Micropython-Stubber

Release 1.4.5

Jos Verlinde

Jan 03, 2022

CONTENTS:

CHAPTER

ONE

BOOST MICROPYTHON PRODUCTIVITY IN VSCODE

The intellisense and code linting that is so prevalent in modern editors, does not work out-of-the-gate for MicroPython projects. While the language is Python, the modules used are different from CPython , and also different ports have different modules and classes , or the same class with different parameters.

Writing MicroPython code in a modern editor should not need to involve keeping a browser open to check for the exact parameters to read a sensor, light-up a led or send a network request.

Fortunately with some additional configuration and data, it is possible to make the editors understand your flavor of MicroPython. even if you run a on-off custom firmware version.

In order to achieve this a few things are needed:

1. Stub files for the native / enabled modules in the firmware using PEP 484 Type Hints
2. Specific configuration of the VSCode Python extensions
3. Specific configuration of Pylint
4. Suppression of warnings that collide with the MicroPython principals or code optimization.

With that in place, VSCode will understand MicroPython for the most part, and help you to write code, and catch more errors before deploying it to your board.

Note that the above is not limited to VSCode and pylint, but it happens to be the combination that I use.

A lot of subs have already been generated and are shared on github or other means, so it is quite likely that you can just grab a copy to be productive in a few minutes.

For now you will need to *configure this by hand*, or use the `micropy cli` *tool*

1. The sister-repo [**MicroPython-stubs**][stubs-repo] contains [all stubs][all-stubs] I have collected with the help of others, and which can be used directly. That repo also contains examples configuration files that can be easily adopted to your setup.
2. A second repo [micropy-stubs repo][stubs-repo2] maintained by BradenM, also contains stubs but in a structure used and distributed by the *micropy-cli* tool. you should use micropy-cli to consume stubs in this repo.

The (stretch) goal is to create a VSCode add-in to simplify the configuration, and allow easy switching between different firmwares and versions.

1.1 Licensing

MicroPython-Stubber is licensed under the MIT license, and all contributions should follow this LICENSE.

CHAPTER

TWO

APPROACH TO COLLECTING STUB INFORMATION

The stubs are used by 3 components.

1. pylint
2. the VSCode Pylance Language Server
3. the VSCode Python add-in

These 3 tools work together to provide code completion/prediction, type checking and all the other good things. For this the order in which these tools use, the stub folders is significant, and best results are when all use the same order.

In most cases the best results are achieved by the below setup:

![stub processing order][]

1. **Your own source files**, including any libraries you add to your project. This can be a single libs folder or multiple directories. There is no need to run stubber on your source or libraries.
2. **The CPython common stubs**. These stubs are handcrafted to allow MicroPython script to run on a CPython system. There are only a limited number of these stubs and while they are not intended to be used to provide type hints, they do provide valuable information. Note that for some modules (such as the `gc`, `time` and `sys` modules) this approach does not work.
3. **Frozen stubs**. Most micropython firmwares include a number of python modules that have been included in the firmware as frozen modules in order to take up less memory. These modules have been extracted from the source code.
4. **Firmware Stubs**. For all other modules that are included on the board, [micropython-stubber] or [micropy-cli] has been used to extract as much information as available, and provide that as stubs. While there is a lot of relevant and useful information for code completion, it does unfortunately not provide all details regarding parameters that the above options may provide.

2.1 Stub collection process

- The **CPython common stubs** are periodically collected from the [micropython-lib][] or the [pycopy-lib][].
- The **Frozen stubs** are collected from the repos of [micropython][] + [micropython-lib][] and from the [loboris][] repo the methods to gather these differs per firmware family , and there are differences between versions how these are stored , and retrieved. where possible this is done per port and board, or if not possible the common configuration for has been included.
- the **Firmware stubs** are generated directly on a MicroPython board.

2.2 Firmware Stubs format and limitations

1. No function parameters are generated
2. No return types are generated
3. Instances of imported classes have no type (due to 2)
4. The stubs use the .py extension rather than .pyi (for autocomplete to work)
5. Due to the method of generation nested modules are included, rather than referenced. While this leads to somewhat larger stubs, this should not be limiting for using the stubs on a PC.
6.

2.3 Firmware naming convention

The firmware naming conventions is most relevant to provide clear folder names when selecting which stubs to use.

for stubfiles: {**firmware**}-{port}-{version}[-{build}]

for frozen modules : {firmware}-{version}-frozen

- ***firmware***: lowercase
  - micropython | loboris | pycopy | ...
- ***port***: lowercase , as reported by os.implementation.platform
  - esp32 | linux | win32 | esp32_lobo
- ***version*** : digits only , dots replaced by underscore, follow version in documentation rather than semver
  - 1_13
  - 1_9_4
- ***build***, only for nightly build, the build nr. extracted from the git tag
  - Nothing , for released versions
  - 103
  - N (short notation)

CHAPTER

THREE

USING STUBS

3.1 Manual configuration

The manual configuration, including sample configuration files is described in detail in the sister-repo [micropython-stubs][] section [using-the-stubs][]

3.2 Using micropy-cli

'micropy-cli' is command line tool for managing MicroPython projects with VSCode If you want a command line interface to setup a new project and configure the settings as described above for you, then take a look at : [micropy-cli]

```
pip install micropy-cli
micropy init
```

Braden has essentially created a front-end for using micropython-stubber, and the configuration of a project folder for pymakr.

micropy-cli maintains its own repository of stubs.

CHAPTER

FOUR

VSCODE AND PYLINT CONFIGURATION

The current configuration section describes how to use [Pylance].

> *To deliver an improved user experience, we've created Pylance as a brand-new language server based on Microsoft's Pyright static type checking tool. Pylance leverages type stubs (.pyi files) and lazy type inferencing to provide a highly-performant development experience. Pylance supercharges your Python IntelliSense experience with rich type information, helping you write better code, faster. The Pylance extension is also shipped with a collection of type stubs for popular modules to provide fast and accurate auto-completions and type checking.*

Some sections may still refer to the use of [Microsoft Python Language Server][mpls], which has been deprecated.

4.1 Recommended order of the stubs in your config:

1. The src/libs folder(s)
2. The CPython common modules
3. The frozen modules offer more information that can be used in code completion, and therefore should be loaded before the firmware stubs.
4. The firmware stubs generated on or for your board

Announcing Pylance: Fast, feature-rich language support for Python in Visual Studio Code | Python (microsoft.com)

4.2 Relevant VSCode settings

Setting	Default	Description	ref
python.autoComplete.extraPaths	[]	Specifies locations of additional packages for which to load autocomplete data.	Autocomplete Settings
typeshedPaths	[]	Specifies paths to local typeshed repository clone(s) for the Python language server.	Git
python.linting.			Linting Settings
enabled	true	Specifies whether to enable linting in general.	
pylintEnabled	true	Specifies whether to enable Pylint.	

4.2.1 Pylance - pyright

[Pylance](Pylance - Visual Studio Marketplace) is replacing MPLS and provides the same and more functionality.

Setting	Default	Description
python.analysis.stubPath	Typings	Used to allow a user to specify a path to a directory that contains custom type stubs. Each package's type stub file(s) are expected to be in its own subdirectory.
python.analysis.autoSearchPaths	true	Used to automatically add search paths based on some predefined names (like `src`).
python.analysis.extraPaths	[]	Used to specify extra search paths for import resolution. This replaces the old `python.autoComplete.extraPaths` setting.

4.2.2 Sample configuration for Pylance

To update a project configuration from MPLS to Pylance is simple :

Open your VSCode settings file : `.vscode/settings.json`

- change the language server to Pylance `"python.languageServer": "Pylance",`
- remove the section: `python.autoComplete.typeshedPaths`
- remove the section : `python.analysis.typeshedPaths`
- optionally add : `"python.analysis.autoSearchPath": true,`

The result should be something like this :

```
{
     "python.languageServer": "Pylance",
     "python.analysis.autoSearchPath": true,
     "python.autoComplete.extraPaths": [
          "src/lib",
          "all-stubs/cpython_patch",
          "all-stubs/mpy_1_13-nightly_frozen/esp32/GENERIC",
          "all-stubs/esp32_1_13_0-103",
     ]
    "python.linting.enabled": true,
    "python.linting.pylintEnabled": true,
}
```

If you notice problems :

- The paths are case sensitive (which may not be apparent for your platform)
- To allow the config to be used cross platform you can use forward slashes `/`, *note that this is also accepted on Windows*
- If you prefer to use a backslash : in JSON notation the \ (backslash) MUST be escaped as \\ (double backslash)
- Remember to put the 'Frozen' module paths before the generated module paths.

References :

Pylance - Visual Studio Marketplace

microsoft/pyright: Static type checker for Python (github.com)

possible testing / diag :

pyright/command-line.md at master · microsoft/pyright (github.com)

4.3 pylint

Pylint needs 2 settings :

1. Specify **init-hook** to inform pylint where the stubs are stored. note that the `src` folder is already automagically included, so you do not need to add that.
2. disable some pesky warnings that make no sense for MicroPython, and that are caused by the stubs that have only limited information

File: .pylintrc

```
[MASTER]
# Loaded Stubs:  esp32-micropython-1.11.0
init-hook='import sys;sys.path[1:1] = ["src/lib", "all-stubs/cpython-core", "all-stubs/
↪mpy_1_12/frozen/esp32/GENERIC", "all-stubs/esp32_1_13_0-103",]'

disable = missing-docstring, line-too-long, trailing-newlines, broad-except, logging-
↪format-interpolation, invalid-name,
        no-method-argument, assignment-from-no-return, too-many-function-args,␣
↪unexpected-keyword-arg
        # the 2nd  line deals with the limited information in the generated stubs.
```

4.4 Microsoft Python Language Server settings - Deprecated

MPLS is being replaced by Pylance , and the below configuration is for reference only .

The language server settings apply when python.jediEnabled is false.

Setting	Default	Description	ref
python.jediEnabled	Default: *true*, must be set to FALSE	Indicates whether to use Jedi as the IntelliSense engine (true) or the Microsoft Python Language Server (false). Note that the language server requires a platform that supports .NET Core 2.1 or newer.	
python.analysis.			code analysis settings)
typeshedPaths	[]	Paths to look for typeshed modules on GitHub.	

Our long-term plan is to transition our Microsoft Python Language Server users over to Pylance and eventually deprecate and remove the old language server as a supported option

CHAPTER

FIVE

CREATE FIRMWARE STUBS

It is possible to create MicroPython stubs using the `createstubs.py` MicroPython script.

the script goes though the following stages

1. it determines the firmware family, the version and the port of the device, and based on that information it creates a firmware identifier (fwid) in the format : {family}-{port}-{version} the fwid is used to name the folder that stores the subs for that device.
    - micropython-pyboard-1_10
    - micropython-esp32-1_12
    - loboris-esp32_LoBo-3_2_4
2. it cleans the stub folder
3. it generates stubs, using a predetermined list of module names. for each found module or submodule a stub file is written to the device and progress is output to the console/repl.
4. a module manifest (`modules.json`) is created that contains the pertinent information determined from the board, the version of createstubs.py and a list of the successful generated stubs

Module duplication

Due to the module naming convention in micropython some modules will be duplicated , ie `uos` and `os` will both be included

5.1 Running the script

The createstubs.py script can either be run as a script or imported as a module depending on your preferences.

Running as a script is used on the linux or win32 platforms in order to pass a –path parameter to the script.

The steps are :

1. connect to your board
2. upload the script to your board [optional]
3. run/import the `createstubs.py` script
4. download the generated stubs to a folder on your PC
5. run the post-processor [optional, but recommended]

![createstubs-flow][]

Note: There is a memory allocation bug in MicroPython 1.30 that prevents createstubs.py to work. this was fixed in nightly build v1.13-103 and newer.

If you try to create stubs on this defective version, the stubber will raise *NotImplementedError*("MicroPython 1.13.0 cannot be stubbed")

5.2 Generating Stubs for a specific Firmware

The stub files are generated on a MicroPython board by running the script `createstubs.py`, this will generate the stubs on the board and store them, either on flash or on the SD card. If your firmware does not include the `logging` module, you will need to upload this to your board, or use the minified version.

```
import createstubs
```

The generation will take a few minutes (2-5 minutes) depending on the speed of the board and the number of included modules.

As the stubs are generated on the board, and as MicroPython is highly optimized to deal with the scarce resources, this unfortunately does mean that the stubs lacks parameters details. So for these you must still use the documentation provided for that firmware.

5.3 Downloading the files

After the sub files have been generated , you will need to download the generated stubs from the micropython board and most likely you will want to copy and save them on a folder on your computer. if you work with multiple firmwares, ports or version it is simple to keep the stub files in a common folder as the firmware id is used to generate unique names

- ./stubs
  - /micropython-pyboard-1_10
  - /micropython-esp32-1_12
  - /micropython-linux-1_11
  - /loboris-esp32_LoBo-3_1_20
  - /loboris-esp32_LoBo-3_2_24

5.4 Custom firmware

The script tries to determine a firmware ID and version from the information provided in `sys.implementation` , `sys.uname()` and the existence of specific modules..

This firmware ID is used in the stubs , and in the folder name to store the subs.

If you need, or prefer, to specify a firmware ID you can do so by setting the firmware_id variable before importing createstubs For this you will need to edit the createstubs.py file.

The recommendation is to keep the firmware id short, and add a version as in the below example.

```
# almost at the end of the file
def main():
    stubber = Stubber(firmware_id='HoverBot v1.2.1')
    # Add specific additional modules to be stubbed
```

(continues on next page)

(continued from previous page)

```
    stubber.add_modules(['hover','rudder'])
```

after this , upload the file and import it to generate the stubs using your custom firmware id.

5.5 The Unstubbables

There are a limited number of modules that cannot be stubbed by createstubs.py for a number of different reasons. Some simply raise errors , others my reboot the MCU, or require a specific configuration or state before they are loaded.

a few of the frozen modules are just included as a sample rather \t would not be very useful to generate stubs for these

the problematic category throw errors or lock up the stubbing process altogether:

```
 self.problematic=["upysh","webrepl_setup","http_client","http_client_ssl","http_server",
↪"http_server_ssl"]
```

the excluded category provides no relevant stub information

```
 self.excluded=["webrepl","_webrepl","port_diag","example_sub_led.py","example_pub_
↪button.py"]
```

`createstubs.py` will not process a module in either category.

Note that some of these modules are in fact included in the frozen modules that are gathered for those ports or boards

CHAPTER

SIX

CREATESTUB VARIANTS

There are two variant of the script available, in 3 levels of optimisation:

variant	full documented script	minified script (no logging)	cross-compiled script
full version	board/createstubs.py	minified/createstubs.py	minified/createstubs.mpy
memory optimized	board/createstubs_mem.py	minified/createstubs_mem.py	minified/createstubs_mem.mpy

In all cases the generation will take a few minutes (2-5 minutes) depending on the speed of the board and the number of included modules. As the stubs are generated on the board, and as MicroPython is highly optimized to deal with the scarce resources, this unfortunately does mean that the stubs lacks parameters details. So for these you must still use the documentation provided for that firmware.

6.1 board/createstubs.py

this is the core version of the script, and is fully self contained, but includes logging with requires the logging module to be avaialble on your device If your firmware does not include the `logging` module, you will need to upload this to your board, or use the minified version.

```
import createstubs
```

6.2 board/createstubs_mem.py

This variant of the createstubs.py script is optimised for use on low-memory devices, and reads the list of modules from a text file, rather than including it in the source file. as a result this requires an additional file `./modulelist.txt`, that should be uploaded to the device. If that cannot be found then only a single module (micropython) is stubbed.

```
import createstubs_mem
```

6.3 Optimisations

In order to run this on low-memory devices two additional steps are recommended:

- Minifification, using python-minifier to reduce overall size, and remove logging overhead. can be used on all devices
- Cross compilation, using mpy-cross, to avoid the compilation step on the micropython device. The cross-compiled version can only run on specific Micropython 1.12 or newer.

6.3.1 Minification

Minified versions, which requires less memory and only very basic logging. this removes the requirement for the logging module on the MCU.

Minifiacation helps reduce the seize of the script, and therefore of the memory usage. As a result the script becomes almost unreadable.

6.3.2 Cross compilation

this is specially suited for low memory devices such as the esp8622

CHAPTER

SEVEN

CPYTHON AND FROZEN MODULES

7.1 Frozen Modules

It is common for Firmwares to include a few (or many) python modules as 'frozen' modules. 'Freezing' modules is a way to pre-process .py modules so they're 'baked-in' to MicroPython' s firmware and use less memory. Once the code is frozen it can be quickly loaded and interpreted by MicroPython without as much memory and processing time.

Most OSS firmwares store these frozen modules as part of their repository, which allows us to:

1. Download the *.py from the (github) repo using `git clone` or a direct download
2. Extract and store the 'unfrozen' modules (ie the *.py files) in a _Frozen folder. if there are different port / boards or releases defined , there may be multiple folders such as:
   - stubs/micropython_1_12_frozen
     - /esp32
       - /GENERIC
       - /RELEASE
       - /TINYPICO
     - /stm32
       - /GENERIC
       - /PYBD_SF2
3. generate typeshed stubs of these files. (the .pyi files will be stored alongside the .py files)
4. Include/use them in the configuration

ref: https://learn.adafruit.com/micropython-basics-loading-modules/frozen-modules

7.2 Collect Frozen Stubs (micropython)

This is run daily though the github action workflow : get-all-frozen in the micropython-stubs repo.

If you want to run this manually

- Check out repos side-by-side:
  - micropython-stubs
  - micropython-stubber
  - micropython

  - micropython-lib

- link repos using all_stubs symlink
- checkout tag / version in the micropython folder
  (for most accurate results should checkout micropython-lib for the same date)
- run `src/get-frozen.py`
- run `src/update-stubs.py`
- create a PR for changes to the stubs repo

7.3 Postprocessing

You can run postprocessing for all stubs by running either of the two scripts. There is an optional parameter to specify the location of the stub folder. The default path is `./all_stubs`

Powershell:

```
./scripts/updates_stubs.ps1 [-path ./mystubs]
```

or python

```
python ./src/update_stubs.py [./mystubs]
```

This will generate or update the `.pyi` stubs for all new (and existing) stubs in the `./all_stubs` or specified folder.

From version '1.3.8' the `.pyi` stubs are generated using `stubgen`, before that the `make_stub_files.py` script was used.

Stubgen is run on each 'collected stub folder' (that contains a `modules.json` manifest) using the options : `--ignore-errors --include-private` and the resulting `.pyi` files are stored in the same folder (`foo.py` and `foo.pyi` are stored next to each other).

In some cases `stubgen` detects duplicate modules in a 'collected stub folder', and subsequently does not generate any stubs for any `.py` module or script. then **Plan B** is to run stubgen for each separate `*.py` file in that folder. THis is significantly slower and according to the stubgen documentation the resulting stubs may of lesser quality, but that is better than no stubs at all.

Note: In several cases `stubgen` creates folders in inappropriate locations (reason undetermined), which would cause issues when re-running `stubgen` at a later time. to compensate for this behaviour the known-incorrect .pyi files are removed before and after stubgen is run see: `cleanup(modules_folder)` in `utils.py`

CHAPTER

EIGHT

REPO STRUCTURE

- *This and sister repos*
- *Structure of this repo*
- *Naming Convention and Stub folder structure*
- 2 python versions

8.1 This and sister repos

repo	Why	Where	example
micropython-stubber	needed to make stubs	in your source folder	develop/micropython-stubber
micropython	to collect frozen modules	submodule of micropython-stubber	develop/micropython-stubber/micropython
micropython-lib	to collect frozen modules	submodule of micropython-stubber	develop/micropython-stubber/micropython-lib
micropython-stubs	stores collected stubs	next to the `stubber`	develop/micropython-stubs

Note:

- recommended is to create a symlink from `develop/micropython-stubber\all-stubs` to `develop/micropython-stubs`

Note:

- For Git submodules please refer to https://git-scm.com/book/en/v2/Git-Tools-Submodules

8.2 Structure of this repo

The file structure is based on my personal windows environment, but you should be able to adapt that without much hardship to you own preference and OS.

What	Details	Where
stub root	symlink to connect the 2 sister-repos	all_stubs
firmware stubber	MicroPython	board/createstubs.py
minified firmware stubber	MicroPython	minified/createstubs.py
PC based scripts	CPython	src/*
PC based scripts	CPython	process.py
pytest tests		test/*

8.3 Naming Convention and Stub folder structure

What	Why	Where
stub root	connect the 2 repos	all_stubs
cpython stubs for micropython core	adapt for differences between CPython and MicroPython	stubs/cpython-core
generated stub files	needed to use stubs	stubs/{firmware}-{port}-{version}-frozen
Frozen stub files	better code intellisense	stubs/{firmware}-{version}-frozen

Note: I found that, for me, using submodules caused more problems than it solved. So instead I link the two main repo's using a *symlink*.

Note: I in the repo tests I have used the folders `TESTREPO-micropython` and `TESTREPO-micropython-lib` to avoid conflicts with any development that you might be doing on similar `micropython` repos at the potential cost of a little disk space.

```
cd /develop

git clone  https://github.com/josverl/micropython-stubber.git
git clone  https://github.com/josverl/micropython-stubs.git
git clone  https://github.com/micropython/micropython.git
git clone  https://github.com/micropython/micropython.git
```

8.4 Create a symbolic link

To create the symbolic link to the `../micropython-stubs/stubs` folder the instructions differ slightly for each OS/ The below examples assume that the micropython-stubs repo is cloned 'next-to' your project folder. please adjust as needed.

8.4.1 Windows 10

Requires `Developer enabled` or elevated powershell prompt.

```
# target must be an absolute path, resolve path is used to resolve the relative path to␣
↪absolute
New-Item -ItemType SymbolicLink -Path "all-stubs" -Target (Resolve-Path -Path ../
↪micropython-stubs/stubs)
```

or use mklink in an (elevated) command prompt

```
rem target must be an absolute path
mklink /d all-stubs c:\develop\micropython-stubs\stubs
```

8.4.2 Linux/Unix/Mac OS

```
# target must be an absolute path
ln -s /path/to/micropython-stubs/stubs all-stubs
```

CHAPTER

NINE

POWERSHELL SCRIPTS

A number of scripts have been written in PowerShell as that is one of my preferred scripting languages. Possibly these scripts could be ported to python , at the cost of more complex handling of OS processes and paths and ports.

(a PR with a port to Python would be appreciated)

9.1 bulk_stubber.ps1

The goal of this script is to run create_stubs on a set of boards connected to my machine in order to generate new stubs for multiple micropython versions

high level operation:

- Scans the serial ports for connected esp32 and esp8266 devices using `get-serialport.ps1 -chip`
- Uses a (hardcoded) list of firmwares including version + chip type
- for each firmware in that list:
  - Selects the corresponding device and serialport
  - Flashes the micropython version to the device using `flash_MPY.ps1`
  - waits for the device to finish processing any initial tasks (file system creation etc)

```
rshell -p $serialport  --rts 1 repl "~ print('connected') ~"
```

> **Note:** This is quite sensitive to timing and requires some delays to allow the device to restart before the script continues.
>
> Also a bit of automated manipulation of the RTS (and DTR) signals is needed to avoid needing to press a device's reset button.

  - Starts the minified version of createstubs.py

```
$createstubs_py = join-path $WSRoot "minified/createstubs.py"
pyboard --device $serialport $createstubs_py | write-host
```

  - Downloads the generated machine-stubs

```
# reverse sync
# $dest = path relative to current directory
# $source = path on board ( all boards are called pyboard)
```

(continues on next page)

(continued from previous page)

```
$source = "/pyboard/stubs"
rshell -p $serialport --buffer-size 512 rsync $source $subfolder  | write-host
```

9.1.1 Minificantion and compilation

in order to allow createstubs to be run on low-memory devices there are a few steps needed to allow for sufficient memory

9.1.2 Requirements & dependencies

Python

- esptool - to flash new firmware to the esp32 and esp8266
- pyboard.py - to upload files and run commands (not the old version on PyPi)
- rshell - to download the folder with stubs

PowerShell ../../Firmware

- get-serialport.ps1
- flash_MPY.ps1

9.1.3 Hardware

- ESP32 board + SPIRAM on USB + Serial drivers
- ESP8266 board on USB + Serial drivers

Note: Multiple boards can be connected at the same time. The script will select the first board of the corresponding type. If a board-type is not present, then no stubs for that device type will be generated.

CHAPTER

TEN

OVERVIEW OF STUBS

Initially I also stored all the generated subs in the same repo. That turned out to be a bit of a hassle and since then I have moved all the stubs to the micropython-stubs repo

Below are the most relevant stub sources referenced in this project.

10.1 Firmware and libraries

10.1.1 MicroPython firmware and frozen modules *[MIT]*

https://github.com/micropython/micropython

https://github.com/micropython/micropython-lib

10.1.2 Pycopy firmware and frozen modules *[MIT]*

https://github.com/pfalcon/pycopy

https://github.com/pfalcon/pycopy-lib

10.1.3 LoBoris ESP32 firmware and frozen modules *[MIT, Apache 2]*

https://github.com/loboris/MicroPython_ESP32_psRAM_LoBo

10.2 Included custom stubs

Github repo	Contributions	License
pfalcon/micropython-lib	CPython backports	MIT
dastultz/micropython-pyb	a pyb.py file for use with IDEs in developing a project for the Pyboard	Apache 2

10.2.1 Stub source: MicroPython-lib > CPython backports *[MIT, Python]*

While micropython-lib focuses on MicroPython, sometimes it may be beneficial to run MicroPython code using CPython, e.g. to use code coverage, debugging, etc. tools available for it. To facilitate such usage, micropython-lib also provides re-implementations ("backports") of MicroPython modules which run on CPython. https://github.com/pfalcon/micropython-lib#cpython-backports

10.2.2 micropython_pyb *[Apache 2]*

This project provides a pyb.py file for use with IDEs in developing a project for the Pyboard. https://github.com/dastultz/micropython-pyb

CHAPTER

ELEVEN

REFERENCES

11.1 Inspiration

11.1.1 Thonny - MicroPython _cmd_dump_api_info *[MIT License]*

The `createstubs.py` script to create the stubs is based on the work of Aivar Annamaa and the Thonny crew. It is somewhere deep in the code and is apparently only used during the development cycle but it showed a way how to extract/generate a representation of the MicroPython modules written in C

While the concepts remain, the code has been rewritten to run on a micropython board, rather than on a connected PC running CPython. Please refer to : Thonny code sample

11.1.2 MyPy Stubgen

MyPy stubgen is used to generate stubs for the frozen modules and for the `*.py` stubs that were generated on a board.

11.1.3 make_stub_files *[Public Domain]*

https://github.com/edreamleo/make-stub-files

This script `make_stub_files.py` makes a stub (.pyi) file in the output directory for each source file listed on the command line (wildcard file names are supported).

The script does no type inference. Instead, the user supplies patterns in a configuration file. The script matches these patterns to: The names of arguments in functions and methods and The text of return expressions. Return expressions are the actual text of whatever follows the "return" keyword. The script removes all comments in return expressions and converts all strings to "str". This preprocessing greatly simplifies pattern matching.

Note: It was found that the stubs / prototypes of some functions with complex arguments were not handled correctly, resulting in incorrectly formatted stubs (.pyi)
Therefore this functionality has been replaced by MyPy `stubgen`

11.2 Documentation on Type hints

- Type hints cheat sheet
- PEP 3107 – Function Annotations
- PEP 484 – Type Hints
- Optional Static Typing for Python
- TypeShed
- SO question

CHAPTER

TWELVE

DOCUMENTATION

This documentation is built using Sphinxs with the bulk of the documents written in markdown and hosted on read the docs.

The markdown files are processed using Myst

Some diagrams have been generated using Mermaid and integrated using the Mermaid plugin

Documentation for the scripts is created using the Spinxs AutoApi plugin

CHAPTER

THIRTEEN

CHANGELOG

13.1 Documentation Stubs

- avoid the use of BaseException

13.2 createstubs.py - v1.4.3

- significant memory optimisation for use on low-memory devices such as the esp8266 family
  - load the list of modules to be stubbed from a text file rather than as part of the source
  - use both minification and the `mpy-cross` compiler to reduce the claim on memory (RAM)

Warning: This is a potential breaking change for external tools that expect to either directly execute the script or upload only a single file to an MCU in order to stub.

- the current process is automated in `remote_stubber.ps1'

13.3 createstubs.py - v1.4.2

- Fixes a regression introduced in 1.4-beta where function definitions would include a self parameter.

13.4 minified createstubs.py - v1.4.1

- Switched to use python-minifier for the minification due to the end-of-life of the previous minification tool The new minification tool produces more compact code, although that is still not sufficient for some memory constrained devices.
  - there are no functional changes,
  - the detection of Micropython was adjusted to avoid the use of eval which blocked a minification rule
  - several tests were adjusted

13.5 documentation

- Add Sphinxs documentation
  - changelog
  - automatic API documentation for
    - createstubs.py (board)
    - scripts to run on PC / Github actions
- Publish documentation to readthedocs

13.6 createstubs - v1.4-beta

- createstubs.py
  - improvements to handle nested classes to be able to create stubs for lvgl. this should also benefit other more complex modules.
- added `stub_lvgl.py` helper script

13.7 createstubs.py - v1.3.16

- createstubs.py
  - fix for micropython v1.16
  - skip _test modules in module list
  - black formatting
  - addition of **init** methods (based on runtime / static)
  - class method decorator
  - additional type information for constants using comment style typing
  - detect if running on MicroPython or CPython
  - improve report formatting to list each module on a separate line to allow for better comparison
- workflows
  - move to ubuntu 20.04
    - move to test/tools/ubuntu_20_04/micropython_v1.xx
  - run more tests in GHA
- postprocessing
  - minification adjusted to work with `black`
  - use `mypy.stubgen`
  - run per folder
    - verify 1:1 relation .py-.pyi
    - run `mypy.stubgen` to generate missing .pyi files

  - publish test results to GH

- develop / repo setup
  - updated dev requirements (requirements-dev.txt)
  - enable developing on GitHub codespaces
  - switched to using submodules to remove external dependencies how to clone : `git submodule init` `git submodule update`
  - added black configuration file to avoid running black on minified version
  - switched to using .venv on all platforms
  - added and improved tests
    - test coverage increased to 82%
  - move to test/tools/ubuntu_20_04/micropython_v1.xx
    - for test (git workflows)
    - for tasks
  - make use of CPYTHON stubs to alle makestubs to run well on CPYTHON
    - allows pytest, and debugging of tests
  - add tasks to :
    - run createstubs in linux version

CHAPTER

FOURTEEN

TO-DO (PROVISIONAL)

14.1 working on it

14.1.1 read RST files

- add prototypes from RST ref: https://github.com/python/mypy/blob/master/mypy/stubdoc.py

14.1.2 documentation

- how to run post-processing
- how the debug setup works

14.1.3 stubber :

- document - that gc and sys modules are somehow ignored by pylint and will keep throwing errors
- add mpy information to manifest
- use 'nightly' naming convention in createstubs.py
- change firmware naming

14.1.4 frozen stubs

- add simple readme.md ?

14.1.5 Stub augmentation/ merging typeinformation from copied / generated type-rich info

https://libcst.readthedocs.io/en/latest/tutorial.html

- add prototypes from Source ? check if https://github.com/python/mypy/blob/master/mypy/stubgenc.py might be useful
- test to auto-merge common prototypes by stubber ie. add common return types to make_stub_files.cfg
- resolve import time issues

14.1.6 SYS en GC

#pylint: disable=no-member ## workaround for sys and gc

Module ‘sys’ has no ‘print_exception’ member Module ‘gc’ has no ‘mem_free’ member Module ‘gc’ has no ‘threshold’ member Module ‘gc’ has no ‘mem_free’ member Module ‘gc’ has no ‘mem_alloc’ member { “resource”: “/c:/develop/MyPython/ESP32-P1Meter/src/main.py”, “owner”: “python”, “code”: “no-member”, “severity”: 8, “message”: “Module ‘gc’ has no ‘mem_free’ member”, “source”: “pylint”, “startLineNumber”: 33, “startColumn”: 22, “endLineNumber”: 33, “endColumn”: 22 }

14.1.7 Webrepl

Unable to import ‘webrepl’ can include in common modules C:\develop\MyPython\micropython\extmod\webrepl\webrepl.py

CHAPTER

FIFTEEN

DEVELOPING

15.1 Cloning the repo

The repo uses two submodules in order to generate the MicroPython frozen stubs. In order to fully check-out the repo you need to run additional commands: how to clone inclusing submodules:

```
git clone https://github.com/Josverl/micropython-stubber.git
cd micropython-stubber
git submodule init
git submodule update --recursive
# install pyright
npm install
```

15.1.1 adding additional submodules

how to add :

```
git submodule add --force -b master https://github.com/micropython/micropython.git
git submodule add --force -b master https://github.com/micropython/micropython-lib.git
```

15.2 Windows 10

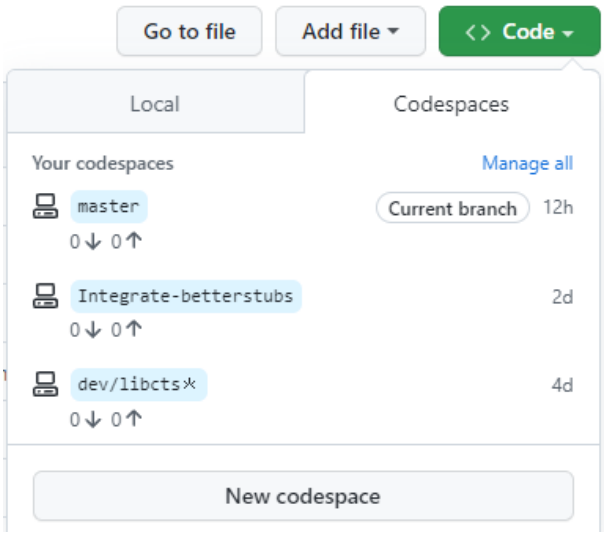
I use Windows 10/11 and use WSL2 to run the linux based parts. if you develop on other platform, it is quite likely that you may need to change some details. if that is needed , please update/add to the documentation and send a documentation PR.

- clone
- create python virtual environment (optional)
- install requirements-dev
- setup sister repos
- run test to verify setup

15.3 Github codespaces

Is is also possible to start a pre-configure development environment in GitHub Codespaces this is probably the fastest and quickest way to start developing.

Note that Codespaces is currently in an extended beta.



15.4 Wrestling with two pythons

This project combines CPython and MicroPython in one project. As a result you may/will need to switch the configuration of pylint and VSCode to match the section of code that you are working on. This is caused by the fact that pylint does not support per-folder configuration

to help switching there are 2 different .pylintrc files stored in the root of the project to simplify switching.

Similar changes will need to be done to the .vscode/settings.json

If / when we can get pylance to work with the micropython stubs , this may become simpler as Pylance natively supports multi-root workspaces, meaning that you can open multiple folders in the same Visual Studio Code session and have Pylance functionality in each folder.

15.5 Minification

If you make changes to the createstubs.py script , you should also update the minified version by running `python process.py minify` at some point.

If you forget to do this there is a github action that should do this for you and create a PR for your branch.

15.6 Testing

MicroPython-Stubber has a number of tests written in Pytest

see below overview

folder	what	how	used where
board	createstubs.pynormal & minified	runs createstubs.py on micropython-linux ports	WSL2 and github actions
check-out_repo	simple_git mod-uleretrieval of frozen modules	does not use mocking but actually retrieves different firmware versions locally using git or dowNloads modules for online	local win-dows
com-mon	all other tests	common	local + github action

Note: Also see *test documentation*

Platform detection to support pytest In order to allow both simple usability om MicroPython and testability on Full Python, createstubs does a runtime test to determine the actual platform it is running on while importing the module This is similar to using the `if __name__ == "__main__":` preamble If running on MicroPython, then it starts stubbing

```
if isMicroPython():
    main()
```

Testing on micropython linux port(s) In order to be able to test `createstubs.py`, it has been updated to run on linux, and accept a –path parameter to indicate the path where the stubs should be stored.

15.7 Debugging Cpython code that run Micropython

Some of the test code run the micropython executable using `subprocess.run()`. When you try to debug these tests the VSCode debugger (debugpy](https://github.com/microsoft/debugpy) then tries to attach to that micropython subprocess in order to facilitate debugging. This will fail as reported in this issue.

The solution to this problem is to disable subprocess debugging using the `"subProcess": false` switch.

```
// launch.json
        {
            // disable pytest coverage report as it conflicts with debugging tests
            "name": "Debug pytest tests",
            "type": "python",
            "purpose": [
                "debug-test"
            ],
            "console": "integratedTerminal",
            "justMyCode": false,
            "stopOnEntry": false,
            "subProcess": false, // Avoid debugpy trying to debug micropython
            "env": {
```

(continues on next page)

(continued from previous page)

```
                "PYTEST_ADDOPTS": "--no-cov"
            }
        },
```

15.8 github actions

15.8.1 pytests.yml

This workflow will :

- test the workstation scripts
- test the createstubs.py script on multiple micropython linux versions
- test the minified createstubs.py script on multiple micropython linux versions

15.8.2 run minify-pr.yml

This workflow will :

- create a minified version of createstubs.py
- run a quick test on that
- and submit a PR to the branch -minify

CHAPTER

SIXTEEN

TESTING

A significant number of tests have been created in pytest.

- The tests are located in the `tests` folder.
- The `tests/data` folder contains folders with subs that are used to verify the correct working of the minification modules
- debugging the tests only works if –no-cov is specified for pytest

16.1 testing & debugging createstubs.py

- the `tests\mocks` folder contains mock-modules that allow the micropython code to be run in CPython. This is used by the unit tests that verify `createstubs.py` and it minified version.
- in order to load / debug the test the python path needs to include the cpython_core modules (Q&D)
- mocking cpython_core/os is missing the implementation attribute so that has been added (Q&D)

16.2 platform detection

In order to allow both simple usability om MicroPython and testability on *full* Python, createstubs does a runtime test to determine the actual platform it is running on while importing the module

This is similar to using the `if __name__ == "__main__":` preamble

```
if isMicroPython():
    main()
```

This allows pytest test running on full Python to import `createstubs.py` and run tests against individual methods, while allowing the script to run directly on import on a MicroPython board.

Note: Some test are platform dependent and have been marked to only run on linux or windows

16.3 Code Coverage

Code coverage is measured and reported in the `coverage/index.html` report. This report is not checked in to the repo, and therefore is only

CHAPTER

SEVENTEEN

API REFERENCE

This page contains auto-generated API reference documentation[1].

17.1 createstubs_mem

Create stubs for (all) modules on a MicroPython board.

> This variant of the createstubs.py script is optimised for use on low-memory devices, and reads the list of modules from a text file *./modulelist.txt* that should be uploaded to the device. If that cannot be found then only a single module (micropython) is stubbed. In order to run this on low-memory devices two additioanl steps are recommended: - minifification, using python-minifier
>
> to reduce overall size, and remove logging overhead.
>
> - cross compilation, using mpy-cross, to avoid the compilation step on the micropython device
>
> you can find a cross-compiled version located here: .minifiedcreatestubs_mem.mpy

Copyright (c) 2019-2021 Jos Verlinde

17.1.1 Module Contents

Classes

Stubber	Generate stubs for modules in firmware

Functions

resetWDT()	
ensure_folder(path: str)	Create nested folders if needed
_info()	collect base information on this runtime
get_root() → str	Determine the root folder of the device
show_help()	
read_path() → str	get --path from cmdline. [unix/win]
isMicroPython() → bool	runtime test to determine full or micropython

continues on next page

[1] Created with sphinx-autoapi

Table 2 – continued from previous page

main()

Attributes

__version__
ENOENT
_MAX_CLASS_LEVEL
_log

createstubs_mem.**__version__ = 1.4.4**

createstubs_mem.**ENOENT = 2**

createstubs_mem.**_MAX_CLASS_LEVEL = 2**

createstubs_mem.**resetWDT**()

class createstubs_mem.**Stubber**(*path: str = None*, *firmware_id: str = None*)
Generate stubs for modules in firmware

Parameters

- **path** (*str*) –
- **firmware_id** (*str*) –

get_obj_attributes(*self*, *item_instance: object*)
extract information of the objects members and attributes

Parameters **item_instance** (*object*) –

add_modules(*self*, *modules: list*)
Add additional modules to be exported

Parameters **modules** (*list*) –

create_all_stubs(*self*)
Create stubs for all configured modules

create_one_stub(*self*, *module_name*)

create_module_stub(*self*, *module_name: str*, *file_name: str = None*)
Create a Stub of a single python module

Args: - module_name (str): name of the module to document. This module will be imported. - file_name (Optional[str]): the 'path/filename.py' to write to. If omitted will be created based on the module name.

Parameters

- **module_name** (*str*) –
- **file_name** (*str*) –

write_object_stub(*self*, *fp*, *object_expr: object*, *obj_name: str*, *indent: str*, *in_class: int = 0*)
Write a module/object stub to an open file. Can be called recursive.

Parameters

- **object_expr** (*object*) –
- **obj_name** (*str*) –
- **indent** (*str*) –
- **in_class** (*int*) –

property flat_fwid(*self*)
Turn _fwid from 'v1.2.3' into '1_2_3' to be used in filename

clean(*self*, *path: str = None*)
Remove all files from the stub folder

Parameters **path** (*str*) –

report(*self*, *filename: str = 'modules.json'*)
create json with list of exported modules

Parameters **filename** (*str*) –

createstubs_mem.**ensure_folder**(*path: str*)
Create nested folders if needed

Parameters **path** (*str*) –

createstubs_mem.**_info**()
collect base information on this runtime

createstubs_mem.**get_root**() → str
Determine the root folder of the device

Return type str

createstubs_mem.**show_help**()

createstubs_mem.**read_path**() → str
get –path from cmdline. [unix/win]

Return type str

createstubs_mem.**isMicroPython**() → bool
runtime test to determine full or micropython

Return type bool

createstubs_mem.**main**()

createstubs_mem.**_log**

17.2 stub_lvgl

Helper module to create stubs for the lvgl modules. Note that the stubs can be very large, and it may be best to directly store them on an SD card if your device supports this.

17.2.1 Module Contents

Functions

main()	Create stubs for the lvgl modules using the lvlg version number.

stub_lvgl.**main**()
Create stubs for the lvgl modules using the lvlg version number.

17.3 main

17.3.1 Module Contents

Functions

countdown()

main.**countdown**()

17.4 createstubs_db

Create stubs for (all) modules on a MicroPython board.

This variant of the createstubs.py script is optimized for use on very-low-memory devices. Note: this version has undergone limited testing.

1) reads the list of modules from a text file *./modulelist.txt* that should be uploaded to the device.
2) **creates a btree database of the files that should be stubbed**
    - todo:
    - add a main.py that starts stubbing
3) **process the modules in the database:**
    - stub the module
    - reboots the device if it runs out of memory
4) creates the modules.json

If that cannot be found then only a single module (micropython) is stubbed. In order to run this on low-memory devices two additional steps are recommended: - minification, using python-minifier

to reduce overall size, and remove logging overhead.

- cross compilation, using mpy-cross, to avoid the compilation step on the micropython device

You should find a cross-compiled version located here: `./minified/createstubs_db.mpy

17.4.1 Module Contents

Classes

Stubber	Generate stubs for modules in firmware

Functions

resetWDT()	
ensure_folder(path: str)	Create nested folders if needed
_info()	collect base information on this runtime
get_root() → str	Determine the root folder of the device
show_help()	
read_path() → str	get --path from cmdline. [unix/win]
isMicroPython() → bool	runtime test to determine full or micropython
main_esp8266()	

Attributes

__version__
ENOENT
_MAX_CLASS_LEVEL
_log

createstubs_db.**__version__ = 1.4.4**

createstubs_db.**ENOENT = 2**

createstubs_db.**_MAX_CLASS_LEVEL = 2**

createstubs_db.**resetWDT**()

class createstubs_db.**Stubber**(*path: str = None*, *firmware_id: str = None*)
Generate stubs for modules in firmware

Parameters

- **path** (*str*) –
- **firmware_id** (*str*) –

get_obj_attributes(*self*, *item_instance: object*)
extract information of the objects members and attributes

Parameters **item_instance** (*object*) –

add_modules(*self*, *modules: list*)
Add additional modules to be exported

Parameters **modules** (*list*) –

create_all_stubs(*self*)
Create stubs for all configured modules

create_one_stub(*self*, *module_name*)

create_module_stub(*self*, *module_name: str*, *file_name: str = None*)
Create a Stub of a single python module

Args: - module_name (str): name of the module to document. This module will be imported. - file_name (Optional[str]): the 'path/filename.py' to write to. If omitted will be created based on the module name.

Parameters

- **module_name** (*str*) –
- **file_name** (*str*) –

write_object_stub(*self*, *fp*, *object_expr: object*, *obj_name: str*, *indent: str*, *in_class: int = 0*)
Write a module/object stub to an open file. Can be called recursive.

Parameters

- **object_expr** (*object*) –
- **obj_name** (*str*) –
- **indent** (*str*) –
- **in_class** (*int*) –

property flat_fwid(*self*)
Turn _fwid from 'v1.2.3' into '1_2_3' to be used in filename

clean(*self*, *path: str = None*)
Remove all files from the stub folder

Parameters **path** (*str*) –

report(*self*, *filename: str = 'modules.json'*)
create json with list of exported modules

Parameters **filename** (*str*) –

createstubs_db.**ensure_folder**(*path: str*)
Create nested folders if needed

Parameters **path** (*str*) –

createstubs_db.**_info**()
collect base information on this runtime

createstubs_db.**get_root**() → str
Determine the root folder of the device

Return type str

createstubs_db.**show_help**()

createstubs_db.**read_path**() → str
get –path from cmdline. [unix/win]

Return type str

createstubs_db.**isMicroPython**() → bool
runtime test to determine full or micropython

Return type bool

createstubs_db.**main_esp8266**()

createstubs_db.**_log**

17.5 createstubs

Create stubs for (all) modules on a MicroPython board Copyright (c) 2019-2021 Jos Verlinde

17.5.1 Module Contents

Classes

Stubber	Generate stubs for modules in firmware

Functions

resetWDT()	
ensure_folder(path: str)	Create nested folders if needed
_info()	collect base information on this runtime
get_root() → str	Determine the root folder of the device
show_help()	
read_path() → str	get --path from cmdline. [unix/win]
isMicroPython() → bool	runtime test to determine full or micropython
main()	

Attributes

__version__
ENOENT
_MAX_CLASS_LEVEL
_log

createstubs.**__version__** = 1.4.4

createstubs.**ENOENT** = 2

createstubs.**_MAX_CLASS_LEVEL** = 2

createstubs.**resetWDT**()

class createstubs.**Stubber**(*path: str = None*, *firmware_id: str = None*)
Generate stubs for modules in firmware

Parameters

- **path** (*str*) –
- **firmware_id** (*str*) –

get_obj_attributes(*self*, *item_instance: object*)
extract information of the objects members and attributes

Parameters **item_instance** (*object*) –

add_modules(*self*, *modules: list*)
Add additional modules to be exported

Parameters **modules** (*list*) –

create_all_stubs(*self*)
Create stubs for all configured modules

create_one_stub(*self*, *module_name*)

create_module_stub(*self*, *module_name: str*, *file_name: str = None*)
Create a Stub of a single python module

Args: - module_name (str): name of the module to document. This module will be imported. - file_name (Optional[str]): the 'path/filename.py' to write to. If omitted will be created based on the module name.

Parameters

- **module_name** (*str*) –
- **file_name** (*str*) –

write_object_stub(*self*, *fp*, *object_expr: object*, *obj_name: str*, *indent: str*, *in_class: int = 0*)
Write a module/object stub to an open file. Can be called recursive.

Parameters

- **object_expr** (*object*) –
- **obj_name** (*str*) –
- **indent** (*str*) –
- **in_class** (*int*) –

property flat_fwid(*self*)
Turn _fwid from 'v1.2.3' into '1_2_3' to be used in filename

clean(*self*, *path: str = None*)
Remove all files from the stub folder

Parameters **path** (*str*) –

report(*self*, *filename: str = 'modules.json'*)
create json with list of exported modules

Parameters **filename** (*str*) –

createstubs.**ensure_folder**(*path: str*)
Create nested folders if needed

Parameters **path** (*str*) –

createstubs._info()
collect base information on this runtime

createstubs.get_root() → str
Determine the root folder of the device

Return type str

createstubs.show_help()

createstubs.read_path() → str
get –path from cmdline. [unix/win]

Return type str

createstubs.isMicroPython() → bool
runtime test to determine full or micropython

Return type bool

createstubs.main()

createstubs._log

17.6 get_mpy

Collect modules and python stubs from MicroPython source projects (v1.12 +) and stores them in the all_stubs folder
The all_stubs folder should be mapped/symlinked to the micropython_stubs/stubs repo/folder

17.6.1 Module Contents

Classes

IncludeOptions

Functions

freeze_as_mpy(path, script=None, opt=0)	
freeze_as_str(path)	
freeze_mpy(path, script=None, opt=0)	Freeze the input (see above), which must be .mpy files that are
freeze(path, script=None, opt=0)	Freeze the input, automatically determining its type. A .py script
freeze_internal(path: str, script: str)	Copy the to-be-frozen module to the destination folder to be stubbed.
include(manifest, **kwargs)	Include another manifest.
convert_path(path)	Perform variable substitution in path
get_frozen(stub_path: str, version: str, mpy_path: str = None, lib_path: str = None)	get and parse the to-be-frozen .py modules for micropython to extract the static type information

continues on next page

Table 13 – continued from previous page

get_frozen_folders(stub_path: str, mpy_path: str, lib_path: str, version: str)	get and parse the to-be-frozen .py modules for micropython to extract the static type information
get_target_names(path: str) → tuple	get path to port and board names from a path
get_frozen_manifest(manifests, stub_path: str, mpy_path: str, lib_path: str, version: str)	get and parse the to-be-frozen .py modules for micropython to extract the static type information

Attributes

log
FAMILY
path_vars
stub_dir
mpy_path

get_mpy.**log**

get_mpy.**FAMILY = micropython**

get_mpy.**path_vars**

get_mpy.**stub_dir**

exception get_mpy.**FreezeError**

Bases: Exception

Common base class for all non-exit exceptions.

class get_mpy.**IncludeOptions**(***kwargs*)

defaults(*self*, ***kwargs*)

__getattr__(*self*, *name*)

get_mpy.**freeze_as_mpy**(*path*, *script=None*, *opt=0*)

get_mpy.**freeze_as_str**(*path*)

get_mpy.**freeze_mpy**(*path*, *script=None*, *opt=0*)

Freeze the input (see above), which must be .mpy files that are frozen directly.

get_mpy.**freeze**(*path*, *script=None*, *opt=0*)

Freeze the input, automatically determining its type. A .py script will be compiled to a .mpy first then frozen, and a .mpy file will be frozen directly.

path must be a directory, which is the base directory to search for files from. When importing the resulting frozen modules, the name of the module will start after *path*, ie *path* is excluded from the module name.

If *path* is relative, it is resolved to the current manifest.py. Use $(MPY_DIR), $(MPY_LIB_DIR), $(PORT_DIR), $(BOARD_DIR) if you need to access specific paths.

If *script* is None all files in *path* will be frozen.

If *script* is an iterable then freeze() is called on all items of the iterable (with the same *path* and *opt* passed through).

If *script* is a string then it specifies the filename to freeze, and can include extra directories before the file. The file will be searched for in *path*.

opt is the optimisation level to pass to mpy-cross when compiling .py to .mpy. (ignored in this implementation)

get_mpy.**freeze_internal**(*path: str*, *script: str*)
Copy the to-be-frozen module to the destination folder to be stubbed.

Parameters: path (str) : the destination script (str): the source script to be frozen

Parameters

- **path** (*str*) –
- **script** (*str*) –

get_mpy.**include**(*manifest*, ***kwargs*)
Include another manifest.

The manifest argument can be a string (filename) or an iterable of strings.

Relative paths are resolved with respect to the current manifest file.

Optional kwargs can be provided which will be available to the included script via the *options* variable.

e.g. include("path.py", extra_features=True)

in path.py: options.defaults(standard_features=True)

freeze minimal modules. if options.standard_features:

freeze standard modules.

if options.extra_features: # freeze extra modules.

get_mpy.**convert_path**(*path*)
Perform variable substitution in path

get_mpy.**get_frozen**(*stub_path: str*, *version: str*, *mpy_path: str = None*, *lib_path: str = None*)

get and parse the to-be-frozen .py modules for micropython to extract the static type information

- requires that the MicroPython and Micropython-lib repos are checked out and available on a local path
- repos should be cloned side-by-side as some of the manifests refer to micropython-lib scripts using a relative path

Parameters

- **stub_path** (*str*) –
- **version** (*str*) –
- **mpy_path** (*str*) –
- **lib_path** (*str*) –

get_mpy.**get_frozen_folders**(*stub_path: str*, *mpy_path: str*, *lib_path: str*, *version: str*)
get and parse the to-be-frozen .py modules for micropython to extract the static type information locates the to-be-frozen files in modules folders - 'ports/<port>/modules/*.py' - 'ports/<port>/boards/<board>/modules/*.py'

Parameters

- **stub_path** (*str*) –
- **mpy_path** (*str*) –
- **lib_path** (*str*) –
- **version** (*str*) –

get_mpy.**get_target_names**(*path: str*) → tuple
get path to port and board names from a path

Parameters **path** (*str*) –

Return type tuple

get_mpy.**get_frozen_manifest**(*manifests*, *stub_path: str*, *mpy_path: str*, *lib_path: str*, *version: str*)
get and parse the to-be-frozen .py modules for micropython to extract the static type information locates the to-be-frozen files through the manifest.py introduced in MicroPython 1.12 - manifest.py is used for board specific and daily builds - manifest_release.py is used for the release builds

Parameters

- **stub_path** (*str*) –
- **mpy_path** (*str*) –
- **lib_path** (*str*) –
- **version** (*str*) –

get_mpy.**mpy_path = ./micropython**

17.7 get_lobo

Collect modules and python stubs from the Loboris MicroPython source project and stores them in the all_stubs folder The all_stubs folder should be mapped/symlinked to the micropython_stubs/stubs repo/folder

17.7.1 Module Contents

Functions

get_frozen(stub_path=None, *, repo=None, version='3.2.24')	Loboris frozen modules

Attributes

FAMILY
PORT
log

get_lobo.**FAMILY = loboris**

get_lobo.**PORT = esp32_lobo**

get_lobo.**log**

get_lobo.**get_frozen**(*stub_path=None*, ***, *repo=None*, *version='3.2.24'*)
Loboris frozen modules

17.8 utils

17.8.1 Module Contents

Functions

clean_version(version: str, *, build: bool = False, patch: bool = False, commit: bool = False, drop_v: bool = False, flat: bool = False)	Clean up and transform the many flavours of versions
stubfolder(path: str) → str	return path in the stub folder
flat_version(version: str, keep_v: bool = False)	Turn version from 'v1.2.3' into '1_2_3' to be used in file-name
cleanup(modules_folder: pathlib.Path, all_pyi: bool = False)	Q&D cleanup
generate_pyi_from_file(file: pathlib.Path) → bool	Generate a .pyi stubfile from a single .py module using mypy/stubgen
generate_pyi_files(modules_folder: pathlib.Path) → bool	generate typeshed files for all scripts in a folder using mypy/stubgen
manifest(family: str = 'micropython', stubtype: str = 'frozen', machine: Optional[str] = None, port: Optional[str] = None, platform: Optional[str] = None, sysname: Optional[str] = None, nodename: Optional[str] = None, version: Optional[str] = None, release: Optional[str] = None, firmware: Optional[str] = None) → dict	create a new empty manifest dict
make_manifest(folder: pathlib.Path, family: str, port: str, version: str, release: str = '', stubtype: str = '', board: str = '') → bool	Create a *module.json* manifest listing all files/stubs in this folder and subfolders.
generate_all_stubs()	just create typeshed stubs
read_exclusion_file(path: pathlib.Path = None) → List[str]	Read a .exclusion file to determine which files should not be automatically re-generated
should_ignore(file: str, exclusions: List[str]) → bool	Check if a file matches a line in the exclusion list.

Attributes

log
STUB_FOLDER
LATEST

utils.**log**

utils.**STUB_FOLDER = ./all-stubs**

utils.**LATEST = Latest**

utils.**clean_version**(*version: str*, *, *build: bool = False*, *patch: bool = False*, *commit: bool = False*, *drop_v: bool = False*, *flat: bool = False*)
Clean up and transform the many flavours of versions

Parameters

- **version** (*str*) –
- **build** (*bool*) –
- **patch** (*bool*) –
- **commit** (*bool*) –
- **drop_v** (*bool*) –
- **flat** (*bool*) –

utils.**stubfolder**(*path: str*) → str
return path in the stub folder

Parameters **path** (*str*) –

Return type str

utils.**flat_version**(*version: str*, *keep_v: bool = False*)
Turn version from 'v1.2.3' into '1_2_3' to be used in filename

Parameters

- **version** (*str*) –
- **keep_v** (*bool*) –

utils.**cleanup**(*modules_folder: pathlib.Path*, *all_pyi: bool = False*)
Q&D cleanup

Parameters

- **modules_folder** (*pathlib.Path*) –
- **all_pyi** (*bool*) –

utils.**generate_pyi_from_file**(*file: pathlib.Path*) → bool
Generate a .pyi stubfile from a single .py module using mypy/stubgen

Parameters **file** (*pathlib.Path*) –

Return type bool

utils.**generate_pyi_files**(*modules_folder: pathlib.Path*) → bool
generate typeshed files for all scripts in a folder using mypy/stubgen

Parameters **modules_folder** (*pathlib.Path*) –

Return type bool

utils.**manifest**(*family: str = 'micropython'*, *stubtype: str = 'frozen'*, *machine: Optional[str] = None*, *port: Optional[str] = None*, *platform: Optional[str] = None*, *sysname: Optional[str] = None*, *nodename: Optional[str] = None*, *version: Optional[str] = None*, *release: Optional[str] = None*, *firmware: Optional[str] = None*) → dict
create a new empty manifest dict

Parameters

- **family** (*str*) –
- **stubtype** (*str*) –
- **machine** (*Optional[str]*) –
- **port** (*Optional[str]*) –
- **platform** (*Optional[str]*) –
- **sysname** (*Optional[str]*) –
- **nodename** (*Optional[str]*) –
- **version** (*Optional[str]*) –
- **release** (*Optional[str]*) –
- **firmware** (*Optional[str]*) –

Return type dict

utils.**make_manifest**(*folder: pathlib.Path*, *family: str*, *port: str*, *version: str*, *release: str = ''*, *stubtype: str = ''*, *board: str = ''*) → bool

Create a *module.json* manifest listing all files/stubs in this folder and subfolders.

Parameters

- **folder** (*pathlib.Path*) –
- **family** (*str*) –
- **port** (*str*) –
- **version** (*str*) –
- **release** (*str*) –
- **stubtype** (*str*) –
- **board** (*str*) –

Return type bool

utils.**generate_all_stubs**()

just create typeshed stubs

utils.**read_exclusion_file**(*path: pathlib.Path = None*) → List[str]

Read a .exclusion file to determine which files should not be automatically re-generated in .GitIgnore format

Parameters **path** (*pathlib.Path*) –

Return type List[str]

utils.**should_ignore**(*file: str*, *exclusions: List[str]*) → bool

Check if a file matches a line in the exclusion list.

Parameters

- **file** (*str*) –
- **exclusions** (*List[str]*) –

Return type bool

17.9 get_cpython

Download or update the micropyton compatibility modules from pycopy and stores them in the all_stubs folder The all_stubs folder should be mapped/symlinked to the micropython_stubs/stubs repo/folder

17.9.1 Module Contents

Functions

get_core(requirements, stub_path=None)	Download MicroPython compatibility modules

Attributes

log
family

get_cpython.**log**

get_cpython.**family = common**

get_cpython.**get_core**(*requirements*, *stub_path=None*)
Download MicroPython compatibility modules

17.10 get_all_frozen

Collect modules and python stubs from other projects and stores them in the all_stubs folder The all_stubs folder should be mapped/symlinked to the micropython_stubs/stubs repo/folder

1) get cpython core modules
2) get micropython frozen modules for the CURRENT checked out version
3) get Loboris frozen modules (no longer maintained)
4) Generate/update type hint files (pyi) for all stubs.

17.10.1 Module Contents

Functions

get_all(stub_folder: str = STUB_FOLDER, mpy_folder: str = MPY_FOLDER, mpy_lib_folder: str = MPY_LIB_FOLDER, version: str = '', core: bool = False, core_type: str = 'pycopy', mpy: bool = False, lobo: bool = False, pyi: bool = True, all: bool = False)	get all frozen modules for the current version of micropython

Attributes

log
STUB_FOLDER
MPY_FOLDER
MPY_LIB_FOLDER

get_all_frozen.**log**

get_all_frozen.**STUB_FOLDER = ./all-stubs**

get_all_frozen.**MPY_FOLDER = ./micropython**

get_all_frozen.**MPY_LIB_FOLDER = ./micropython-lib**

get_all_frozen.**get_all**(*stub_folder: str = STUB_FOLDER*, *mpy_folder: str = MPY_FOLDER*, *mpy_lib_folder: str = MPY_LIB_FOLDER*, *version: str = ''*, *core: bool = False*, *core_type: str = 'pycopy'*, *mpy: bool = False*, *lobo: bool = False*, *pyi: bool = True*, *all: bool = False*)

get all frozen modules for the current version of micropython

Parameters

- **stub_folder** (*str*) –
- **mpy_folder** (*str*) –
- **mpy_lib_folder** (*str*) –
- **version** (*str*) –
- **core** (*bool*) –
- **core_type** (*str*) –
- **mpy** (*bool*) –
- **lobo** (*bool*) –
- **pyi** (*bool*) –
- **all** (*bool*) –

17.11 basicgit

simple Git module, where needed via powershell

17.11.1 Module Contents

Functions

_run_git(cmd: List[str], repo: str = None, expect_stderr=False)	run a external (git) command in the repo's folder and deal with some of the errors
get_tag(repo: str = None, abbreviate: bool = True) → Union[str, None]	get the most recent git version tag of a local repo
checkout_tag(tag: str, repo: str = None) → bool	get the most recent git version tag of a local repo"
switch_tag(tag: str, repo: str = None) → bool	get the most recent git version tag of a local repo"
switch_branch(branch: str, repo: str = None) → bool	get the most recent git version tag of a local repo"
fetch(repo: str) → bool	fetches a repo
pull(repo: str, branch='master') → bool	pull a repo origin into master

basicgit._run_git(*cmd: List[str]*, *repo: str = None*, *expect_stderr=False*)
run a external (git) command in the repo's folder and deal with some of the errors

Parameters

- **cmd** (*List[str]*) –
- **repo** (*str*) –

basicgit.get_tag(*repo: str = None*, *abbreviate: bool = True*) → Union[str, None]
get the most recent git version tag of a local repo repo should be in the form of : repo = "./micropython"

returns the tag or None

Parameters

- **repo** (*str*) –
- **abbreviate** (*bool*) –

Return type Union[str, None]

basicgit.checkout_tag(*tag: str*, *repo: str = None*) → bool
get the most recent git version tag of a local repo" repo should be in the form of : repo = "../micropython/.git"

returns the tag or None

Parameters

- **tag** (*str*) –
- **repo** (*str*) –

Return type bool

basicgit.switch_tag(*tag: str*, *repo: str = None*) → bool
get the most recent git version tag of a local repo" repo should be in the form of : path/.git repo = '../micropython/.git' returns the tag or None

Parameters

- **tag** (*str*) –
- **repo** (*str*) –

Return type bool

basicgit.**switch_branch**(*branch: str*, *repo: str = None*) → bool
get the most recent git version tag of a local repo" repo should be in the form of : path/.git repo = '../micropython/.git' returns the tag or None

Parameters

- **branch** (*str*) –
- **repo** (*str*) –

Return type bool

basicgit.**fetch**(*repo: str*) → bool
fetches a repo repo should be in the form of : path/.git repo = '../micropython/.git' returns True on success

Parameters **repo** (*str*) –

Return type bool

basicgit.**pull**(*repo: str*, *branch='master'*) → bool
pull a repo origin into master repo should be in the form of : path/.git repo = '../micropython/.git' returns True on success

Parameters **repo** (*str*) –

Return type bool

17.12 stubs_from_docs

Read the Micropython library documentation files and use them to build stubs that can be used for static typechecking using a custom-built parser to read and process the micropython RST files - generates:

- **modules**
  - docstrings
- **function definitions**
  - function parameters based on documentation
  - docstrings

classes

- docstrings
- __init__ method
- parameters based on documentation for class
- **methods**
  - parameters based on documentation for the method
  - docstrings
- exceptions

- **tries to determine the return type by parsing the docstring.**
  - Imperative verbs used in docstrings have a strong correlation to return -> None
  - recognizes documented Generators, Iterators, Callable
  - Coroutines are identified based tag "This is a Coroutine". Then if the return type was Foo, it will be transformed to : Coroutine[Foo]

  - a static Lookup list is used for a few methods/functions for which the return type cannot be determined from the docstring.
  - add NoReturn to a few functions that never return (stop / deepsleep / reset)
  - if no type can be detected the type *Any* is used

The generated stub files are formatted using *black* and checked for validity using *pyright* Note: black on python 3.7 does not like some function defs *def sizeof(struct, layout_type=NATIVE, /) -> int:*

- ordering of inter-dependent classes in the same module
- **Literals / constants**
  - documentation contains repeated vars with the same indentation
  - Module level:

```
.. data:: IPPROTO_UDP
          IPPROTO_TCP
```

  - class level:

```
.. data:: Pin.IRQ_FALLING
          Pin.IRQ_RISING
          Pin.IRQ_LOW_LEVEL
          Pin.IRQ_HIGH_LEVEL

          Selects the IRQ trigger type.
```

  - literals documented using a wildcard are added as comments only
- add GLUE imports to allow specific modules to import specific others.
- **repeats of definitions in the rst file for similar functions or literals**
  - CONSTANTS (module and Class level)
  - functions
  - methods
- **Child/ Parent classes** are added based on a (manual) lookup table CHILD_PARENT_CLASS

17.12.1 Not yet implemented

- **manual tweaks for uasync** https://docs.python.org/3/library/typing.html#asynchronous-programming

correct warnings for 'Unsupported escape sequence in string literal'

17.12.2 Module Contents

Classes

RSTReader

Functions

_color(c, *args) → str
_green(*args) → str
_red(*args) → str
generate_from_rst(rst_path: pathlib.Path, dst_path: pathlib.Path, v_tag: str, release: str = None, black=True, stubgen=True, pattern: str = '*.rst', verbose: bool = False) → int
cli_docstubs(source: str = './micropython', target: str = './all-stubs', verbose: bool = False, black: bool = True, stubgen: bool = True, basename='micropython')

Attributes

log
NEW_OUTPUT
SEPERATOR

stubs_from_docs.**log**

stubs_from_docs.**NEW_OUTPUT = True**

stubs_from_docs.**SEPERATOR = ::**

stubs_from_docs.**_color**(*c*, **args*) → str

Return type str

stubs_from_docs.**_green**(**args*) → str

Return type str

stubs_from_docs.**_red**(**args*) → str

Return type str

class stubs_from_docs.**RSTReader**(*v_tag='v1.xx'*)

verbose = False

gather_docs = False

target = .py

property line(*self*) → str
get the current line from input, also stores this as last_line to allow for inspection and dumping the json file

Return type str

property module_names(*self*) → List[str]
list of possible module names [uname , name] (longest first)

Return type List[str]

strip_prefixes(*self*, *name: str*, *strip_mod: bool = True*, *strip_class: bool = False*)
Remove the modulename. and or the classname. from the begining of a name

Parameters

- **name** (*str*) –
- **strip_mod** (*bool*) –
- **strip_class** (*bool*) –

leave_class(*self*)

read_file(*self*, *filename: pathlib.Path*)

Parameters **filename** (*pathlib.Path*) –

prepare_output(*self*)
clean up some trailing spaces and commas

write_file(*self*, *filename: pathlib.Path*) → bool

Parameters **filename** (*pathlib.Path*) –

Return type bool

parse_docstring(*self*) → List[str]
Read a textblock that will be used as a docstring, or used to process a toc tree The textblock is terminated at the following RST line structures/tags

– Heading == Heading ~~ Heading

The blank lines at the start and end are removed to limit the space the docstring takes up.

Return type List[str]

fix_parameters(*self*, *params: str*)
change documentation parameter notation to a supported format that works for linting

Parameters **params** (*str*) –

create_update_class(*self*, *name: str*, *params: str*, *docstr: List[str]*)

Parameters

- **name** (*str*) –

- **params** (*str*) –
- **docstr** (*List[str]*) –

parse_toc(*self*)
process table of content with additional rst files, and add / include them in the current module

parse_module(*self*)
parse a module tag and set the module's docstring

parse_current_module(*self*)

parse_function(*self*)

parse_class(*self*)

get_rst_hint(*self*)
parse the '.. <rst hint>:: ' from the current line

parse_method(*self*)

parse_exception(*self*)

parse_name(*self*, *line: str = None*)
get the constant/function/class name from a line with an identifier

Parameters **line** (*str*) –

parse_names(*self*, *oneliner: bool = True*)
get a list of constant/function/class names from and following a line with an identifier advances the linecounter

oneliner : treat a line with commas as multiple names (used for constants)

Parameters **oneliner** (*bool*) –

parse_data(*self*)

parse(*self*)

stubs_from_docs.**generate_from_rst**(*rst_path: pathlib.Path*, *dst_path: pathlib.Path*, *v_tag: str*, *release: str = None*, *black=True*, *stubgen=True*, *pattern: str = '*.rst'*, *verbose: bool = False*) → int

Parameters

- **rst_path** (*pathlib.Path*) –
- **dst_path** (*pathlib.Path*) –
- **v_tag** (*str*) –
- **release** (*str*) –
- **pattern** (*str*) –
- **verbose** (*bool*) –

Return type int

stubs_from_docs.**cli_docstubs**(*source: str = './micropython'*, *target: str = './all-stubs'*, *verbose: bool = False*, *black: bool = True*, *stubgen: bool = True*, *basename='micropython'*)
Read the Micropython library documentation files and use them to build stubs that can be used for static type-checking. Generates: - modules

- docstrings

- module constants
- **function definitions**
  - docstrings
  - function parameters based on documentation

classes

- docstrings
- __init__ method
- class constants
- parameters based on documentation for class
- **methods**
  - parameters based on documentation for the method
  - docstrings

- exceptions

Parameters

- **source** (*str*) –
- **target** (*str*) –
- **verbose** (*bool*) –
- **black** (*bool*) –
- **stubgen** (*bool*) –

17.13 downloader

Download files from a public github repo

17.13.1 Module Contents

Functions

download_file(url: str, module: str, folder: str = './')	dowload a file from a public github repo
download_files(repo, frozen_modules, savepath)	dowload multiple files from a public github repo

Attributes

log	

downloader.**log**

downloader.**download_file**(*url: str*, *module: str*, *folder: str = './'*)
dowload a file from a public github repo

Parameters

- **url** (*str*) –
- **module** (*str*) –
- **folder** (*str*) –

downloader.**download_files**(*repo*, *frozen_modules*, *savepath*)
dowload multiple files from a public github repo

17.14 update_pyi

Collect modules and python stubs from other projects and stores them in the all_stubs folder The all_stubs folder should be mapped/symlinked to the micropython_stubs/stubs repo/folder

17.14.1 Module Contents

update_pyi.**log**

update_pyi.**stub_path**

17.15 visitors

17.15.1 Submodules

visitors._apply_stubber_annotations

Module Contents

Classes

FunctionAnnotation	
TypeCollector	Collect type annotations from a stub module.
Annotations	
ApplyStubberAnnotationsVisitor	Apply type annotations to a source module using the given stub modules.

Functions

_get_import_alias_names(import_aliases: Sequence[libcst.ImportAlias]) → Set[str]	
_get_import_names(imports: Sequence[Union[libcst.Import, libcst.ImportFrom]]) → Set[str]	

visitors._apply_stubber_annotations.**_get_import_alias_names**(*import_aliases: Sequence[libcst.ImportAlias]*) → Set[str]

Parameters **import_aliases** (*Sequence[libcst.ImportAlias]*) –

Return type Set[str]

visitors._apply_stubber_annotations.**_get_import_names**(*imports: Sequence[Union[libcst.Import, libcst.ImportFrom]]*) → Set[str]

Parameters **imports** (*Sequence[Union[libcst.Import, libcst.ImportFrom]]*) –

Return type Set[str]

class visitors._apply_stubber_annotations.**FunctionAnnotation**

parameters :libcst.Parameters

returns :Optional[libcst.Annotation]

class visitors._apply_stubber_annotations.**TypeCollector**(*existing_imports: Set[str]*, *context: libcst.codemod._context.CodemodContext*)

Bases: libcst.CSTVisitor

Collect type annotations from a stub module.

Parameters

- **existing_imports** (*Set[str]*) –
- **context** (*libcst.codemod._context.CodemodContext*) –

visit_ClassDef(*self*, *node: libcst.ClassDef*) → None

Parameters **node** (*libcst.ClassDef*) –

Return type None

leave_ClassDef(*self*, *original_node: libcst.ClassDef*) → None

Parameters **original_node** (*libcst.ClassDef*) –

Return type None

visit_FunctionDef(*self*, *node: libcst.FunctionDef*) → bool

Parameters **node** (*libcst.FunctionDef*) –

Return type bool

leave_FunctionDef(*self*, *original_node: libcst.FunctionDef*) → None

Parameters **original_node** (*libcst.FunctionDef*) –

Return type None

visit_AnnAssign(*self*, *node: libcst.AnnAssign*) → bool

Parameters **node** (*libcst.AnnAssign*) –

Return type bool

leave_AnnAssign(*self*, *original_node: libcst.AnnAssign*) → None

Parameters **original_node** (*libcst.AnnAssign*) –

Return type None

visit_ImportFrom(*self*, *node: libcst.ImportFrom*) → None

Parameters **node** (*libcst.ImportFrom*) –

Return type None

_add_annotation_to_imports(*self*, *annotation: libcst.Attribute*) → Union[libcst.Name, libcst.Attribute]

Parameters **annotation** (*libcst.Attribute*) –

Return type Union[libcst.Name, libcst.Attribute]

_handle_Index(*self*, *slice: libcst.Index*, *node: libcst.Subscript*) → libcst.Subscript

Parameters

- **slice** (*libcst.Index*) –
- **node** (*libcst.Subscript*) –

Return type libcst.Subscript

_handle_Subscript(*self*, *node: libcst.Subscript*) → libcst.Subscript

Parameters **node** (*libcst.Subscript*) –

Return type libcst.Subscript

_create_import_from_annotation(*self*, *returns: libcst.Annotation*) → libcst.Annotation

Parameters **returns** (*libcst.Annotation*) –

Return type libcst.Annotation

_import_parameter_annotations(*self*, *parameters: libcst.Parameters*) → libcst.Parameters

Parameters **parameters** (*libcst.Parameters*) –

Return type libcst.Parameters

class visitors._apply_stubber_annotations.**Annotations**

function_annotations :Dict[str, FunctionAnnotation]

attribute_annotations :Dict[str, libcst.Annotation]

class_definitions :Dict[str, libcst.ClassDef]

class visitors._apply_stubber_annotations.**ApplyStubberAnnotationsVisitor**(*context: libcst.codemod._context.CodemodConte annotations: Optional[Annotations] = None, overwrite_existing_annotations: bool = False*)

Bases: libcst.codemod._visitor.ContextAwareTransformer

Apply type annotations to a source module using the given stub modules. You can also pass in explicit annotations for functions and attributes and pass in new class definitions that need to be added to the source module.

This is one of the transforms that is available automatically to you when running a codemod. To use it in this manner, import `ApplyStubberAnnotationsVisitor` and then call the static `store_stub_in_context()` method, giving it the current context (found as `self.context` for all subclasses of `Codemod`), the stub module from which you wish to add annotations.

For example, you can store the type annotation `int` for `x` using:

```
stub_module = parse_module("x: int = ...")

ApplyStubberAnnotationsVisitor.store_stub_in_context(self.context, stub_module)
```

You can apply the type annotation using:

```
source_module = parse_module("x = 1")
ApplyStubberAnnotationsVisitor.transform_module(source_module)
```

This will produce the following code:

```
x: int = 1
```

If the function or attribute already has a type annotation, it will not be overwritten.

To overwrite existing annotations when applying annotations from a stub, use the keyword argument `overwrite_existing_annotations=True` when constructing the codemod or when calling `store_stub_in_context`.

Parameters

- **context** (*libcst.codemod._context.CodemodContext*) –
- **annotations** (*Optional[Annotations]*) –
- **overwrite_existing_annotations** (*bool*) –

CONTEXT_KEY = ApplyStubberAnnotationsVisitor

static store_stub_in_context(*context: libcst.codemod._context.CodemodContext, stub: libcst.Module, overwrite_existing_annotations: bool = False*) → None

Store a stub module in the `CodemodContext` so that type annotations from the stub can be applied in a later invocation of this class.

If the `overwrite_existing_annotations` flag is `True`, the codemod will overwrite any existing annotations.

If you call this function multiple times, only the last values of `stub` and `overwrite_existing_annotations` will take effect.

Parameters

- **context** (*libcst.codemod._context.CodemodContext*) –
- **stub** (*libcst.Module*) –
- **overwrite_existing_annotations** (*bool*) –

Return type None

transform_module_impl(*self*, *tree: libcst.Module*) → libcst.Module

Collect type annotations from all stubs and apply them to `tree`.

Gather existing imports from `tree` so that we don't add duplicate imports.

Parameters **tree** (*libcst.Module*) –

Return type libcst.Module

_qualifier_name(*self*) → str

Return type str

_annotate_single_target(*self*, *node: libcst.Assign*, *updated_node: libcst.Assign*) → Union[libcst.Assign, libcst.AnnAssign]

Parameters

- **node** (*libcst.Assign*) –
- **updated_node** (*libcst.Assign*) –

Return type Union[libcst.Assign, libcst.AnnAssign]

_split_module(*self*, *module: libcst.Module*, *updated_module: libcst.Module*) → Tuple[List[Union[libcst.SimpleStatementLine, libcst.BaseCompoundStatement]], List[Union[libcst.SimpleStatementLine, libcst.BaseCompoundStatement]]]

Parameters

- **module** (*libcst.Module*) –
- **updated_module** (*libcst.Module*) –

Return type Tuple[List[Union[libcst.SimpleStatementLine, libcst.BaseCompoundStatement]], List[Union[libcst.SimpleStatementLine, libcst.BaseCompoundStatement]]]

_add_to_toplevel_annotations(*self*, *name: str*) → None

Parameters **name** (*str*) –

Return type None

_update_parameters(*self*, *annotations:* FunctionAnnotation, *updated_node: libcst.FunctionDef*) → libcst.Parameters

Parameters

- **annotations** (FunctionAnnotation) –
- **updated_node** (*libcst.FunctionDef*) –

Return type libcst.Parameters

_insert_empty_line(*self*, *statements: List[Union[libcst.SimpleStatementLine, libcst.BaseCompoundStatement]]*) → List[Union[libcst.SimpleStatementLine, libcst.BaseCompoundStatement]]

Parameters **statements** (*List[Union[libcst.SimpleStatementLine, libcst.BaseCompoundStatement]]*) –

Return type List[Union[libcst.SimpleStatementLine, libcst.BaseCompoundStatement]]

visit_ClassDef(*self*, *node: libcst.ClassDef*) → None

Parameters **node** (*libcst.ClassDef*) –

Return type None

leave_ClassDef(*self*, *original_node: libcst.ClassDef*, *updated_node: libcst.ClassDef*) → libcst.ClassDef

Parameters

- **original_node** (*libcst.ClassDef*) –
- **updated_node** (*libcst.ClassDef*) –

Return type libcst.ClassDef

visit_FunctionDef(*self*, *node: libcst.FunctionDef*) → bool

Parameters **node** (*libcst.FunctionDef*) –

Return type bool

leave_FunctionDef(*self*, *original_node: libcst.FunctionDef*, *updated_node: libcst.FunctionDef*) → libcst.FunctionDef

Parameters

- **original_node** (*libcst.FunctionDef*) –
- **updated_node** (*libcst.FunctionDef*) –

Return type libcst.FunctionDef

leave_Assign(*self*, *original_node: libcst.Assign*, *updated_node: libcst.Assign*) → Union[libcst.Assign, libcst.AnnAssign]

Parameters

- **original_node** (*libcst.Assign*) –
- **updated_node** (*libcst.Assign*) –

Return type Union[libcst.Assign, libcst.AnnAssign]

leave_ImportFrom(*self*, *original_node: libcst.ImportFrom*, *updated_node: libcst.ImportFrom*) → libcst.ImportFrom

Parameters

- **original_node** (*libcst.ImportFrom*) –
- **updated_node** (*libcst.ImportFrom*) –

Return type libcst.ImportFrom

leave_Module(*self*, *original_node: libcst.Module*, *updated_node: libcst.Module*) → libcst.Module

Parameters

- **original_node** (*libcst.Module*) –
- **updated_node** (*libcst.Module*) –

Return type libcst.Module

17.16 commands

17.16.1 Subpackages

commands.tests

17.16.2 Submodules

commands.constant_folding

Module Contents

Classes

ConvertConstantCommand

class commands.constant_folding.**ConvertConstantCommand**(*context: libcst.codemod.CodemodContext*, *string: str*, *constant: str*)

Bases: libcst.codemod.VisitorBasedCodemodCommand

Parameters

- **context** (*libcst.codemod.CodemodContext*) –
- **string** (*str*) –
- **constant** (*str*) –

DESCRIPTION :str = Converts raw strings to constant accesses.

static add_args(*arg_parser: argparse.ArgumentParser*) → None

Parameters **arg_parser** (*argparse.ArgumentParser*) –

Return type None

leave_SimpleString(*self*, *original_node: libcst.SimpleString*, *updated_node: libcst.SimpleString*) → Union[libcst.SimpleString, libcst.Name]

Parameters

- **original_node** (*libcst.SimpleString*) –
- **updated_node** (*libcst.SimpleString*) –

Return type Union[libcst.SimpleString, libcst.Name]

commands.noop

Module Contents

Classes

NOOPCommand

class commands.noop.**NOOPCommand**

Bases: libcst.codemod.CodemodCommand

DESCRIPTION :str = Does absolutely nothing.

transform_module_impl(*self*, *tree: libcst.Module*) → libcst.Module

Parameters **tree** (*libcst.Module*) –

Return type libcst.Module

commands.samples

Module Contents

commands.samples.**simple_stub = Multiline-String**

```
"""
Module: 'machine' on micropython-esp32-1.15
"""
# MCU: {'ver': '1.15', 'port': 'esp32', 'arch': 'xtensawin', 'sysname':
↪'esp32', 'release': '1.15.0', 'name': 'micropython', 'mpy': 10757,
↪'version': '1.15.0', 'machine': 'ESP32 module (spiram) with ESP32', 'build
↪': '', 'nodename': 'esp32', 'platform': 'esp32', 'family': 'micropython'}
# Stubber: 1.3.11
from typing import Any

class Signal:
    ''
    def __init__(self):
```

(continues on next page)

(continued from previous page)

```
        pass

    def off(self) -> Any:
        pass

    def on(self) -> Any:
        pass

    def value(self) -> Any:
        pass

def time_pulse_us() -> Any:
    pass

def unique_id() -> Any:
    pass

def wake_reason() -> Any:
    pass
```

commands.samples.**rich_source = Multiline-String**

```
def time_pulse_us(pin:Pin, pulse_level:int, timeout_us:int=1000000, /) ->␣
↪int:
    """
    Time a pulse on the given *pin*, and return the duration of the pulse in
    microseconds.  The *pulse_level* argument should be 0 to time a low␣
↪pulse
    or 1 to time a high pulse.

    If the current input value of the pin is different to *pulse_level*,
    the function first (*) waits until the pin input becomes equal to␣
↪*pulse_level*,
    then (**) times the duration that the pin is equal to *pulse_level*.
    If the pin is already equal to *pulse_level* then timing starts␣
↪straight away.

    The function will return -2 if there was timeout waiting for condition␣
↪marked
    (*) above, and -1 if there was timeout during the main measurement,␣
↪marked (**)
    above. The timeout is the same for both cases and given by *timeout_us*␣
↪(which
    is in microseconds).
    """
    ...


class Signal(Pin):
    """The Signal class is a simple extension of the Pin class. Unlike Pin,␣
↪which can be only in "absolute"
    0 and 1 states, a Signal can be in "asserted" (on) or "deasserted"␣
↪(off) states, while being inverted (active-low) or not.
```

(continues on next page)

(continued from previous page)

```
    In other words, it adds logical inversion support to Pin functionality. While this may seem a simple addition, it is exactly what
    is needed to support wide array of simple digital devices in a way portable across different boards, which is one of the major
    MicroPython goals. Regardless of whether different users have an active-high or active-low LED, a normally open or normally closed
    relay - you can develop a single, nicely looking application which works with each of them, and capture hardware configuration
    differences in few lines in the config file of your app.

    """
    def __init__(self, pin_obj:Pin, *,invert:bool=False):
        """ Create a Signal object. There're two ways to create it:
        By wrapping existing Pin object - universal method which works for any board.
        By passing required Pin parameters directly to Signal constructor, skipping the need to create intermediate Pin object. Available on many, but not all boards.
        The arguments are:
        pin_obj is existing Pin object.
        pin_arguments are the same arguments as can be passed to Pin constructor.
        invert - if True, the signal will be inverted (active low).
        """
        pass


    def off(self) -> None:
        """ Activate signal.
        """
        pass


    def on(self) -> None:
        """ Deactivate signal.
        """
        pass

    def value(self) -> None:
        """ This method allows to set and get the value of the signal, depending on whether the argument x is supplied or not.
            If the argument is omitted then this method gets the signal level, 1 meaning signal is asserted (active) and 0 - signal inactive.
            If the argument is supplied then this method sets the signal level. The argument x can be anything that converts to a boolean.
            If it converts to True, the signal is active, otherwise it is inactive.
            Correspondence between signal being active and actual logic level on the underlying pin depends on whether signal is inverted (active-low) or not.
            For non-inverted signal, active status corresponds to logical 1, inactive - to logical 0. For inverted/active-low signal, active status corresponds to logical 0,
```

(continues on next page)

(continued from previous page)

```
60              while inactive - to logical 1.
61          """
62          pass
```

17.17 rst

17.17.1 Submodules

rst.classsort

Sort list of classess in parent-child order note that this does not take multiple inheritance into account ref : https://stackoverflow.com/questions/34964878/python-generate-a-dictionarytree-from-a-list-of-tuples/35049729#35049729 with modification

Module Contents

Functions

sort_classes(classes: List[str])	sort a list of classes to respect the parent-child order

rst.classsort.**sort_classes**(*classes: List[str]*)
sort a list of classes to respect the parent-child order

Parameters **classes** (*List[str]*) –

rst.lookup

this is an list with manual overrides for function returns that could not efficiently be determined from their docstring description Format: a dictionary with : - key = module.[class.]function name - value : two-tuple with (return type , priority)

Module Contents

rst.lookup.**LOOKUP_LIST**

rst.lookup.**NONE_VERBS = ['Activate ', 'Build a ', 'Cancel ', 'Clear ', 'Close ', 'cancel ', 'Configure ', 'Connect ',...**

rst.lookup.**MODULE_GLUE**

rst.lookup.**PARAM_FIXES = [['*', '*'], ['**', '*'], ['/*', '*'], ['**', '*'], ["'param'", 'param'], ['(adcx, adcy,...**

rst.lookup.**CHILD_PARENT_CLASS**

rst.output_dict

ModuleSourceDict represents a source file with the following components

- docstr
- version
- comment
- typing
- Optional: list of constants
- optional: ClassSourcedicts
- optional: FunctionSourcedicts
- optional: individual lines of code

ClassSourceDict represents a source file with the following components

- comment
- class
- docstr
- Optional: list of constants
- __init__ : class signature
- optional: FunctionSourcedicts
- optional: individual lines of code

FunctionSourceDict represents a source file with the following components

- # comments - todo
- optional: decorator
- def - function definition
- docstr
- constants
- body - ...
- optional: individual lines of code

SourceDict is the 'base class' it

Module Contents

Classes

SourceDict	(abstract) dict to store source components respecting parent child dependencies and proper definition order
ModuleSourceDict	(abstract) dict to store source components respecting parent child dependencies and proper definition order

continues on next page

Table 34 – continued from previous page

ClassSourceDict	(abstract) dict to store source components respecting parent child dependencies and proper definition order
FunctionSourceDict	(abstract) dict to store source components respecting parent child dependencies and proper definition order

class rst.output_dict.**SourceDict**(*base: List*, *indent: int = 0*, *body: int = 0*, *lf: str = '\n'*)

Bases: OrderedDict

(abstract) dict to store source components respecting parent child dependencies and proper definition order

Parameters

- **base** (*List*) –
- **indent** (*int*) –
- **body** (*int*) –
- **lf** (*str*) –

__str__(*self*) → str

convert the OD into a string

Return type str

__add__(*self*, *dict: SourceDict*)

Parameters **dict** (SourceDict) –

add_docstr(*self*, *docstr: Union[str, List[str]]*, *extra: int = 0*)

Parameters

- **docstr** (*Union[str, List[str]]*) –
- **extra** (*int*) –

add_comment(*self*, *line: Union[str, List[str]]*)

Add a comment, or list of comments, to this block.

Parameters **line** (*Union[str, List[str]]*) –

add_constant(*self*, *line: str*, *autoindent: bool = True*)

add constant to the constant scope of this block

Parameters

- **line** (*str*) –
- **autoindent** (*bool*) –

add_constant_smart(*self*, *name: str*, *type: str = 'Any'*, *docstr: List[str] = []*, *autoindent: bool = True*)

add constant to the constant scope of this block, or a class in this block

Parameters

- **name** (*str*) –
- **type** (*str*) –
- **docstr** (*List[str]*) –
- **autoindent** (*bool*) –

abstract find(*self*, *name: str*) → Union[str, None]

Parameters **name** (*str*) –

Return type Union[str, None]

add_line(*self*, *line: str*, *autoindent: bool = True*)

Parameters

- **line** (*str*) –
- **autoindent** (*bool*) –

index(*self*, *key: str*)

Parameters **key** (*str*) –

class rst.output_dict.ModuleSourceDict(*name: str*, *indent=0*, *lf: str = '\n'*)

Bases: *SourceDict*

(abstract) dict to store source components respecting parent child dependencies and proper definition order

Parameters

- **name** (*str*) –
- **lf** (*str*) –

sort(*self*)

make sure all classdefs are in order

__str__(*self*)

convert the OD into a string

find(*self*, *name: str*) → Union[str, None]

find a classnode based on the name with or without the superclass

Parameters **name** (*str*) –

Return type Union[str, None]

classes(*self*)

get a list of the class names in parent-child order

add_import(*self*, *imports: Union[str, List[str]]*)

add a [list of] imports this module

Parameters **imports** (*Union[str, List[str]]*) –

class rst.output_dict.ClassSourceDict(*name: str*, ***, *docstr: List[str] = ['""" """']*, *init: str = ''*, *indent: int = 0*, *lf='\n'*)

Bases: *SourceDict*

(abstract) dict to store source components respecting parent child dependencies and proper definition order

Parameters

- **name** (*str*) –
- **docstr** (*List[str]*) –
- **init** (*str*) –
- **indent** (*int*) –

class rst.output_dict.**FunctionSourceDict**(*name: str*, *, *definition: List[str] = []*, *docstr: List[str] = ['"""\n"""']*, *indent: int = 0*, *decorators: List[str] = []*, *lf='\n'*)

Bases: *SourceDict*

(abstract) dict to store source components respecting parent child dependencies and proper definition order

Parameters

- **name** (*str*) –
- **definition** (*List[str]*) –
- **docstr** (*List[str]*) –
- **indent** (*int*) –
- **decorators** (*List[str]*) –

rst.report_return

Work in Progress

build test and % report Will need to be updated after new_output has been implemented.

Module Contents

Functions

process(folder: pathlib.Path, pattern: str)

rst.report_return.**process**(*folder: pathlib.Path*, *pattern: str*)

Parameters

- **folder** (*pathlib.Path*) –
- **pattern** (*str*) –

rst.rst_utils

Tries to determine the return type by parsing the docstring and the function signature

- if the signature contains a return type –> <something> then that is returned
- **check a lookup dictionary of type overrides,** if the functionnae is listed, then use the override
- **use re to find phrases such as:**
  - 'Returns ….. '
  - 'Gets ….. '
- docstring is joined without newlines to simplify parsing
- then parses the docstring to find references to known types and give then a rating though a hand coded model ()
- builds a list return type candidates

- selects the highest ranking candidate
- the default Type is 'Any'

to do:

- **regex :**
  - 'With no arguments the frequency in Hz is returned.'
  - 'Get or set' –> indicates overloaded/optional return Union[None|...]
  - add regex for 'Query' ` Otherwise, query current state if no argument is provided. `
- **regex :**
  - 'With no arguments the frequency in Hz is returned.'
  - 'Get or set' –> indicates overloaded/optional return Union[None|...]
  - add regex for 'Query' ` Otherwise, query current state if no argument is provided. `
- **try if an Azure Machine Learning works as well** https://docs.microsoft.com/en-us/azure/machine-learning/quickstart-create-resources
-

Module Contents

Functions

simple_candidates(type: str, match_string: str, keywords: List[str], rate: float = 0.5, exclude: List[str] = [])	find and rate possible types and confidence weighting for simple types.
compound_candidates(type: str, match_string: str, keywords: List[str], rate: float = 0.85, exclude: List[str] = [])	find and rate possible types and confidence weighting for compound types that can have a subscription.
object_candidates(match_string: str, rate: float = 0.81, exclude: List[str] = ['IRQ'])	find and rate possible types and confidence weighting for Object types.
distill_return(return_text: str) → List[Dict]	Find return type and confidence.
return_type_from_context(docstring: Union[str, List[str]], signature: str, module: str, literal: bool = False)	
_type_from_context(docstring: Union[str, List[str]], signature: str, module: str, literal: bool = False)	Determine the return type of a function or method based on:

Attributes

TYPING_IMPORT

rst.rst_utils.**TYPING_IMPORT :List[str] = ['from typing import IO, Any, Callable, Coroutine, Dict, Generator, Iterator, List, NoReturn,...**

rst.rst_utils.**simple_candidates**(*type: str*, *match_string: str*, *keywords: List[str]*, *rate: float = 0.5*, *exclude: List[str] = []*)

find and rate possible types and confidence weighting for simple types. Case sensitive

Parameters

- **type** (*str*) –
- **match_string** (*str*) –
- **keywords** (*List[str]*) –
- **rate** (*float*) –
- **exclude** (*List[str]*) –

rst.rst_utils.**compound_candidates**(*type: str*, *match_string: str*, *keywords: List[str]*, *rate: float = 0.85*, *exclude: List[str] = []*)

find and rate possible types and confidence weighting for compound types that can have a subscription. Case sensitive

Parameters

- **type** (*str*) –
- **match_string** (*str*) –
- **keywords** (*List[str]*) –
- **rate** (*float*) –
- **exclude** (*List[str]*) –

rst.rst_utils.**object_candidates**(*match_string: str*, *rate: float = 0.81*, *exclude: List[str] = ['IRQ']*)

find and rate possible types and confidence weighting for Object types. Case sensitive

Parameters

- **match_string** (*str*) –
- **rate** (*float*) –
- **exclude** (*List[str]*) –

rst.rst_utils.**distill_return**(*return_text: str*) → List[Dict]

Find return type and confidence. Returns a list of possible types and confidence weighting. {

type :str # the return type confidence: float # the confidence between 0.0 and 1 match: Optional[str] # for debugging : the reason the match was made

}

Parameters **return_text** (*str*) –

Return type List[Dict]

rst.rst_utils.**return_type_from_context**(*docstring: Union[str, List[str]]*, *signature: str*, *module: str*, *literal: bool = False*)

Parameters

- **docstring** (*Union[str, List[str]]*) –
- **signature** (*str*) –
- **module** (*str*) –

- **literal** (*bool*) –

rst.rst_utils._type_from_context(*docstring: Union[str, List[str]]*, *signature: str*, *module: str*, *literal: bool = False*)

Determine the return type of a function or method based on:

- the function signature
- the terminology used in the docstring

Logic: - if the signature contains a return type –> <something> then that is returned - use re to find phrases such as:

- 'Returns ….. '
- 'Gets ….. '
- docstring is joined without newlines to simplify parsing
- then parses the docstring to find references to known types and give then a rating though a hand coded model ()
- builds a list return type candidates
- selects the highest ranking candidate
- the default Type is 'Any'

Parameters

- **docstring** (*Union[str, List[str]]*) –
- **signature** (*str*) –
- **module** (*str*) –
- **literal** (*bool*) –

17.17.2 Package Contents

Classes

SourceDict	(abstract) dict to store source components respecting parent child dependencies and proper definition order
ModuleSourceDict	(abstract) dict to store source components respecting parent child dependencies and proper definition order
ClassSourceDict	(abstract) dict to store source components respecting parent child dependencies and proper definition order
FunctionSourceDict	(abstract) dict to store source components respecting parent child dependencies and proper definition order

Functions

sort_classes(classes: List[str])	sort a list of classes to respect the parent-child order
simple_candidates(type: str, match_string: str, keywords: List[str], rate: float = 0.5, exclude: List[str] = [])	find and rate possible types and confidence weighting for simple types.
compound_candidates(type: str, match_string: str, keywords: List[str], rate: float = 0.85, exclude: List[str] = [])	find and rate possible types and confidence weighting for compound types that can have a subscription.
object_candidates(match_string: str, rate: float = 0.81, exclude: List[str] = ['IRQ'])	find and rate possible types and confidence weighting for Object types.
distill_return(return_text: str) → List[Dict]	Find return type and confidence.
return_type_from_context(docstring: Union[str, List[str]], signature: str, module: str, literal: bool = False)	
_type_from_context(docstring: Union[str, List[str]], signature: str, module: str, literal: bool = False)	Determine the return type of a function or method based on:

Attributes

LOOKUP_LIST
NONE_VERBS
CHILD_PARENT_CLASS
PARAM_FIXES
MODULE_GLUE
TYPING_IMPORT
__all__

rst.**sort_classes**(*classes: List[str]*)
sort a list of classes to respect the parent-child order

Parameters **classes** (*List[str]*) –

rst.**LOOKUP_LIST**

rst.**NONE_VERBS = ['Activate ', 'Build a ', 'Cancel ', 'Clear ', 'Close ', 'cancel ', 'Configure ', 'Connect ',...**

rst.**CHILD_PARENT_CLASS**

rst.**PARAM_FIXES = [['*', '*'], ['**', '*'], ['/*', '*'], ['**', '*'], ["'param'", 'param'], ['(adcx, adcy,...**

rst.**MODULE_GLUE**

class rst.**SourceDict**(*base: List*, *indent: int = 0*, *body: int = 0*, *lf: str = '\n'*)
Bases: OrderedDict

(abstract) dict to store source components respecting parent child dependencies and proper definition order

Parameters

- **base** (*List*) –
- **indent** (*int*) –
- **body** (*int*) –
- **lf** (*str*) –

__str__(*self*) → str
convert the OD into a string

Return type str

__add__(*self*, *dict:* SourceDict)

Parameters **dict** (SourceDict) –

add_docstr(*self*, *docstr: Union[str, List[str]]*, *extra: int = 0*)

Parameters

- **docstr** (*Union[str, List[str]]*) –
- **extra** (*int*) –

add_comment(*self*, *line: Union[str, List[str]]*)
Add a comment, or list of comments, to this block.

Parameters **line** (*Union[str, List[str]]*) –

add_constant(*self*, *line: str*, *autoindent: bool = True*)
add constant to the constant scope of this block

Parameters

- **line** (*str*) –
- **autoindent** (*bool*) –

add_constant_smart(*self*, *name: str*, *type: str = 'Any'*, *docstr: List[str] = []*, *autoindent: bool = True*)
add constant to the constant scope of this block, or a class in this block

Parameters

- **name** (*str*) –
- **type** (*str*) –
- **docstr** (*List[str]*) –
- **autoindent** (*bool*) –

abstract find(*self*, *name: str*) → Union[str, None]

Parameters **name** (*str*) –

Return type Union[str, None]

add_line(*self*, *line: str*, *autoindent: bool = True*)

Parameters

- **line** (*str*) –
- **autoindent** (*bool*) –

index(*self*, *key: str*)

Parameters **key** (*str*) –

class rst.**ModuleSourceDict**(*name: str*, *indent=0*, *lf: str = '\n'*)

Bases: *SourceDict*

(abstract) dict to store source components respecting parent child dependencies and proper definition order

Parameters

- **name** (*str*) –
- **lf** (*str*) –

sort(*self*)

make sure all classdefs are in order

__str__(*self*)

convert the OD into a string

find(*self*, *name: str*) → Union[str, None]

find a classnode based on the name with or without the superclass

Parameters **name** (*str*) –

Return type Union[str, None]

classes(*self*)

get a list of the class names in parent-child order

add_import(*self*, *imports: Union[str, List[str]]*)

add a [list of] imports this module

Parameters **imports** (*Union[str, List[str]]*) –

class rst.**ClassSourceDict**(*name: str*, ***, *docstr: List[str] = ['""" """']*, *init: str = ''*, *indent: int = 0*, *lf='\n'*)

Bases: *SourceDict*

(abstract) dict to store source components respecting parent child dependencies and proper definition order

Parameters

- **name** (*str*) –
- **docstr** (*List[str]*) –
- **init** (*str*) –
- **indent** (*int*) –

class rst.**FunctionSourceDict**(*name: str*, ***, *definition: List[str] = []*, *docstr: List[str] = ['""" """']*, *indent: int = 0*, *decorators: List[str] = []*, *lf='\n'*)

Bases: *SourceDict*

(abstract) dict to store source components respecting parent child dependencies and proper definition order

Parameters

- **name** (*str*) –
- **definition** (*List[str]*) –

- **docstr** (*List[str]*) –
- **indent** (*int*) –
- **decorators** (*List[str]*) –

rst.**simple_candidates**(*type: str*, *match_string: str*, *keywords: List[str]*, *rate: float = 0.5*, *exclude: List[str] = []*)

find and rate possible types and confidence weighting for simple types. Case sensitive

Parameters

- **type** (*str*) –
- **match_string** (*str*) –
- **keywords** (*List[str]*) –
- **rate** (*float*) –
- **exclude** (*List[str]*) –

rst.**compound_candidates**(*type: str*, *match_string: str*, *keywords: List[str]*, *rate: float = 0.85*, *exclude: List[str] = []*)

find and rate possible types and confidence weighting for compound types that can have a subscription. Case sensitive

Parameters

- **type** (*str*) –
- **match_string** (*str*) –
- **keywords** (*List[str]*) –
- **rate** (*float*) –
- **exclude** (*List[str]*) –

rst.**object_candidates**(*match_string: str*, *rate: float = 0.81*, *exclude: List[str] = ['IRQ']*)

find and rate possible types and confidence weighting for Object types. Case sensitive

Parameters

- **match_string** (*str*) –
- **rate** (*float*) –
- **exclude** (*List[str]*) –

rst.**distill_return**(*return_text: str*) → List[Dict]

Find return type and confidence. Returns a list of possible types and confidence weighting. {

type :str # the return type confidence: float # the confidence between 0.0 and 1 match: Optional[str] # for debugging : the reason the match was made

}

Parameters **return_text** (*str*) –

Return type List[Dict]

rst.**return_type_from_context**(*docstring: Union[str, List[str]]*, *signature: str*, *module: str*, *literal: bool = False*)

Parameters

- **docstring** (*Union[str, List[str]]*) –
- **signature** (*str*) –
- **module** (*str*) –
- **literal** (*bool*) –

rst.**_type_from_context**(*docstring: Union[str, List[str]]*, *signature: str*, *module: str*, *literal: bool = False*)

Determine the return type of a function or method based on:

- the function signature
- the terminology used in the docstring

Logic: - if the signature contains a return type –> <something> then that is returned - use re to find phrases such as:

- 'Returns ….. '
- 'Gets ….. '
- docstring is joined without newlines to simplify parsing
- then parses the docstring to find references to known types and give then a rating though a hand coded model ()
- builds a list return type candidates
- selects the highest ranking candidate
- the default Type is 'Any'

Parameters

- **docstring** (*Union[str, List[str]]*) –
- **signature** (*str*) –
- **module** (*str*) –
- **literal** (*bool*) –

rst.**TYPING_IMPORT :List[str] = ['from typing import IO, Any, Callable, Coroutine, Dict, Generator, Iterator, List, NoReturn,...**

rst.**__all__**

17.18 add_class_init

17.18.1 Module Contents

Functions

add_init_methods(filename) → int	Add (missing) __init__ methods to a class using a regex

Attributes

empty_classdef
re_classdef
repl_classdef
x

add_class_init.**empty_classdef = (?P<indent1>
?)class\s*(?P<class>\s*.+\s*):(?P<LF>\r?\n)(?P<indent2> +)''\r?\n**

add_class_init.**re_classdef**

add_class_init.**repl_classdef = \g<indent1>class \g<class>:\g<LF>\g<indent2>def
__init__(self):\g<LF>\g<indent2> ...**

add_class_init.**add_init_methods**(*filename*) → int

Add (missing) __init__ methods to a class using a regex this assumes the (incorrect) classdef format that has been used by stubbers prior to version 1.4.0 and updates that to add the init.

Return type int

add_class_init.**x**

17.19 basics

17.19.1 Module Contents

Classes

TypingCollector
TypingTransformer

Attributes

source_tree
info_tree
visitor
transformer

continues on next page

Table 44 – continued from previous page

modified_tree

class basics.**TypingCollector**

Bases: libcst.CSTVisitor

visit_ClassDef(*self*, *node: libcst.ClassDef*) → Optional[bool]

Parameters **node** (*libcst.ClassDef*) –

Return type Optional[bool]

leave_ClassDef(*self*, *node: libcst.ClassDef*) → None

Parameters **node** (*libcst.ClassDef*) –

Return type None

visit_FunctionDef(*self*, *node: libcst.FunctionDef*) → Optional[bool]

Parameters **node** (*libcst.FunctionDef*) –

Return type Optional[bool]

leave_FunctionDef(*self*, *node: libcst.FunctionDef*) → None

Parameters **node** (*libcst.FunctionDef*) –

Return type None

class basics.**TypingTransformer**(*annotations*)

Bases: libcst.CSTTransformer

visit_ClassDef(*self*, *node: libcst.ClassDef*) → Optional[bool]

Parameters **node** (*libcst.ClassDef*) –

Return type Optional[bool]

leave_ClassDef(*self*, *original_node: libcst.ClassDef*, *updated_node: libcst.ClassDef*) → libcst.CSTNode

Parameters

- **original_node** (*libcst.ClassDef*) –
- **updated_node** (*libcst.ClassDef*) –

Return type libcst.CSTNode

visit_FunctionDef(*self*, *node: libcst.FunctionDef*) → Optional[bool]

Parameters **node** (*libcst.FunctionDef*) –

Return type Optional[bool]

leave_FunctionDef(*self*, *original_node: libcst.FunctionDef*, *updated_node: libcst.FunctionDef*) → libcst.CSTNode

Parameters

- **original_node** (*libcst.FunctionDef*) –
- **updated_node** (*libcst.FunctionDef*) –

Return type libcst.CSTNode

basics.**source_tree**

basics.**info_tree**

basics.**visitor**

basics.**transformer**

basics.**modified_tree**

17.20 ata_script

17.20.1 Module Contents

Functions

prRed(skk)
prGreen(skk)

Attributes

context
visitor
stubs_dir
sources_dir
stubs_dict
sources_dict
stubs_pathlist
rel_path
sources_pathlist
rel_path

continues on next page

Table 46 – continued from previous page

stub

ata_script.**prRed**(*skk*)

ata_script.**prGreen**(*skk*)

ata_script.**context**

ata_script.**visitor**

ata_script.**stubs_dir = all-stubs/micropython-v1_16-documentation/**

ata_script.**sources_dir = dev-stubs/micropython-v1_16-esp32/**

ata_script.**stubs_dict**

ata_script.**sources_dict**

ata_script.**stubs_pathlist**

ata_script.**rel_path**

ata_script.**sources_pathlist**

ata_script.**rel_path**

ata_script.**stub**

17.21 testfile

17.21.1 Module Contents

testfile.**code = Multiline-String**

```
print("hello {}".format("Jos"))
```

testfile.**code**

testfile.**output**

testfile.**work_with_bytes = False**

testfile.**output**

17.22 samples

17.22.1 Module Contents

samples.**simple_stub = Multiline-String**

```
"""
Module: 'machine' on micropython-esp32-1.15
"""
# MCU: {'ver': '1.15', 'port': 'esp32', 'arch': 'xtensawin', 'sysname':
↪'esp32', 'release': '1.15.0', 'name': 'micropython', 'mpy': 10757,
↪'version': '1.15.0', 'machine': 'ESP32 module (spiram) with ESP32', 'build
↪': '', 'nodename': 'esp32', 'platform': 'esp32', 'family': 'micropython'}
```

(continues on next page)

(continued from previous page)

```
# Stubber: 1.3.11
from typing import Any

class Signal:
    ''
    def __init__(self):
        pass

    def off(self) -> Any:
        pass

    def on(self) -> Any:
        pass

    def value(self) -> Any:
        pass

def time_pulse_us() -> Any:
    pass

def unique_id() -> Any:
    pass

def wake_reason() -> Any:
    pass
```

samples.**rich_source = Multiline-String**

```
def time_pulse_us(pin:Pin, pulse_level:int, timeout_us:int=1000000, /) ->␣
↪int:
    """
    Time a pulse on the given *pin*, and return the duration of the pulse in
    microseconds.  The *pulse_level* argument should be 0 to time a low␣
↪pulse
    or 1 to time a high pulse.

    If the current input value of the pin is different to *pulse_level*,
    the function first (*) waits until the pin input becomes equal to␣
↪*pulse_level*,
    then (**) times the duration that the pin is equal to *pulse_level*.
    If the pin is already equal to *pulse_level* then timing starts␣
↪straight away.

    The function will return -2 if there was timeout waiting for condition␣
↪marked
    (*) above, and -1 if there was timeout during the main measurement,␣
↪marked (**)
    above. The timeout is the same for both cases and given by *timeout_us*␣
↪(which
    is in microseconds).
    """
    ...
```

(continues on next page)

(continued from previous page)

```


class Signal(Pin):
    """The Signal class is a simple extension of the Pin class. Unlike Pin, which can be only in "absolute"
    0 and 1 states, a Signal can be in "asserted" (on) or "deasserted" (off) states, while being inverted (active-low) or not.
    In other words, it adds logical inversion support to Pin functionality. While this may seem a simple addition, it is exactly what
    is needed to support wide array of simple digital devices in a way portable across different boards, which is one of the major
    MicroPython goals. Regardless of whether different users have an active-high or active-low LED, a normally open or normally closed
    relay - you can develop a single, nicely looking application which works with each of them, and capture hardware configuration
    differences in few lines in the config file of your app.

    """
    def __init__(self, pin_obj:Pin, *,invert:bool=False):
        """ Create a Signal object. There're two ways to create it:
        By wrapping existing Pin object - universal method which works for any board.
        By passing required Pin parameters directly to Signal constructor, skipping the need to create intermediate Pin object. Available on many, but not all boards.
        The arguments are:
        pin_obj is existing Pin object.
        pin_arguments are the same arguments as can be passed to Pin constructor.
        invert - if True, the signal will be inverted (active low).
        """
        pass


    def off(self) -> None:
        """ Activate signal.
        """
        pass


    def on(self) -> None:
        """ Deactivate signal.
        """
        pass

    def value(self) -> None:
        """ This method allows to set and get the value of the signal, depending on whether the argument x is supplied or not.
            If the argument is omitted then this method gets the signal level, 1 meaning signal is asserted (active) and 0 - signal inactive.
            If the argument is supplied then this method sets the signal level. The argument x can be anything that converts to a boolean.
```

(continues on next page)

(continued from previous page)

```
            If it converts to True, the signal is active, otherwise it is inactive.
            Correspondence between signal being active and actual logic level on the underlying pin depends on whether signal is inverted (active-low) or not.
            For non-inverted signal, active status corresponds to logical 1, inactive - to logical 0. For inverted/active-low signal, active status corresponds to logical 0,
            while inactive - to logical 1.
        """
        pass
```

17.23 simple

17.23.1 Module Contents

simple.**py_source = Multiline-String**

```
'moduledoc'
class Signal:
    ''
    def value(self):
        ''
        pass

class Foo:
    """
    """
    def __init__(self):
        ''
        pass
```

simple.**config**

simple.**source_tree**

simple.**classdef**

simple.**body**

simple.**expr**

simple.**expr**

simple.**expr**

simple.**new_docstr**

simple.**newtree**

17.24 docstrings

CHAPTER

EIGHTEEN

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

INDEX

Symbols

A

B

C

D

E

F

G

I

L

M

N

O

P

R

S

T

U

V

W

X