CodeQuick Documentation

Release 0.9.0

William Forde

Mar 04, 2021

Contents

CHAPTER 1

Codequick

Codequick is a framework for kodi add-on's. The goal for this framework is to simplify add-on development. This is achieved by reducing the amount of boilerplate code to a minimum, while automating as many tasks that can be automated. Ultimately, allowing the developer to focus primarily on scraping content from websites and passing it to Kodi.

- Route dispatching (callbacks)
- Callback arguments can be any Python object that can be "pickled"
- Delayed execution (execute code after callbacks have returned results)
- No need to set "isplayable" or "isfolder" properties
- Supports both Python 2 and 3
- Auto sort method selection
- Better error reporting
- Full unicode support
- Sets "mediatype" to "video" or "music" depending on listitem type, if not set
- Sets "xbmcplugin.setContent", base off mediatype infolabel.
- Sets "xbmcplugin.setPluginCategory" to the title of current folder
- Sets "thumbnail" to add-on icon image, if not set
- Sets "fanart" to add-on fanart image, if not set
- Sets "icon" to "DefaultFolder.png" or "DefaultVideo.png', if not set
- Sets "plot" to the listitem title, if not set
- Auto type convertion for (str, unicode, int, float, long) infolables and stream info
- Support for media flags e.g. High definition '720p', audio channels '2.0'
- Reimplementation of the "listitem" class, that makes heavy use of dictionaries
- Youtube.DL intergration (https://forum.kodi.tv/showthread.php?tid=200877)

- URLQuick intergration (http://urlquick.readthedocs.io/en/stable/)
- Built-in support for saved searches
- Youtube intergration

CHAPTER 2

Contents

2.1 Tutorial

Here we will document the creation of an "add-on". In this instance, "plugin.video.metalvideo". This will be a simplified version of the full add-on that can be found over at: https://github.com/willforde/plugin.video.metalvideo

First of all, import the required "Codequick" components.

- *Route* will be used to list folder items.
- *Resolver* will be used to resolve video URLs.
- *Listitem* is used to create "items" within Kodi.
- *utils* is a module, containing some useful functions.
- *run* is the function that controls the execution of the add-on.

```
from codequick import Route, Resolver, Listitem, utils, run
```

Next we will import "urlquick", which is a "light-weight" HTTP client with a "requests" like interface, featuring caching support.

```
import urlquick
```

Now, we will use *utils.urljoin_partial* to create a URL constructor with the "base" URL of the site. This is use to convert relative URLs to absolute URLs. Normally HTML is full of relative URLs and this makes it easier to work with them, guaranteeing that you will always have an absolute URL to use.

```
# Base url constructor
url_constructor = utils.urljoin_partial("https://metalvideo.com")
```

Next we will create the "Root" function which will be the starting point for the add-on. It is very important that the "Root" function is called "root". This function will first have to be registered as a "callback" function. Since this is a function that will return "listitems", this will be registered as a *Route* callback. It is expected that a *Route* callback

should return a "generator" or "list", of *codequick.Listitem* objects. The first argument that will be passed to a *Route* callback, will always be the *Route* instance.

This "callback" will parse the list of "Music Video Categories" available on: http://metalvideo.com, This will return a "generator" of "listitems" linking to a sub-directory of videos within that category. Parsing of the HTML source will be done using "HTMLement" which is integrated into the "urlquick" request response.

See also:

URLQuick: http://urlquick.readthedocs.io/en/stable/

HTMLement: http://python-htmlement.readthedocs.io/en/stable/

```
@Route.register
def root(plugin):
    # Request the online resource
    url = url_constructor("/browse.html")
    resp = urlquick.get(url)

    # Filter source down to required section by giving the name and
    # attributes of the element containing the required data.
    # It's a lot faster to limit the parser to required section.
    root_elem = resp.parse("div", attrs={"id": "primary"})

    # Parse each category
    for elem in root_elem.iterfind("ul/li"):
        item = Listitem()

        # The image tag contains both the image url and title
        img = elem.find(".//img")

        # Set the thumbnail image
        item.art["thumb"] = img.get("src")

        # Set the title
        item.label = img.get("alt")

        # Fetch the url
        url = elem.find("div/a").get("href")

        # This will set the callback that will be called when listitem is activated.
        # 'video_list' is the route callback function that we will create later.
        # The 'url' argument is the url of the category that will be passed
        # to the 'video_list' callback.
        item.set_callback(video_list, url=url)

        # Return the listitem as a generator.
        yield item
```

Now, we can create the "video parser" callback that will return "playable" listitems. Since this is another function that will return listitems, it will be registered as a *Route* callback.

```
@Route.register
def video_list(plugin, url):
    # Request the online resource.
    url = url_constructor(url)
    resp = urlquick.get(url)

    # Parse the html source
```

(continues on next page)

(continued from previous page)

```
    root_elem = resp.parse("div", attrs={"class": "primary-content"})

    # Parse each video
    for elem in root_elem.find("ul").iterfind("./li/div"):
        item = Listitem()

        # Set the thumbnail image of the video.
        item.art["thumb"] = elem.find(".//img").get("src")

        # Set the duration of the video
        item.info["duration"] = elem.find("span/span/span").text.strip()

        # Set thel plot info
        item.info["plot"] = elem.find("p").text.strip()

        # Set view count
        views = elem.find("./div/span[@class='pm-video-attr-numbers']/small").text
        item.info["count"] = views.split(" ", 1)[0].strip()

        # Set the date that the video was published
        date = elem.find(".//time[@datetime]").get("datetime")
        date = date.split("T", 1)[0]
        item.info.date(date, "%Y-%m-%d")  # 2018-10-19

        # Url to video & name
        a_tag = elem.find("h3/a")
        url = a_tag.get("href")
        item.label = a_tag.text

        # Extract the artist name from the title
        item.info["artist"] = [a_tag.text.split("-", 1)[0].strip()]

        # 'play_video' is the resolver callback function that we will create later.
        # The 'url' argument is the url of the video that will be passed
        # to the 'play_video' resolver callback.
        item.set_callback(play_video, url=url)

        # Return the listitem as a generator.
        yield item

    # Extract the next page url if one exists.
    next_tag = root_elem.find("./div[@class='pagination pagination-centered']/ul")
    if next_tag is not None:
        # Find all page links
        next_tag = next_tag.findall("li/a")
        # Reverse list of links so the next page link should be the first item
        next_tag.reverse()
        # Attempt to find the next page link with the text of '>>'
        for node in next_tag:
            if node.text == u"\xbb":
                yield Listitem.next_page(url=node.get("href"), callback=video_list)
                # We found the link so we can now break from the loop
                break
```

Finally we need to create the *Resolver* "callback", and register it as so. This callback is expected to return a playable video URL. The first argument that will be passed to a *Resolver* callback, will always be a *Resolver* instance.

```
@Resolver.register
def play_video(plugin, url):
    # Sence https://metalvideo.com uses enbeaded youtube videos,
    # we can use 'plugin.extract_source' to extract the video url.
    url = url_constructor(url)
    return plugin.extract_source(url)
```

plugin.extract_source uses "YouTube.DL" to extract the video URL. Since it uses YouTube.DL, it will work with way-more than just youtube.

See also:

https://rg3.github.io/youtube-dl/supportedsites.html

So to finish, we need to initiate the "codequick" startup process. This will call the "callback functions" automatically for you.

```
if __name__ == "__main__":
    run()
```

2.2 API

Table of Contents.

2.2.1 Script

This module is used for creating "Script" callback's, which are also used as the base for all other types of callbacks.

codequick.**run**(*process_errors=True*, *redirect=None*)

The starting point of the add-on.

This function will handle the execution of the "callback" functions. The callback function that will be executed, will be auto selected.

The "root" callback, is the callback that will be the initial starting point for the add-on.

Parameters **process_errors** (*bool*) – Enable/Disable internal error handler. (default => True)

Returns Returns None if no errors were raised, or if errors were raised and process_errors is True (default) then the error Exception that was raised will be returned.

returns the error Exception if an error ocurred. :rtype: Exception or None

class codequick.script.**Script**

This class is used to create "Script" callbacks. Script callbacks are callbacks that just execute code and return nothing.

This class is also used as the base for all other types of callbacks i.e. *codequick.Route* and *codequick.Resolver*.

handle = -1

The Kodi handle that this add-on was started with.

CRITICAL = 50

Critical logging level, maps to "xbmc.LOGFATAL".

WARNING = 30
Critical logging level, maps to "xbmc.LOGWARNING".

ERROR = 40
Critical logging level, maps to "xbmc.LOGERROR".

DEBUG = 10
Critical logging level, maps to "xbmc.LOGDEBUG".

INFO = 20
Critical logging level, maps to "xbmc.LOGINFO".

NOTIFY_WARNING = 'warning'
Kodi notification warning image.

NOTIFY_ERROR = 'error'
Kodi notification error image.

NOTIFY_INFO = 'info'
Kodi notification info image.

setting = <codequick.script.Settings object>
Dictionary like interface of "add-on" settings. See *script.Settings* for more details.

logger = <Logger CodeQuick (DEBUG)>
Underlining logger object, for advanced use. See logging.Logger for more details.

params = {}
Dictionary of all callback parameters, for advanced use.

classmethod ref(*path*)
When given a path to a callback function, will return a reference to that callback function.

This is used as a way to link to a callback without the need to import it first. With this only the required module containing the callback is imported when callback is executed. This can be used to improve performance when dealing with lots of different callback functions.

The path structure is '/<package>/<module>:function' where 'package' is the full package path. 'module' is the name of the modules containing the callback. And 'function' is the name of the callback function.

Example

```
>>> from codequick import Route, Resolver, Listitem
>>> item = Listitem()
>>>
>>> # Example of referencing a Route callback
>>> item.set_callback(Route.ref("/resources/lib/videos:video_
↪list"))
>>>
>>> # Example of referencing a Resolver callback
>>> item.set_callback(Resolver.ref("/resources/lib/
↪resolvers:play_video"))
```

Parameters **path** (*str*) – The path to a callback function.
Returns A callback reference object.

classmethod register(*func=None*, ***kwargs*)
Decorator used to register callback functions.

Can be called with or without arguments. If arguments are given, they have to be "keyword only" arguments. The keyword arguments are parameters that are used by the plugin class instance. e.g. autosort=False to disable auto sorting for Route callbacks

Example

```
>>> from codequick import Route, Listitem
>>>
>>> @Route.register
>>> def root(_):
>>>     yield Listitem.from_dict("Extra videos", subfolder)
>>>
>>> @Route.register(cache_ttl=240, autosort=False, content_
↪type="videos")
>>> def subfolder(_):
>>>     yield Listitem.from_dict("Play video", "http://www.
↪example.com/video1.mkv")
```

Parameters
- **func** (*function*) – The callback function to register.
- **kwargs** – Keyword only arguments to pass to callback handler.

Returns A callback instance.
Return type Callback

static register_delayed(*func*, **args*, ***kwargs*)

Registers a function that will be executed after Kodi has finished listing all "listitems". Since this function is called after the listitems has been shown, it will not slow down the listing of content. This is very useful for fetching extra metadata for later use.

Note: Functions will be called in reverse order to the order they are added (LIFO).

Parameters
- **func** – Function that will be called after "xbmcplugin.endOfDirectory" is called.
- **args** – "Positional" arguments that will be passed to function.
- **kwargs** – "Keyword" arguments that will be passed to function.

Note: There is one optional keyword only argument `function_type`. Values are as follows. * `0` Only run if no errors are raised. (Default) * `1` Only run if an error has occurred. * `2` Run regardless if an error was raised or not.

Note: If there is an argument called exception in the delayed function callback and an error was raised, then that exception argument will be set to the raised exception object. Otherwise it will be set to None.

---

static log(*msg*, *args=None*, *lvl=10*)

Logs a message with logging level of "lvl".

Logging Levels.
- *Script.DEBUG*
- *Script.INFO*
- *Script.WARNING*
- *Script.ERROR*
- *Script.CRITICAL*

Parameters
- **msg** (*str*) – The message format string.
- **args** (*list or tuple*) – List of arguments which are merged into msg using the string formatting operator.

- **lvl** (*int*) – The logging level to use. default => 10 (Debug).

Note: When a log level of 50(CRITICAL) is given, all debug messages that were previously logged will now be logged as level 30(WARNING). This allows for debug messages to show in the normal Kodi log file when a CRITICAL error has occurred, without having to enable Kodi's debug mode.

---

static notify(*heading*, *message*, *icon=None*, *display_time=5000*, *sound=True*)

Send a notification to Kodi.

Options for icon are.

- *Script.NOTIFY_INFO*
- *Script.NOTIFY_ERROR*
- *Script.NOTIFY_WARNING*

Parameters

- **heading** (*str*) – Dialog heading label.
- **message** (*str*) – Dialog message label.
- **icon** (*str*) – [opt] Icon image to use. (default => 'add-on icon image')
- **display_time** (*int*) – [opt] Ttime in "milliseconds" to show dialog. (default => 5000)
- **sound** (*bool*) – [opt] Whether or not to play notification sound. (default => True)

static localize(*string_id*)

Returns a translated UI string from addon localization files.

Note: *utils.string_map* needs to be populated before you can pass in a string as the reference.

Parameters **string_id** (*str or int*) – The numeric ID or gettext string ID of the localized string

Returns Localized unicode string.

Return type str

Raises **Keyword** – if a gettext string ID was given but the string is not found in English `strings.po`.

Example

```
>>> Script.localize(30001)
"Toutes les vidéos"
>>> Script.localize("All Videos")
"Toutes les vidéos"
```

static get_info(*key*, *addon_id=None*)

Returns the value of an add-on property as a unicode string.

Properties.

- author
- changelog
- description
- disclaimer
- fanart
- icon
- id
- name

- path
- profile
- stars
- summary
- type
- version

Parameters
- **key** (*str*) – "Name" of the property to access.
- **addon_id** (*str*) – [opt] ID of another add-on to extract properties from.

Returns Add-on property as a unicode string.
Return type str
Raises **RuntimeError** – If add-on ID is given and there is no add-on with given ID.

class codequick.script.**Settings**
Settings class to handle the getting and setting of "add-on" settings.

__getitem__(*key*)
Returns the value of a setting as a "unicode string".
Parameters **key** (*str*) – ID of the setting to access.
Returns Setting as a "unicode string".
Return type str

__setitem__(*key*, *value*)
Set add-on setting.
Parameters
- **key** (*str*) – ID of the setting.
- **value** (*str*) – Value of the setting.

static get_string(*key*, *addon_id=None*)
Returns the value of a setting as a "unicode string".
Parameters
- **key** (*str*) – ID of the setting to access.
- **addon_id** (*str*) – [opt] ID of another add-on to extract settings from.

Raises **RuntimeError** – If `addon_id` is given and there is no add-on with given ID.
Returns Setting as a "unicode string".
Return type str

static get_boolean(*key*, *addon_id=None*)
Returns the value of a setting as a "Boolean".
Parameters
- **key** (*str*) – ID of the setting to access.
- **addon_id** (*str*) – [opt] ID of another add-on to extract settings from.

Raises **RuntimeError** – If `addon_id` is given and there is no add-on with given ID.
Returns Setting as a "Boolean".
Return type bool

static get_int(*key*, *addon_id=None*)
Returns the value of a setting as a "Integer".
Parameters
- **key** (*str*) – ID of the setting to access.
- **addon_id** (*str*) – [opt] ID of another add-on to extract settings from.

Raises **RuntimeError** – If `addon_id` is given and there is no add-on with given ID.
Returns Setting as a "Integer".

Return type int

static get_number(*key*, *addon_id=None*)
Returns the value of a setting as a "Float".

Parameters
- **key** (*str*) – ID of the setting to access.
- **addon_id** (*str*) – [opt] ID of another addon to extract settings from.

Raises **RuntimeError** – If addon_id is given and there is no addon with given ID.

Returns Setting as a "Float".

Return type float

2.2.2 Route

This module is used for the creation of "Route callbacks".

class codequick.route.**Route**
This class is used to create "Route" callbacks. "Route" callbacks, are callbacks that return "listitems" which will show up as folders in Kodi.

Route inherits all methods and attributes from *codequick.Script*.

The possible return types from Route Callbacks are.

- iterable: "List" or "tuple", consisting of *codequick.listitem* objects.
- generator: A Python "generator" that return's *codequick.listitem* objects.
- False: This will cause the "plugin call" to quit silently, without raising a RuntimeError.

Raises **RuntimeError** – If no content was returned from callback.

Example

```
>>> from codequick import Route, Listitem
>>>
>>> @Route.register
>>> def root(_):
>>>     yield Listitem.from_dict("Extra videos", subfolder)
>>>     yield Listitem.from_dict("Play video", "http://www.
↪example.com/video1.mkv")
>>>
>>> @Route.register
>>> def subfolder(_):
>>>     yield Listitem.from_dict("Play extra video", "http://
↪www.example.com/video2.mkv")
```

autosort = True
Set to False to disable auto sortmethod selection.

Note: If autosort is disabled and no sortmethods are given, then SORT_METHOD_UNSORTED will be set.

---

category
Manualy specifiy the category for the current folder view. Equivalent to setting xbmcplugin.setPluginCategory()

redirect_single_item = False
When this attribute is set to `True` and there is only one folder listitem available in the folder view, then that listitem will be automaticly called for you.

update_listing = False
When set to `True`, the current page of listitems will be updated, instead of creating a new page of listitems.

content_type = None
The add-on's "content type".

If not given, then the "content type" is based on the "mediatype" infolabel of the listitems. If the "mediatype" infolabel" was not set, then it defaults to "files/videos", based on type of content.
- "files" when listing folders.
- "videos" when listing videos.

See also:

The full list of "content types" can be found at:

https://codedocs.xyz/xbmc/xbmc/group__python__xbmcplugin.html#gaa30572d1e5d9d589e1cd3bfc1e2318d6

add_sort_methods(**methods*, ***kwargs*)
Add sorting method(s).

Any number of sort method's can be given as multiple positional arguments. Normally this should not be needed, as sort method's are auto detected.

You can pass an optional keyword only argument, 'disable_autosort' to disable auto sorting.
Parameters **methods** (*int*) – One or more Kodi sort method's.
See also:

The full list of sort methods can be found at.

https://codedocs.xyz/xbmc/xbmc/group__python__xbmcplugin.html#ga85b3bff796fd644fb28f87b136025f40

2.2.3 Resolver

This module is used for the creation of "Route callbacks".

class codequick.resolver.**Resolver**
This class is used to create "Resolver" callbacks. Resolver callbacks are callbacks that return playable video URL's which Kodi can play.

Resolver inherits all methods and attributes from *script.Script*.

The possible return types from Resolver Callbacks are.

- `str`: URL as type "str".
- `iterable`: "List" or "tuple", consisting of URL's, "listItem's" or a "tuple" consisting of (title, URL).
- `dict`: "Dictionary" consisting of "title" as the key and the URL as the value.
- `listItem`: A *codequick.Listitem* object with required data already set e.g. "label" and "path".
- `generator`: A Python "generator" that return's one or more URL's.
- `False`: This will cause the "resolver call" to quit silently, without raising a RuntimeError.

Note: If multiple URL's are given, a playlist will be automaticly created.

Raises

- **RuntimeError** – If no content was returned from callback.
- **ValueError** – If returned url is invalid.

Example

```
>>> from codequick import Resolver, Route, Listitem
>>>
>>> @Route.register
>>> def root(_):
>>>     yield Listitem.from_dict("Play video", play_video,
>>>           params={"url": "https://www.youtube.com/watch?
↪v=RZuVTOk6ePM"})
>>>
>>> @Resolver.register
>>> def play_video(plugin, url):
>>>     # Extract a playable video url using youtubeDL
>>>     return plugin.extract_source(url)
```

extract_source(*url*, *quality=None*, ***params*)

Extract video URL using "YouTube.DL".

YouTube.DL provides access to hundreds of sites.

See also:

The list of supported sites can be found at:

https://rg3.github.io/youtube-dl/supportedsites.html

Quality options are.

- 0 = SD,
- 1 = 720p,
- 2 = 1080p,
- 3 = Highest Available

Parameters

- **url** (*str*) – URL of the video source, where the playable video can be extracted from.
- **quality** (*int*) – [opt] Override YouTube.DL's quality setting.
- **params** – Optional "Keyword" arguments of YouTube.DL parameters.

Returns The playable video url

Return type str

See also:

The list of available parameters can be found at.

https://github.com/rg3/youtube-dl#options

2.2.4 Listitem

The "listitem" control is used for the creating of item lists in Kodi.

class codequick.listing.**Listitem**(*content_type='video'*)

The "listitem" control is used for the creating "folder" or "video" items within Kodi.

Parameters **content_type** (*str*) – [opt] Type of content been listed. e.g. "video", "music", "pictures".

subtitles = list()

List of paths to subtitle files.

art = Art()

Dictionary like object for adding "listitem art". See *listing.Art* for more details.

info = Info()

Dictionary like object for adding "infoLabels". See *listing.Info* for more details.

stream = Stream()

Dictionary like object for adding "stream details". See *listing.Stream* for more details.

context = Context()

List object for "context menu" items. See *listing.Context* for more details.

property = dict()

Dictionary like object that allows you to add "listitem properties". e.g. "StartOffset".

Some of these are processed internally by Kodi, such as the "StartOffset" property, which is the offset in seconds at which to start playback of an item. Others may be used in the skin to add extra information, such as "WatchedCount" for tvshow items.

Examples

```
>>> item = Listitem()
>>> item.property['StartOffset'] = '256.4'
```

params = dict()

Dictionary like object for parameters that will be passed to the "callback" function.

Example

```
>>> item = Listitem()
>>> item.params['videoid'] = 'kqmdIV_gBfo'
```

listitem = None

The underlining kodi listitem object, for advanced use.

label

The listitem label property.

Example

```
>>> item = Listitem()
>>> item.label = "Video Title"
```

set_path(*path*, *is_folder=False*, *is_playable=True*)

Set the listitem's path.

The path can be any of the following:

- Any kodi path, e.g. "plugin://" or "script://"
- Directly playable URL or filepath.

Note: When specifying a external 'plugin' or 'script' as the path, Kodi will treat it as a playable item. To override this behavior, you can set the `is_playable` and `is_folder` parameters.

Parameters

- **path** – A playable URL or plugin/script path.
- **is_folder** – Tells kodi if path is a folder (default -> `False`).
- **is_playable** – Tells kodi if path is a playable item (default -> `True`).

set_callback(*callback*, **args*, ***kwargs*)

Set the "callback" function for this listitem.

The "callback" parameter can be any of the following:

- *codequick.Script* callback.
- *codequick.Route* callback.
- *codequick.Resolver* callback.
- A callback reference object *Script.ref*.

Parameters

- **callback** – The "callback" function or reference object.
- **args** – "Positional" arguments that will be passed to the callback.
- **kwargs** – "Keyword" arguments that will be passed to the callback.

classmethod from_dict(*callback*, *label*, *art=None*, *info=None*, *stream=None*, *context=None*, *properties=None*, *params=None*, *subtitles=None*)

Constructor to create a "listitem".

This method will create and populate a listitem from a set of given values.

Parameters

- **callback** (*Callback*) – The "callback" function or playable URL.
- **label** (*str*) – The listitem's label.
- **art** (*dict*) – Dictionary of listitem art.
- **info** (*dict*) – Dictionary of infoLabels.
- **stream** (*dict*) – Dictionary of stream details.
- **context** (*list*) – List of "context menu" item(s) containing "tuples" of ("label", "command") pairs.
- **properties** (*dict*) – Dictionary of listitem properties.
- **params** (*dict*) – Dictionary of parameters that will be passed to the "callback" function.
- **subtitles** (*list*) – List of paths to subtitle files.

Returns A listitem object.

Return type *Listitem*

Example

```
>>> params = {"url": "http://example.com"}
>>> item = {"label": "Video Title", "art": {"thumb": "http://
↪example.com/image.jpg"}, "params": params}
>>> listitem = Listitem.from_dict(**item)
```

classmethod next_page(**args*, ***kwargs*)

Constructor for adding link to "Next Page" of content.

By default the current running "callback" will be called with all of the parameters that are given here. You can specify which "callback" will be called by setting a keyword only argument called 'callback'.

Parameters

- **args** – "Positional" arguments that will be passed to the callback.
- **kwargs** – "Keyword" arguments that will be passed to the callback.

Example

```
>>> item = Listitem()
>>> item.next_page(url="http://example.com/videos?page2")
```

classmethod recent(*callback*, **args*, ***kwargs*)
Constructor for adding "Recent Videos" folder.

This is a convenience method that creates the listitem with "name", "thumbnail" and "plot", already preset.

Parameters

- **callback** (*Callback*) – The "callback" function.
- **args** – "Positional" arguments that will be passed to the callback.
- **kwargs** – "Keyword" arguments that will be passed to the callback.

classmethod search(*callback*, **args*, ***kwargs*)
Constructor to add "saved search" support to add-on.

This will first link to a "sub" folder that lists all saved "search terms". From here, "search terms" can be created or removed. When a selection is made, the "callback" function that was given will be executed with all parameters forwarded on. Except with one extra parameter, `search_query`, which is the "search term" that was selected.

Parameters

- **callback** (*Callback*) – Function that will be called when the "listitem" is activated.
- **args** – "Positional" arguments that will be passed to the callback.
- **kwargs** – "Keyword" arguments that will be passed to the callback.

classmethod youtube(*content_id*, *label=None*, *enable_playlists=True*)
Constructor to add a "YouTube channel" to add-on.

This listitem will list all videos from a "YouTube", channel or playlist. All videos will have a "Related Videos" option via the context menu. If `content_id` is a channel ID and `enable_playlists` is `True`, then a link to the "channel playlists" will also be added to the list of videos.

Parameters

- **content_id** (*str*) – Channel ID or playlist ID, of video content.
- **label** (*str*) – [opt] Listitem Label. (default => "All Videos").
- **enable_playlists** (*bool*) – [opt] Set to `False` to disable linking to channel playlists. (default => `True`)

Example

```
>>> item = Listitem()
>>> item.youtube("UC4QZ_LsYcvcq7qOsOhpAX4A")
```

class codequick.listing.**Art**
Dictionary like object, that allows you to add various images. e.g. "thumb", "fanart".

if "thumb", "fanart" or "icon" is not set, then they will be set automaticly based on the add-on's fanart and icon images if available.

Note: The automatic image values can be disabled by setting them to an empty string. e.g. item.art.thumb = "".

Expected art values are.

- thumb
- poster
- banner
- fanart

- clearart
- clearlogo
- landscape
- icon

Example

```
>>> item = Listitem()
>>> item.art.icon = "http://www.example.ie/icon.png"
>>> item.art["fanart"] = "http://www.example.ie/fanart.jpg"
>>> item.art.local_thumb("thumbnail.png")
```

local_thumb(*image*)
Set the "thumbnail" image to a image file, located in the add-on "resources/media" directory.
Parameters **image** (*str*) – Filename of the image.

global_thumb(*image*)
Set the "thumbnail" image to a image file, located in the codequick "resources/media" directory.
The available global thumbnail images are.

- next.png - Arrow pointing to the right.
- videos.png - Circle with a play button in the middle.
- search.png - An image of a magnifying glass.
- search_new.png - A magnifying glass with plus symbol in the middle.
- playlist.png - Image of three bulleted lines.
- recent.png - Image of a clock.

Parameters **image** (*str*) – Filename of the image.

class codequick.listing.**Info**
Dictionary like object, that allow's you to add listitem "infoLabels".

"InfoLabels" are like metadata for listitems. e.g. "duration", "genre", "size", "rating" and or "plot". They are also used for sorting purpose's, sort methods will be automatically selected.

Some "infolabels" need to be of a given type e.g. "size" as "long", "rating" as "float". For the most part, this conversion will be done automatically.

Example of what would happen is.

- "duration" would be converted to `int` and "xbmcplugin.SORT_METHOD_VIDEO_RUNTIME" sort method will be selected.
- "size" would be converted to `long` and "xbmcplugin.SORT_METHOD_SIZE" sort method will be selected.

See also:

The full list of listitem "infoLabels" can be found at:

https://codedocs.xyz/xbmc/xbmc/group__python__xbmcgui__listitem.html#ga0b71166869bda87ad744942888fb5f14

Note: Duration infolabel value can be either in "seconds" or as a "hh:mm:ss" string.

Examples

```
>>> item = Listitem()
>>> item.info.genre = "Science Fiction"
>>> item.info["size"] = 256816
```

date(*date*, *date_format*)

Set the date infolabel.

Parameters

- **date** (*str*) – The date for the listitem.
- **date_format** (*str*) – The format of the date as a strftime directive e.g. "june 27, 2017" => "%B %d, %Y"

See also:

The full list of directives can be found at:

https://docs.python.org/3.6/library/time.html#time.strftime

Example

```
>>> item = Listitem()
>>> item.info.date('june 27, 2017', '%B %d, %Y')
```

class codequick.listing.**Stream**

Dictionary like object, that allows you to add "stream details". e.g. "video_codec", "audio_codec".

Expected stream values are.

- video_codec - str (h264)
- aspect - float (1.78)
- width - integer (1280)
- height - integer (720)
- channels - integer (2)
- audio_codec - str (AAC)
- audio_language - str (en)
- subtitle_language - str (en)

Type convertion will be done automatically, so manual convertion is not required.

Example

```
>>> item = Listitem()
>>> item.stream.video_codec = "h264"
>>> item.stream.audio_codec = "aac"
```

hd(*quality*, *aspect=None*)

Convenient method to set required stream info to show "SD/HD/4K" logos.

The values witch are set are "width", "height" and "aspect". If no aspect ratio is given, then a ratio of 1.78(16:9) is set when the quality is 720p or greater.

Quality options are.

- 0 = 480p
- 1 = 720p
- 2 = 1080p
- 3 = 4K.

Parameters

- **quality** (*int or None*) – Quality of the stream.
- **aspect** (*float*) – [opt] The "aspect ratio" of the video.

Example

```
>>> item = Listitem()
>>> item.stream.hd(2, aspect=1.78) # 1080p
```

class codequick.listing.**Context**

Adds item(s) to the context menu of the listitem.

This is a list containing "tuples" consisting of ("label", "command") pairs.

This class inherits all methods and attributes from the build-in data type `list`.

See also:

The full list of built-in functions can be found at:

http://kodi.wiki/view/List_of_Built_In_Functions

related(*callback*, **args*, ***kwargs*)

Convenient method to add a "Related Videos" context menu item.

All this really does is to call "context.container" and sets "label" for you.

Parameters

- **callback** (*Callback*) – The function that will be called when menu item is activated.
- **args** – [opt] "Positional" arguments that will be passed to the callback.
- **kwargs** – [opt] "Keyword" arguments that will be passed to the callback.

container(*callback*, *label*, **args*, ***kwargs*)

Convenient method to add a context menu item that links to a "container".

Parameters

- **callback** (*Callback*) – The function that will be called when menu item is activated.
- **label** (*str*) – The label of the context menu item.
- **args** – [opt] "Positional" arguments that will be passed to the callback.
- **kwargs** – [opt] "Keyword" arguments that will be passed to the callback.

script(*callback*, *label*, **args*, ***kwargs*)

Convenient method to add a context menu item that links to a "script".

Parameters

- **callback** (*Callback*) – The function that will be called when menu item is activated.
- **label** (*str or unicode*) – The label of the context menu item.
- **args** – [opt] "Positional" arguments that will be passed to the callback.
- **kwargs** – [opt] "Keyword" arguments that will be passed to the callback.

2.2.5 Storage

Persistent data storage objects. These objects will act like normal built-in data types, except all data will be saved to disk for later access when flushed.

class codequick.storage.**PersistentDict**(*name*, *ttl=None*)

Persistent storage with a `dictionary` like interface.

Parameters

- **name** (*str*) – Filename or path to storage file.

- **ttl** (*int*) – [opt] The amount of time in "seconds" that a value can be stored before it expires.

Note: `name` can be a filename, or the full path to a file. The add-on profile directory will be the default location for files, unless a full path is given.

Note: If the `ttl` parameter is given, "any" expired data will be removed on initialization.

Note: This class is also designed as a "Context Manager".

Note: Data will only be synced to disk when connection to file is "closed" or when "flush" method is explicitly called.

Example

```
>>> with PersistentDict("dictfile.pickle") as db:
>>>     db["testdata"] = "testvalue"
>>>     db.flush()
```

flush()
Synchronize data back to disk.

Data will only be written to disk if content has changed.

close()
Flush content to disk & close file object.

items() → a set-like object providing a view on D's items

class codequick.storage.**PersistentList**(*name*, *ttl=None*)
Persistent storage with a `list` like interface.

Parameters

- **name** (*str*) – Filename or path to storage file.
- **ttl** (*int*) – [opt] The amount of time in "seconds" that a value can be stored before it expires.

Note: `name` can be a filename, or the full path to a file. The add-on profile directory will be the default location for files, unless a full path is given.

Note: If the `ttl` parameter is given, "any" expired data will be removed on initialization.

Note: This class is also designed as a "Context Manager".

Note: Data will only be synced to disk when connection to file is "closed" or when "flush" method is explicitly called.

Example

```
>>> with PersistentList("listfile.pickle") as db:
>>>     db.append("testvalue")
>>>     db.extend(["test1", "test2"])
>>>     db.flush()
```

flush()
Synchronize data back to disk.

Data will only be written to disk if content has changed.

close()
Flush content to disk & close file object.

insert(*index*, *value*)
S.insert(index, value) – insert value before index

append(*value*)
S.append(value) – append value to the end of the sequence

2.2.6 Utils

A collection of useful functions.

codequick.utils.**string_map = {}**
Dict of localized string references used in conjunction with *Script.localize*. Allowing you to use the string as the localized string reference.

Note: It is best if you set the string references at the top of your add-on python file.

Example

```
>>> Script.localize(30001)
"Toutes les vidéos"
>>>
>>> # Add reference id for "All Videos" so you can use the␣
↪string name instead.
>>> utils.string_map["All Videos": 30001]
>>> Script.localize("All Videos")
"Toutes les vidéos"
```

codequick.utils.**keyboard**(*heading*, *default=''*, *hidden=False*)
Show a keyboard dialog.

Parameters

- **heading** (*str*) – Keyboard heading.
- **default** (*str*) – [opt] Default text.

- **hidden** (*bool*) – [opt] `True` for hidden text entry.

Returns Returns the user input as unicode.

Return type str

codequick.utils.**parse_qs**(*qs*, *keep_blank_values=False*, *strict_parsing=False*)

Parse a "urlencoded" query string, and return the data as a dictionary.

Parse a query string given as a string or unicode argument (data of type application/x-www-form-urlencoded). Data is returned as a dictionary. The dictionary keys are the "Unique" query variable names and the values are "Unicode" values for each name.

The optional argument `keep_blank_values`, is a flag indicating whether blank values in percent-encoded queries should be treated as a blank string. A `True` value indicates that blanks should be retained as a blank string. The default `False` value indicates that blank values are to be ignored and treated as if they were not included.

The optional argument `strict_parsing`, is a flag indicating what to do with parsing errors. If `False` (the default), errors are silently ignored. If `True`, errors raise a "ValueError" exception.

Parameters

- **qs** (*str*) – Percent-encoded "query string" to be parsed, or a URL with a "query string".
- **keep_blank_values** (*bool*) – `True` to keep blank values, else discard.
- **strict_parsing** (*bool*) – `True` to raise "ValueError" if there are parsing errors, else silently ignore.

Returns Returns a dictionary of key/value pairs, with all keys and values as "Unicode".

Return type dict

Raises **ValueError** – If duplicate query field names exists or if there is a parsing error.

Example

```
>>> parse_qs("http://example.com/path?q=search&safe=no")
{u"q": u"search", u"safe": u"no"}
>>> parse_qs(u"q=search&safe=no")
{u"q": u"search", u"safe": u"no"}
```

codequick.utils.**urljoin_partial**(*base_url*)

Construct a full (absolute) URL by combining a base URL with another URL.

This is useful when parsing HTML, as the majority of links would be relative links.

Informally, this uses components of the base URL, in particular the addressing scheme, the network location and (part of) the path, to provide missing components in the relative URL.

Returns a new "partial" object which when called, will pass `base_url` to `urlparse.urljoin()` along with the supplied relative URL.

Parameters **base_url** (*str*) – The absolute URL to use as the base.

Returns A partial function that accepts a relative URL and returns a full absolute URL.

Example

```
>>> url_constructor = urljoin_partial("https://google.ie/")
>>> url_constructor("/path/to/something")
"https://google.ie/path/to/something"
```

(continues on next page)

(continued from previous page)

```
>>> url_constructor("/gmail")
"https://google.ie/gmail"
```

codequick.utils.**strip_tags**(*html*)

Strips out HTML tags and return plain text.

Parameters **html** (*str*) – HTML with text to extract.

Returns Html with tags striped out

Return type str

Example

```
>>> strip_tags('<a href="http://example.com/">I linked to <i>
↪example.com</i></a>')
"I linked to example.com"
```

codequick.utils.**ensure_native_str**(*data*, *encoding='utf8'*)

Ensures that the given string is returned as a native str type, `bytes` on Python 2, `unicode` on Python 3.

Parameters

- **data** – String to convert if needed.
- **encoding** (*str*) – [opt] The encoding to use if needed..

Returns The given string as a native `str` type.

Return type str

codequick.utils.**ensure_unicode**(*data*, *encoding='utf8'*)

Ensures that the given string is return as type `unicode`.

Parameters

- **data** (*str or bytes*) – String to convert if needed.
- **encoding** (*str*) – [opt] The encoding to use if needed..

Returns The given string as type `unicode`.

Return type str

codequick.utils.**bold**(*text*)

Return Bolded text.

Parameters **text** (*str*) – Text to bold.

Returns Bolded text.

Return type str

codequick.utils.**italic**(*text*)

Return Italic text.

Parameters **text** (*str*) – Text to italic.

Returns Italic text.

Return type str

codequick.utils.**color**(*text*, *color_code*)

Return Colorized text of givin color.

Parameters

- **text** (*str*) – Text to italic.
- **color_code** (*str*) – Color to change text to.

Returns Colorized text.

Return type str

2.3 Examples

There are no better examples than actual add-on's that use the codequick framework. So here is a list of add-ons that use this framework.

- https://github.com/willforde/plugin.video.watchmojo
- https://github.com/willforde/plugin.video.earthtouch
- https://github.com/willforde/plugin.video.metalvideo
- https://github.com/willforde/plugin.video.science.friday

Python Module Index

C

Index

Symbols

A

B

C

D

E

F

G

H

I

K

L

N

P

R

S

U

W

Y