Kurt Documentation

Release 2.0.0

Tim Radvan

Jul 23, 2017

Contents

Contents:

CHAPTER 1

Changes from kurt 1.4

This section describes the changes between kurt version 1.4 and version 2.0, and how to upgrade your code for the new interface. If you've never used kurt before, skip this section.

Kurt 2.0 includes support for multiple file formats, and so has a brand-new, shiny interface. As the API breaks support with previous versions, the major version has been updated.

Ideally you should rewrite your code to use the *new interface*. It's much cleaner, and you get support for multiple file formats for free!

A quick, very incomplete list of some of the names that have changed:

- `kurt.ScratchProjectFile.new()` -> `kurt.Project()`
- `kurt.ScratchProjectFile(path)` -> `kurt.Project.load(path)`
- `project.stage.variables` -> `project.variables`
- `project.stage.lists` -> `project.lists`
- `sprite.scripts.append(kurt.parse_block_plugin(text)` -> `sprite.parse(text)`
- `kurt.Image.from_image(name, pil_image)` -> `kurt.Costume(name, kurt.Image(pil_image))`
- `sprite.lists[name] = kurt.ScratchListMorph(name='bob', items=[1, 2])` -> `sprite.lists['bob'] = kurt.List([1, 2])`
- `kurt.Point(20, 100)` -> `(20, 100)`

CHAPTER 2

API

This is the documentation for Kurt's interface, mostly the data structures for storing and accessing the information contained in Scratch files.

kurt A Python module for reading and writing Scratch project files.

Scratch is created by the Lifelong Kindergarten Group at the MIT Media Lab. See their website: http://scratch.mit.edu/

Classes

The main interface:

- *Project*

The following *Actors* may be found on the project stage:

- *Stage*
- *Sprite*
- *Watcher*

The two *Scriptables* (*Stage* and *Sprite*) have instances of the following contained in their attributes:

- *Variable*
- *List*

Scripts use the following classes:

- *Block*
- *Script*
- *Comment*
- *BlockType*

Media files use the following classes:

- *Costume*
- *Image*
- *Sound*
- *Waveform*

File Formats

Supported file formats:

Format Name	Description	Extension
"scratch14"	Scratch 1.4	.sb
"scratch20"	Scratch 2.0	.sb2

Pass "Format name" as the argument to *Project.convert*.

Kurt provides a superset of the information in each individual format, but will only convert features between a subset of formats.

---

class kurt.**Project**

Bases: object

The main kurt class. Stores the contents of a project file.

Contents include global variables and lists, the *stage* and *sprites*, each with their own scripts, costumes, sounds, *variables* and *lists*.

A Project can be loaded from or saved to disk in a format which can be read by a Scratch program or one of its derivatives.

Loading a project:

```
p = kurt.Project.load("tests/game.sb")
```

Getting all the scripts:

```
for scriptable in p.sprites + [p.stage]:
    for script in scriptable.scripts:
        print script
```

Creating a new project:

```
p = kurt.Project()
```

Converting between formats:

```
p = kurt.Project.load("tests/game.sb")
p.convert("scratch20")
# []
p.save()
# 'tests/game.sb2'
```

name = None
The name of the project.

May be displayed to the user. Doesn't have to match the filename in *path*. May not be saved for some formats.

path = None
The path to the project file.

stage = None
The *Stage*.

sprites = None
List of *Sprites*.

Use *get_sprite* to get a sprite by name.

actors = None
List of each *Actor* on the stage.

Includes *Watchers* as well as *Sprites*.

Sprites in *sprites* but not in actors will be added to actors on save.

variables = None
`dict` of global *Variables* by name.

lists = None
`dict` of global *Lists* by name.

thumbnail = None
An *Image* with a screenshot of the project.

tempo = None
The tempo in BPM used for note blocks.

notes = None
Notes about the project, aka project comments.

Displayed on the website next to the project.

Line endings will be converted to `\n`.

author = None
The username of the project's author, eg. `'blob8108'`.

get_sprite(*name*)
Get a sprite from *sprites* by name.

Returns None if the sprite isn't found.

format
The file format of the project.

Project is mainly a universal representation, and so a project has no specfic format. This is the format the project was loaded with. To convert to a different format, use *save()*.

classmethod load(*path*, *format=None*)
Load project from file.

Use `format` to specify the file format to use.

Path can be a file-like object, in which case format is required. Otherwise, can guess the appropriate format from the extension.

If you pass a file-like object, you're responsible for closing the file.

Parameters

- **path** – Path or file pointer.
- **format** – `KurtFileFormat.name` eg. `"scratch14"`. Overrides the extension.

Raises *UnknownFormat* if the extension is unrecognised.

Raises `ValueError` if the format doesn't exist.

copy()
Return a new Project instance, deep-copying all the attributes.

convert(*format*)
Convert the project in-place to a different file format.

Returns a list of *UnsupportedFeature* objects, which may give warnings about the conversion.

Parameters **format** – `KurtFileFormat.name` eg. `"scratch14"`.

Raises `ValueError` if the format doesn't exist.

save(*path=None*, *debug=False*)
Save project to file.

Parameters

- **path** – Path or file pointer.

  If you pass a file pointer, you're responsible for closing it.

  If path is not given, the *path* attribute is used, usually the original path given to *load()*.

  If *path* has the extension of an existing plugin, the project will be converted using *convert*. Otherwise, the extension will be replaced with the extension of the current plugin.

  (Note that log output for the conversion will be printed to stdout. If you want to deal with the output, call *convert* directly.)

  If the path ends in a folder instead of a file, the filename is based on the project's *name*.

- **debug** – If true, return debugging information from the format plugin instead of the path.

Raises `ValueError` if there's no path or name.

Returns path to the saved file.

get_broadcasts()

class kurt.**UnsupportedFeature**(*feature*, *obj*)
Bases: `object`

The plugin doesn't support this Feature.

Output once by Project.convert for each occurence of the feature.

exception kurt.**UnknownFormat**
Bases: `exceptions.Exception`

The file extension is not recognised.

Raised when *Project* can't find a valid format plugin to handle the file extension.

exception kurt.**UnknownBlock**
Bases: `exceptions.Exception`

A *Block* with the given command or type cannot be found.

Raised by *BlockType.get*.

exception kurt.**BlockNotSupported**
Bases: exceptions.Exception

The plugin doesn't support this Block.

Raised by Block.convert when it can't find a *PluginBlockType* for the given plugin.

exception kurt.**VectorImageError**
Bases: exceptions.Exception

Tried to construct a raster image from a vector format image file.

You shouldn't usally get this error, because Feature("Vector Images") will give a warning instead when the Project is converted.

class kurt.**Actor**
Bases: object

An object that goes on the project stage.

Subclasses include *Watcher* or *Sprite*.

class kurt.**Scriptable**(*project*)
Bases: object

Superclass for all scriptable objects.

Subclasses are *Stage* and *Sprite*.

project = None
The *Project* this belongs to.

scripts = None
The contents of the scripting area.

List containing *Scripts* and *Comments*.

Will be sorted by y position on load/save.

custom_blocks = None
Scripts for custom blocks, indexed by *CustomBlockType*.

variables = None
dict of *Variables* by name.

lists = None
dict of *Lists* by name.

costumes = None
List of *Costumes*.

sounds = None
List of *Sounds*.

costume = None
The currently selected *Costume*.

Defaults to the first costume in self.costumes on save.

If a sprite doesn't have a costume, a black 1x1 pixel square will be used.

volume = None
The volume in percent used for note and sound blocks.

copy(*o=None*)
Return a new instance, deep-copying all the attributes.

costume_index
The index of *costume* in *costumes*.

None if no costume is selected.

parse(*text*)
Parse the given code and add it to *scripts*.

The syntax matches *Script.stringify()*. See `kurt.text` for reference.

class `kurt.`**Stage**(*project*)
Bases: *kurt.Scriptable*

Represents the background of the project. The stage is similar to a *Sprite*, but has a fixed position. The stage has a fixed size of `480x360` pixels.

The stage does not require a costume. If none is given, it is assumed to be white (#FFF).

Not all formats have stage-specific variables and lists. Global variables and lists are stored on the *Project*.

Parameters **project** – The *Project* this Stage belongs to. Note that you still need to set *Project.stage* to this Stage instance.

name = 'Stage'

is_draggable = False

is_visible = True

SIZE = (480, 360)

COLOR = (255, 255, 255)

backgrounds
Alias for `costumes`.

class `kurt.`**Sprite**(*project*, *name*)
Bases: *kurt.Scriptable*, *kurt.Actor*

A scriptable object displayed on the project stage. Can be moved and rotated, unlike the *Stage*.

Sprites require a `costume`, and will raise an error when saving without one.

Parameters **project** – The *Project* this Sprite belongs to. Note that you still need to add this sprite to *Project.sprites*.

name = None
The name of the sprite, as referred to from scripts and displayed in the Scratch interface.

position = None
The `(x, y)` position of the centre of the sprite in Scratch co-ordinates.

direction = None
The angle in degrees the sprite is rotated to.

rotation_style = None
How the sprite's costume rotates with the sprite. Valid values are:

'normal' Continuous rotation with *direction*. The default.

'leftRight' Don't rotate. Instead, flip the costume for directions with x component < 0. Useful for side-views.

'none' Don't rotate with direction.

size = None
The scale factor of the sprite in percent. Defaults to 100.

is_draggable = None
True if the sprite can be dragged using the mouse in the player/presentation mode.

is_visible = None
Whether the sprite is shown on the stage. False if the sprite is hidden.

copy()
Return a new instance, deep-copying all the attributes.

class kurt.**Watcher**(*target*, *block*, *style='normal'*, *is_visible=True*, *pos=None*)
Bases: *kurt.Actor*

A monitor for displaying a data value on the stage.

Some formats won't save hidden watchers, and so their position won't be remembered.

target = None
The *Scriptable* or *Project* the watcher belongs to.

block = None
The *Block* to evaluate on *target*.

For variables:

```
kurt.Block('readVariable', 'variable name')
```

For lists:

```
kurt.Block('contentsOfList:', 'list name')
```

style = None
How the watcher should appear.

Valid values:

'normal' The name of the data is displayed next to its value. The only valid value for list watchers.

'large' The data is displayed in a larger font with no describing text.

'slider' Like the normal style, but displayed with a slider that can change the variable's value. Not valid for reporter block watchers.

pos = None
`(x, y)` position of the top-left of the watcher from the top-left of the stage in pixels. None if not specified.

is_visible = None
Whether the watcher is displayed on the screen.

Some formats won't save hidden watchers, and so their position won't be remembered.

slider_min = None
Minimum value for slider. Only applies to `"slider"` style.

slider_max = None
Maximum value for slider. Only applies to `"slider"` style.

copy()
Return a new instance with the same attributes.

kind
The type of value to watch, based on *block*.

One of `variable`, `list`, or `block`.

`block` watchers watch the value of a reporter block.

value
Return the *Variable* or *List* to watch.

Returns `None` if it's a block watcher.

class `kurt.`**Variable**(*value=0*, *is_cloud=False*)
Bases: `object`

A memory value used in scripts.

There are both *global variables* and `sprite-specific variables`.

Some formats also have `stage-specific variables`.

value = None
The value of the variable, usually a number or a string.

For some formats, variables can take list values, and *List* is not used.

is_cloud = None
Whether the value of the variable is shared with other users.

For Scratch 2.0.

watcher = None
The *Watcher* instance displaying this Variable's value.

copy()
Return a new instance with the same attributes.

class `kurt.`**List**(*items=None*, *is_cloud=False*)
Bases: `object`

A sequence of items used in scripts.

Each item takes a *Variable*-like value.

Lists cannot be nested. However, for some formats, variables can take list values, and this class is not used.

items = None
The items contained in the list. A Python list of unicode strings.

is_cloud = None
Whether the value of the list is shared with other users.

For Scratch 2.0.

watcher = None
The *Watcher* instance displaying this List's value.

copy()
Return a new instance with the same attributes.

class `kurt.`**Color**(*r*, *g=None*, *b=None*)
Bases: `object`

A 24-bit RGB color value.

Accepts tuple or hexcode arguments:

```
>>> kurt.Color('#f08')
kurt.Color(255, 0, 136)

>>> kurt.Color((255, 0, 136))
kurt.Color(255, 0, 136)

>>> kurt.Color('#f0ffee')
kurt.Color(240, 255, 238)
```

r = None
Red component, 0-255

g = None
Green component, 0-255

b = None
Blue component, 0-255

value
Return `(r, g, b)` tuple.

stringify()
Returns the color value in hexcode format.

eg. `'#ff1056'`

classmethod random()

class kurt.**Insert**(*shape*, *kind=None*, *default=None*, *name=None*, *unevaluated=None*)
Bases: object

The specification for an argument to a *BlockType*.

SHAPE_DEFAULTS = {'color': kurt.Color(255, 0, 0), 'inline': 'nil', 'number-menu': 0, 'number': 0, 'stack': []}

SHAPE_FMTS = {'number-menu': '(%s v)', 'boolean': '<%s>', 'string': '[%s]', 'readonly-menu': '[%s v]', 'color': '[%s]

KIND_OPTIONS = {'attribute': ['x position', 'y position', 'direction', 'costume #', 'size', 'volume'], 'booleanSensor': ['bu

shape = None
What kind of values this argument accepts.

Shapes that accept a simple data value or a reporter block:

'number' An integer or float number. Defaults to `0`.

'string' A unicode text value.

'readonly-menu' A choice of string value from a menu.

Some readonly inserts do not accept reporter blocks.

'number-menu' Either a number value, or a choice of special value from a menu.

Defaults to `0`.

'color' A *Color* value. Defaults to a random color.

Shapes that only accept blocks with the corresponding *shape*:

'boolean' Accepts a boolean block.

'stack' Accepts a list of stack blocks. Defaults to `[]`.

The block is rendered with a "mouth" into which blocks can be inserted.

Special shapes:

'inline' Not actually an insert – used for variable and list reporters.

'block' Used for the argument to the "define ..." hat block.

kind = None

Valid arguments for a "menu"-shaped insert. Default is `None`.

Valid values include:

- `'attribute'`
- `'booleanSensor'`
- `'broadcast'`
- `'costume'`
- `'direction'`
- `'drum'`
- `'effect'`
- `'instrument'`
- `'key'`
- `'list'`
- `'listDeleteItem'`
- `'listItem'`
- `'mathOp'`
- `'motorDirection'`
- `'note'`
- `'sensor'`
- `'sound'`
- `'spriteOrMouse'`
- `'spriteOrStage'`
- `'touching'`
- `'var'`

Scratch 2.0-specific:

- `'backdrop'`
- `'rotationStyle'`
- `'spriteOnly'`
- `'stageOrThis'`
- `'stop'`
- `'timeAndDate'`
- `'triggerSensor'`
- `'videoMotionType'`

- `'videoState'`

default = None
The default value for the insert.

unevaluated = None
True if the interpreter should evaluate the argument to the block.

Defaults to True for 'stack' inserts, False for all others.

name = None
The name of the parameter to a *CustomBlockType*.

Not used for *BlockTypes*.

copy()

stringify(*value=None*, *block_plugin=False*)

options(*scriptable=None*)
Return a list of valid options to a menu insert, given a Scriptable for context.

Mostly complete, excepting 'attribute'.

class kurt.**BaseBlockType**(*shape*, *parts*)
Bases: object

Base for *BlockType* and *PluginBlockType*.

Defines common attributes.

SHAPE_FMTS = {'boolean': '<%s>', 'reporter': '(%s)'}

shape = None
The shape of the block. Valid values:

'stack' The default. Can connect to blocks above and below. Appear jigsaw-shaped.

'cap' Stops the script executing after this block. No blocks can be connected below them.

'hat' A block that starts a script, such as by responding to an event. Can connect to blocks below.

'reporter' Return a value. Can be placed into insert slots of other blocks as an argument to that block. Appear rounded.

'boolean' Like reporter blocks, but return a true/false value. Appear hexagonal.

"C"-shaped blocks with "mouths" for stack blocks, such as `"doIf"`, are specified by adding `Insert('stack')` to the end of *parts*.

parts = None
A list describing the text and arguments of the block.

Contains strings, which are part of the text displayed on the block, and *Insert* instances, which are arguments to the block.

text
The text displayed on the block.

String containing `"%s"` in place of inserts.

eg. `'say %s for %s secs'`

inserts
The type of each argument to the block.

List of *Insert* instances.

defaults
Default values for block inserts. (See *Block.args*.)

stripped_text
The *text*, with spaces and inserts removed.

Used by *BlockType.get* to look up blocks.

stringify(*args=None*, *block_plugin=False*, *in_insert=False*)

has_insert(*shape*)
Returns True if any of the inserts have the given shape.

class kurt.**BlockType**(*pbt*)
Bases: *kurt.BaseBlockType*

The specification for a type of *Block*.

These are initialiased by Kurt by combining *PluginBlockType* objects from individual format plugins to create a single *BlockType* for each command.

convert(*plugin=None*)
Return a *PluginBlockType* for the given plugin name.

If plugin is None, return the first registered plugin.

conversions
Return the list of *PluginBlockType* instances.

has_conversion(*plugin*)
Return True if the plugin supports this block.

has_command(*command*)
Returns True if any of the plugins have the given command.

shape

parts

classmethod get(*block_type*)
Return a *BlockType* instance from the given parameter.

- If it's already a BlockType instance, return that.
- If it exactly matches the command on a *PluginBlockType*, return the corresponding BlockType.
- If it loosely matches the text on a PluginBlockType, return the corresponding BlockType.
- If it's a PluginBlockType instance, look for and return the corresponding BlockType.

class kurt.**PluginBlockType**(*category*, *shape*, *command*, *parts*, *match=None*)
Bases: *kurt.BaseBlockType*

Holds plugin-specific *BlockType* attributes.

For each block concept, Kurt builds a single BlockType that references a corresponding PluginBlockType for each plugin that supports that block.

Note that whichever plugin is loaded first takes precedence.

format = None
The format plugin the block belongs to.

command = None
The method name from the source code, used to identify the block.

eg. `'say:duration:elapsed:from:'`

category = None
Where the block is found in the interface.

The same blocks may have different categories in different formats.

Possible values include:

```
'motion', 'looks', 'sound', 'pen', 'control', 'events', 'sensing',
'operators', 'data', 'variables', 'list', 'more blocks', 'motor',
'sensor', 'wedo', 'midi', 'obsolete'
```

copy()

class kurt.**CustomBlockType**(*shape*, *parts*)
Bases: *kurt.BaseBlockType*

A user-specified *BlockType*.

The script defining the custom block starts with:

```
kurt.Block("procDef", <CustomBlockType>)
```

And the scripts defininining the block follow.

The same CustomBlockType instance can then be used in a block in another script:

```
kurt.Block(<CustomBlocktype>, [args ...,])
```

is_atomic = None
True if the block should run without screen refresh.

class kurt.**Block**(*block_type*, **args*)
Bases: object

A statement in a graphical programming language. Blocks can connect together to form sequences of commands, which are stored in a *Script*. Blocks perform different commands depending on their type.

Parameters

- **type** – A *BlockType* instance, used to identify the command the block performs. Will also exact match a `command` or loosely match `text`.
- ***args** – List of the block's arguments. Arguments can be numbers, strings, Blocks, or lists of Blocks (for 'stack' shaped Inserts).

The following constructors are all equivalent:

```
>>> block = kurt.Block('say:duration:elapsed:from:', 'Hello!', 2)
>>> block = kurt.Block("say %s for %s secs", "Hello!", 2)
>>> block = kurt.Block("sayforsecs", "Hello!", 2)
```

Using BlockType:

```
>>> block.type
<kurt.BlockType('say [Hello!] for (2) secs', 'stack')>
>>> block.args
['Hello!', 2]
>>> block2 = kurt.Block(block.type, "Goodbye!", 5)
>>> block.stringify()
'say [Hello!] for (2) secs'
>>> block2.stringify()
'say [Goodbye!] for (5) secs'
```

type = None
BlockType instance. The command this block performs.

comment = None
The text of the comment attached to the block. Empty if no comment is attached.

Comments can only be attached to stack blocks.

args = None
List of arguments to the block.

The block's parameters are found in `type.inserts`. Default values come from `type.defaults <BlockType.defaults`.

copy()
Return a new Block instance with the same attributes.

stringify(*block_plugin=False*, *in_insert=False*)

class kurt.**Script**(*blocks=None*, *pos=None*)
Bases: object

A single sequence of blocks. Each *Scriptable* can have many Scripts.

The first block, `self.blocks[0]` is usually a "when" block, eg. an EventHatMorph.

Scripts implement the `list` interface, so can be indexed directly, eg. `script[0]`. All other methods like `append` also work.

blocks = None
The list of *Blocks*.

pos = None
`(x, y)` position from the top-left of the script area in pixels.

copy()
Return a new instance with the same attributes.

stringify(*block_plugin=False*)

class kurt.**Comment**(*text*, *pos=None*)
Bases: object

A free-floating comment in *Scriptable.scripts*.

text = None
The text of the comment.

pos = None
`(x, y)` position from the top-left of the script area in pixels.

copy()

stringify()

class kurt.**Costume**(*name*, *image*, *rotation_center=None*)
Bases: object

Describes the look of a sprite.

The raw image data is stored in *image*.

name = None
Name used by scripts to refer to this Costume.

rotation_center = None
`(x, y)` position from the top-left corner of the point about which the image rotates.

Defaults to the center of the image.

image = None
An *Image* instance containing the raw image data.

copy()
Return a new instance with the same attributes.

classmethod load(*path*)
Load costume from image file.

Uses *Image.load*, but will set the Costume's name based on the image filename.

save(*path*)
Save the costume to an image file at the given path.

Uses *Image.save*, but if the path ends in a folder instead of a file, the filename is based on the costume's *name*.

The image format is guessed from the extension. If path has no extension, the image's `format` is used.

Returns Path to the saved file.

resize(*size*)
Resize *image* in-place.

class kurt.**Image**(*contents*, *format=None*)
Bases: `object`

The contents of an image file.

Constructing from raw file contents:

```
Image(file_contents, "JPEG")
```

Constructing from a `PIL.Image.Image` instance:

```
pil_image = PIL.Image.new("RGBA", (480, 360))
Image(pil_image)
```

Loading from file path:

```
Image.load("path/to/image.jpg")
```

Images are immutable. If you want to modify an image, get a `PIL.Image.Image` instance from *pil_image*, modify that, and use it to construct a new Image. Modifying images in-place may break things.

The reason for having multiple constructors is so that kurt can implement lazy loading of image data – in many cases, a PIL image will never need to be created.

pil_image
A `PIL.Image.Image` instance containing the image data.

contents
The raw file contents as a string.

format
The format of the image file.

An uppercase string corresponding to the `PIL.ImageFile.ImageFile.format` attribute. Valid values include `"JPEG"` and `"PNG"`.

extension
The extension of the image's *format* when written to file.

eg `".png"`

size
`(width, height)` in pixels.

width

height

classmethod load(*path*)
Load image from file.

convert(**formats*)
Return an Image instance with the first matching format.

For each format in `*args`: If the image's *format* attribute is the same as the format, return self, otherwise try the next format.

If none of the formats match, return a new Image instance with the last format.

save(*path*)
Save image to file path.

The image format is guessed from the extension. If path has no extension, the image's *format* is used.

Returns Path to the saved file.

classmethod new(*size*, *fill*)
Return a new Image instance filled with a color.

resize(*size*)
Return a new Image instance with the given size.

paste(*other*)
Return a new Image with the given image pasted on top.

This image will show through transparent areas of the given image.

static image_format(*format_or_extension*)

static image_extension(*format_or_extension*)

class `kurt.`**Sound**(*name*, *waveform*)
Bases: `object`

A sound a *Scriptable* can play.

The raw sound data is stored in *waveform*.

name = None
Name used by scripts to refer to this Sound.

waveform = None
A *Waveform* instance containing the raw sound data.

copy()
Return a new instance with the same attributes.

classmethod load(*path*)
Load sound from wave file.

Uses *Waveform.load*, but will set the Waveform's name based on the sound filename.

save(*path*)
Save the sound to a wave file at the given path.

Uses *Waveform.save*, but if the path ends in a folder instead of a file, the filename is based on the project's *name*.

Returns Path to the saved file.

class kurt.**Waveform**(*contents*, *rate=None*, *sample_count=None*)
Bases: object

The contents of a wave file. Only WAV format files are supported.

Constructing from raw file contents:

```
Sound(file_contents)
```

Loading from file path:

```
Sound.load("path/to/sound.wav")
```

Waveforms are immutable.

extension = '.wav'

contents
The raw file contents as a string.

rate
The sampling rate of the sound.

sample_count
The number of samples in the sound.

classmethod load(*path*)
Load Waveform from file.

save(*path*)
Save waveform to file path as a WAV file.

Returns Path to the saved file.

CHAPTER 3

Plugin API

Writing plugins

To add support for a new file format, write a new *KurtPlugin* subclass:

```
import kurt
from kurt.plugin import Kurt, KurtPlugin

class MyScratchModPlugin(KurtPlugin):
    def load(self, fp):
        kurt_project = kurt.Project()
        # ... set kurt_project attributes ... #
        return kurt_project

    def save(self, fp, kurt_project):
        # ... save kurt_project attributes to file ...

Kurt.register(MyScratchModPlugin())
```

Take a look at `kurt.scratch20` for a more detailed example.

List available plugins

To get a list of the plugins registered with *Kurt*:

```
>>> kurt.plugin.Kurt.plugins.keys()
['scratch20', 'scratch14']
```

You should see your plugin in the output, unless you forgot to *register* it.

Notes

Some things to keep in mind:

- Most Scratch file formats have the *stage* as the base object – so project attributes, such as the notes and the list of sprites, are stored on the stage object.
- For Scratch, which doesn't support stage-specific variables, global variables and lists are stored on the Project, not the Stage.
- If your plugin contains obsolete blocks, they should be at the *end* of the `blocks` list. Otherwise things might not work properly.

KurtPlugin

class kurt.plugin.**KurtPlugin**
Bases: object

Handles a specific file format.

Loading and saving converts between a `Project`, kurt's internal representation, and a file of this format.

name = 'scratch14'
Short name of this file format, Python identifier style. Used internally by kurt.

Examples: `"scratch14"`, `"scratch20.sprite"`, `"byob3"`, `"snap"`

display_name = 'Scratch 2.0 Sprite'
Human-readable name of this file format. May be displayed to the user. Should not contain "Project" or "File".

Examples: `"Scratch 1.4"`, `"Scratch 2.0 Sprite"`, `"BYOB 3.1"`

extension = '.sb'
The extension used by this format, with leading dot.

Used by `Project.load` to recognise its files.

features = []
A list of the `Features` that the plugin supports.

blocks = []
The list of `PluginBlockType` objects supported by this plugin, in the order they appear in the program's interface.

load(*fp*)
Load a project from a file with this format.

`Project.path` will be set later. `Project.name` will be set to the filename of the path to the file if unset.

Parameters **fp** – A file pointer to the file, opened in binary mode.

Returns `Project`

save(*fp*, *project*)
Save a project to a file with this format.

Parameters

- **path** – A file pointer to the file, opened in binary mode.

- **project** – a Project

Kurt

class kurt.plugin.**Kurt**
Bases: object

The Kurt file format loader.

This class manages the registering and selection of file formats. Used by Project.

classmethod register(*plugin*)
Register a new *KurtPlugin*.

Once registered, the plugin can be used by Project, when:

- Project.load sees a file with the right extension
- Project.convert is called with the format as a parameter

classmethod get_plugin(*name=None*, ***kwargs*)
Returns the first format plugin whose attributes match kwargs.

For example:

```
get_plugin(extension="scratch14")
```

Will return the *KurtPlugin* whose *extension* attribute is "scratch14".

The *name* is used as the format parameter to Project.load and Project.save.

Raises ValueError if the format doesn't exist.

Returns *KurtPlugin*

classmethod block_by_command(*command*)
Return the block with the given command.

Returns None if the block is not found.

classmethod blocks_by_text(*text*)
Return a list of blocks matching the given text.

Capitalisation and spaces are ignored.

CHAPTER 4

Indices and tables

- genindex
- modindex
- search

Python Module Index

k

Index

T

U

V

W