment supports that functionality.

`BaseContour.`**`setStartSegment`**`(`*segment*`)`
Set the first segment on the contour. segment can be a segment object or an index.

`BaseContour.`**`autoStartSegment`**`()`
Automatically calculate and set the first segment in this contour.

The behavior of this may vary accross environments.

bPoints

`BaseContour.`**`bPoints`**

`BaseContour.`**`appendBPoint`**(*type=None*, *anchor=None*, *bcpIn=None*, *bcpOut=None*, *bPoint=None*)
Append a bPoint to the contour.

`BaseContour.`**`insertBPoint`**(*index*, *type=None*, *anchor=None*, *bcpIn=None*, *bcpOut=None*, *bPoint=None*)
Insert a bPoint at index in the contour.

Points

`BaseContour.`**`points`**

`BaseContour.`**`appendPoint`**(*position=None*, *type='line'*, *smooth=False*, *name=None*, *identifier=None*, *point=None*)
Append a point to the contour.

`BaseContour.`**`insertPoint`**(*index*, *position=None*, *type='line'*, *smooth=False*, *name=None*, *identifier=None*, *point=None*)
Insert a point into the contour.

`BaseContour.`**`removePoint`**(*point*, *preserveCurve=False*)
Remove the point from the contour. point can be a point object or an index. If `preserveCurve` is set to `True` an attempt will be made to preserve the shape of the curve if the environment supports that functionality.

Transformations

`BaseContour.`**`transformBy`**(*matrix*, *origin=None*)
Transform the object.

```
>>> obj.transformBy((0.5, 0, 0, 2.0, 10, 0))
>>> obj.transformBy((0.5, 0, 0, 2.0, 10, 0), origin=(500, 500))
```

matrix must be a *Transformation Matrix*. **origin** defines the point at with the transformation should originate. It must be a *Coordinate* or `None`. The default is `(0, 0)`.

`BaseContour.`**`moveBy`**(*value*)
Move the object.

```
>>> obj.moveBy((10, 0))
```

value must be an iterable containing two *Integer/Float* values defining the x and y values to move the object by.

`BaseContour.`**`scaleBy`**(*value*, *origin=None*)
Scale the object.

```
>>> obj.scaleBy(2.0)
>>> obj.scaleBy((0.5, 2.0), origin=(500, 500))
```

value must be an iterable containing two *Integer/Float* values defining the x and y values to scale the object by. **origin** defines the point at with the scale should originate. It must be a *Coordinate* or `None`. The default is `(0, 0)`.

`BaseContour.`**`rotateBy`**(*value*, *origin=None*)
Rotate the object.

```
>>> obj.rotateBy(45)
>>> obj.rotateBy(45, origin=(500, 500))
```

value must be a *Integer/Float* values defining the angle to rotate the object by. **origin** defines the point at with the rotation should originate. It must be a *Coordinate* or `None`. The default is `(0, 0)`.

`BaseContour.`**`skewBy`**`(`*value*, *origin=None*`)`

Skew the object.

```
>>> obj.skewBy(11)
>>> obj.skewBy((25, 10), origin=(500, 500))
```

value must be rone of the following:

- single *Integer/Float* indicating the value to skew the x direction by.
- iterable cointaining type *Integer/Float* defining the values to skew the x and y directions by.

origin defines the point at with the skew should originate. It must be a *Coordinate* or `None`. The default is `(0, 0)`.

Normalization

`BaseContour.`**`round`**`()`

Round coordinates in all points to integers.

Environment

`BaseContour.`**`naked`**`()`

Return the environment's native object that has been wrapped by this object.

```
>>> loweLevelObj = obj.naked()
```

`BaseContour.`**`changed`**`(`**args*, ***kwargs*`)`

Tell the environment that something has changed in the object. The behavior of this method will vary from environment to environment.

```
>>> obj.changed()
```

2.1.10 Segment

Description

A `Contour` object is a list of segments. A `Segment` is a list of points with some special attributes and methods.

Overview

Parents

`BaseSegment.contour`	The segment's parent contour.
`BaseSegment.glyph`	The segment's parent glyph.
`BaseSegment.layer`	The segment's parent layer.
`BaseSegment.font`	The segment's parent font.

Identification

`BaseSegment.index`	The index of the segment within the ordered list of the parent contour's segments.

Attributes

`BaseSegment.type`	The segment type.
`BaseSegment.smooth`	Boolean indicating if the segment is smooth or not.

Points

`BaseSegment.points`	A list of points in the segment.
`BaseSegment.onCurve`	The on curve point in the segment.
`BaseSegment.offCurve`	The off curve points in the segment.

Transformations

`BaseSegment.transformBy`	Transform the object.
`BaseSegment.moveBy`	Move the object.
`BaseSegment.scaleBy`	Scale the object.
`BaseSegment.rotateBy`	Rotate the object.
`BaseSegment.skewBy`	Skew the object.

Normalization

`BaseSegment.round`	Round coordinates in all points.

Environment

`BaseSegment.naked`	Return the environment's native object that has been wrapped by this object.
`BaseSegment.changed`	Tell the environment that something has changed in the object.

Reference

class `fontParts.base.`**`BaseSegment`**(**args*, ***kwargs*)

Parents

`BaseSegment.`**`contour`**

The segment's parent contour.

`BaseSegment.`**`glyph`**
The segment's parent glyph.

`BaseSegment.`**`layer`**
The segment's parent layer.

`BaseSegment.`**`font`**
The segment's parent font.

Identification

`BaseSegment.`**`index`**
The index of the segment within the ordered list of the parent contour's segments.

Attributes

`BaseSegment.`**`type`**
The segment type. The possible types are move, line, curve, qcurve.

`BaseSegment.`**`smooth`**
Boolean indicating if the segment is smooth or not.

Points

`BaseSegment.`**`points`**
A list of points in the segment.

`BaseSegment.`**`onCurve`**
The on curve point in the segment.

`BaseSegment.`**`offCurve`**
The off curve points in the segment.

Transformations

`BaseSegment.`**`transformBy`**(*matrix*, *origin=None*)
Transform the object.

```
>>> obj.transformBy((0.5, 0, 0, 2.0, 10, 0))
>>> obj.transformBy((0.5, 0, 0, 2.0, 10, 0), origin=(500, 500))
```

matrix must be a *Transformation Matrix*. **origin** defines the point at with the transformation should originate. It must be a *Coordinate* or `None`. The default is `(0, 0)`.

`BaseSegment.`**`moveBy`**(*value*)
Move the object.

```
>>> obj.moveBy((10, 0))
```

value must be an iterable containing two *Integer/Float* values defining the x and y values to move the object by.

`BaseSegment.`**`scaleBy`**(*value*, *origin=None*)
Scale the object.

```
>>> obj.scaleBy(2.0)
>>> obj.scaleBy((0.5, 2.0), origin=(500, 500))
```

value must be an iterable containing two *Integer/Float* values defining the x and y values to scale the object by. **origin** defines the point at with the scale should originate. It must be a *Coordinate* or `None`. The default is `(0, 0)`.

`BaseSegment.`**`rotateBy`**`(`*value*, *origin=None*`)`

Rotate the object.

```
>>> obj.rotateBy(45)
>>> obj.rotateBy(45, origin=(500, 500))
```

value must be a *Integer/Float* values defining the angle to rotate the object by. **origin** defines the point at with the rotation should originate. It must be a *Coordinate* or `None`. The default is `(0, 0)`.

`BaseSegment.`**`skewBy`**`(`*value*, *origin=None*`)`

Skew the object.

```
>>> obj.skewBy(11)
>>> obj.skewBy((25, 10), origin=(500, 500))
```

value must be rone of the following:

- single *Integer/Float* indicating the value to skew the x direction by.
- iterable cointaining type *Integer/Float* defining the values to skew the x and y directions by.

origin defines the point at with the skew should originate. It must be a *Coordinate* or `None`. The default is `(0, 0)`.

Normalization

`BaseSegment.`**`round`**`()`

Round coordinates in all points.

Environment

`BaseSegment.`**`naked`**`()`

Return the environment's native object that has been wrapped by this object.

```
>>> loweLevelObj = obj.naked()
```

`BaseSegment.`**`changed`**`(`**args*, ***kwargs*`)`

Tell the environment that something has changed in the object. The behavior of this method will vary from environment to environment.

```
>>> obj.changed()
```

2.1.11 bPoint

Description

The `bPoint` is a point object which mimics the old "Bezier Point" from RoboFog. It has attributes for `bcpIn`, anchor, bcpOut and type. The coordinates in bcpIn and bcpOut are relative to the position of the anchor. For instance, if the bcpIn is 20 units to the left of the anchor, its coordinates would be (-20,0), regardless of the coordinates of the anchor itself. Also: bcpIn will be (0,0) when it is "on top of the anchor", i.e. when there is no bcp it will still have a value. The parent of a bPoint is usually a `Contour`.

Overview

Parents

`BaseBPoint.contour`	The bPoint's parent contour.
`BaseBPoint.glyph`	The bPoint's parent glyph.
`BaseBPoint.layer`	The bPoint's parent layer.
`BaseBPoint.font`	The bPoint's parent font.

Identification

`BaseBPoint.index`	The index of the bPoint within the ordered list of the parent contour's bPoints.

Attributes

`BaseBPoint.type`	The bPoint type.

Points

`BaseBPoint.anchor`	The anchor point.
`BaseBPoint.bcpIn`	The incoming off curve.
`BaseBPoint.bcpOut`	The outgoing off curve.

Transformations

`BaseBPoint.transformBy`	Transform the object.
`BaseBPoint.moveBy`	Move the object.
`BaseBPoint.scaleBy`	Scale the object.
`BaseBPoint.rotateBy`	Rotate the object.
`BaseBPoint.skewBy`	Skew the object.

Normalization

`BaseBPoint.round`	Round coordinates.

Environment

BaseBPoint.naked	Return the environment's native object that has been wrapped by this object.
BaseBPoint.changed	Tell the environment that something has changed in the object.

Reference

class fontParts.base.**BaseBPoint**(**args*, ***kwargs*)

Parents

BaseBPoint.**contour**
The bPoint's parent contour.

BaseBPoint.**glyph**
The bPoint's parent glyph.

BaseBPoint.**layer**
The bPoint's parent layer.

BaseBPoint.**font**
The bPoint's parent font.

Identification

BaseBPoint.**index**
The index of the bPoint within the ordered list of the parent contour's bPoints. None if the bPoint does not belong to a contour.

Attributes

BaseBPoint.**type**
The bPoint type.

Points

BaseBPoint.**anchor**
The anchor point.

BaseBPoint.**bcpIn**
The incoming off curve.

BaseBPoint.**bcpOut**
The outgoing off curve.

Transformations

`BaseBPoint.`**`transformBy`**`(`*matrix*, *origin=None*`)`

Transform the object.

```
>>> obj.transformBy((0.5, 0, 0, 2.0, 10, 0))
>>> obj.transformBy((0.5, 0, 0, 2.0, 10, 0), origin=(500, 500))
```

matrix must be a *Transformation Matrix*. **origin** defines the point at with the transformation should originate. It must be a *Coordinate* or `None`. The default is `(0, 0)`.

`BaseBPoint.`**`moveBy`**`(`*value*`)`

Move the object.

```
>>> obj.moveBy((10, 0))
```

value must be an iterable containing two *Integer/Float* values defining the x and y values to move the object by.

`BaseBPoint.`**`scaleBy`**`(`*value*, *origin=None*`)`

Scale the object.

```
>>> obj.scaleBy(2.0)
>>> obj.scaleBy((0.5, 2.0), origin=(500, 500))
```

value must be an iterable containing two *Integer/Float* values defining the x and y values to scale the object by. **origin** defines the point at with the scale should originate. It must be a *Coordinate* or `None`. The default is `(0, 0)`.

`BaseBPoint.`**`rotateBy`**`(`*value*, *origin=None*`)`

Rotate the object.

```
>>> obj.rotateBy(45)
>>> obj.rotateBy(45, origin=(500, 500))
```

value must be a *Integer/Float* values defining the angle to rotate the object by. **origin** defines the point at with the rotation should originate. It must be a *Coordinate* or `None`. The default is `(0, 0)`.

`BaseBPoint.`**`skewBy`**`(`*value*, *origin=None*`)`

Skew the object.

```
>>> obj.skewBy(11)
>>> obj.skewBy((25, 10), origin=(500, 500))
```

value must be rone of the following:

- single *Integer/Float* indicating the value to skew the x direction by.
- iterable cointaining type *Integer/Float* defining the values to skew the x and y directions by.

origin defines the point at with the skew should originate. It must be a *Coordinate* or `None`. The default is `(0, 0)`.

Normalization

`BaseBPoint.`**`round`**`()`

Round coordinates.

Environment

`BaseBPoint.naked()`

Return the environment's native object that has been wrapped by this object.

```
>>> loweLevelObj = obj.naked()
```

`BaseBPoint.changed(*args, **kwargs)`

Tell the environment that something has changed in the object. The behavior of this method will vary from environment to environment.

```
>>> obj.changed()
```

2.1.12 Point

Description

`Point` represents one single point with a particular coordinate in a contour. It is used to access off-curve and on-curve points alike. Its cousin *`BPoint`* also provides access to incoming and outgoing bcps. `Point` is exclusively only one single point.

```
glyph = CurrentGlyph()
for contour in glyph:
    for point in contour.points:
        print(point)
```

Overview

Copy

BasePoint.copy	Copy this object into a new object of the same type.

Parents

BasePoint.contour	The point's parent *BaseContour*.
BasePoint.glyph	The point's parent *BaseGlyph*.
BasePoint.layer	The point's parent *BaseLayer*.
BasePoint.font	The point's parent *BaseFont*.

Identification

BasePoint.name	The name of the point.
BasePoint.identifier	The unique identifier for the object.
BasePoint.index	The index of the point within the ordered list of the parent glyph's point.

Coordinate

BasePoint.x	The x coordinate of the point.
BasePoint.y	The y coordinate of the point.

Type

BasePoint.type	The point type defined with a *String*.
BasePoint.smooth	A `bool` indicating if the point is smooth or not.

Transformations

BasePoint.transformBy	Transform the object.
BasePoint.moveBy	Move the object.
BasePoint.scaleBy	Scale the object.
BasePoint.rotateBy	Rotate the object.
BasePoint.skewBy	Skew the object.

Normalization

BasePoint.round	Round the point's coordinate.

Environment

BasePoint.naked	Return the environment's native object that has been wrapped by this object.
BasePoint.changed	Tell the environment that something has changed in the object.

Reference

Copy

`BasePoint.`**`copy`**`()`
Copy this object into a new object of the same type. The returned object will not have a parent object.

Parents

`BasePoint.`**`contour`**
The point's parent *BaseContour*.

`BasePoint.`**`glyph`**
The point's parent *BaseGlyph*.

`BasePoint.`**`layer`**
The point's parent *BaseLayer*.

`BasePoint.`**`font`**

The point's parent *BaseFont*.

Identification

`BasePoint.`**`name`**

The name of the point. This will be a *String* or `None`.

```
>>> point.name
'my point'
>>> point.name = None
```

`BasePoint.`**`identifier`**

The unique identifier for the object. This value will be an *Identifier* or a `None`. This attribute is read only.

```
>>> object.identifier
'ILHGJlygfds'
```

To request an identifier if it does not exist use *object.getIdentifier()*

`BasePoint.`**`index`**

The index of the point within the ordered list of the parent glyph's point. This attribute is read only.

```
>>> point.index
0
```

Coordinate

`BasePoint.`**`x`**

The x coordinate of the point. It must be an *Integer/Float*.

```
>>> point.x
100
>>> point.x = 101
```

`BasePoint.`**`y`**

The y coordinate of the point. It must be an *Integer/Float*.

```
>>> point.y
100
>>> point.y = 101
```

Type

`BasePoint.`**`type`**

The point type defined with a *String*. The possible types are:

move	An on-curve move to.
line	An on-curve line to.
curve	An on-curve cubic curve to.
qcurve	An on-curve quadratic curve to.
offcurve	An off-curve.

`BasePoint.`**`smooth`**
A `bool` indicating if the point is smooth or not.

```
>>> point.smooth
False
>>> point.smooth = True
```

Transformations

`BasePoint.`**`transformBy`**`(`*matrix*, *origin=None*`)`
Transform the object.

```
>>> obj.transformBy((0.5, 0, 0, 2.0, 10, 0))
>>> obj.transformBy((0.5, 0, 0, 2.0, 10, 0), origin=(500, 500))
```

matrix must be a *Transformation Matrix*. **origin** defines the point at with the transformation should originate. It must be a *Coordinate* or `None`. The default is `(0, 0)`.

`BasePoint.`**`moveBy`**`(`*value*`)`
Move the object.

```
>>> obj.moveBy((10, 0))
```

value must be an iterable containing two *Integer/Float* values defining the x and y values to move the object by.

`BasePoint.`**`scaleBy`**`(`*value*, *origin=None*`)`
Scale the object.

```
>>> obj.scaleBy(2.0)
>>> obj.scaleBy((0.5, 2.0), origin=(500, 500))
```

value must be an iterable containing two *Integer/Float* values defining the x and y values to scale the object by. **origin** defines the point at with the scale should originate. It must be a *Coordinate* or `None`. The default is `(0, 0)`.

`BasePoint.`**`rotateBy`**`(`*value*, *origin=None*`)`
Rotate the object.

```
>>> obj.rotateBy(45)
>>> obj.rotateBy(45, origin=(500, 500))
```

value must be a *Integer/Float* values defining the angle to rotate the object by. **origin** defines the point at with the rotation should originate. It must be a *Coordinate* or `None`. The default is `(0, 0)`.

`BasePoint.`**`skewBy`**`(`*value*, *origin=None*`)`
Skew the object.

```
>>> obj.skewBy(11)
>>> obj.skewBy((25, 10), origin=(500, 500))
```

value must be rone of the following:

- single *Integer/Float* indicating the value to skew the x direction by.
- iterable cointaining type *Integer/Float* defining the values to skew the x and y directions by.

origin defines the point at with the skew should originate. It must be a *Coordinate* or `None`. The default is `(0, 0)`.

Normalization

`BasePoint.round()`

Round the point's coordinate.

```
>>> point.round()
```

This applies to the following:

- x
- y

Environment

`BasePoint.naked()`

Return the environment's native object that has been wrapped by this object.

```
>>> loweLevelObj = obj.naked()
```

`BasePoint.changed(*args, **kwargs)`

Tell the environment that something has changed in the object. The behavior of this method will vary from environment to environment.

```
>>> obj.changed()
```

2.1.13 Component

Description

A component can be a part of a glyph, and it is a reference to another glyph in the same font. With components you can make glyphs depend on other glyphs. Changes to the base glyph will reflect in the component as well.

The parent of a component is usually a glyph. Components can be decomposed: they replace themselves with the actual outlines from the base glyph. When that happens, the link between the original and the component is broken: changes to the base glyph will no longer reflect in the glyph that had the component.

Overview

Parents

BaseComponent.glyph	The component's parent glyph.
BaseComponent.layer	The component's parent layer.
BaseComponent.font	The component's parent font.

Copy

BaseComponent.copy	Copy this object into a new object of the same type.

Identification

BaseComponent.identifier	The unique identifier for the object.
BaseComponent.index	The index of the component within the ordered list of the parent glyph's components.

Attributes

BaseComponent.baseGlyph	The name of the glyph the component references.
BaseComponent.transformation	The component's transformation matrix.
BaseComponent.offset	The component's offset.
BaseComponent.scale	The component's scale.

Queries

BaseComponent.bounds	The bounds of the component: (xMin, yMin, xMax, yMax) or None.
BaseComponent.pointInside	Determine if point is in the black or white of the component.

Pens and Drawing

BaseComponent.draw	Draw the component with the given Pen.
BaseComponent.drawPoints	Draw the contour with the given PointPen.

Transformations

BaseComponent.transformBy	Transform the object.
BaseComponent.moveBy	Move the object.
BaseComponent.scaleBy	Scale the object.
BaseComponent.rotateBy	Rotate the object.
BaseComponent.skewBy	Skew the object.

Normalization

BaseComponent.decompose	Decompose the component.
BaseComponent.round	Round offset coordinates.

Environment

BaseComponent.naked	Return the environment's native object that has been wrapped by this object.

Continued on next page

Table 87 – continued from previous page

BaseComponent.changed	Tell the environment that something has changed in the object.

Reference

class fontParts.base.**BaseComponent**(**args*, ***kwargs*)

Parents

BaseComponent.**glyph**
The component's parent glyph.

BaseComponent.**layer**
The component's parent layer.

BaseComponent.**font**
The component's parent font.

Copy

BaseComponent.**copy**()
Copy this object into a new object of the same type. The returned object will not have a parent object.

Identification

BaseComponent.**identifier**
The unique identifier for the object. This value will be an *Identifier* or a None. This attribute is read only.

```
>>> object.identifier
'ILHGJlygfds'
```

To request an identifier if it does not exist use *object.getIdentifier()*

BaseComponent.**index**
The index of the component within the ordered list of the parent glyph's components.

Attributes

BaseComponent.**baseGlyph**
The name of the glyph the component references.

BaseComponent.**transformation**
The component's transformation matrix.

BaseComponent.**offset**
The component's offset.

BaseComponent.**scale**
The component's scale.

Queries

`BaseComponent.`**`bounds`**
The bounds of the component: (xMin, yMin, xMax, yMax) or None.

`BaseComponent.`**`pointInside`**`(`*point*`)`
Determine if point is in the black or white of the component.

point must be an (x, y) tuple.

Pens and Drawing

`BaseComponent.`**`draw`**`(`*pen*`)`
Draw the component with the given Pen.

`BaseComponent.`**`drawPoints`**`(`*pen*`)`
Draw the contour with the given PointPen.

Transformations

`BaseComponent.`**`transformBy`**`(`*matrix*, *origin=None*`)`
Transform the object.

```
>>> obj.transformBy((0.5, 0, 0, 2.0, 10, 0))
>>> obj.transformBy((0.5, 0, 0, 2.0, 10, 0), origin=(500, 500))
```

matrix must be a *Transformation Matrix*. **origin** defines the point at with the transformation should originate. It must be a *Coordinate* or `None`. The default is `(0, 0)`.

`BaseComponent.`**`moveBy`**`(`*value*`)`
Move the object.

```
>>> obj.moveBy((10, 0))
```

value must be an iterable containing two *Integer/Float* values defining the x and y values to move the object by.

`BaseComponent.`**`scaleBy`**`(`*value*, *origin=None*`)`
Scale the object.

```
>>> obj.scaleBy(2.0)
>>> obj.scaleBy((0.5, 2.0), origin=(500, 500))
```

value must be an iterable containing two *Integer/Float* values defining the x and y values to scale the object by. **origin** defines the point at with the scale should originate. It must be a *Coordinate* or `None`. The default is `(0, 0)`.

`BaseComponent.`**`rotateBy`**`(`*value*, *origin=None*`)`
Rotate the object.

```
>>> obj.rotateBy(45)
>>> obj.rotateBy(45, origin=(500, 500))
```

value must be a *Integer/Float* values defining the angle to rotate the object by. **origin** defines the point at with the rotation should originate. It must be a *Coordinate* or `None`. The default is `(0, 0)`.

`BaseComponent.`**`skewBy`**`(`*value*, *origin=None*`)`
Skew the object.

```
>>> obj.skewBy(11)
>>> obj.skewBy((25, 10), origin=(500, 500))
```

value must be rone of the following:

- single *Integer/Float* indicating the value to skew the x direction by.
- iterable cointaining type *Integer/Float* defining the values to skew the x and y directions by.

origin defines the point at with the skew should originate. It must be a *Coordinate* or `None`. The default is `(0, 0)`.

Normalization

`BaseComponent.`**`decompose`**`()`
Decompose the component.

`BaseComponent.`**`round`**`()`
Round offset coordinates.

Environment

`BaseComponent.`**`naked`**`()`
Return the environment’s native object that has been wrapped by this object.

```
>>> loweLevelObj = obj.naked()
```

`BaseComponent.`**`changed`**`(`**args*, ***kwargs*`)`
Tell the environment that something has changed in the object. The behavior of this method will vary from environment to environment.

```
>>> obj.changed()
```

2.1.14 Anchor

Description

Anchors are single points in a glyph which are not part of a contour. They can be used as reference positions for doing things like assembling components. In most font editors, anchors have a special appearance and can be edited.

```
glyph = CurrentGlyph()
for anchor in glyph.anchors:
    print(anchor)
```

Overview

Copy

BaseAnchor.copy	Copy this object into a new object of the same type.

Parents

BaseAnchor.glyph	The anchor's parent *BaseGlyph*.
BaseAnchor.layer	The anchor's parent *BaseLayer*.
BaseAnchor.font	The anchor's parent *BaseFont*.

Identification

BaseAnchor.name	The name of the anchor.
BaseAnchor.color	The anchor's color.
BaseAnchor.identifier	The unique identifier for the object.
BaseAnchor.index	The index of the anchor within the ordered list of the parent glyph's anchors.

Coordinate

BaseAnchor.x	The x coordinate of the anchor.
BaseAnchor.y	The y coordinate of the anchor.

Transformations

BaseAnchor.transformBy	Transform the object.
BaseAnchor.moveBy	Move the object.
BaseAnchor.scaleBy	Scale the object.
BaseAnchor.rotateBy	Rotate the object.
BaseAnchor.skewBy	Skew the object.

Normalization

BaseAnchor.round	Round the anchor's coordinate.

Environment

BaseAnchor.naked	Return the environment's native object that has been wrapped by this object.
BaseAnchor.changed	Tell the environment that something has changed in the object.

Reference

class `fontParts.base.`**`BaseAnchor`**(**args*, ***kwargs*)

An anchor object. This object is almost always created with *BaseGlyph.appendAnchor*. An orphan anchor can be created like this:

```
>>> anchor = RAnchor()
```

Copy

`BaseAnchor.copy()`
Copy this object into a new object of the same type. The returned object will not have a parent object.

Parents

`BaseAnchor.glyph`
The anchor's parent `BaseGlyph`.

`BaseAnchor.layer`
The anchor's parent `BaseLayer`.

`BaseAnchor.font`
The anchor's parent `BaseFont`.

Identification

`BaseAnchor.name`
The name of the anchor. This will be a *String* or `None`.

```
>>> anchor.name
'my anchor'
>>> anchor.name = None
```

`BaseAnchor.color`
The anchor's color. This will be a *Color* or `None`.

```
>>> anchor.color
None
>>> anchor.color = (1, 0, 0, 0.5)
```

`BaseAnchor.identifier`
The unique identifier for the object. This value will be an *Identifier* or a `None`. This attribute is read only.

```
>>> object.identifier
'ILHGJlygfds'
```

To request an identifier if it does not exist use *object.getIdentifier()*

`BaseAnchor.index`
The index of the anchor within the ordered list of the parent glyph's anchors. This attribute is read only.

```
>>> anchor.index
0
```

Coordinate

`BaseAnchor.x`
The x coordinate of the anchor. It must be an *Integer/Float*.

```
>>> anchor.x
100
>>> anchor.x = 101
```

`BaseAnchor.`**`y`**

The y coordinate of the anchor. It must be an *Integer/Float*.

```
>>> anchor.y
100
>>> anchor.y = 101
```

Transformations

`BaseAnchor.`**`transformBy`**`(`*matrix*, *origin=None*`)`

Transform the object.

```
>>> obj.transformBy((0.5, 0, 0, 2.0, 10, 0))
>>> obj.transformBy((0.5, 0, 0, 2.0, 10, 0), origin=(500, 500))
```

matrix must be a *Transformation Matrix*. **origin** defines the point at with the transformation should originate. It must be a *Coordinate* or `None`. The default is `(0, 0)`.

`BaseAnchor.`**`moveBy`**`(`*value*`)`

Move the object.

```
>>> obj.moveBy((10, 0))
```

value must be an iterable containing two *Integer/Float* values defining the x and y values to move the object by.

`BaseAnchor.`**`scaleBy`**`(`*value*, *origin=None*`)`

Scale the object.

```
>>> obj.scaleBy(2.0)
>>> obj.scaleBy((0.5, 2.0), origin=(500, 500))
```

value must be an iterable containing two *Integer/Float* values defining the x and y values to scale the object by. **origin** defines the point at with the scale should originate. It must be a *Coordinate* or `None`. The default is `(0, 0)`.

`BaseAnchor.`**`rotateBy`**`(`*value*, *origin=None*`)`

Rotate the object.

```
>>> obj.rotateBy(45)
>>> obj.rotateBy(45, origin=(500, 500))
```

value must be a *Integer/Float* values defining the angle to rotate the object by. **origin** defines the point at with the rotation should originate. It must be a *Coordinate* or `None`. The default is `(0, 0)`.

`BaseAnchor.`**`skewBy`**`(`*value*, *origin=None*`)`

Skew the object.

```
>>> obj.skewBy(11)
>>> obj.skewBy((25, 10), origin=(500, 500))
```

value must be rone of the following:

- single *Integer/Float* indicating the value to skew the x direction by.

- iterable cointaining type *Integer/Float* defining the values to skew the x and y directions by.

origin defines the point at with the skew should originate. It must be a *Coordinate* or `None`. The default is `(0, 0)`.

Normalization

`BaseAnchor.`**`round`**`()`

Round the anchor's coordinate.

```
>>> anchor.round()
```

This applies to the following:

- x
- y

Environment

`BaseAnchor.`**`naked`**`()`

Return the environment's native object that has been wrapped by this object.

```
>>> loweLevelObj = obj.naked()
```

`BaseAnchor.`**`changed`**`(*args, **kwargs)`

Tell the environment that something has changed in the object. The behavior of this method will vary from environment to environment.

```
>>> obj.changed()
```

2.1.15 Image

Overview

BaseImage.copy	Copy this object into a new object of the same type.
BaseImage.glyph	The image's parent *BaseGlyph*.
BaseImage.layer	The image's parent *BaseLayer*.
BaseImage.font	The image's parent *BaseFont*.
BaseImage.data	The image's raw byte data.
BaseImage.color	The image's color.
BaseImage.transformation	The image's *Transformation Matrix*.
BaseImage.offset	The image's offset.
BaseImage.scale	The image's scale.
BaseImage.transformBy	Transform the object.
BaseImage.moveBy	Move the object.
BaseImage.scaleBy	Scale the object.
BaseImage.rotateBy	Rotate the object.
BaseImage.skewBy	Skew the object.
BaseImage.round	Round offset coordinates.

Continued on next page

Table 95 – continued from previous page

BaseImage.naked	Return the environment's native object that has been wrapped by this object.
BaseImage.changed	Tell the environment that something has changed in the object.

Reference

class fontParts.base.**BaseImage**(**args*, ***kwargs*)

Copy

BaseImage.**copy**()

Copy this object into a new object of the same type. The returned object will not have a parent object.

Parents

BaseImage.**glyph**

The image's parent *BaseGlyph*.

BaseImage.**layer**

The image's parent *BaseLayer*.

BaseImage.**font**

The image's parent *BaseFont*.

Attributes

BaseImage.**data**

The image's raw byte data. The possible formats are defined by each environment.

BaseImage.**color**

The image's color. This will be a *Color* or None.

```
>>> image.color
None
>>> image.color = (1, 0, 0, 0.5)
```

BaseImage.**transformation**

The image's *Transformation Matrix*. This defines the image's position, scale, and rotation.

```
>>> image.transformation
(1, 0, 0, 1, 0, 0)
>>> image.transformation = (2, 0, 0, 2, 100, -50)
```

BaseImage.**offset**

The image's offset. This is a shortcut to the offset values in *transformation*. This must be an iterable containing two *Integer/Float* values defining the x and y values to offset the image by.

```
>>> image.offset
(0, 0)
>>> image.offset = (100, -50)
```

`BaseImage.`**`scale`**

The image's scale. This is a shortcut to the scale values in *`transformation`*. This must be an iterable containing two *Integer/Float* values defining the x and y values to scale the image by.

```
>>> image.scale
(1, 1)
>>> image.scale = (2, 2)
```

Transformations

`BaseImage.`**`transformBy`**`(`*matrix*, *origin=None*`)`

Transform the object.

```
>>> obj.transformBy((0.5, 0, 0, 2.0, 10, 0))
>>> obj.transformBy((0.5, 0, 0, 2.0, 10, 0), origin=(500, 500))
```

matrix must be a *Transformation Matrix*. **origin** defines the point at with the transformation should originate. It must be a *Coordinate* or `None`. The default is `(0, 0)`.

`BaseImage.`**`moveBy`**`(`*value*`)`

Move the object.

```
>>> obj.moveBy((10, 0))
```

value must be an iterable containing two *Integer/Float* values defining the x and y values to move the object by.

`BaseImage.`**`scaleBy`**`(`*value*, *origin=None*`)`

Scale the object.

```
>>> obj.scaleBy(2.0)
>>> obj.scaleBy((0.5, 2.0), origin=(500, 500))
```

value must be an iterable containing two *Integer/Float* values defining the x and y values to scale the object by. **origin** defines the point at with the scale should originate. It must be a *Coordinate* or `None`. The default is `(0, 0)`.

`BaseImage.`**`rotateBy`**`(`*value*, *origin=None*`)`

Rotate the object.

```
>>> obj.rotateBy(45)
>>> obj.rotateBy(45, origin=(500, 500))
```

value must be a *Integer/Float* values defining the angle to rotate the object by. **origin** defines the point at with the rotation should originate. It must be a *Coordinate* or `None`. The default is `(0, 0)`.

`BaseImage.`**`skewBy`**`(`*value*, *origin=None*`)`

Skew the object.

```
>>> obj.skewBy(11)
>>> obj.skewBy((25, 10), origin=(500, 500))
```

value must be rone of the following:

- single *Integer/Float* indicating the value to skew the x direction by.
- iterable cointaining type *Integer/Float* defining the values to skew the x and y directions by.

origin defines the point at with the skew should originate. It must be a *Coordinate* or `None`. The default is `(0, 0)`.

Normalization

`BaseImage.round()`
Round offset coordinates.

Environment

`BaseImage.naked()`
Return the environment's native object that has been wrapped by this object.

```
>>> loweLevelObj = obj.naked()
```

`BaseImage.changed(*args, **kwargs)`
Tell the environment that something has changed in the object. The behavior of this method will vary from environment to environment.

```
>>> obj.changed()
```

2.1.16 Guideline

Description

Guidelines are reference lines in a glyph that are not part of a contour or the generated font data. They are defined by a point and an angle; the guideline extends from the point in both directions on the specified angle. They are most often used to keep track of design information for a font ('my overshoots should be here') or to measure positions in a glyph ('line the ends of my serifs on this line'). They can also be used as reference positions for doing things like assembling components. In most font editors, guidelines have a special appearance and can be edited.

```
glyph = CurrentGlyph()
for guideline in glyph.guidelines:
    print(guideline)
```

Overview

Copy

BaseGuideline.copy	Copy this object into a new object of the same type.

Parents

BaseGuideline.glyph	The guideline's parent *BaseGlyph*.
BaseGuideline.layer	The guideline's parent *BaseLayer*.
BaseGuideline.font	The guideline's parent *BaseFont*.

Identification

BaseGuideline.name	The name of the guideline.
BaseGuideline.color	”
BaseGuideline.identifier	The unique identifier for the object.
BaseGuideline.index	The index of the guideline within the ordered list of the parent glyph’s guidelines.

Attributes

BaseGuideline.x	The x coordinate of the guideline.
BaseGuideline.y	The y coordinate of the guideline.
BaseGuideline.angle	The angle of the guideline.

Transformations

BaseGuideline.transformBy	Transform the object.
BaseGuideline.moveBy	Move the object.
BaseGuideline.scaleBy	Scale the object.
BaseGuideline.rotateBy	Rotate the object.
BaseGuideline.skewBy	Skew the object.

Normalization

BaseGuideline.round	Round the guideline’s coordinate.

Environment

BaseGuideline.naked	Return the environment’s native object that has been wrapped by this object.
BaseGuideline.changed	Tell the environment that something has changed in the object.

Reference

class fontParts.base.**BaseGuideline**(**args*, ***kwargs*)

A guideline object. This object is almost always created with *BaseGlyph.appendGuideline*. An orphan guideline can be created like this:

```
>>> guideline = RGuideline()
```

Copy

`BaseGuideline.`**`copy`**`()`

Copy this object into a new object of the same type. The returned object will not have a parent object.

Parents

`BaseGuideline.`**`glyph`**

The guideline's parent *`BaseGlyph`*.

`BaseGuideline.`**`layer`**

The guideline's parent *`BaseLayer`*.

`BaseGuideline.`**`font`**

The guideline's parent *`BaseFont`*.

Identification

`BaseGuideline.`**`name`**

The name of the guideline. This will be a *String* or `None`.

```
>>> guideline.name
'my guideline'
>>> guideline.name = None
```

`BaseGuideline.`**`color`**

" The guideline's color. This will be a *Color* or `None`.

```
>>> guideline.color
None
>>> guideline.color = (1, 0, 0, 0.5)
```

`BaseGuideline.`**`identifier`**

The unique identifier for the object. This value will be an *Identifier* or a `None`. This attribute is read only.

```
>>> object.identifier
'ILHGJlygfds'
```

To request an identifier if it does not exist use *object.getIdentifier()*

`BaseGuideline.`**`index`**

The index of the guideline within the ordered list of the parent glyph's guidelines. This attribute is read only.

```
>>> guideline.index
0
```

Attributes

`BaseGuideline.`**`x`**

The x coordinate of the guideline. It must be an *Integer/Float*.

```
>>> guideline.x
100
>>> guideline.x = 101
```

`BaseGuideline.`**`y`**

The y coordinate of the guideline. It must be an *Integer/Float*.

```
>>> guideline.y
100
>>> guideline.y = 101
```

`BaseGuideline.`**`angle`**

The angle of the guideline. It must be an *Angle*. Please check how `normalizers.normalizeRotationAngle` handles the angle. There is a special case, when angle is `None`. If so, when x and y are not 0, the angle will be 0. If x is 0 but y is not, the angle will be 0. If y is 0 and x is not, the angle will be 90. If both x and y are 0, the angle will be 0.

```
>>> guideline.angle
45.0
>>> guideline.angle = 90
```

Transformations

`BaseGuideline.`**`transformBy`**`(`*matrix*, *origin=None*`)`

Transform the object.

```
>>> obj.transformBy((0.5, 0, 0, 2.0, 10, 0))
>>> obj.transformBy((0.5, 0, 0, 2.0, 10, 0), origin=(500, 500))
```

matrix must be a *Transformation Matrix*. **origin** defines the point at with the transformation should originate. It must be a *Coordinate* or `None`. The default is `(0, 0)`.

`BaseGuideline.`**`moveBy`**`(`*value*`)`

Move the object.

```
>>> obj.moveBy((10, 0))
```

value must be an iterable containing two *Integer/Float* values defining the x and y values to move the object by.

`BaseGuideline.`**`scaleBy`**`(`*value*, *origin=None*`)`

Scale the object.

```
>>> obj.scaleBy(2.0)
>>> obj.scaleBy((0.5, 2.0), origin=(500, 500))
```

value must be an iterable containing two *Integer/Float* values defining the x and y values to scale the object by. **origin** defines the point at with the scale should originate. It must be a *Coordinate* or `None`. The default is `(0, 0)`.

`BaseGuideline.`**`rotateBy`**`(`*value*, *origin=None*`)`

Rotate the object.

```
>>> obj.rotateBy(45)
>>> obj.rotateBy(45, origin=(500, 500))
```

value must be a *Integer/Float* values defining the angle to rotate the object by. **origin** defines the point at with the rotation should originate. It must be a *Coordinate* or `None`. The default is `(0, 0)`.

`BaseGuideline.`**`skewBy`**`(`*value*, *origin=None*`)`

Skew the object.

```
>>> obj.skewBy(11)
>>> obj.skewBy((25, 10), origin=(500, 500))
```

value must be rone of the following:

- single *Integer/Float* indicating the value to skew the x direction by.
- iterable cointaining type *Integer/Float* defining the values to skew the x and y directions by.

origin defines the point at with the skew should originate. It must be a *Coordinate* or `None`. The default is `(0, 0)`.

Normalization

`BaseGuideline.`**`round`**`()`

Round the guideline's coordinate.

```
>>> guideline.round()
```

This applies to the following:

- x
- y

It does not apply to

- angle

Environment

`BaseGuideline.`**`naked`**`()`

Return the environment's native object that has been wrapped by this object.

```
>>> loweLevelObj = obj.naked()
```

`BaseGuideline.`**`changed`**`(*args, **kwargs)`

Tell the environment that something has changed in the object. The behavior of this method will vary from environment to environment.

```
>>> obj.changed()
```

2.2 Common Value Types

FontParts scripts are built on with objects that represent fonts, glyphs, contours and so on. The objects are obtained through fontparts-world.

FontParts uses some common value types.

2.2.1 String

Unicode (unencoded) or string. Internally everything is a unicode string.

2.2.2 Integer/Float

Integers and floats are interchangeable in FontParts (unless the specification states that only one is allowed).

2.2.3 Coordinate

An immutable iterable containing two *Integer/Float* representing:

1. x
2. y

2.2.4 Angle

XXX define the angle specifications here. Direction, degrees, etc. This will always be a float.

2.2.5 Identifier

A *String* following the UFO identifier conventions.

2.2.6 Color

An immutable iterable containing four *Integer/Float* representing:

1. red
2. green
3. blue
4. alpha

Values are from 0 to 1.0.

2.2.7 Transformation Matrix

An immutable iterable defining a 2x2 transformation plus offset (aka Affine transform). The default is `(1, 0, 0, 1, 0, 0)`.

2.2.8 Immutable List

This must be an immutable, ordered iterable like a `tuple`.

2.3 fontParts.world

Note: We still need to decide if we need a `world` module or if we should recommend namespace injection.

`fontParts.world.`**`AllFonts`**`(`*sortOptions=None*`)`

Get a list of all open fonts. Optionally, provide a value for `sortOptions` to sort the fonts. See `world.FontList.sortBy` for options.

```
from fontParts.world import *

fonts = AllFonts()
for font in fonts:
    # do something

fonts = AllFonts("magic")
for font in fonts:
    # do something

fonts = AllFonts(["familyName", "styleName"])
for font in fonts:
    # do something
```

`fontParts.world.`**`NewFont`**`(`*familyName=None*, *styleName=None*, *showInterface=True*`)`

Create a new font. **familyName** will be assigned to `font.info.familyName` and **styleName** will be assigned to `font.info.styleName`. These are optional and default to `None`. If **showInterface** is `False`, the font should be created without graphical interface. The default for **showInterface** is `True`.

```
from fontParts.world import *

font = NewFont()
font = NewFont(familyName="My Family", styleName="My Style")
font = NewFont(showInterface=False)
```

`fontParts.world.`**`OpenFont`**`(`*path*, *showInterface=True*`)`

Open font located at **path**. If **showInterface** is `False`, the font should be opened without graphical interface. The default for **showInterface** is `True`.

```
from fontParts.world import *

font = OpenFont("/path/to/my/font.ufo")
font = OpenFont("/path/to/my/font.ufo", showInterface=False)
```

`fontParts.world.`**`OpenFonts`**`(`*directory=None*, *showInterface=True*, *fileExtensions=None*`)`

Open all fonts with the given **fileExtensions** located in **directory**. If **directory** is `None`, a dialog for selecting a directory will be opened. **directory** may also be a list of directories. If **showInterface** is `False`, the font should be opened without graphical interface. The default for **showInterface** is `True`.

The fonts are located within the directory using the *glob* <https://docs.python.org/library/glob.html>'_ module. The patterns are created with `os.path.join(glob, "*" + fileExtension)` for every file extension in `fileExtensions`. If `fileExtensions` if `None` the environment will use its default fileExtensions.

```
from fontParts.world import *

fonts = OpenFonts()
fonts = OpenFonts(showInterface=False)
```

`fontParts.world.`**`CurrentFont`**`()`

Get the "current" font.

`fontParts.world.`**`CurrentLayer`**`()`

Get the "current" layer from *`CurrentGlyph`*.

```
from fontParts.world import *

layer = CurrentLayer()
```

fontParts.world.**CurrentGlyph**()

Get the "current" glyph from *CurrentFont*.

```
from fontParts.world import *

glyph = CurrentGlyph()
```

fontParts.world.**CurrentContours**()

Get the "currently" selected contours from *CurrentGlyph*.

```
from fontParts.world import *

contours = CurrentContours()
```

This returns an immutable list, even when nothing is selected.

fontParts.world.**CurrentSegments**()

Get the "currently" selected segments from *CurrentContours*.

```
from fontParts.world import *

segments = CurrentSegments()
```

This returns an immutable list, even when nothing is selected.

fontParts.world.**CurrentPoints**()

Get the "currently" selected points from *CurrentContours*.

```
from fontParts.world import *

points = CurrentPoints()
```

This returns an immutable list, even when nothing is selected.

fontParts.world.**CurrentComponents**()

Get the "currently" selected components from *CurrentGlyph*.

```
from fontParts.world import *

components = CurrentComponents()
```

This returns an immutable list, even when nothing is selected.

fontParts.world.**CurrentAnchors**()

Get the "currently" selected anchors from *CurrentGlyph*.

```
from fontParts.world import *

anchors = CurrentAnchors()
```

This returns an immutable list, even when nothing is selected.

fontParts.world.**CurrentGuidelines**()

Get the "currently" selected guidelines from *CurrentGlyph*. This will include both font level and glyph level guidelines.

```
from fontParts.world import *

guidelines = CurrentGuidelines()
```

This returns an immutable list, even when nothing is selected.

`fontParts.world.`**`FontList`**(*fonts=None*)

Get a list with font specific methods.

```
from fontParts.world import *

fonts = FontList()
```

Refer to *`BaseFontList`* for full documentation.

`class` `fontParts.world.`**`BaseFontList`**

CHAPTER 3

Developers

3.1 Implementing FontParts

The whole point of FontParts is to present a common API to scripters. So, obviously, the way to implement it is to develop an API that is compliant with the *object documentation*. That's going to be a non-trivial amount of work, so we offer a less laborious alternative: we provide a set of *base objects* that can be subclassed and privately mapped to an environment's native API. If you don't want to use these base objects, you can implement the API all on your own. You just have to make sure that your implementation is compatible.

3.1.1 Testing

A test suite is provided to test any implementation, either subclassed from the base objects or implemented independently. The suite has been designed to be environment and format agnostic. Environment developers only need to implement a function that provides objects for testing and a simple Python script that sends the function to the test suite.

Testing an environment

The main thing that an environment needs to implement is the test object generator. This should create an object for the requested class identifier.

```
def MyAppObjectGenerator(classIdentifier):
    unrequested = []
    obj = myApp.foo.bar.something.hi(classIdentifier)
    return obj, unrequested
```

If an environment does not allow orphan objects, parent objects may create the parent objects and store them in a list. The function must return the generated objects and the list of unrequested objects (or an empty list if no parent objects were generated).

The class identifiers are as follows:

- font
- info
- groups
- kerning
- features
- lib
- layer
- glyph
- contour
- segment
- bpoint
- point
- component
- anchor
- image
- guideline

Once an environment has developed this function, all that remains is to pass the function to the test runner:

```
from fontParts.test import testEnvironment

if __name__ == "__main__":
    testEnvironment(MyAppObjectGenerator)
```

This can then be executed and the report will be printed.

It is up to each environment to ensure that the bridge from the environment's native objects to the fontParts wrappers is working properly. This has to be done on an environment by environment basis since the native objects are not consistently implemented.

3.1.2 Subclassing fontObjects.base

The base objects have been designed to provide common behavior, normalization and type consistency for environments and scripters alike. Environments wrap their native objects with subclasses of fontParts' base objects and implement the necessary translation to the native API. Once this is done, the environment will inherit all of the base behavior from fontParts.

Environments will need to implement their own subclasses of:

Font

Must Override

`BaseFont.`**`_close`**`(`*`**kwargs`*`)`

This is the environment implementation of `BaseFont.close`.

Subclasses must override this method.

`BaseFont.`**`_generate`**`(`*format*, *path*, *environmentOptions*, ***kwargs*`)`

This is the environment implementation of *`BaseFont.generate`*. **format** will be a *String* defining the output format. Refer to the *`BaseFont.generate`* documentation for the standard format identifiers. If the value given for **format** is not supported by the environment, the environment must raise `FontPartsError`. **path** will be a *String* defining the location where the file should be created. It will have been normalized with `normalizers.normalizeFilePath`. **environmentOptions** will be a dictionary of names validated with `BaseFont._isValidGenerateEnvironmentOption` nd the given values. These values will not have been passed through any normalization functions.

Subclasses must override this method.

`BaseFont.`**`_getGuideline`**`(`*index*, ***kwargs*`)`

This must return a *`BaseGuideline`* object. **index** will be a valid **index**.

Subclasses must override this method.

`BaseFont.`**`_get_defaultLayer`**`()`

`BaseFont.`**`_get_features`**`()`

This is the environment implementation of *`BaseFont.features`*. This must return an instance of a *`BaseFeatures`* subclass.

Subclasses must override this method.

`BaseFont.`**`_get_glyphOrder`**`()`

This is the environment implementation of *`BaseFont.glyphOrder`*. This must return an *Immutable List* containing glyph names representing the glyph order in the font. The value will be normalized with `normalizers.normalizeGlyphOrder`.

Subclasses must override this method.

`BaseFont.`**`_get_groups`**`()`

This is the environment implementation of *`BaseFont.groups`*. This must return an instance of a *`BaseGroups`* subclass.

Subclasses must override this method.

`BaseFont.`**`_get_info`**`()`

This is the environment implementation of *`BaseFont.info`*. This must return an instance of a *`BaseInfo`* subclass.

Subclasses must override this method.

`BaseFont.`**`_get_kerning`**`()`

This is the environment implementation of *`BaseFont.kerning`*. This must return an instance of a *`BaseKerning`* subclass.

Subclasses must override this method.

`BaseFont.`**`_get_layerOrder`**`(`***kwargs*`)`

This is the environment implementation of *`BaseFont.layerOrder`*. This must return an *Immutable List* defining the order of the layers in the font. The contents of the list must be layer names as *String*. The list will be normalized with `normalizers.normalizeLayerOrder`.

Subclasses must override this method.

`BaseFont.`**`_get_layers`**`(`***kwargs*`)`

This is the environment implementation of *`BaseFont.layers`*. This must return an *Immutable List* containing instances of *`BaseLayer`* subclasses. The items in the list should be in the order defined by *`BaseFont.layerOrder`*.

Subclasses must override this method.

`BaseFont.`**`_get_lib`**`()`

This is the environment implementation of *`BaseFont.lib`*. This must return an instance of a *`BaseLib`* subclass.

Subclasses must override this method.

`BaseFont.`**`_get_path`**`(`*`**kwargs`*`)`

This is the environment implementation of *`BaseFont.path`*.

This must return a *String* defining the location of the file or `None` indicating that the font does not have a file representation. If the returned value is not `None` it will be normalized with `normalizers.normalizeFilePath`.

Subclasses must override this method.

`BaseFont.`**`_init`**`(`*pathOrObject=None*, *showInterface=True*, *`**kwargs`*`)`

Initialize this object. This should wrap a native font object based on the values for **pathOrObject**:

None	Create a new font.
string	Open the font file located at the given location.
native font object	Wrap the given object.

If **showInterface** is `False`, the font should be created without graphical interface.

Subclasses must override this method.

`BaseFont.`**`_lenGuidelines`**`(`*`**kwargs`*`)`

This must return an integer indicating the number of font-level guidelines in the font.

Subclasses must override this method.

`BaseFont.`**`_newLayer`**`(`*name*, *color*, *`**kwargs`*`)`

This is the environment implementation of *`BaseFont.newLayer`*. **name** will be a *String* representing a valid layer name. The value will have been normalized with `normalizers.normalizeLayerName` and **name** will not be the same as the name of an existing layer. **color** will be a *Color* or `None`. If the value is not `None` the value will have been normalized with `normalizers.normalizeColor`. This must return an instance of a *`BaseLayer`* subclass that represents the new layer.

Subclasses must override this method.

`BaseFont.`**`_removeGuideline`**`(`*index*, *`**kwargs`*`)`

This is the environment implementation of *`BaseFont.removeGuideline`*. **index** will be a valid index.

Subclasses must override this method.

`BaseFont.`**`_removeLayer`**`(`*name*, *`**kwargs`*`)`

This is the environment implementation of *`BaseFont.removeLayer`*. **name** will be a *String* defining the name of an existing layer. The value will have been normalized with `normalizers.normalizeLayerName`.

Subclasses must override this method.

`BaseFont.`**`_save`**`(`*path=None*, *showProgress=False*, *formatVersion=None*, *fileStructure=None*, *`**kwargs`*`)`

This is the environment implementation of *`BaseFont.save`*. **path** will be a *String* or `None`. If **path** is not `None`, the value will have been normalized with `normalizers.normalizeFilePath`. **showProgress** will be a `bool` indicating if the environment should display a progress bar during the operation. Environments are not *required* to display a progress bar even if **showProgess** is `True`. **formatVersion** will be *Integer/Float* or `None` indicating the file format version to write the data into. It will have been normalized with `normalizers.normalizeFileFormatVersion`.

Subclasses must override this method.

BaseFont.**_set_defaultLayer**(*layer*)

BaseFont.**_set_glyphOrder**(*value*)
This is the environment implementation of `BaseFont.glyphOrder`. **value** will be a list of *String*. It will have been normalized with `normalizers.normalizeGlyphOrder`.

Subclasses must override this method.

BaseFont.**_set_layerOrder**(*value*, ***kwargs*)
This is the environment implementation of `BaseFont.layerOrder`. **value** will be a **list** of *String* representing layer names. The list will have been normalized with `normalizers.normalizeLayerOrder`.

Subclasses must override this method.

May Override

BaseFont.**_appendGuideline**(*position*, *angle*, *name=None*, *color=None*, *identifier=None*, ***kwargs*)
This is the environment implementation of `BaseFont.appendGuideline`. **position** will be a valid *Coordinate*. **angle** will be a valid angle. **name** will be a valid *String* or `None`. **color** will be a valid *Color* or `None`. This must return the newly created `BaseGuideline` object.

Subclasses may override this method.

BaseFont.**_autoUnicodes**()
This is the environment implementation of `BaseFont.autoUnicodes`.

Subclasses may override this method.

BaseFont.**_clearGuidelines**()
This is the environment implementation of `BaseFont.clearGuidelines`.

Subclasses may override this method.

BaseFont.**_contains**(*name*, ***kwargs*)
This is the environment implementation of `BaseLayer.__contains__` and `BaseFont.__contains__` This must return `bool` indicating if the layer has a glyph with the defined name. **name** will be a :ref-type-string‘ representing a glyph name. It will have been normalized with `normalizers.normalizeGlyphName`.

Subclasses may override this method.

BaseFont.**_getItem**(*name*, ***kwargs*)
This is the environment implementation of `BaseFont.__getitem__`. **name** will be a *String* defining an existing glyph in the default layer. The value will have been normalized with `normalizers.normalizeGlyphName`.

Subclasses may override this method.

BaseFont.**_getLayer**(*name*, ***kwargs*)
This is the environment implementation of `BaseFont.getLayer`. **name** will be a *String*. It will have been normalized with `normalizers.normalizeLayerName` and it will have been verified as an existing layer. This must return an instance of `BaseLayer`.

Subclasses may override this method.

BaseFont.**_get_guidelines**()
This is the environment implementation of `BaseFont.guidelines`. This must return an *Immutable List* of `BaseGuideline` objects.

Subclasses may override this method.

`BaseFont._insertGlyph`(*glyph*, *name*, ***kwargs*)
This is the environment implementation of `BaseLayer.__setitem__` and `BaseFont.__setitem__`. This must return an instance of a *BaseGlyph* subclass. **glyph** will be a glyph object with the attributes necessary for copying as defined in *BaseGlyph.copy* An environment must not insert **glyph** directly. Instead the data from **glyph** should be copied to a new glyph instead. **name** will be a *String* representing a glyph name. It will have been normalized with `normalizers.normalizeGlyphName`. **name** will have been tested to make sure that no glyph with the same name exists in the layer.

Subclasses may override this method.

`BaseFont._interpolate`(*factor*, *minFont*, *maxFont*, *round=True*, *suppressError=True*)
This is the environment implementation of *BaseFont.interpolate*.

Subclasses may override this method.

`BaseFont._isCompatible`(*other*, *reporter*)
This is the environment implementation of *BaseFont.isCompatible*.

Subclasses may override this method.

`BaseFont._iter`(***kwargs*)
This is the environment implementation of *BaseLayer.__iter__* and *BaseFont.__iter__* This must return an iterator that returns instances of a *BaseGlyph* subclass.

Subclasses may override this method.

`BaseFont._keys`(***kwargs*)
This is the environment implementation of *BaseFont.keys*. This must return an *Immutable List* of all glyph names in the default layer.

Subclasses may override this method.

`BaseFont._len`(***kwargs*)
This is the environment implementation of *BaseLayer.__len__* and *BaseFont.__len__* This must return an `int` indicating the number of glyphs in the layer.

Subclasses may override this method.

`BaseFont._newGlyph`(*name*, ***kwargs*)
This is the environment implementation of *BaseFont.newGlyph*. **name** will be a *String* representing a valid glyph name. The value will have been tested to make sure that an existing glyph in the default layer does not have an identical name. The value will have been normalized with `normalizers.normalizeGlyphName`. This must return an instance of *BaseGlyph* representing the new glyph.

Subclasses may override this method.

`BaseFont._removeGlyph`(*name*, ***kwargs*)
This is the environment implementation of *BaseFont.removeGlyph*. **name** will be a *String* representing an existing glyph in the default layer. The value will have been normalized with `normalizers.normalizeGlyphName`.

Subclasses may override this method.

`BaseFont._round`()
This is the environment implementation of *BaseFont.round*.

Subclasses may override this method.

Info

Must Override

May Override

`BaseInfo.`**`_getAttr`**`(`*attr*`)`
Subclasses may override this method.

If a subclass does not override this method, it must implement '_get_attributeName' methods for all Info methods.

`BaseInfo.`**`_init`**`(`**args*, ***kwargs*`)`
Subclasses may override this method.

`BaseInfo.`**`_interpolate`**`(`*factor*, *minInfo*, *maxInfo*, *round=True*, *suppressError=True*`)`
Subclasses may override this method.

`BaseInfo.`**`_round`**`(`***kwargs*`)`
Subclasses may override this method.

`BaseInfo.`**`_setAttr`**`(`*attr*, *value*`)`
Subclasses may override this method.

If a subclass does not override this method, it must implement '_set_attributeName' methods for all Info methods.

`BaseInfo.`**`copyData`**`(`*source*`)`
Subclasses may override this method. If so, they should call the super.

Groups

Must Override

`BaseGroups.`**`_contains`**`(`*key*`)`
Subclasses must override this method.

`BaseGroups.`**`_delItem`**`(`*key*`)`
Subclasses must override this method.

`BaseGroups.`**`_getItem`**`(`*key*`)`
Subclasses must override this method.

`BaseGroups.`**`_items`**`()`
Subclasses must override this method.

`BaseGroups.`**`_setItem`**`(`*key*, *value*`)`
Subclasses must override this method.

May Override

`BaseGroups.`**`_clear`**`()`
Subclasses may override this method.

`BaseGroups.`**`_findGlyph`**`(`*glyphName*`)`
This is the environment implementation of `BaseGroups.findGlyph`. **glyphName** will be an *String*.

Subclasses may override this method.

`BaseGroups.`**`_get`**`(`*key*, *default=None*`)`
Subclasses may override this method.

`BaseGroups.`**`_init`**`(`**args*, ***kwargs*`)`
Subclasses may override this method.

`BaseGroups.`**`_iter`**`()`
Subclasses may override this method.

`BaseGroups.`**`_keys`**`()`
Subclasses may override this method.

`BaseGroups.`**`_len`**`()`
Subclasses may override this method.

`BaseGroups.`**`_pop`**`(`*key*, *default=None*`)`
Subclasses may override this method.

`BaseGroups.`**`_update`**`(`*other*`)`
Subclasses may override this method.

`BaseGroups.`**`_values`**`()`
Subclasses may override this method.

Kerning

Must Override

`BaseKerning.`**`_contains`**`(`*key*`)`
Subclasses must override this method.

`BaseKerning.`**`_delItem`**`(`*key*`)`
Subclasses must override this method.

`BaseKerning.`**`_getItem`**`(`*key*`)`
Subclasses must override this method.

`BaseKerning.`**`_items`**`()`
Subclasses must override this method.

`BaseKerning.`**`_setItem`**`(`*key*, *value*`)`
Subclasses must override this method.

May Override

`BaseKerning.`**`_clear`**`()`
Subclasses may override this method.

`BaseKerning.`**`_get`**`(`*key*, *default=None*`)`
Subclasses may override this method.

`BaseKerning.`**`_init`**`(`**args*, ***kwargs*`)`
Subclasses may override this method.

`BaseKerning.`**`_interpolate`**`(`*factor*, *minKerning*, *maxKerning*, *round=True*, *suppressError=True*`)`
This is the environment implementation of `BaseKerning.interpolate`.

- **factor** will be an *Integer/Float*, `tuple` or `list`.
- **minKerning** will be a `BaseKerning` object.
- **maxKerning** will be a `BaseKerning` object.

- **round** will be a `bool` indicating if the interpolated kerning should be rounded.
- **suppressError** will be a `bool` indicating if incompatible data should be ignored.

Subclasses may override this method.

`BaseKerning.`**`_iter`**`()`
Subclasses may override this method.

`BaseKerning.`**`_keys`**`()`
Subclasses may override this method.

`BaseKerning.`**`_len`**`()`
Subclasses may override this method.

`BaseKerning.`**`_pop`**`(`*key*, *default=None*`)`
Subclasses may override this method.

`BaseKerning.`**`_round`**`(`*multiple=1*`)`
This is the environment implementation of *`BaseKerning.round`*. **multiple** will be an `int`.

Subclasses may override this method.

`BaseKerning.`**`_scale`**`(`*factor*`)`
This is the environment implementation of *`BaseKerning.scaleBy`*. **factor** will be a `tuple`.

Subclasses may override this method.

`BaseKerning.`**`_update`**`(`*other*`)`
Subclasses may override this method.

`BaseKerning.`**`_values`**`()`
Subclasses may override this method.

Features

Must Override

`BaseFeatures.`**`_get_text`**`()`
This is the environment implementation of *`BaseFeatures.text`*. This must return a *String*.

Subclasses must override this method.

`BaseFeatures.`**`_set_text`**`(`*value*`)`
This is the environment implementation of *`BaseFeatures.text`*. **value** will be a *String*.

Subclasses must override this method.

May Override

`BaseFeatures.`**`_init`**`(`**args*, ***kwargs*`)`
Subclasses may override this method.

`BaseFeatures.`**`copyData`**`(`*source*`)`
Subclasses may override this method. If so, they should call the super.

Lib

Must Override

`BaseLib._contains(key)`
Subclasses must override this method.

`BaseLib._delItem(key)`
Subclasses must override this method.

`BaseLib._getItem(key)`
Subclasses must override this method.

`BaseLib._items()`
Subclasses must override this method.

`BaseLib._setItem(key, value)`
Subclasses must override this method.

May Override

`BaseLib._clear()`
Subclasses may override this method.

`BaseLib._get(key, default=None)`
Subclasses may override this method.

`BaseLib._init(*args, **kwargs)`
Subclasses may override this method.

`BaseLib._iter()`
Subclasses may override this method.

`BaseLib._keys()`
Subclasses may override this method.

`BaseLib._len()`
Subclasses may override this method.

`BaseLib._pop(key, default=None)`
Subclasses may override this method.

`BaseLib._update(other)`
Subclasses may override this method.

`BaseLib._values()`
Subclasses may override this method.

Layer

Must Override

`BaseLayer._getItem(name, **kwargs)`
This is the environment implementation of `BaseLayer.__getitem__` and `BaseFont.__getitem__` This must return an instance of a `BaseGlyph` subclass. **name** will be a *String* representing a name of a glyph that is in the layer. It will have been normalized with `normalizers.normalizeGlyphName`.

Subclasses must override this method.

`BaseLayer.`**`_get_color`**`()`

This is the environment implementation of *`BaseLayer.color`*. This must return a *Color* defining the color assigned to the layer. If the layer does not have an assigned color, the returned value must be `None`. It will be normalized with `normalizers.normalizeColor`.

Subclasses must override this method.

`BaseLayer.`**`_get_lib`**`()`

This is the environment implementation of *`BaseLayer.lib`*. This must return an instance of a *`BaseLib`* subclass.

`BaseLayer.`**`_get_name`**`()`

This is the environment implementation of *`BaseLayer.name`*. This must return a *String* defining the name of the layer. If the layer is the default layer, the returned value must be `None`. It will be normalized with `normalizers.normalizeLayerName`.

Subclasses must override this method.

`BaseLayer.`**`_keys`**`(`***kwargs*`)`

This is the environment implementation of *`BaseLayer.keys`* and *`BaseFont.keys`* This must return an *Immutable List* of the names representing all glyphs in the layer. The order is not defined.

Subclasses must override this method.

`BaseLayer.`**`_newGlyph`**`(`*name*, ***kwargs*`)`

This is the environment implementation of *`BaseLayer.newGlyph`* and *`BaseFont.newGlyph`* This must return an instance of a *`BaseGlyph`* subclass. **name** will be a *String* representing a glyph name. It will have been normalized with `normalizers.normalizeGlyphName`. The name will have been tested to make sure that no glyph with the same name exists in the layer.

Subclasses must override this method.

`BaseLayer.`**`_removeGlyph`**`(`*name*, ***kwargs*`)`

This is the environment implementation of *`BaseLayer.removeGlyph`* and *`BaseFont.removeGlyph`*. **name** will be a *String* representing a glyph name of a glyph that is in the layer. It will have been normalized with `normalizers.normalizeGlyphName`. The newly created *`BaseGlyph`* must be returned.

Subclasses must override this method.

`BaseLayer.`**`_set_color`**`(`*value*, ***kwargs*`)`

This is the environment implementation of *`BaseLayer.color`*. **value** will be a *Color* or `None` defining the color to assign to the layer. It will have been normalized with `normalizers.normalizeColor`.

Subclasses must override this method.

`BaseLayer.`**`_set_name`**`(`*value*, ***kwargs*`)`

This is the environment implementation of *`BaseLayer.name`*. **value** will be a *String* defining the name of the layer. It will have been normalized with `normalizers.normalizeLayerName`. No layer with the same name will exist.

Subclasses must override this method.

May Override

`BaseLayer.`**`_autoUnicodes`**`()`

This is the environment implementation of *`BaseLayer.autoUnicodes`*.

Subclasses may override this method.

`BaseLayer.`**`_contains`**`(`*name*, ***kwargs*`)`

This is the environment implementation of *`BaseLayer.__contains__`* and *`BaseFont.__contains__`* This must return `bool` indicating if the layer has a glyph with the defined name. **name** will be a :ref-type-string‘ representing a glyph name. It will have been normalized with `normalizers.normalizeGlyphName`.

Subclasses may override this method.

`BaseLayer.`**`_init`**`(`**args*, ***kwargs*`)`

Subclasses may override this method.

`BaseLayer.`**`_insertGlyph`**`(`*glyph*, *name*, ***kwargs*`)`

This is the environment implementation of `BaseLayer.__setitem__` and `BaseFont.__setitem__`. This must return an instance of a *`BaseGlyph`* subclass. **glyph** will be a glyph object with the attributes necessary for copying as defined in *`BaseGlyph.copy`* An environment must not insert **glyph** directly. Instead the data from **glyph** should be copied to a new glyph instead. **name** will be a *String* representing a glyph name. It will have been normalized with `normalizers.normalizeGlyphName`. **name** will have been tested to make sure that no glyph with the same name exists in the layer.

Subclasses may override this method.

`BaseLayer.`**`_interpolate`**`(`*factor*, *minLayer*, *maxLayer*, *round=True*, *suppressError=True*`)`

This is the environment implementation of *`BaseLayer.interpolate`*.

Subclasses may override this method.

`BaseLayer.`**`_isCompatible`**`(`*other*, *reporter*`)`

This is the environment implementation of *`BaseLayer.isCompatible`*.

Subclasses may override this method.

`BaseLayer.`**`_iter`**`(`***kwargs*`)`

This is the environment implementation of *`BaseLayer.__iter__`* and *`BaseFont.__iter__`* This must return an iterator that returns instances of a *`BaseGlyph`* subclass.

Subclasses may override this method.

`BaseLayer.`**`_len`**`(`***kwargs*`)`

This is the environment implementation of *`BaseLayer.__len__`* and *`BaseFont.__len__`* This must return an `int` indicating the number of glyphs in the layer.

Subclasses may override this method.

`BaseLayer.`**`_round`**`()`

This is the environment implementation of *`BaseLayer.round`*.

Subclasses may override this method.

Glyph

Must Override

`BaseGlyph.`**`_addImage`**`(`*data*, *transformation=None*, *color=None*`)`

data will be raw, unnormalized image data. Each environment may have different possible formats, so this is unspecified.

transformation will be a valid transformation matrix.

color will be a color tuple or None.

This must return an Image object. Assigning it to the glyph will be handled by the base class.

Subclasses must override this method.

`BaseGlyph._autoUnicodes()`
Subclasses must override this method.

`BaseGlyph._clearImage(**kwargs)`
Subclasses must override this method.

`BaseGlyph._getAnchor(index, **kwargs)`
This must return a wrapped anchor.

index will be a valid index.

Subclasses must override this method.

`BaseGlyph._getComponent(index, **kwargs)`
This must return a wrapped component.

index will be a valid index.

Subclasses must override this method.

`BaseGlyph._getContour(index, **kwargs)`
This must return a wrapped contour.

index will be a valid index.

Subclasses must override this method.

`BaseGlyph._getGuideline(index, **kwargs)`
This must return a wrapped guideline.

index will be a valid index.

Subclasses must override this method.

`BaseGlyph._get_height()`
This must return an int or float.

Subclasses must override this method.

`BaseGlyph._get_image()`
Subclasses must override this method.

`BaseGlyph._get_lib()`
Subclasses must override this method.

`BaseGlyph._get_markColor()`
Return the mark color value as a color tuple or None.

Subclasses must override this method.

`BaseGlyph._get_name()`
Get the name of the glyph. This must return a unicode string.

Subclasses must override this method.

`BaseGlyph._get_note()`
Subclasses must override this method.

`BaseGlyph._get_unicodes()`
Get the unicodes assigned to the glyph. This must return a tuple of zero or more integers.

Subclasses must override this method.

BaseGlyph.**_get_width**()
This must return an int or float.

Subclasses must override this method.

BaseGlyph.**_lenAnchors**(***kwargs*)
This must return an integer indicating the number of anchors in the glyph.

Subclasses must override this method.

BaseGlyph.**_lenComponents**(***kwargs*)
This must return an integer indicating the number of components in the glyph.

Subclasses must override this method.

BaseGlyph.**_lenContours**(***kwargs*)
This must return an integer.

Subclasses must override this method.

BaseGlyph.**_lenGuidelines**(***kwargs*)
This must return an integer indicating the number of guidelines in the glyph.

Subclasses must override this method.

BaseGlyph.**_newLayer**(*name*, ***kwargs*)
name will be a string representing a valid layer name. The name will have been tested to make sure that no layer in the glyph already has the name.

This must returned the new glyph.

Subclasses must override this method.

BaseGlyph.**_removeAnchor**(*index*, ***kwargs*)
index will be a valid index.

Subclasses must override this method.

BaseGlyph.**_removeComponent**(*index*, ***kwargs*)
index will be a valid index.

Subclasses must override this method.

BaseGlyph.**_removeContour**(*index*, ***kwargs*)
index will be a valid index.

Subclasses must override this method.

BaseGlyph.**_removeGuideline**(*index*, ***kwargs*)
index will be a valid index.

Subclasses must override this method.

BaseGlyph.**_removeOverlap**()
Subclasses must implement this method.

BaseGlyph.**_set_height**(*value*)
value will be an int or float.

Subclasses must override this method.

BaseGlyph.**_set_markColor**(*value*)
value will be a color tuple or None.

Subclasses must override this method.

`BaseGlyph.`**`_set_name`**`(`*value*`)`
Set the name of the glyph. This will be a unicode string.

Subclasses must override this method.

`BaseGlyph.`**`_set_note`**`(`*value*`)`
Subclasses must override this method.

`BaseGlyph.`**`_set_unicodes`**`(`*value*`)`
Assign the unicodes to the glyph. This will be a list of zero or more integers.

Subclasses must override this method.

`BaseGlyph.`**`_set_width`**`(`*value*`)`
value will be an int or float.

Subclasses must override this method.

May Override

`BaseGlyph.`**`__add__`**`(`*other*`)`
Subclasses may override this method.

`BaseGlyph.`**`__div__`**`(`*factor*`)`
Subclasses may override this method.

`BaseGlyph.`**`__mul__`**`(`*factor*`)`
Subclasses may override this method.

`BaseGlyph.`**`__rmul__`**`(`*factor*`)`
Subclasses may override this method.

`BaseGlyph.`**`__sub__`**`(`*other*`)`
Subclasses may override this method.

`BaseGlyph.`**`_appendAnchor`**`(`*name*, *position=None*, *color=None*, *identifier=None*, ***kwargs*`)`
name will be a valid anchor name. position will be a valid position (x, y). color will be None or a valid color. identifier will be a valid, nonconflicting identifier.

This must return the new anchor.

Subclasses may override this method.

`BaseGlyph.`**`_appendComponent`**`(`*baseGlyph*, *transformation=None*, *identifier=None*, ***kwargs*`)`
baseGlyph will be a valid glyph name. The baseGlyph may or may not be in the layer.

offset will be a valid offset (x, y). scale will be a valid scale (x, y). identifier will be a valid, nonconflicting identifier.

This must return the new component.

Subclasses may override this method.

`BaseGlyph.`**`_appendContour`**`(`*contour*, *offset=None*, ***kwargs*`)`
contour will be an object with a drawPoints method.

offset will be a valid offset (x, y).

This must return the new contour.

Subclasses may override this method.

`BaseGlyph.`**`_appendGlyph`**`(`*other*, *offset=None*`)`
Subclasses may override this method.

`BaseGlyph.`**`_appendGuideline`**(*position*, *angle*, *name=None*, *color=None*, *identifier=None*, ***kwargs*)

position will be a valid position (x, y). angle will be a valid angle. name will be a valid guideline name or None. color will be a valid color or None . identifier will be a valid, nonconflicting identifier.

This must return the new guideline.

Subclasses may override this method.

`BaseGlyph.`**`_clear`**(*contours=True*, *components=True*, *anchors=True*, *guidelines=True*, *image=True*)

Subclasses may override this method.

`BaseGlyph.`**`_clearAnchors`**()

Subclasses may override this method.

`BaseGlyph.`**`_clearComponents`**()

Subclasses may override this method.

`BaseGlyph.`**`_clearContours`**()

Subclasses may override this method.

`BaseGlyph.`**`_clearGuidelines`**()

Subclasses may override this method.

`BaseGlyph.`**`_decompose`**()

Subclasses may override this method.

`BaseGlyph.`**`_getLayer`**(*name*, ***kwargs*)

name will be a string, but there may not be a layer with a name matching the string. If not, a `ValueError` must be raised.

Subclasses may override this method.

`BaseGlyph.`**`_get_anchors`**()

Subclasses may override this method.

`BaseGlyph.`**`_get_bottomMargin`**()

This must return an int or float. If the glyph has no outlines, this must return *None*.

Subclasses may override this method.

`BaseGlyph.`**`_get_bounds`**()

Subclasses may override this method.

`BaseGlyph.`**`_get_components`**()

Subclasses may override this method.

`BaseGlyph.`**`_get_contours`**()

Subclasses may override this method.

`BaseGlyph.`**`_get_guidelines`**()

Subclasses may override this method.

`BaseGlyph.`**`_get_leftMargin`**()

This must return an int or float. If the glyph has no outlines, this must return *None*.

Subclasses may override this method.

`BaseGlyph.`**`_get_rightMargin`**()

This must return an int or float. If the glyph has no outlines, this must return *None*.

Subclasses may override this method.

`BaseGlyph.`**`_get_topMargin`**()

This must return an int or float. If the glyph has no outlines, this must return *None*.

Subclasses may override this method.

`BaseGlyph.`**`_get_unicode`**`()`

Get the primary unicode assigned to the glyph. This must return an integer or None.

Subclasses may override this method.

`BaseGlyph.`**`_init`**`(`**args*, ***kwargs*`)`

Subclasses may override this method.

`BaseGlyph.`**`_interpolate`**`(`*factor*, *minGlyph*, *maxGlyph*, *round=True*, *suppressError=True*`)`

Subclasses may override this method.

`BaseGlyph.`**`_isCompatible`**`(`*other*, *reporter*`)`

This is the environment implementation of `BaseGlyph.isCompatible`.

Subclasses may override this method.

`BaseGlyph.`**`_iterContours`**`(`***kwargs*`)`

This must return an iterator that returns wrapped contours.

Subclasses may override this method.

`BaseGlyph.`**`_moveBy`**`(`*value*, ***kwargs*`)`

This is the environment implementation of `BaseObject.moveBy`.

value will be an iterable containing two *Integer/Float* values defining the x and y values to move the object by. It will have been normalized with `normalizers.normalizeTransformationOffset`.

Subclasses may override this method.

`BaseGlyph.`**`_pointInside`**`(`*point*`)`

Subclasses may override this method.

`BaseGlyph.`**`_removeLayer`**`(`*name*, ***kwargs*`)`

name will be a valid layer name. It will represent an existing layer in the font.

Subclasses may override this method.

`BaseGlyph.`**`_rotateBy`**`(`*value*, *origin=None*, ***kwargs*`)`

This is the environment implementation of `BaseObject.rotateBy`.

value will be a *Integer/Float* value defining the value to rotate the object by. It will have been normalized with `normalizers.normalizeRotationAngle`. **origin** will be a *Coordinate* defining the point at which the rotation should orginate.

Subclasses may override this method.

`BaseGlyph.`**`_round`**`()`

Subclasses may override this method.

`BaseGlyph.`**`_scaleBy`**`(`*value*, *origin=None*, ***kwargs*`)`

This is the environment implementation of `BaseObject.scaleBy`.

value will be an iterable containing two *Integer/Float* values defining the x and y values to scale the object by. It will have been normalized with `normalizers.normalizeTransformationScale`. **origin** will be a *Coordinate* defining the point at which the scale should orginate.

Subclasses may override this method.

`BaseGlyph.`**`_set_bottomMargin`**`(`*value*`)`

value will be an int or float.

Subclasses may override this method.

`BaseGlyph.`**`_set_leftMargin`**`(`*value*`)`
value will be an int or float.

Subclasses may override this method.

`BaseGlyph.`**`_set_rightMargin`**`(`*value*`)`
value will be an int or float.

Subclasses may override this method.

`BaseGlyph.`**`_set_topMargin`**`(`*value*`)`
value will be an int or float.

Subclasses may override this method.

`BaseGlyph.`**`_set_unicode`**`(`*value*`)`
Assign the primary unicode to the glyph. This will be an integer or None.

Subclasses may override this method.

`BaseGlyph.`**`_skewBy`**`(`*value*, *origin=None*, ***kwargs*`)`
This is the environment implementation of `BaseObject.skewBy`.

value will be an iterable containing two *Integer/Float* values defining the x and y values to skew the object by. It will have been normalized with `normalizers.normalizeTransformationSkewAngle`. **origin** will be a *Coordinate* defining the point at which the skew should orginate.

Subclasses may override this method.

`BaseGlyph.`**`_transformBy`**`(`*matrix*, ***kwargs*`)`
Subclasses may override this method.

Contour

Must Override

`BaseContour.`**`_getPoint`**`(`*index*, ***kwargs*`)`
This must return a wrapped point.

index will be a valid index.

Subclasses must override this method.

`BaseContour.`**`_get_clockwise`**`()`
Subclasses must override this method.

`BaseContour.`**`_get_identifier`**`()`
This is the environment implementation of `BaseObject.identifier`. This must return an *Identifier*. If the native object does not have an identifier assigned one should be assigned and returned.

Subclasses must override this method.

`BaseContour.`**`_insertPoint`**`(`*index*, *position*, *type='line'*, *smooth=False*, *name=None*, *identifier=None*, ***kwargs*`)`
position will be a valid position (x, y). type will be a valid type. smooth will be a valid boolean. name will be a valid name or None. identifier will be a valid identifier or None. The identifier will not have been tested for uniqueness.

Subclasses must override this method.

`BaseContour.`**`_lenPoints`**`(`***kwargs*`)`
This must return an integer indicating the number of points in the contour.

Subclasses must override this method.

`BaseContour.`**`_removePoint`**`(`*index*, *preserveCurve*, ***kwargs*`)`
index will be a valid index. preserveCurve will be a boolean.

Subclasses must override this method.

`BaseContour.`**`_set_index`**`(`*value*`)`
Subclasses must override this method.

May Override

`BaseContour.`**`_appendBPoint`**`(`*type*, *anchor*, *bcpIn=None*, *bcpOut=None*, ***kwargs*`)`
Subclasses may override this method.

`BaseContour.`**`_appendSegment`**`(`*type=None*, *points=None*, *smooth=False*, ***kwargs*`)`
Subclasses may override this method.

`BaseContour.`**`_autoStartSegment`**`(`***kwargs*`)`
Subclasses may override this method.

XXX port this from robofab

`BaseContour.`**`_draw`**`(`*pen*, ***kwargs*`)`
Subclasses may override this method.

`BaseContour.`**`_drawPoints`**`(`*pen*, ***kwargs*`)`
Subclasses may override this method.

`BaseContour.`**`_get_bounds`**`()`
Subclasses may override this method.

`BaseContour.`**`_get_index`**`()`
Subclasses may override this method.

`BaseContour.`**`_get_points`**`()`
Subclasses may override this method.

`BaseContour.`**`_get_segments`**`()`
Subclasses may override this method.

`BaseContour.`**`_init`**`(`**args*, ***kwargs*`)`
Subclasses may override this method.

`BaseContour.`**`_insertBPoint`**`(`*index*, *type*, *anchor*, *bcpIn*, *bcpOut*, ***kwargs*`)`
Subclasses may override this method.

`BaseContour.`**`_insertSegment`**`(`*index=None*, *type=None*, *points=None*, *smooth=False*, ***kwargs*`)`
Subclasses may override this method.

`BaseContour.`**`_len__segments`**`(`***kwargs*`)`
Subclasses may override this method.

`BaseContour.`**`_moveBy`**`(`*value*, ***kwargs*`)`
This is the environment implementation of `BaseObject.moveBy`.

value will be an iterable containing two *Integer/Float* values defining the x and y values to move the object by. It will have been normalized with `normalizers.normalizeTransformationOffset`.

Subclasses may override this method.

`BaseContour.`**`_pointInside`**`(`*point*`)`
Subclasses may override this method.

`BaseContour.`**`_removeSegment`**`(`*segment*, *preserveCurve*, ***kwargs*`)`
segment will be a valid segment index. preserveCurve will be a boolean.

Subclasses may override this method.

`BaseContour.`**`_reverse`**`(`***kwargs*`)`
Subclasses may override this method.

`BaseContour.`**`_rotateBy`**`(`*value*, *origin=None*, ***kwargs*`)`
This is the environment implementation of `BaseObject.rotateBy`.

value will be a *Integer/Float* value defining the value to rotate the object by. It will have been normalized with `normalizers.normalizeRotationAngle`. **origin** will be a *Coordinate* defining the point at which the rotation should orginate.

Subclasses may override this method.

`BaseContour.`**`_round`**`(`***kwargs*`)`
Subclasses may override this method.

`BaseContour.`**`_scaleBy`**`(`*value*, *origin=None*, ***kwargs*`)`
This is the environment implementation of `BaseObject.scaleBy`.

value will be an iterable containing two *Integer/Float* values defining the x and y values to scale the object by. It will have been normalized with `normalizers.normalizeTransformationScale`. **origin** will be a *Coordinate* defining the point at which the scale should orginate.

Subclasses may override this method.

`BaseContour.`**`_setStartSegment`**`(`*segmentIndex*, ***kwargs*`)`
Subclasses may override this method.

`BaseContour.`**`_set_clockwise`**`(`*value*`)`
Subclasses may override this method.

`BaseContour.`**`_skewBy`**`(`*value*, *origin=None*, ***kwargs*`)`
This is the environment implementation of `BaseObject.skewBy`.

value will be an iterable containing two *Integer/Float* values defining the x and y values to skew the object by. It will have been normalized with `normalizers.normalizeTransformationSkewAngle`. **origin** will be a *Coordinate* defining the point at which the skew should orginate.

Subclasses may override this method.

`BaseContour.`**`_transformBy`**`(`*matrix*, ***kwargs*`)`
Subclasses may override this method.

Segment

Must Override

May Override

`BaseSegment.`**`_getItem`**`(`*index*`)`
Subclasses may override this method.

`BaseSegment.`**`_get_base_offCurve`**`()`
Subclasses may override this method.

`BaseSegment.`**`_get_index`**`()`
Subclasses may override this method.

`BaseSegment.`**`_get_offCurve`**`()`
Subclasses may override this method.

`BaseSegment.`**`_get_onCurve`**`()`
Subclasses may override this method.

`BaseSegment.`**`_get_points`**`()`
Subclasses may override this method.

`BaseSegment.`**`_get_smooth`**`()`
Subclasses may override this method.

`BaseSegment.`**`_get_type`**`()`
Subclasses may override this method.

`BaseSegment.`**`_init`**`(`**args*, ***kwargs*`)`
Subclasses may override this method.

`BaseSegment.`**`_iterPoints`**`(`***kwargs*`)`
Subclasses may override this method.

`BaseSegment.`**`_len`**`(`***kwargs*`)`
Subclasses may override this method.

`BaseSegment.`**`_moveBy`**`(`*value*, ***kwargs*`)`
This is the environment implementation of `BaseObject.moveBy`.

value will be an iterable containing two *Integer/Float* values defining the x and y values to move the object by. It will have been normalized with `normalizers.normalizeTransformationOffset`.

Subclasses may override this method.

`BaseSegment.`**`_rotateBy`**`(`*value*, *origin=None*, ***kwargs*`)`
This is the environment implementation of `BaseObject.rotateBy`.

value will be a *Integer/Float* value defining the value to rotate the object by. It will have been normalized with `normalizers.normalizeRotationAngle`. **origin** will be a *Coordinate* defining the point at which the rotation should orginate.

Subclasses may override this method.

`BaseSegment.`**`_scaleBy`**`(`*value*, *origin=None*, ***kwargs*`)`
This is the environment implementation of `BaseObject.scaleBy`.

value will be an iterable containing two *Integer/Float* values defining the x and y values to scale the object by. It will have been normalized with `normalizers.normalizeTransformationScale`. **origin** will be a *Coordinate* defining the point at which the scale should orginate.

Subclasses may override this method.

`BaseSegment.`**`_set_smooth`**`(`*value*`)`
Subclasses may override this method.

`BaseSegment.`**`_set_type`**`(`*newType*`)`
Subclasses may override this method.

`BaseSegment.`**`_skewBy`**`(`*value*, *origin=None*, ***kwargs*`)`
This is the environment implementation of `BaseObject.skewBy`.

value will be an iterable containing two *Integer/Float* values defining the x and y values to skew the object by. It will have been normalized with `normalizers.normalizeTransformationSkewAngle`. **origin** will be a *Coordinate* defining the point at which the skew should orginate.

Subclasses may override this method.

`BaseSegment.`**`_transformBy`**`(`*matrix*, ***kwargs*`)`
Subclasses may override this method.

`BaseSegment.`**`copyData`**`(`*source*`)`
Subclasses may override this method. If so, they should call the super.

BPoint

Must Override

May Override

`BaseBPoint.`**`_get_anchor`**`()`
Subclasses may override this method.

`BaseBPoint.`**`_get_bcpIn`**`()`
Subclasses may override this method.

`BaseBPoint.`**`_get_bcpOut`**`()`
Subclasses may override this method.

`BaseBPoint.`**`_get_index`**`()`
Subclasses may override this method.

`BaseBPoint.`**`_get_type`**`()`
Subclasses may override this method.

`BaseBPoint.`**`_init`**`(`**args*, ***kwargs*`)`
Subclasses may override this method.

`BaseBPoint.`**`_moveBy`**`(`*value*, ***kwargs*`)`
This is the environment implementation of `BaseObject.moveBy`.

value will be an iterable containing two *Integer/Float* values defining the x and y values to move the object by. It will have been normalized with `normalizers.normalizeTransformationOffset`.

Subclasses may override this method.

`BaseBPoint.`**`_rotateBy`**`(`*value*, *origin=None*, ***kwargs*`)`
This is the environment implementation of `BaseObject.rotateBy`.

value will be a *Integer/Float* value defining the value to rotate the object by. It will have been normalized with `normalizers.normalizeRotationAngle`. **origin** will be a *Coordinate* defining the point at which the rotation should orginate.

Subclasses may override this method.

`BaseBPoint.`**`_scaleBy`**`(`*value*, *origin=None*, ***kwargs*`)`
This is the environment implementation of `BaseObject.scaleBy`.

value will be an iterable containing two *Integer/Float* values defining the x and y values to scale the object by. It will have been normalized with `normalizers.normalizeTransformationScale`. **origin** will be a *Coordinate* defining the point at which the scale should orginate.

Subclasses may override this method.

`BaseBPoint.`**`_set_anchor`**`(`*value*`)`
Subclasses may override this method.

`BaseBPoint.`**`_set_bcpIn`**`(`*value*`)`
Subclasses may override this method.

`BaseBPoint._set_bcpOut(value)`
Subclasses may override this method.

`BaseBPoint._set_type(value)`
Subclasses may override this method.

`BaseBPoint._skewBy(value, origin=None, **kwargs)`
This is the environment implementation of `BaseObject.skewBy`.

value will be an iterable containing two *Integer/Float* values defining the x and y values to skew the object by. It will have been normalized with `normalizers.normalizeTransformationSkewAngle`. **origin** will be a *Coordinate* defining the point at which the skew should orginate.

Subclasses may override this method.

`BaseBPoint._transformBy(matrix, **kwargs)`
Subclasses may override this method.

`BaseBPoint.copyData(source)`
Subclasses may override this method. If so, they should call the super.

Point

Must Override

`BasePoint._get_identifier()`
This is the environment implementation of `BaseObject.identifier`. This must return an *Identifier*. If the native object does not have an identifier assigned one should be assigned and returned.

Subclasses must override this method.

`BasePoint._get_name()`
This is the environment implementation of `BasePoint.name`. This must return a *String* or `None`. The returned value will be normalized with `normalizers.normalizePointName`.

Subclasses must override this method.

`BasePoint._get_smooth()`
This is the environment implementation of `BasePoint.smooth`. This must return a `bool` indicating the smooth state.

Subclasses must override this method.

`BasePoint._get_type()`
This is the environment implementation of `BasePoint.type`. This must return a *String* defining the point type.

Subclasses must override this method.

`BasePoint._get_x()`
This is the environment implementation of `BasePoint.x`. This must return an *Integer/Float*.

Subclasses must override this method.

`BasePoint._get_y()`
This is the environment implementation of `BasePoint.y`. This must return an *Integer/Float*.

Subclasses must override this method.

`BasePoint._set_name(value)`
This is the environment implementation of `BasePoint.name`. **value** will be a *String* or `None`. It will have been normalized with `normalizers.normalizePointName`.

Subclasses must override this method.

`BasePoint._set_smooth(value)`
This is the environment implementation of `BasePoint.smooth`. **value** will be a `bool` indicating the smooth state. It will have been normalized with `normalizers.normalizeBoolean`.

Subclasses must override this method.

`BasePoint._set_type(value)`
This is the environment implementation of `BasePoint.type`. **value** will be a *String* defining the point type. It will have been normalized with `normalizers.normalizePointType`.

Subclasses must override this method.

`BasePoint._set_x(value)`
This is the environment implementation of `BasePoint.x`. **value** will be an *Integer/Float*.

Subclasses must override this method.

`BasePoint._set_y(value)`
This is the environment implementation of `BasePoint.y`. **value** will be an *Integer/Float*.

Subclasses must override this method.

May Override

`BasePoint._get_index()`
Get the point's index. This must return an `int`.

Subclasses may override this method.

`BasePoint._init(*args, **kwargs)`
Subclasses may override this method.

`BasePoint._moveBy(value, **kwargs)`
This is the environment implementation of `BaseObject.moveBy`.

value will be an iterable containing two *Integer/Float* values defining the x and y values to move the object by. It will have been normalized with `normalizers.normalizeTransformationOffset`.

Subclasses may override this method.

`BasePoint._rotateBy(value, origin=None, **kwargs)`
This is the environment implementation of `BaseObject.rotateBy`.

value will be a *Integer/Float* value defining the value to rotate the object by. It will have been normalized with `normalizers.normalizeRotationAngle`. **origin** will be a *Coordinate* defining the point at which the rotation should orginate.

Subclasses may override this method.

`BasePoint._round(**kwargs)`
This is the environment implementation of `BasePoint.round`.

Subclasses may override this method.

`BasePoint._scaleBy(value, origin=None, **kwargs)`
This is the environment implementation of `BaseObject.scaleBy`.

value will be an iterable containing two *Integer/Float* values defining the x and y values to scale the object by. It will have been normalized with `normalizers.normalizeTransformationScale`. **origin** will be a *Coordinate* defining the point at which the scale should orginate.

Subclasses may override this method.

`BasePoint.`**`_skewBy`**`(`*value*, *origin=None*, ***kwargs*`)`

This is the environment implementation of `BaseObject.skewBy`.

value will be an iterable containing two *Integer/Float* values defining the x and y values to skew the object by. It will have been normalized with `normalizers.normalizeTransformationSkewAngle`. **origin** will be a *Coordinate* defining the point at which the skew should orginate.

Subclasses may override this method.

`BasePoint.`**`_transformBy`**`(`*matrix*, ***kwargs*`)`

This is the environment implementation of *`BasePoint.transformBy`*.

matrix will be a *Transformation Matrix*. that has been normalized with `normalizers.normalizeTransformationMatrix`.

Subclasses may override this method.

`BasePoint.`**`copyData`**`(`*source*`)`

Subclasses may override this method. If so, they should call the super.

Component

Must Override

`BaseComponent.`**`_decompose`**`()`

Subclasses must override this method.

`BaseComponent.`**`_get_baseGlyph`**`()`

Subclasses must override this method.

`BaseComponent.`**`_get_identifier`**`()`

This is the environment implementation of `BaseObject.identifier`. This must return an *Identifier*. If the native object does not have an identifier assigned one should be assigned and returned.

Subclasses must override this method.

`BaseComponent.`**`_get_transformation`**`()`

Subclasses must override this method.

`BaseComponent.`**`_set_baseGlyph`**`(`*value*`)`

Subclasses must override this method.

`BaseComponent.`**`_set_index`**`(`*value*`)`

Subclasses must override this method.

`BaseComponent.`**`_set_transformation`**`(`*value*`)`

Subclasses must override this method.

May Override

`BaseComponent.`**`_draw`**`(`*pen*, ***kwargs*`)`

Subclasses may override this method.

`BaseComponent.`**`_drawPoints`**`(`*pen*, ***kwargs*`)`

Subclasses may override this method.

`BaseComponent.`**`_get_bounds`**`()`

Subclasses may override this method.

`BaseComponent.`**`_get_index`**`()`
Subclasses may override this method.

`BaseComponent.`**`_get_offset`**`()`
Subclasses may override this method.

`BaseComponent.`**`_get_scale`**`()`
Subclasses may override this method.

`BaseComponent.`**`_init`**`(`**args*, ***kwargs*`)`
Subclasses may override this method.

`BaseComponent.`**`_moveBy`**`(`*value*, ***kwargs*`)`
This is the environment implementation of `BaseObject.moveBy`.

value will be an iterable containing two *Integer/Float* values defining the x and y values to move the object by. It will have been normalized with `normalizers.normalizeTransformationOffset`.

Subclasses may override this method.

`BaseComponent.`**`_pointInside`**`(`*point*`)`
Subclasses may override this method.

`BaseComponent.`**`_rotateBy`**`(`*value*, *origin=None*, ***kwargs*`)`
This is the environment implementation of `BaseObject.rotateBy`.

value will be a *Integer/Float* value defining the value to rotate the object by. It will have been normalized with `normalizers.normalizeRotationAngle`. **origin** will be a *Coordinate* defining the point at which the rotation should orginate.

Subclasses may override this method.

`BaseComponent.`**`_round`**`()`
Subclasses may override this method.

`BaseComponent.`**`_scaleBy`**`(`*value*, *origin=None*, ***kwargs*`)`
This is the environment implementation of `BaseObject.scaleBy`.

value will be an iterable containing two *Integer/Float* values defining the x and y values to scale the object by. It will have been normalized with `normalizers.normalizeTransformationScale`. **origin** will be a *Coordinate* defining the point at which the scale should orginate.

Subclasses may override this method.

`BaseComponent.`**`_set_offset`**`(`*value*`)`
Subclasses may override this method.

`BaseComponent.`**`_set_scale`**`(`*value*`)`
Subclasses may override this method.

`BaseComponent.`**`_skewBy`**`(`*value*, *origin=None*, ***kwargs*`)`
This is the environment implementation of `BaseObject.skewBy`.

value will be an iterable containing two *Integer/Float* values defining the x and y values to skew the object by. It will have been normalized with `normalizers.normalizeTransformationSkewAngle`. **origin** will be a *Coordinate* defining the point at which the skew should orginate.

Subclasses may override this method.

`BaseComponent.`**`_transformBy`**`(`*matrix*, ***kwargs*`)`
Subclasses may override this method.

`BaseComponent.`**`copyData`**`(`*source*`)`
Subclasses may override this method. If so, they should call the super.

Anchor

Must Override

`BaseAnchor.`**`_get_color`**`()`

This is the environment implementation of *`BaseAnchor.color`*. This must return a *Color* or `None`. The returned value will be normalized with `normalizers.normalizeColor`.

Subclasses must override this method.

`BaseAnchor.`**`_get_identifier`**`()`

This is the environment implementation of `BaseObject.identifier`. This must return an *Identifier*. If the native object does not have an identifier assigned one should be assigned and returned.

Subclasses must override this method.

`BaseAnchor.`**`_get_name`**`()`

This is the environment implementation of *`BaseAnchor.name`*. This must return a *String* or `None`. The returned value will be normalized with `normalizers.normalizeAnchorName`.

Subclasses must override this method.

`BaseAnchor.`**`_get_x`**`()`

This is the environment implementation of *`BaseAnchor.x`*. This must return an *Integer/Float*.

Subclasses must override this method.

`BaseAnchor.`**`_get_y`**`()`

This is the environment implementation of *`BaseAnchor.y`*. This must return an *Integer/Float*.

Subclasses must override this method.

`BaseAnchor.`**`_set_color`**`(`*value*`)`

This is the environment implementation of *`BaseAnchor.color`*. **value** will be a *Color* or `None`. It will have been normalized with `normalizers.normalizeColor`.

Subclasses must override this method.

`BaseAnchor.`**`_set_name`**`(`*value*`)`

This is the environment implementation of *`BaseAnchor.name`*. **value** will be a *String* or `None`. It will have been normalized with `normalizers.normalizeAnchorName`.

Subclasses must override this method.

`BaseAnchor.`**`_set_x`**`(`*value*`)`

This is the environment implementation of *`BaseAnchor.x`*. **value** will be an *Integer/Float*.

Subclasses must override this method.

`BaseAnchor.`**`_set_y`**`(`*value*`)`

This is the environment implementation of *`BaseAnchor.y`*. **value** will be an *Integer/Float*.

Subclasses must override this method.

May Override

`BaseAnchor.`**`_init`**`(`**args*, ***kwargs*`)`

Subclasses may override this method.

`BaseAnchor.`**`_moveBy`**`(`*value*, ***kwargs*`)`

This is the environment implementation of `BaseObject.moveBy`.

value will be an iterable containing two *Integer/Float* values defining the x and y values to move the object by. It will have been normalized with `normalizers.normalizeTransformationOffset`.

Subclasses may override this method.

`BaseAnchor.`**`_rotateBy`**`(`*value*, *origin=None*, ***kwargs*`)`

This is the environment implementation of `BaseObject.rotateBy`.

value will be a *Integer/Float* value defining the value to rotate the object by. It will have been normalized with `normalizers.normalizeRotationAngle`. **origin** will be a *Coordinate* defining the point at which the rotation should orginate.

Subclasses may override this method.

`BaseAnchor.`**`_scaleBy`**`(`*value*, *origin=None*, ***kwargs*`)`

This is the environment implementation of `BaseObject.scaleBy`.

value will be an iterable containing two *Integer/Float* values defining the x and y values to scale the object by. It will have been normalized with `normalizers.normalizeTransformationScale`. **origin** will be a *Coordinate* defining the point at which the scale should orginate.

Subclasses may override this method.

`BaseAnchor.`**`_skewBy`**`(`*value*, *origin=None*, ***kwargs*`)`

This is the environment implementation of `BaseObject.skewBy`.

value will be an iterable containing two *Integer/Float* values defining the x and y values to skew the object by. It will have been normalized with `normalizers.normalizeTransformationSkewAngle`. **origin** will be a *Coordinate* defining the point at which the skew should orginate.

Subclasses may override this method.

`BaseAnchor.`**`_transformBy`**`(`*matrix*, ***kwargs*`)`

This is the environment implementation of *`BaseAnchor.transformBy`*.

matrix will be a *Transformation Matrix*. that has been normalized with `normalizers.normalizeTransformationMatrix`.

Subclasses may override this method.

`BaseAnchor.`**`copyData`**`(`*source*`)`

Subclasses may override this method. If so, they should call the super.

Guideline

Must Override

`BaseGuideline.`**`_get_angle`**`()`

This is the environment implementation of *`BaseGuideline.angle`*. This must return an *Angle*.

Subclasses must override this method.

`BaseGuideline.`**`_get_color`**`()`

This is the environment implementation of *`BaseGuideline.color`*. This must return a *Color* or `None`. The returned value will be normalized with `normalizers.normalizeColor`.

Subclasses must override this method.

`BaseGuideline.`**`_get_identifier`**`()`

This is the environment implementation of `BaseObject.identifier`. This must return an *Identifier*. If the native object does not have an identifier assigned one should be assigned and returned.

Subclasses must override this method.

BaseGuideline.**_get_name**()

This is the environment implementation of *BaseGuideline.name*. This must return a *String* or None. The returned value will be normalized with normalizers.normalizeGuidelineName.

Subclasses must override this method.

BaseGuideline.**_get_x**()

This is the environment implementation of *BaseGuideline.x*. This must return an *Integer/Float*.

Subclasses must override this method.

BaseGuideline.**_get_y**()

This is the environment implementation of *BaseGuideline.y*. This must return an *Integer/Float*.

Subclasses must override this method.

BaseGuideline.**_set_angle**(*value*)

This is the environment implementation of *BaseGuideline.angle*. **value** will be an *Angle*.

Subclasses must override this method.

BaseGuideline.**_set_color**(*value*)

This is the environment implementation of *BaseGuideline.color*. **value** will be a *Color* or None. It will have been normalized with normalizers.normalizeColor.

Subclasses must override this method.

BaseGuideline.**_set_name**(*value*)

This is the environment implementation of *BaseGuideline.name*. **value** will be a *String* or None. It will have been normalized with normalizers.normalizeGuidelineName.

Subclasses must override this method.

BaseGuideline.**_set_x**(*value*)

This is the environment implementation of *BaseGuideline.x*. **value** will be an *Integer/Float*.

Subclasses must override this method.

BaseGuideline.**_set_y**(*value*)

This is the environment implementation of *BaseGuideline.y*. **value** will be an *Integer/Float*.

Subclasses must override this method.

May Override

BaseGuideline.**_get_index**()

Get the guideline's index. This must return an int.

Subclasses may override this method.

BaseGuideline.**_init**(**args*, ***kwargs*)

Subclasses may override this method.

BaseGuideline.**_moveBy**(*value*, ***kwargs*)

This is the environment implementation of BaseObject.moveBy.

value will be an iterable containing two *Integer/Float* values defining the x and y values to move the object by. It will have been normalized with normalizers.normalizeTransformationOffset.

Subclasses may override this method.

`BaseGuideline.`**`_rotateBy`**`(`*value*, *origin=None*, ***kwargs*`)`

This is the environment implementation of `BaseObject.rotateBy`.

value will be a *Integer/Float* value defining the value to rotate the object by. It will have been normalized with `normalizers.normalizeRotationAngle`. **origin** will be a *Coordinate* defining the point at which the rotation should orginate.

Subclasses may override this method.

`BaseGuideline.`**`_round`**`(`***kwargs*`)`

This is the environment implementation of *`BaseGuideline.round`*.

Subclasses may override this method.

`BaseGuideline.`**`_scaleBy`**`(`*value*, *origin=None*, ***kwargs*`)`

This is the environment implementation of `BaseObject.scaleBy`.

value will be an iterable containing two *Integer/Float* values defining the x and y values to scale the object by. It will have been normalized with `normalizers.normalizeTransformationScale`. **origin** will be a *Coordinate* defining the point at which the scale should orginate.

Subclasses may override this method.

`BaseGuideline.`**`_skewBy`**`(`*value*, *origin=None*, ***kwargs*`)`

This is the environment implementation of `BaseObject.skewBy`.

value will be an iterable containing two *Integer/Float* values defining the x and y values to skew the object by. It will have been normalized with `normalizers.normalizeTransformationSkewAngle`. **origin** will be a *Coordinate* defining the point at which the skew should orginate.

Subclasses may override this method.

`BaseGuideline.`**`_transformBy`**`(`*matrix*, ***kwargs*`)`

This is the environment implementation of *`BaseGuideline.transformBy`*.

matrix will be a *Transformation Matrix*. that has been normalized with `normalizers.normalizeTransformationMatrix`.

Subclasses may override this method.

`BaseGuideline.`**`copyData`**`(`*source*`)`

Subclasses may override this method. If so, they should call the super.

Image

Must Override

`BaseImage.`**`_get_color`**`()`

Return the color value as a color tuple or None.

Subclasses must override this method.

`BaseImage.`**`_get_data`**`()`

This must return raw byte data.

Subclasses must override this method.

`BaseImage.`**`_get_transformation`**`()`

Subclasses must override this method.

`BaseImage.`**`_set_color`**`(`*value*`)`

value will be a color tuple or None.

Subclasses must override this method.

`BaseImage._set_data(value)`
value will be raw byte data.

Subclasses must override this method.

`BaseImage._set_transformation(value)`
Subclasses must override this method.

May Override

`BaseImage._get_offset()`
Subclasses may override this method.

`BaseImage._get_scale()`
Subclasses may override this method.

`BaseImage._init(*args, **kwargs)`
Subclasses may override this method.

`BaseImage._moveBy(value, **kwargs)`
This is the environment implementation of `BaseObject.moveBy`.

value will be an iterable containing two *Integer/Float* values defining the x and y values to move the object by. It will have been normalized with `normalizers.normalizeTransformationOffset`.

Subclasses may override this method.

`BaseImage._rotateBy(value, origin=None, **kwargs)`
This is the environment implementation of `BaseObject.rotateBy`.

value will be a *Integer/Float* value defining the value to rotate the object by. It will have been normalized with `normalizers.normalizeRotationAngle`. **origin** will be a *Coordinate* defining the point at which the rotation should orginate.

Subclasses may override this method.

`BaseImage._round()`
Subclasses may override this method.

`BaseImage._scaleBy(value, origin=None, **kwargs)`
This is the environment implementation of `BaseObject.scaleBy`.

value will be an iterable containing two *Integer/Float* values defining the x and y values to scale the object by. It will have been normalized with `normalizers.normalizeTransformationScale`. **origin** will be a *Coordinate* defining the point at which the scale should orginate.

Subclasses may override this method.

`BaseImage._set_offset(value)`
Subclasses may override this method.

`BaseImage._set_scale(value)`
Subclasses may override this method.

`BaseImage._skewBy(value, origin=None, **kwargs)`
This is the environment implementation of `BaseObject.skewBy`.

value will be an iterable containing two *Integer/Float* values defining the x and y values to skew the object by. It will have been normalized with `normalizers.normalizeTransformationSkewAngle`. **origin** will be a *Coordinate* defining the point at which the skew should orginate.

Subclasses may override this method.

`BaseImage._transformBy(matrix, **kwargs)`
Subclasses may override this method.

`BaseImage.copyData(source)`
Subclasses may override this method. If so, they should call the super.

Each of these require their own specific environment overrides, but the general structure follows this form:

```
from fontParts.base import BaseSomething

class MySomething(BaseSomething):

    # Initialization.
    # This will be called when objects are initialized.
    # The behavior, args and kwargs may be designed by the
    # subclass to implement specific behaviors.

    def _init(self, myObj):
        self.myObj = myObj

    # Comparison.
    # The __eq__ method must be implemented by subclasses.
    # It must return a boolean indicating if the lower level
    # objects are the same object. This does not mean that two
    # objects that have the same content should be considered
    # equal. It means that the object must be the same. The
    # corrilary __ne__ is optional to define.
    #
    # Note that the base implentation of fontParts provides
    # __eq__ and __ne__ methods that test the naked objects
    # for equality. Depending on environmental needs this can
    # be overridden.

    def __eq__(self, other):
        return self.myObj == other.myObj

    def __ne__(self, other):
        return self.myObj != other.myObj

    # Properties.
    # Properties are get and set through standard method names.
    # Within these methods, the subclass may do whatever is
    #   necessary to get/set the value from/to the environment.

    def _get_something(self):
        return self.myObj.getSomething()

    def _set_something(self, value):
        self.myObj.setSomething(value)

    # Methods.
    # Generally, the public methods call internal methods with
    # the same name, but preceded with an underscore. Subclasses
    # may implement the internal method. Any values passed to
    # the internal methods will have been normalized and will
    # be a standard type.
```

(continues on next page)

(continued from previous page)

```
    def _whatever(self, value):
        self.myObj.doWhatever(value)

    # Copying.
    # Copying is handled in most cases by the base objects.
    # If subclasses have a special class that should be used
    # when creating a copy of an object, the class must be
    # defined with the copyClass attribute. If anything special
    # needs to be done during the copying process, the subclass
    # can implement the copyData method. This method will be
    # called automatically. The subclass must call the base class
    # method with super.

    copyClass = MyObjectWithoutUI

    def copyData(self, source):
        super(MySomething, self).copyData(source)
        self.myObj.internalThing = source.internalThing

    # Environment updating.
    # If the environment requires the scripter to manually
    # notify the environment that the object has been changed,
    # the subclass must implement the changed method. Please
    # try to avoid requiring this.

    def changed(self):
        myEnv.goUpdateYourself()

    # Wrapped objects.
    # It is very useful for scripters to have access to the
    # lower level, wrapped object. Subclasses implement this
    # with the naked method.

    def naked(self):
        return self.myObj
```

All methods that must be overridden are labeled with “Subclasses must override this method.” in the method’s documentation string. If a method may optionally be overridden, the documentation string is labeled with “Subclasses may override this method.” All other methods, attributes and properties **must not** be overridden.

An example implementation that wraps the defcon library with fontParts is located in fontParts/objects/fontshell.

Data Normalization

When possible, incoming and outgoing values are checked for type validity and are coerced to a common type for return. This is done with a set of functions located here:

These are done in a central place rather than within the objects for consitency. There are many cases where a `(x, y)` tuple is defined and than rewriting all of the code to check if there are exactly two values, that each is an `int` or a `float` and so on before finally making sure that the value to be returned is a `tuple` not an instance of `list`, `OrderedDict` or some native object we consolidate the code into a single function and call that.

Environment Specific Methods, Attributes and Arguments

FontParts is designed to be environment agnostic. Therefore, it is a 100% certainty that it doesn't do *something* that your environment does. You will want to allow your environment's *something* to be accessible through FontParts. *We* want you to allow this, too. The problem is, how do you implement *something* in a way that doesn't conflict with current or future things in FontParts? For example, let's say that you want to add a support for the "Do Something to the Font" feature you have built in your environment. You add a new method to support this:

```
class MyFont(BaseFont):

  def doSomething(self, skip=None, double=None):
    # go
```

This *will* work. However, if FontParts adds a `doSomething` method in a later version that does something other than what or accepts different arguments than your method does, it's not going to work. Either the `doSomething` method will have to be changed in your implementation or you will not support the FontParts `doSomething` method. This is going to be lead to you being mad at FontParts, your scripters being mad at you or something else unpleasant.

The solution to this problem is to prevent it from happening in the first place. To do this, environment specific methods, proprties and attributes must be prefixed with an environment specific tag followed by an _ and then your method name. For example:

```
class MyFont(BaseFont):

  def myapp_doSomething(self, skip=None, double=None):
    # go
```

This applies to any method, attribute or property additions to the FontParts objects. The environment tag is up to you. The only requirement is that it needs to be unique to your own environment.

Method Arguments

In some cases, you are likely to discover that your environment supports specific options in a method that are not supported by the environment agnostic API. For example, your environment may have an optional heuristic that can be used in the `font.autoUnicodes` method. However, the `font.autoUnicodes` method does not have a `useHeuristics` argument. Unfortunately, Python doesn't offer a way to handle this in a way that is both flexible for developers and friendly for scripters. The only two options for handling this are:

1. Create an environment specific clone of the `font.autoUnicodes` method as `myapp_autoUnicodes` and add your `useHeuristics` argument there.
2. Contact the FontParts developers by opening a GitHub issue requesting support for your argument. If it is generic enough, we may add support for it.

We're experimenting with a third way to handle this. You can see it as the `**environmentOptions` argument in the `BaseFont.generate` method. This may or may not move to other methods. Please contact us if you are interested in this being applied to other methods.

3.1.3 Layers

There are two primary layer models in the font world:

- font level layers: In this model, all glyphs have the same layers. A good example of this is a chromatic font.
- glyph level layers: In this model, individual glyphs may have their own unique layers.

fontParts supports both of these models. Both fonts and glyphs have fully developed layer APIs:

```
font = CurrentFont()
foregroundLayer = font.getLayer("foreground")
backgroundLayer = font.getLayer("background")

glyph = font["A"]
foregroundGlyph = glyph.getLayer("foreground")
backgroundGlyph = glyph.getLayer("background")
```

A font-level layer is a font-like object. Essentially, a layer has the same glyph management behavior as a font:

```
font = CurrentFont()
foreground = font.getLayer("foreground")
glyph = foreground.newGlyph("A")
```

A glyph-level layer is identical to a glyph object:

```
font = CurrentFont()
glyph = font["A"]
foreground = glyph.getLayer("foreground")
background = glyph.getLayer("background")
```

When a scripter is addressing a font or glyph without specifying a specific layer, the action is performed on the "default" (or primary) layer. For example, in the original Fontographer there were two layers: foreground and background. The foreground was the primary layer and it contained the primary data that would be compiled into a font binary. In multi-layered glyph editing environments, designers can specify which layer should be considered primary. This layer is the "default" layer in fontParts. Thus:

```
font = CurrentFont()
glyph1 = font["A"]
glyph2 = font.newGlyph("B")
```

The *glyph1* object will reference the A's "foreground" layer and the "foreground" layer will contain a new glyph named "B".

fontParts delegates the implementation to the environment subclasses. Given that an environment can only support font-level layers *or* glyph-level layers, the following algorithms can be used to simulate the model that the environment doesn't support.

Simulating glyph-level layers.

1. Get the parent font.
2. Iterate through all of the font's layers.
3. If the glyph's name is in the layer, grab the glyph from the layer.
4. Return all found glyphs.

Simulating font-level layers.

1. Iterate over all glyphs.
2. For every layer in the glyph, create a global mapping of layer name to glyphs containing a layer with the same name.

3.2 Developing FontParts

You want to help with developing FontParts? Yay!

We are mostly focused on documenting the objects and building a test suite. We'll eventually need bits of code here and there. If you have an idea for a new API or want to discuss one of the testing APIs, cool.

3.2.1 Proposals

Want to suggest a new font part for FontParts? It's best to do this as an issue on the FontParts GitHub repository. Please present why you think this needs to be added. Before you do so, please make sure you understand the goals of the project, the existing API and so on.

3.2.2 Bug Reports

Notice a bug when using FontParts? Is it a bug in a specific application? If so, please report the bug to the application developer. If it's not specific to a particular application, please open an issue on GitHub or, if you really can't open an issue on GitHub, send a message to the RoboFab mailing list

3.2.3 Coding

Take a look at the open issues and see if there is anything there that you want to work on. Please try to follow the general coding style of the library so that everything has the same level of readability.

This library follows much of PEP8, with a couple of exceptions. You'll see camelCase. We like camelCase. The standard line length is also 90 characters. If possible, try to keep lines to 80, but 90 comes in handy occasionally. You'll also notice that some builtin names are redefined in as variables in methods. It's impossible not to use `type` in a package dealing with fonts.

3.2.4 Writing Documentation

We really need help with adding the formatted documentation strings to the base objects. The API documentation is generated from those. Here's a style guide. Please look at the Documentation project on GitHub and see if there is anything you want to work on. If there is, ask to be assigned to that issue, and then follow the style guide. A good place to look for examples of the object documentation is the glyph object.

3.2.5 Test Suite

We also really need help in finishing up the test suite. You can see what needs to be done in the Tests project on GitHub. Pick something you want to write tests for and ask to be assigned to that issue. More information about writing tests is here.

Python Module Index

Index