micropy-cli

Release 3.4.0

Braden Mars

Sep 28, 2020

DOCUMENTATION

Micropy Cli is a project management/generation tool for writing Micropython code in modern IDEs such as VSCode. Its primary goal is to automate the process of creating a workspace complete with:

- **Linting** compatible with Micropython
- VSCode **Intellisense**
- **Autocompletion**
- Dependency Management
- VCS Compatibility

CHAPTER

ONE

INSTALLATION

You can download and install the latest version of this software from the Python package index (PyPI) as follows:

```
pip install --upgrade micropy-cli
```

1.1 Installation

You can download and install the latest version of this software from the Python package index (PyPI) as follows:

```
pip install --upgrade micropy-cli
```

If applicable, you can test out a pre-release by executing:

```
pip install --upgrade --pre micropy-cli
```

1.2 Getting Started

1.2.1 Creating a Project

Creating a new project folder is as simple as:

1. Executing `micropy init <PROJECT NAME>`
2. Selecting which features to enable
3. Selecting your target device/firmware
4. Boom. Your workspace is ready.

1.2.2 Micropy Project Environment

When creating a project with `micropy-cli`, two special items are added:

- A `.micropy/` folder
- A `micropy.json` file

The `.micropy/` contains symlinks from your project to your `$HOME/.micropy/stubs` folder. By doing this, micropy can reference the required stub files for your project as relative to it, rather than using absolute paths to `$HOME/.micropy`. How does this benefit you? Thanks to this feature, you can feel free to push common setting files such as `settings.json` and `.pylint.rc` to your remote git repository. This way, others who clone your repo can achieve a matching workspace in their local environment.

> Note: The generated `.micropy/` folder should be *IGNORED* by your VCS. It is created locally for each environment via the `micropy.json` file.

The `micropy.json` file contains information micropy needs in order to resolve your projects required files when other clone your repo. Think of it as a `package.json` for micropython.

1.2.3 Cloning a Micropy Environment

To setup a Micropy environment locally, simply:

- Install `micropy-cli`
- Navigate to the project directory
- Execute `micropy`

Micropy will automatically configure and install any stubs required by a project thanks to its `micropy.json` file.

1.2.4 Project Dependencies

While all modules that are included in your targeted micropython firmware are available with autocompletion, intellisense, and linting, most projects require external dependencies.

Currently, handling dependencies with micropython is a bit tricky. Maybe you can install a cpython version of your requirement? Maybe you could just copy and paste it? What if it needs to be frozen?

Micropy handles all these issues for you automatically. Not only does it track your project's dependencies, it keeps both `requirements.txt` and `dev-requirements.txt` updated, enables autocompletion/intellisense for each dep, and allows you to import them just as you would on your device.

This allows you to include your requirement however you want, whether that be as a frozen module in your custom built firmware, or simply in the `/lib` folder on your device.

Installing Packages

To add a package as a requirement for your project, run:

```
micropy install <PACKAGE_NAMES>
```

while in your project's root directory.

This will automatically execute the following:

- Source `PACKAGE_NAMES` from pypi, as a url, or a local path
- Retrieve the module/package and stub it, adding it to your local `.micropy` folder.
- Add requirement to your `micropy.json`
- Update `requirements.txt`

To install dev packages that are not needed on your device, but are needed for local development, add the `--dev` flag. This will do everything above **except** stub the requirement.

You can also install all requirements found in `micropy.json`/`requirements.txt`/`dev-requirements.txt` by executing `micropy install` without passing any packages. Micropy will automatically do this when setting up a local environment of an existing micropy project.

Example

Lets say your new project will depend on picoweb and blynklib. Plus, you'd like to use rshell to communicate directly with your device. After creating your project via `micropy init`, you can install your requirements as so:

Now you or anybody cloning your project can import those requirements normally, and have the benefits of all the features micropy brings:

1.2.5 Stub Management

Stub files are the magic behind how micropy allows features such as linting, Intellisense, and autocompletion to work. To achieve the best results with MicropyCli, its important that you first add the appropriate stubs for the device/firmware your project uses.

> Note: When working in a micropy project, all stub related commands will also be executed on the active project. (i.e if in a project and you run `micropy stubs add <stub-name>`, then that stub retrieved AND added to the active project.)

Adding Stubs

Adding stubs to Micropy is a breeze. Simply run: `micropy stubs add <STUB_NAME>` By sourcing micropy-stubs, MicroPy has several premade stub packages to choose from.

These packages generally use the following naming schema:

`<device>-<firmware>-<version>`

For example, running `micropy stubs add esp32-micropython-1.11.0` will install the following:

- Micropython Specific Stubs
- ESP32 Micropython v1.11 Device Specific Stubs
- Frozen Modules for both device and firmware

You can search stubs that are made available to Micropy via `micropy stubs search <QUERY>`

Alternatively, using `micropy stubs add <PATH>`, you can manually add stubs to Micropy. For manual stub generation, please see Josvel/micropython-stubber.

Creating Stubs

Using `micropy stubs create <PORT/IP_ADDRESS>`, MicropyCli can automatically generate and add stubs from any Micropython device you have on hand. This can be done over both USB and WiFi.

> Note: For stub creation, micropy-cli has additional dependencies.
>
> These can be installed by executing: `pip install micropy-cli[create_stubs]`

Viewing Stubs

To list stubs you have installed, simply run `micropy stubs list`.

To search for stubs for your device, use `micropy stubs search <QUERY>`.

1.3 See Also

- VSCode IntelliSense, Autocompletion & Linting capabilities
  - An awesome article written by lemariva. It covers creating a micropython project environment from scratch using `micropy-cli` and pymakr-vsc. Great place to start if you're new to this!

1.4 Acknowledgements

1.4.1 Micropython-Stubber

Josvel/micropython-stubber

Josverl's Repo is full of information regarding Micropython compatibility with VSCode and more. To find out more about how this process works, take a look at it.

micropy-cli and micropy-stubs depend on micropython-stubber for its ability to generate frozen modules, create stubs on a pyboard, and more.

1.5 CLI Usage

1.5.1 micropy

CLI Application for creating/managing Micropython Projects.

```
micropy [OPTIONS] COMMAND [ARGS]...
```

Options

--version
Show the version and exit.

-s, --skip-checks
Skip Project Checks. Defaults to False.

Commands

init
Create new Micropython Project

install
Install Project Requirements

stubs
Manage Micropy Stubs

1.5.2 micropy init

Create new Micropython Project.

When creating a new project, all files will be placed under the generated <PROJECT_NAME> folder.

```
micropy init [OPTIONS] [PATH]
```

Options

-n, --name <name>
Project Name. Defaults to Path name.

-t, --template <template>
Templates to generate for project. Multiple options can be passed.

Options vscode|pymakr|pylint|gitignore|bootstrap

Arguments

PATH
Optional argument

1.5.3 micropy stubs

Manage Micropy Stubs.

Stub files are what enable linting,
Intellisense, Autocompletion, and more.

To achieve the best results, you can install
stubs specific to your device/firmware using:

micropy stubs add <STUB_NAME>

For more info, please check micropy stubs add –help

```
micropy stubs [OPTIONS] COMMAND [ARGS]...
```

add

Add Stubs from package or path.

In general, stub package names follow this schema:
<device>-<firmware>-<version>

For example:

esp32-micropython-1.11.0

You can search premade stub packages using:
micropy stubs search <QUERY>

Checkout the docs on Github for more info.

```
micropy stubs add [OPTIONS] STUB_NAME
```

Options

-f, --force
Overwrite Stub if it exists.

Arguments

STUB_NAME
Required argument

create

Create stubs from a pyboard at <PORT>

MicropyCli uses Josverl's micropython-stubber for stub creation.
For more information, please visit the repository
at: https://github.com/Josverl/micropython-stubber

```
micropy stubs create [OPTIONS] PORT
```

Options

-v, --verbose
Enable verbose output

Arguments

PORT
Required argument

list

List installed stubs.

```
micropy stubs list [OPTIONS]
```

search

Search available Stubs.

```
micropy stubs search [OPTIONS] QUERY
```

Arguments

QUERY
Required argument

1.5.4 micropy install

Install Packages as Project Requirements.

Install a project dependency while enabling
intellisense, autocompletion, and linting for it.

If no packages are passed and a requirements.txt file is found,
then micropy will install all packages listed in it.

If the –dev flag is passed, then the packages are only
added to micropy.json. They are not stubbed.

To add a dependency from a path, use the –path option
and provide a name for your package:

$ micropy install –path ./src/lib/mypackage MyCustomPackage

You can import installed packages just as you would
on your actual device:

>>> # main.py
>>> import <package_name>

```
micropy install [OPTIONS] [PACKAGES]...
```

Options

-d, --dev
Add dependency to dev requirements

-p, --path <path>
Add dependency from local path. Can be a file or directory.

Arguments

PACKAGES
Optional argument(s)

1.6 API Reference

micropy	Micropy Cli.
micropy.main	Main Module.
micropy.exceptions	Micropy Exceptions.
micropy.stubs	micropy.stubs
micropy.stubs.source	micropy.stubs.source
micropy.project	Module for generating/managing projects.
micropy.project.modules	Project Modules.
micropy.utils	micropy.utils
micropy.config	Configuration files and interfaces for them.
micropy.config.config_source	Config Abstract.
micropy.packages	Packages Module.

1.6.1 micropy

Micropy Cli.

Micropy Cli is a project management/generation tool for writing Micropython code in modern IDEs such as VSCode. Its primary goal is to automate the process of creating a workspace complete with:

Linting compatible with Micropython, VSCode Intellisense, Autocompletion, Dependency Management, VCS Compatibility and more.

class micropy.**MicroPy**
Bases: object

Handles App State Management.

create_stubs(*port*, *verbose=False*)
Create and add stubs from Pyboard.

Parameters **port** (*str*) – Port of Pyboard

Returns generated stub

Return type Stub

resolve_project(*path*, *verbose=True*)
Returns project from path if it exists.

Parameters

- **path** (*str*) – Path to test
- **verbose** (*bool*) – Log to stdout. Defaults to True.

Returns Project if it exists

setup()
Creates necessary directories for micropy.

1.6.2 micropy.main

Main Module.

Classes

MicroPy()	Handles App State Management.

class micropy.main.**MicroPy**
Bases: object

Handles App State Management.

create_stubs(*port*, *verbose=False*)
Create and add stubs from Pyboard.

Parameters **port** (*str*) – Port of Pyboard

Returns generated stub

Return type Stub

resolve_project(*path*, *verbose=True*)
Returns project from path if it exists.

Parameters

- **path** (*str*) – Path to test
- **verbose** (*bool*) – Log to stdout. Defaults to True.

Returns Project if it exists

setup()
Creates necessary directories for micropy.

1.6.3 micropy.exceptions

Micropy Exceptions.

Exceptions

StubError([message, stub])	Exception for any errors raised by stubs.
StubNotFound([stub_name])	Raised when a stub cannot be found.
StubValidationError(path, errors, *args, ...)	Raised when a stub fails validation.

exception micropy.exceptions.**MicropyException**
Bases: Exception

Generic MicroPy Exception.

exception micropy.exceptions.**RequirementException**(**args*, ***kwargs*)
Bases: *micropy.exceptions.MicropyException*

A Requirement Exception Occurred.

exception micropy.exceptions.**RequirementNotFound**(**args*, ***kwargs*)
Bases: *micropy.exceptions.RequirementException*

A requirement could not be found.

exception micropy.exceptions.**StubError**(*message=None*, *stub=None*)
Bases: *micropy.exceptions.MicropyException*

Exception for any errors raised by stubs.

exception micropy.exceptions.**StubNotFound**(*stub_name=None*)
Bases: *micropy.exceptions.StubError*

Raised when a stub cannot be found.

exception micropy.exceptions.**StubValidationError**(*path*, *errors*, **args*, ***kwargs*)
Bases: *micropy.exceptions.StubError*

Raised when a stub fails validation.

1.6.4 micropy.stubs

micropy.stubs

This module contains all functionality relating to stub files/frozen modules and their usage in MicropyCli

Classes

StubManager([resource, repos])	Manages a collection of Stubs.
source	micropy.stubs.source

class micropy.stubs.**StubManager**(*resource=None*, *repos=None*)
Bases: object

Manages a collection of Stubs.

Kwargs: resource (str): Default resource path repos ([StubRepo]): Repos for Remote Stubs

Raises

- ***StubError*** – a stub is missing a def file
- ***StubValidationError*** – a stubs def file is not valid

Returns Instance of StubManager

Return type object

add(*location*, *dest=None*, *force=False*)
Add stub(s) from source.

Parameters

- **source** (*str*) – path to stub(s)
- **dest** (*str, optional*) – path to copy stubs to. Defaults to self.resource
- **force** (*bool, optional*) – overwrite existing stubs. Defaults to False.

Raises **TypeError** – No resource or destination provided

from_stubber(*path*, *dest*)

Formats stubs generated by createstubs.py.

Creates a stub package from the stubs generated by createstubs.py. Also attempts to auto-resolve the stubs firmware name.

Parameters

- **path** (*str*) – path to generated stubs
- **dest** (*str*) – path to output

Returns formatted stubs

Return type str

is_valid(*path*)

Check if stub is valid without raising an exception.

Parameters **path** (*str*) – path to stub

Returns True if stub is valid

Return type bool

iter_by_firmware(*stubs=None*)

Iterate stubs sorted by firmware.

Parameters **stubs** (*[Stub], optional*) – Sublist of Stubs to iterate over. Defaults to None. If none, uses all installed stubs.

load_from(*directory*, **args*, ***kwargs*)

Recursively loads stubs from a directory.

Parameters **directory** (*str*) – Path to load from

Returns List of loaded Stubs

Return type [DeviceStub]

resolve_firmware(*stub*)

Resolves FirmwareStub for DeviceStub instance.

Parameters **stub** (*DeviceStub*) – Stub to resolve

Returns

Instance of FirmwareStub NoneType: None if an appropriate

FirmwareStub cannot be found

Return type FirmwareStub

resolve_subresource(*stubs*, *subresource*)

Resolve or Create StubManager from list of stubs.

Parameters

- **stubs** (*[Stub]*) – List of stubs to use in subresource

- **subresource** (*str*) – path to subresource

Returns StubManager with subresource stubs

Return type *StubManager*

search_remote(*query*)
Search all repositories for query.

Parameters **query** (*str*) – query to search for

Returns

List of result tuples. The first item is the package name, and the second is a bool based on whether the package is installed or not

Return type [tuple]

validate(*path*, *schema=None*)
Validates given stub path against its schema.

Parameters

- **path** (*str*) – path to validate
- **schema** (*str, optional*) – Path to schema. Defaults to None. If None, the DeviceStub schema is used.

Raises

- ***StubError*** – Raised if no info file can be found
- ***StubValidationError*** – Raised if the info file fails validation

verbose_log(*state*)
Enable Stub logging to stdout.

Parameters **state** (*bool*) – State to set

Returns state

Return type bool

1.6.5 micropy.stubs.source

micropy.stubs.source

This module contains abstractions for handling stub sources and their location.

Functions

get_source(location, **kwargs)	Factory for StubSource Instance.

Classes

LocalStubSource(path, **kwargs)	Stub Source Subclass for local locations.
RemoteStubSource(name, **kwargs)	Stub Source for remote locations.
StubRepo(name, location, path, **kwargs)	Represents a remote repository for stubs.

Continued on next page

Table 6 – continued from previous page

StubSource(location[, log])	Abstract Base Class for Stub Sources.

class micropy.stubs.source.**LocalStubSource**(*path*, ***kwargs*)
Bases: *micropy.stubs.source.StubSource*

Stub Source Subclass for local locations.

Parameters **path** (*str*) – Path to Stub Source

Returns Instance of LocalStubSource

Return type obj

class micropy.stubs.source.**RemoteStubSource**(*name*, ***kwargs*)
Bases: *micropy.stubs.source.StubSource*

Stub Source for remote locations.

Parameters **url** (*str*) – URL to Stub Source

Returns Instance of RemoteStubSource

Return type obj

ready()
Retrieves and unpacks source.

Prepares remote stub resource by downloading and unpacking it into a temporary directory. This directory is removed on exit of the superclass context manager

Returns StubSource.ready parent method

Return type callable

class micropy.stubs.source.**StubRepo**(*name*, *location*, *path*, ***kwargs*)
Bases: object

Represents a remote repository for stubs.

Parameters

- **name** (*str*) – Repo Name
- **location** (*str*) – Valid url
- **ref** (*str*) – path to repo definition file

classmethod from_json(*content*)
Create StubRepo Instances from JSON file.

Parameters **file_obj** (*str or bytes*) – json content

Returns iterable of created repos

get_url(*path*)
Returns formatted url to provided path.

Parameters **path** (*str*) – path to format

Returns formatted url

Return type str

has_package(*name*)
Checks if package is available in repo.

Parameters **name** (*str*) – name of package

Returns True if package is available

Return type bool

classmethod resolve_package(*name*)
Attempts to resolve package from all repos.

Parameters **name** (*str*) – package to resolve

Raises ***StubNotFound*** – Package could not be resolved

Returns url to package

Return type str

search(*query*)
Searches repository packages.

Parameters **query** (*str*) – query to search by

Returns List of matching results

Return type [str]

class micropy.stubs.source.**StubSource**(*location*, *log=None*)
Bases: object

Abstract Base Class for Stub Sources.

ready(*path=None*, *teardown=None*)
Yields prepared Stub Source.

Allows StubSource subclasses to have a preperation method before providing a local path to itself.

Parameters

- **path** (*str, optional*) – path to stub source. Defaults to location.
- **teardown** (*func, optional*) – callback to execute on exit. Defaults to None.

Yields Resolved PathLike object to stub source

micropy.stubs.source.**get_source**(*location*, ***kwargs*)
Factory for StubSource Instance.

Parameters **location** (*str*) – PathLike object or valid URL

Returns Either Local or Remote StubSource Instance

Return type obj

1.6.6 micropy.project

Module for generating/managing projects.

Classes

Project(path[, name])	Micropy Project.

class micropy.project.**Project**(*path*, *name=None*, ***kwargs*)
Bases: micropy.project.modules.modules.ProjectModule

Micropy Project.

Parameters

- **path** (*str*) – Path to project root.
- **name** (*str, optional*) – Name of Project. Defaults to None. If none, uses name of current directory.

add(*component*, **args*, ***kwargs*)
Adds project component.

Parameters **component** (*Any*) – Component to add.

property cache
Project wide cache.

Return type Config

Returns Cache instance

property config
Project Configuration.

Returns Project Config Instance

Return type *Config*

property context
Project context used in templates.

Returns Current context

Return type *Config*

create()
Creates new Project.

Returns Path relative to current active directory.

Return type Path

property exists
Whether this project exists.

Returns True if it exists

Return type bool

iter_children_by_priority()
Iterate project modules by priority.

Yields the next child item

Return type Iterator[Type[ProjectModule]]

load(***kwargs*)
Loads all components in Project.

Return type Project

Returns Current Project Instance

remove(*component*)
Removes project component.

Parameters **component** (*Any*) – Component to remove.

update()
Updates all project components.

Returns Current active project.

1.6.7 micropy.project.modules

Project Modules.

Classes

DevPackagesModule(path, **kwargs)	Project Module for Dev Packages.
PackagesModule(path, **kwargs)	Project Module for handling requirements.
ProjectModule([parent, log])	Abstract Base Class for Project Modules.
StubsModule(stub_manager[, stubs])	Project module for handling Stubs.
TemplatesModule([templates, run_checks])	Project Templates Module.
HookProxy(name)	Proxy for Project Hooks.

class micropy.project.modules.**TemplatesModule**(*templates=None*, *run_checks=True*, ***kwargs*)

Bases: micropy.project.modules.modules.ProjectModule

Project Templates Module.

Generates and manages project files using the Projects context.

Parameters

- **templates** (*List[str]*) – List of templates to use.
- **run_checks** (*bool, optional*) – Whether to execute checks or not. Defaults to True.

property config

Template config.

Returns Current configuration

Return type dict

create()

Generates project files.

Returns Project context

Return type dict

load(***kwargs*)

Loads project templates.

update()

Updates project files.

Returns Project context

Return type dict

class micropy.project.modules.**PackagesModule**(*path*, ***kwargs*)

Bases: micropy.project.modules.modules.ProjectModule

Project Module for handling requirements.

Parameters

- **path** (*str*) – Path to create requirements file at.

- **packages** (*dict, optional*) – Initial packages to use. Defaults to None.

add_from_file(*path=None*, *dev=False*, ***kwargs*)
Loads all requirements from file.

Parameters

- **path** (Optional[Path]) – Path to file. Defaults to self.path.
- **dev** (bool) – If dev requirements should be loaded. Defaults to False.

Return type dict

add_package(*package*, *dev=False*, ***kwargs*)
Add requirement to project.

Parameters

- **package** (*str*) – package name/spec
- **dev** (*bool, optional*) – If dev requirements should be loaded. Defaults to False.

Returns Dictionary of packages

Return type dict

property cache
Project Cache.

Return type Config

Returns Project wide cache

property config
Config values specific to component.

Return type Config

Returns Component config.

property context
Context values specific to component.

Return type Config

Returns Context values.

create()
Create project files.

load(*fetch=True*, ***kwargs*)
Retrieves and stubs project requirements.

property path
Path to requirements file.

Returns Path to file

Return type Path

property pkg_path
Path to package data folder.

Returns Path to folder.

Return type Path

update()
Dumps packages to file at path.

class micropy.project.modules.**StubsModule**(*stub_manager*, *stubs=None*, ***kwargs*)
Bases: micropy.project.modules.modules.ProjectModule

Project module for handling Stubs.

Parameters

- **stub_manager** (StubManager) – StubManager instance.
- **stubs** (*List[Type[Stub]], optional*) – Initial Stubs to use.

add_stub(*stub*, ***kwargs*)
Add stub to project.

Parameters **stub** (*Stub*) – Stub object to add

Returns Project Stubs

Return type [Stubs]

property config
Component specific config values.

Returns Current config.

Return type dict

property context
Component stub context.

create()
Create stub project files.

get_stub_tree(*stubs*)
Retrieve and order paths to base stubs and any stubs they depend on.

Parameters **stubs** – List of Stub Items

Return type Sequence[Path]

Returns Paths to all stubs project depends on.

load(***kwargs*)
Loads stubs from info file.

Parameters **stub_list** (*dict*) – Dict of Stubs

property stubs
Component stubs.

Returns List of stubs used in project.

Return type List[micropy.stubs.Stub]

update()
Update current project stubs.

class micropy.project.modules.**ProjectModule**(*parent=None*, *log=None*)
Bases: object

Abstract Base Class for Project Modules.

add(*component*, **args*, ***kwargs*)
Adds component.

Parameters **component** (*Any*) – Component to add.

Return type Any

abstract property config
Config values specific to component.

Return type Union[dict, Config]

abstract create(**args*, ***kwargs*)
Method to create component.

Return type Any

classmethod hook(**args*, ***kwargs*)
Decorator for creating a Project Hook.

Allows decorated method to be called from parent container.

Returns Decorated function.

Return type Callable

abstract load()
Method to load component.

property parent
Component Parent.

remove(*component*)
Removes component.

Parameters **component** (*Any*) – Component to remove.

Return type Any

resolve_hook(*name*)
Resolves appropriate hook for attribute name.

Parameters **name** (*str*) – Attribute name to resolve hook for.

Returns Callable Proxy for ProjectHook. NoneType: Name could not be resolved.

Return type Optional[*HookProxy*]

abstract update()
Method to update component.

class micropy.project.modules.**DevPackagesModule**(*path*, ***kwargs*)
Bases: micropy.project.modules.packages.PackagesModule

Project Module for Dev Packages.

add_from_file(*path=None*, ***kwargs*)
Adds packages from file.

add_package(*package*, ***kwargs*)
Adds package.

create()
Creates component.

load(**args*, ***kwargs*)
Load component.

class micropy.project.modules.**HookProxy**(*name*)
Bases: object

Proxy for Project Hooks.

Allows multiple project hooks with the same name by creating individual hooks for any defined permutations of kwargs.

This is accomplished by creating a unique name for each permutation proxying the original attribute name to the appropriate method determined from the provided kwargs.

Parameters **name** (*str*) – Name of Proxy

add_instance(*inst*)
Add instance to Proxy.

Parameters **inst** (*Any*) – Instance to add.

Return type Any

add_method(*func*, ***kwargs*)
Adds method to Proxy.

Any kwargs provided will be used to generate the unique hook name.

Parameters **func** (*Callable*) – Method to add

Example

```
>>> def test_func(arg1, kwarg1=False):
        pass
```

```
>>> self.add_method(test_func, {'kwarg1': False})
(test_func, '_hook__test_func__kwarg1_False')
```

Returns Tuple containing method and unique hook name.

Return type Tuple[Callable, str]

get()
Get initial method descriptor value.

Return type ~T

get_name(*func*, *params=None*)
Generates name from method and provided kwargs.

Parameters

- **func** (*Callable*) – Method to generate name for.
- **params** (*Dict[Any, Any], optional*) – Any kwargs to update the defaults with. Defaults to None. If none, uses default kwargs.

Returns Generated name

Return type str

is_descriptor()
Determine if initial method provided is a descriptor.

Return type bool

resolve_proxy(***kwargs*)
Resolves appropriate instance and method to proxy to.

If additional kwargs are provided and a proxy is not found, the function will continue to remove one kwarg and recurse into itself until either a match is found or it runs out of kwargs.

Return type (typing.Type[micropy.project.modules.modules.ProjectModule], <class 'str'>)

Returns Instance and method name if resolved, otherwise None.

1.6.8 micropy.utils

micropy.utils

This module provides utility functions that are used within MicropyCli.

Functions

create_dir_link(source, target)	Creates a platform appropriate directory link.
ensure_existing_dir(path)	Ensure path exists and is a directory.
ensure_valid_url(url)	Ensure a url is valid.
extract_tarbytes(file_bytes, path)	Extract tarfile as bytes.
generate_stub(path[, log_func])	Create Stub from local .py file.
get_package_meta(name, url)	Retrieve package metadata from PyPi.
get_url_filename(url)	Parse filename from url.
is_dir_link(path)	Test if path is either a symlink or directory junction.
is_downloadable(url)	Checks if the url can be downloaded from.
is_existing_dir(path)	Check if path is an existing directory.
is_url(url)	Check if provided string is a url.
iter_requirements(path)	Iterate requirements from a requirements.txt file.
search_xml(url, node)	Search xml from url by node.
stream_download(url, **kwargs)	Stream download with tqdm progress bar.
is_update_available()	Check if micropy-cli update is available.
get_cached_data(url)	Wrap requests with a short cache.
get_class_that_defined_method(meth)	Determines Class that defined a given method.

Classes

PyboardWrapper(port[, connect, verbose])	Wrapper for rshell/pyboard.
Validator(schema_path)	"jsonschema wrapper for file validation.

class micropy.utils.**Validator**(*schema_path*)
Bases: object

"jsonschema wrapper for file validation.

Returns Validator Instance

Return type object

validate(*path*)
Validates json file against a schema.

Parameters **path** (*str*) – path to json file to validate

Returns jsonschema.validate

class micropy.utils.**PyboardWrapper**(*port*, *connect=True*, *verbose=False*)
Bases: object

Wrapper for rshell/pyboard.

Exposes the basic run/copy functionality Micropy needs

Parameters **port** (*str*) – Port of Pyboard

Kwargs: connect (bool): Connect on init. Defaults to True

connect()
connect to pyboard.

copy_dir(*path*, *dest*, *rsync={}*)
Copy directory from pyboard to machine.

Parameters

- **path** (*str*) – path to directory
- **dest** (*str*) – destination to copy to
- **rsync** (*dict, optional*) – additonal args to pass to rsync call. Defaults to {}

copy_file(*source*, *dest=None*)
Copies file to pyboard.

Parameters

- **source** (*str*) – path to file
- **dest** (*str, optional*) – dest on pyboard. Defaults to None. If None, file is copied to pyboard root.

Returns path to dest on pyboard

Return type str

list_dir(*path*)
List directory on pyboard.

Parameters **path** (*str*) – path to directory

property pyb_root
pyboard root dirname.

property pyboard
rshell pyboard instance.

repl()
Pyboard raw repl context manager.

run(*file*, *format_output=None*)
Execute a local script on the pyboard.

Parameters

- **file** (*str*) – path to file or string to run
- **format_output** (*callable, optional*) – Callback to format output. Defaults to None. If none, uses print.

micropy.utils.**is_url**(*url*)
Check if provided string is a url.

Parameters **url** (*str*) – url to check

Returns True if arg url is a valid url

Return type bool

micropy.utils.**get_url_filename**(*url*)
Parse filename from url.

Parameters **url** (*str*) – url to parse

Returns filename of url

Return type str

micropy.utils.**ensure_existing_dir**(*path*)
Ensure path exists and is a directory.

If path does exist, it will be returned as a pathlib.PurePath object

Parameters **path** (*str*) – path to validate and return

Raises

- **NotADirectoryError** – path does not exist
- **NotADirectoryError** – path is not a directory

Returns pathlib.PurePath object

Return type object

micropy.utils.**ensure_valid_url**(*url*)
Ensure a url is valid.

Parameters **url** (*str*) – URL to validate

Raises

- **InvalidURL** – URL is not a valid url
- **ConnectionError** – Failed to connect to url
- **HTTPError** – Reponse was not 200 <OK>

Returns valid url

Return type str

micropy.utils.**is_downloadable**(*url*)
Checks if the url can be downloaded from.

Parameters **url** (*str*) – url to check

Returns True if contains a downloadable resource

Return type bool

micropy.utils.**is_existing_dir**(*path*)
Check if path is an existing directory.

Parameters **path** (*str*) – path to check

Returns True if path exists and is a directory

Return type bool

micropy.utils.**stream_download**(*url*, ***kwargs*)
Stream download with tqdm progress bar.

Parameters **url** (*str*) – url to file

Returns bytearray of content

Return type bytearray

micropy.utils.**search_xml**(*url*, *node*)
Search xml from url by node.

Parameters

- **url** (*str*) – url to xml
- **node** (*str*) – node to search for

Returns matching nodes

Return type [str]

micropy.utils.**generate_stub**(*path*, *log_func=None*)
Create Stub from local .py file.

Parameters

- **path** (*str*) – Path to file
- **log_func** (*func, optional*) – Callback function for logging. Defaults to None.

Returns Tuple of file path and generated stub path.

Return type tuple

micropy.utils.**get_package_meta**(*name*, *url*)
Retrieve package metadata from PyPi.

Parameters

- **name** (*str*) – Name of package with specs.
- **url** (*str*) – Url to package.

Returns Dictionary of Metadata

Return type dict

micropy.utils.**extract_tarbytes**(*file_bytes*, *path*)
Extract tarfile as bytes.

Parameters

- **file_bytes** (*bytearray*) – Bytes of file to extract
- **path** (*str*) – Path to extract it to

Returns destination path

Return type path

micropy.utils.**iter_requirements**(*path*)
Iterate requirements from a requirements.txt file.

Parameters **path** (*str*) – path to file

micropy.utils.**create_dir_link**(*source*, *target*)
Creates a platform appropriate directory link.

On POSIX systems it will create a symlink. On Windows it will fallback on a directory junction if needed

Parameters

- **source** (*os.Pathlike*) – Path to create link at.
- **target** (*os.Pathlike*) – Path to link to.

Raises

- **OSError** – Symlink Creation Failed
- **OSError** – Symlink and Directory Junction Fallback Failed

micropy.utils.**is_dir_link**(*path*)
Test if path is either a symlink or directory junction.

Parameters **path** (*os.Pathlike*) – Path to test.

Returns True if path is a type of link.

Return type bool

micropy.utils.**is_update_available**()
Check if micropy-cli update is available.

Returns True if update available, else False.

Return type bool

micropy.utils.**get_cached_data**(*url*)
Wrap requests with a short cache.

micropy.utils.**get_class_that_defined_method**(*meth*)
Determines Class that defined a given method.

See - https://stackoverflow.com/a/25959545

Parameters **meth** (*Callable*) – Method to determine class from

Returns Class that defined method

Return type Callable

1.6.9 micropy.config

Configuration files and interfaces for them.

Classes

Config(*args[, source_format, default])	Configuration File Interface.
JSONConfigSource(path)	JSON Config File Source.
DictConfigSource([config])	

class micropy.config.**Config**(**args*, *source_format=<class 'micropy.config.config_json.JSONConfigSource'>*, *default={}*)
Bases: object

Configuration File Interface.

Automatically syncs config in memory with config saved to disk.

Parameters

- **path** (*Path*) – Path to save file at.
- **source_format** (ConfigSource, *optional*) – Configuration File Format. Defaults to JSONConfigSource.
- **default** (*dict, optional*) – Default configuration. Defaults to {}.

add(*key*, *value*)
Overwrite or add config value.

Parameters

- **key** (str) – Key to set
- **value** (Any) – Value to add or update too

Return type Any

Returns Updated config

extend(*key*, *value*, *unique=False*)
Extend a list in config at key path.

Parameters

- **key** (str) – Key to path to extend.
- **value** (List[Any]) – List of values to extend by.
- **unique** (bool) – Only extend values if not already in values.

Return type dict

Returns Updated Config

get(*key*, *default=None*)
Retrieve config value.

Parameters

- **key** (*str*) – Key (in dot-notation) of value to return.
- **default** (*Any, optional*) – Default value to return. Defaults to None.

Returns Value at key given

Return type Any

parse_key(*key*)
Parses key.

Splits it into a path and 'final key' object. Each key is seperates by a: "/"

Example

```
>>> self.parse_key('item/subitem/value')
(('item', 'subitem'), 'value')
```

Parameters **key** (*str*) – key in dot notation

Returns Parsed key

Return type Tuple[Sequence[str], str]

pop(*key*)
Delete and return value at key.

Parameters **key** (*str*) – Key to pop.

Returns Popped value.

Return type Any

search(*key*)
Retrieve all values at key (with glob pattern).

Parameters **key** – Key with pattern to search with.

Returns Values matching key and pattern.

set(*key*, *value*)
Set config value.

Parameters

- **key** (*str*) – Key (in dot-notation) to update.
- **value** (*Any*) – Value to set

Returns Updated config

Return type Any

sync()
Sync in-memory config with disk.

Returns updated config

Return type dict

upsert(*key*, *value*)
Update or insert values into key list or dict.

Parameters

- **key** (str) – Key to value to upsert.
- **value** (Union[List[Any], dict]) – Value to upsert by.

Return type dict

Returns Updated config.

class micropy.config.**JSONConfigSource**(*path*)
Bases: *micropy.config.config_source.ConfigSource*

JSON Config File Source.

Parameters **path** (*Path*) – Path to save config too.

property exists
Property to check if source exists.

Return type bool

property file_path
Path to config file.

Return type Path

prepare()
Method to prepare on enter.

process()
Load config from JSON file.

Returns config in file

Return type dict

save(*content*)
Save current config.

Parameters **content** (*dict*) – content to write to file.

Returns path to config file.

Return type Path

class micropy.config.**DictConfigSource**(*config={}*)
Bases: *micropy.config.config_source.ConfigSource*

property exists
Property to check if source exists.

Return type bool

prepare()
Method to prepare on enter.

process()
Read and process config file.

Returns Config file content

Return type dict

save(*content*)
Method to save config.

Return type dict

1.6.10 micropy.config.config_source

Config Abstract.

Classes

ConfigSource([initial_config])	Abstract Base Class for Config Sources.

class micropy.config.config_source.**ConfigSource**(*initial_config={}*)
Bases: contextlib.AbstractContextManager

Abstract Base Class for Config Sources.

Parameters **initial_config** (*dict, optional*) – Initial config values. Defaults to {}.

property config
Current Config Content.

Return type dict

abstract property exists
Property to check if source exists.

Return type `bool`

abstract prepare()
Method to prepare on enter.

Return type `Any`

abstract process()
Read and process config file.

Returns Config file content

Return type dict

abstract save(*content*)
Method to save config.

Return type `Any`

1.6.11 micropy.packages

Packages Module.

Allows user to address different dependency types (package, module, path, pypi, etc.) through a single uniform api.

Functions

create_dependency_source(requirement[, name])	Factory for creating a dependency source object.

Classes

LocalDependencySource(package, path)	Dependency Source that is available locally.
Package(name, specs[, path])	
PackageDependencySource(package[, format_desc])	Dependency Source for pypi packages.

class micropy.packages.**PackageDependencySource**(*package*, *format_desc=None*)
Bases: micropy.packages.source.DependencySource

Dependency Source for pypi packages.

Parameters

- **package** (*Package*) – Package source points too.
- **format_desc** (Optional[Callable[..., Any]]) – Callback to format progress bar description. Defaults to None.

fetch()
Fetch package contents into memory.

Returns Package archive contents.

Return type bytes

class micropy.packages.**LocalDependencySource**(*package*, *path*)
Bases: micropy.packages.source.DependencySource

Dependency Source that is available locally.

Parameters

- **package** (*Package*) – Package source points too.
- **path** (*Path*) – Path to package.

micropy.packages.**create_dependency_source**(*requirement*, *name=None*, ***kwargs*)
Factory for creating a dependency source object.

Parameters

- **requirement** (*str*) – Package name/path/constraints in string form.
- **name** (*str, optional*) – Override package name. Defaults to None.

Return type Union[LocalDependencySource, PackageDependencySource]

Returns Appropriate Dependency Source

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

m

INDEX

Symbols

A

C

D

E

F

G

H

I

J

L

M

P

Q

R

S

T

U

V