Mu Documentation

Release 1.2.0

Nicholas H.Tollervey

Nov 15, 2022

CONTENTS

Note: This documentation is NOT for users of Mu. Rather, it is for software developers who want to improve Mu. Read our *Developer Setup* documentation for the technical details needed to get started.

For tutorials, how-to guides and user related discussion, please see the project's website for users of Mu at: https://codewith.mu/

If you're interested in the fun, educational, inspiring and sometimes hilarious ways in which people use Mu, check out: https://madewith.mu/

CHAPTER

ONE

QUICKSTART

Mu works with Python 3.5 to 3.8 (both inclusive). You need to have one of these Python versions installed in order to work on developing Mu (here is a comprehensive guide for how to do this). We also assume you know how to use Python virtual environments.

Clone the repository:

```
git clone https://github.com/mu-editor/mu.git
```

Create a virtualenv in Python 3.8:

```
python3.8 -m venv .venv
```

Activate the environment, on macOS and Linux the command is:

```
source .venv/bin/activate
```

For Windows, the command is:

```
. .\.venv\Scripts\activate.ps1
```

Once your virtual environment is activated, upgrade `pip`:

```
python -m pip install --upgrade pip
```

Install Mu and its development dependencies:

```
pip install -e ".[dev]"
```

Start Mu:

```
mu-editor
```

Run the test suite:

```
make check
```

Read on to learn more about Mu, its aims and how you can contribute.

CHAPTER

TWO

WHAT?

Mu is a very simple Python editor for kids, teachers and beginner programmers. It's written in Python and works on Windows, OSX, Linux and Raspberry Pi.

> "[Papert] realized, 'Oh, we could take the real content out here as a version in the child's world that is still the real thing.' It's not a fake version of math. It's kind of like little league, or even T-ball. In sports they do this all the time. In music, they do it all the time. The idea is, you never let the child do something that isn't the real thing – but you have to work your ass off to figure out what the real thing is in the context of the way their minds are working at that developmental level." – Alan Kay

Mu aspires to be "the real thing" as a development environment for beginner programmers taking their first steps with Python.

As a rule of thumb, if you're able to ask "why doesn't Mu have [feature X]?" then you're probably too advanced for using Mu as a development environment. In which case, you should graduate to a more advanced editor.

CHAPTER

THREE

WHY?

There isn't a cross platform Python code editor that is:

- Easy to use;
- Available on all major platforms;
- Well documented (even for beginners);
- Simply coded;
- Currently maintained; and,
- Thoroughly tested.

Mu addresses these needs.

Mu was originally created as a contribution from the Python Software Foundation for the BBC's micro:bit project. Many people asked if Mu could be turned into a generic beginner's code editor and, thanks to the wonderful support of the Raspberry Pi Foundation the work needed to make such changes was done over the summer of 2017.

The following video of a talk given at PyCon 2018 outlines the story of Mu:

CHAPTER

FOUR

HOW?

Mu's outlook is:

- Less is more (remove all unnecessary distractions);
- Keep it simple (so Mu is easy to understand);
- Walk the path of least resistance (Mu should be easy to use);
- Have fun (learning should be a positive experience).

Mu's own code is simple, clearly organised and well tested. It's copiously commented and mostly found in a few obviously named Python files.

This has been done on purpose: we want teachers and kids to take ownership of this project and organising the code in this way aids the first steps required to get involved.

If you're looking for ways to get involved check out some of the *Suggested First Steps* for new contributors.

Furthermore, we put our users at the centre of our development work. Extensive interviews with teachers, observations of lessons and exceptionally clear and helpful feedback from the education team at the Raspberry Pi Foundation (perhaps the most successful computing in education project in history) have informed the design choices for Mu.

CHAPTER

FIVE

WHO?

You!

Contributions are welcome without prejudice from *anyone* irrespective of age, gender, religion, race or sexuality. If you're thinking, "but they don't mean me", *then we especially mean YOU*. Good quality code and engagement with respect, humour and intelligence wins every time.

Read about *Contributing to Mu* and perhaps try out some *Suggested First Steps*.

We want the Mu community to be a friendly place. Therefore, we expect contributors to follow our *Code of Conduct*.

CHAPTER

SIX

CONTENTS:

6.1 Contributing to Mu

Hey! Many thanks for wanting to improve Mu.

Contributions are welcome without prejudice from *anyone* irrespective of age, gender, religion, race or sexuality. If you're thinking, "but they don't mean me", *then we especially mean YOU*. Good quality code and engagement with respect, humour and intelligence wins every time.

- If you're from a background which isn't well-represented in most geeky groups, get involved - *we want to help you make a difference*.
- If you're from a background which *is* well-represented in most geeky groups, get involved - *we want your help making a difference*.
- If you're worried about not being technical enough, get involved - *your fresh perspective will be invaluable*.
- If you think you're an imposter, get involved.
- If your day job isn't code, get involved.
- This isn't a group of experts, just people. Get involved!
- We are interested in educational, social and technical problems. If you are too, get involved.
- This is a new community. *No-one knows what they are doing*, so, get involved.

We expect contributors to follow our code_of_conduct.

Check out our developer setup documentation for instructions to configure a working development environment for Mu.

Feedback may be given for contributions and, where necessary, changes will be politely requested and discussed with the originating author. Respectful yet robust argument is most welcome.

Warning: Contributions are subject to the following caveats: the contribution was created by the contributor who, by submitting the contribution, is confirming that they have the authority to submit the contribution and place it under the license as defined in the LICENSE file found within this repository (see *GNU General Public License*). If this is a significant contribution the contributor should add themselves to the AUTHORS file found in the root of Mu's repository, otherwise they agree, for the sake of convenience, that copyright passes exclusively to Nicholas H.Tollervey on behalf of the Mu project.

6.1.1 Checklist

- If your contribution includes non-obvious technical decision making please make sure you document this in the design decisions section.
- Your code should be commented in *plain English* (British spelling).
- If your contribution is for a major block of work and you've not done so already, add yourself to the AUTHORS file following the convention found therein.
- We have 100% test coverage - include tests to maintain this!
- **Before submitting code ensure coding standards and test coverage by running**:

```
make check
```

- If in doubt, ask a question. The only stupid question is the one that's never asked.
- Most importantly, **Have fun!** :-)

6.2 Code of Conduct

We expect contributors to follow the Python Software Foundation's Code of Conduct, reproduced below.

The Python community is made up of members from around the globe with a diverse set of skills, personalities, and experiences. It is through these differences that our community experiences great successes and continued growth. When you're working with members of the community, we encourage you to follow these guidelines which help steer our interactions and strive to keep Python a positive, successful, and growing community.

A member of the Python community is:

6.2.1 Open

Members of the community are open to collaboration, whether it's on PEPs, patches, problems, or otherwise. We're receptive to constructive comment and criticism, as the experiences and skill sets of other members contribute to the whole of our efforts. We're accepting of all who wish to take part in our activities, fostering an environment where anyone can participate and everyone can make a difference.

6.2.2 Considerate

Members of the community are considerate of their peers – other Python users. We're thoughtful when addressing the efforts of others, keeping in mind that often times the labor was completed simply for the good of the community. We're attentive in our communications, whether in person or online, and we're tactful when approaching differing views.

6.2.3 Respectful

Members of the community are respectful. We're respectful of others, their positions, their skills, their commitments, and their efforts. We're respectful of the volunteer efforts that permeate the Python community. We're respectful of the processes set forth in the community, and we work within them. When we disagree, we are courteous in raising our issues.

Overall, we're good to each other. We contribute to this community not because we have to, but because we want to. If we remember that, these guidelines will come naturally.

6.3 Developer Setup

The source code is hosted on GitHub. Fork the repository with the following command:

```
git clone https://github.com/mu-editor/mu.git
```

Mu does not and never will use or support Python 2. You should use Python 3.5 or above.

6.3.1 Windows, OSX, Linux

Create a working development environment by installing all the dependencies into your virtualenv with:

```
pip install -e ".[dev]"
```

Note: The Mu package distribution, as specified in `setup.py`, declares both runtime and extra dependencies.

The above mentioned `pip install -e ".[dev]"` installs all runtime dependencies and most development ones: it should serve nearly everyone.

For the sake of completeness, however, here are a few additional details. The `[dev]` extra is actually the aggregation of the following extras:

- `[tests]` specifies the testing dependencies, needed by `make test`.
- `[docs]` specifies the doc building dependencies, needed by `make docs`.
- `[i18n]` specifies the translation dependencies, needed by `make translate_*`.
- `[package]` specifies the packaging dependencies needed by `make win32`, `make win64`, `make macos`, or `make dist`.

Additionally, the following extras are defined:

- `[utils]` specifies the dependencies needed to run the utilities under the `utils` directory. It has been specifically excluded from the `[dev]` extra for two reasons: i) on the Windows platform, it requires a C compiler to be installed (as of this writing), and ii) running such utilities is seldom needed in Mu's development process.
- `[all]` includes all the dependencies in all extras.

Warning: Sometimes, having several different versions of PyQt installed on your machine can cause problems (see this issue for example).

Using a virtualenv will ensure your development environment is safely isolated from such problematic version conflicts.

If in doubt, throw away your virtualenv and start again with a fresh install as per the instructions above.

On Windows, use the venv module from the standard library to avoid an issue with the Qt modules missing a DLL:

```
py -3 -mvenv .venv
```

Virtual environment setup can vary depending on your operating system. To learn more about virtual environments, see this in-depth guide from Real Python.

6.3.2 Running Development Mu

Note: From this point onwards, instructions assume that you're using a virtual environment.

To run the local development version of Mu, in the root of the repository type:

```
python run.py
```

An alternative form is to type:

```
python -m mu
```

Yet another one is typing:

```
mu-editor
```

6.3.3 Raspberry Pi

If you are working on a Raspberry Pi there are additional steps to create a working development environment:

1. Install required dependencies from Raspbian repository:

   ```
   sudo apt-get install python3-pyqt5 python3-pyqt5.qsci python3-pyqt5.qtserialport␣
   ↪python3-pyqt5.qtsvg python3-dev python3-gpiozero python3-pgzero libxmlsec1-dev␣
   ↪libxml2 libxml2-dev
   ```

2. If you are running Raspbian Buster or newer you can also install this optional package:

   ```
   sudo apt-get install python3-pyqt5.qtchart
   ```

3. Create a virtualenv that uses Python 3 and allows the virtualenv access to the packages installed on your system via the `--system-site-packages` flag:

   ```
   sudo pip3 install virtualenv
   virtualenv -p /usr/bin/python3 --system-site-packages ~/mu-venv
   ```

4. Activate the virtual environment

   ```
   source ~/mu-venv/bin/activate
   ```

5. Clone mu:

```
(mu-venv) $ git clone https://github.com/mu-editor/mu.git ~/mu-source
```

6. With the virtualenv enabled, pip install the Python packages for the Raspberry Pi with:

```
(mu-venv) $ cd ~/mu-source
(mu-venv) $ pip install -e ".[dev]"
```

7. Run mu:

```
python run.py
```

An alternative form is to type:

```
python -m mu
```

Or even:

```
mu-editor
```

Warning: These instructions for Raspberry Pi only work with Raspbian version "Stretch".

If you use `pip` to install Mu on a Raspberry Pi, then the PyQt related packages will not be automatically installed from PyPI. This is why you need to use `apt-get` to install them instead, as described in step 1, above.

6.3.4 Using `make`

There is a Makefile that helps with most of the common workflows associated with development. Typing `make` on its own will list the options thus:

```
$ make

There is no default Makefile target right now. Try:

make run - run the local development version of Mu.
make clean - reset the project and remove auto-generated assets.
make pyflakes - run the PyFlakes code checker.
make pycodestyle - run the PEP8 style checker.
make test - run the test suite.
make coverage - view a report on test coverage.
make check - run all the checkers and tests.
make dist - make a dist/wheel for the project.
make publish-test - publish the project to PyPI test instance.
make publish-live - publish the project to PyPI production.
make docs - run sphinx to create project documentation.
make translate - create a messages.pot file for translations.
make translateall - as with translate but for all API strings.
make win32 - create a 32bit Windows installer for Mu.
make win64 - create a 64bit Windows installer for Mu.
make macos - create a macOS native application for Mu.
make video - create an mp4 video representing code commits.
```

Everything should be working if you can successfully run:

```
make check
```

(You'll see the results from various code quality tools, the test suite and code coverage.)

Note: On Windows there is a `make.cmd` file that works in a similar way to the `make` command on Unix-like operating systems.

Warning: In order to use the MicroPython REPL via USB serial you may need to add yourself to the `dialout` group on Linux.

6.3.5 Before Submitting

Before contributing code please make sure you've read *Contributing to Mu* and follow the checklist for contributing changes. We expect everyone participating in the development of Mu to act in accordance with the PSF's *Code of Conduct*.

6.4 Suggested First Steps

We love helpful, collaborative and friendly contributions!

If you would like to ease into contributing to Mu we'd like to suggest the following things to try out, depending upon your skills and interests.

If your contribution includes changes to code or documentation, you should read *Contributing to Mu* to learn about our expectations for submitting changes and improvements.

6.4.1 Bug Reports

If you think you have found a problem, then we want to hear about it!

We keep track of issues via our GitHub repository. You'll need to have an account on GitHub (joining GitHub is very easy) in order to submit such feedback.

When you create an issue we expect certain pieces of information from you:

- What you were doing (including all the necessary steps needed to recreate the situation you encountered).
- What you expected to happen, what actually happened and why you think this difference is problematic.
- Attach a copy of the logs generated by Mu (click on the cog icon in the bottom-right corner of Mu to display these logs, click on the logs and use CTRL-A to select all, then CTRL-C to copy and CTRL-V to paste the contents into the issue).

Please try to be precise and provide as much information as possible.

For what are obvious reaons, I hope you can see why we're unable to respond to issues that say some variation of, "when I click this button, it breaks". ;-)

6.4.2 Coding

The first thing to do is follow the instructions for *Developer Setup*.

You should read the explanation of *Mu's Architecture* to learn how Mu fits together. As of time of writing, Mu is a very small project with only around 4000 lines of Python code. However, it's important to know where to find different aspects of Mu's functionality and understand why Mu was put together in the way that it has been.

Assuming you've read and understood the architecture documentation an obvious and simple way to get started is to change the code in `logic.py` to suggest an alternative (better) message of the day. When Mu starts up, so the user sees that the status bar may contain textual messages from the application, a "message of the day" is displayed. These messages are defined near the top of `logic.py`.

If you'd rather try something more substantial, why not explore the list of currently open issues, fix one of them and create a pull request for your solution?

6.4.3 Translations

Mu uses Python's standard libraries and tools to translate the application into other languages. If you are fluent in a language that is currently not covered by Mu, then we would love you to help by providing us with a translation.

Full details of this process can be found in our guide on the *Internationalisation of Mu*.

6.4.4 Documentation

The documentation associated with Mu is not limited to addressing technical aspects of the editor (like the documentation you're reading right now). Our documentation encompasses tutorials, how-to guides, learning resources and other non-technical information for our users.

Such non-technical documentation is a part of Mu's website. If you are a teacher, learner or other interested party who wishes to contribute a tutorial, how-to or learning resource you should learn how to make such changes by reading the guide to *Developing Mu's Website*.

6.4.5 User Experience Research

Our users are at the centre of everything we do. We have two sorts of user in mind:

- Beginner programmers with little or no experience using a development environment.
- Those who support beginner programmers: teachers, club leaders, parents and other mentors.

When it comes to teaching and learning sometimes you just have to do what the experienced person says: for example, the professional musician explaining to the beginner how to hold an instrument "correctly". There is some notion of correctness that the experienced person understands is the best way to do something.

This also applies to learning to write code: we need to find ways to introduce the practice and conventions of programming in an effective manner. As Alan Kay said of Papert:

> "He realized, 'Oh, we could take the real content out here as a version in the child's world that is still the real thing.' It's not a fake version of math. It's kind of like little league, or even T-ball. In sports they do this all the time. In music, they do it all the time. The idea is, you never let the child do something that isn't the real thing – but you have to work your ass off to figure out what the real thing is in the context of the way their minds are working at that developmental level." – Alan Kay

How do we know what "the real thing" is in the context of a code editor for a beginner programmer? That's where *User Experience* comes in and we would love contributions from professional developers, beginner programmers and teachers to make sure Mu is an effective educational tool.

6.5 User Experience

We care deeply about all our users - we want using Mu to be a positive experience. In particular we focus on our primary users:

- Beginner programmers,
- Those who support beginner programmers.

They are at the centre of all Mu related development.

In order to understand how Mu can best support our users we need to learn about their needs, world view and how this reflects upon and influences their experience of using Mu (hence the name of this section).

This includes taking into account cultural differences, special educational needs, level of education and other aspects of a person's life that may impact on their accessibility to technology.

These are not problems to be "solved". Rather, this illustrates that Mu is an application to be evolved in a way to inclusively address the needs of users. As a result, it is important to keep Mu simple so that it is easy to make the inevitable changes needed. It's also important to point out that we will make mistakes and may need to revise how Mu works. Therefore Mu should be simple enough that it is easy to fix.

It's important to differentiate between design and usability. Plenty of software looks beautiful but is difficult to use. With Mu, we aim to put usability and a great experience before looks.

This beautiful yet inconvenient wine glass from the uncomfortable illustrates what I mean (used with permission, see *Copyright Information*).


© Katerina Kamprani - The Uncomfortable

6.5.1 What is UX?

I have some wonderful friends in the tech world and one in particular, Ann Carrier, was kind enough to explain her work in "user experience" which I'll reproduce below. It beautifully captures how I'd like user experience to relate to Mu. I sent her a series of questions to help me understand what I needed to do to bring about great UX in Mu and Ann gave some great answers.

What is UX and why is it important / useful?

> UX is short for User Experience. User Experience means the overall experience of a person using a product such as a website or computer application, especially in terms of how easy or pleasing it is to use.
>
> It's a really important part of the process of creating a product for people to use. It focuses on finding out about the needs of the people who will be using the product or service. This can also include how they behave, or what they do right now to achieve whatever it is they need to do. Once you understand this, you can then go about the process of designing something to meet those needs. You can even include users in the process through interviews, usability assessments or other workshops. There is absolutely nothing better than seeing people use a thing you've designed to help you figure out all the things you haven't quite got right!

Can you describe the processes and techniques you use as part of your job?

> I start with research. That takes many forms. Whether it's desk research into the latest apps or design patterns, or into the other products in the area we want to build a product in, or speaking to the people who are going to be using the product, this stage is vital.
>
> Once we have a better idea of the problem we're trying to solve, and the context which surrounds that problem, we can begin to try and solve it. There are many ways to do this, but I always find it helpful to draw things out. This is especially helpful when you're working in a team. Having a diagram (even if it's just boxes, lines and dodgy handwriting) that everyone can see and suggest changes to helps you know that everyone has a shared understanding.
>
> Once we have a shared understanding of the problem, and if there is one, the current workflows or processes we can then begin to look ahead to how this thing could work, in a magical perfect world that in reality, rarely exists. When we have this, we can then look at what steps we are going to take to get there. This allows us to have a firm "North Star" in mind, and with every step, re-assess not only whether that's the same direction we want to keep going in, but whether or not any solutions will take us closer towards, or further away from it. At this stage, we can and should also write down the success criteria - in other words, how will we know this is working?
>
> Next up is sketching out ideas for how the interface will look. Coming up with lots of ideas at this point is really useful. The first idea you have is rarely the best, so it's good to try and get a lot of ideas on to paper so that you can figure out the best ones. One technique you can use is to try to come up with 6 different solutions to a problem. And if you're struggling to come up with all 6, then try things like solving the problem the opposite way to the last idea, or something completely wacky. It's amazing how useful this can be. Sometimes, the right idea is just a tiny bit to the side of the absolutely wrong way of doing a thing.
>
> Sketching doesn't need to be just done by designers, and you certainly don't need to be an artist. As long as you can draw boxes and arrows, and your handwriting is neat enough that you can read it afterwards, you're good. The point of a sketch is to communicate an idea. If everyone you're working with can understand what you're trying to describe with the sketch, then that's all it needs to do.
>
> Once you've got more of an idea of what the solution will be, and talked that over with anyone who needs to be involved (Product Managers, Engineers, Testers, Users) then you can move on to more high fidelity designs. These are called mockups and they show what the user interface could look like, once it's been built. Even at this stage, changes - known as iterations - can and will still happen, because at every point you're finding out more and more and refining your idea, until you get to the point where it's built and in the hands of your users.

How does your UX work fit in with the wider software development project?

> In a good team, UX people are involved the whole way through, from the first ideas, through research, exploration of solutions right to the end when the product is out in the world. That said, user experience isn't just the responsibility of people with UX in their job titles. Everyone has a part to play in delivering a good user experience to the people using the product.

What advice would you give to people doing UX for the first time?

> If a good carpenter's rule is "measure twice, cut once" a good rule for UX people is "listen more than you talk". Do your research. Find out about the people who will be using the product. What they need. What they want. What problems they've currently got. How they work. Then keep talking to them as you design and build.

Is there anything else we should know about UX that's not been covered by your answers to the above?

> There's a great Shaker proverb which says:
>
> > "Don't make something unless it is both made necessary and useful; but if it is both necessary and useful, don't hesitate to make it beautiful."
>
> This gives you a great set of questions to ask of yourself whenever you're approaching a project.
>
> Necessary:
>
> 1. what problem are we trying to solve?
> 2. is the proposed solution needed (can it be solved a different way?)
> 3. will it solve the problem?
>
> Useful (and usable):
>
> 4. does the solution solve the problem for the people who need it?
> 5. does it work well?
>
> Beautiful:
>
> 6. does it look good? (beautiful things make people happy!)

6.5.2 UX and Mu

The "story so far" of Mu and UX starts with Carrie Anne Philbin's (director of education at the Raspberry Pi Foundation) keynote address to EuroPython 2015. This formed the basis for usability decisions when Mu was first created. While running workshops to test a browser based editor for the BBC's micro:bit, we'd heard from teachers that while the browser was very convenient in terms of setting things up, it was a pain to have to continually download scripts and then copy them onto the device and they also wanted easy access to MicroPython's REPL. I wondered "how hard can it be?" and set out to create an editor based on Carrie Anne's comments about the needs of teachers and learners when it came to code editing.

Halfway into the keynote Carrie Anne talks about a development environments for beginner programmers in Python:

She starts by explaining the problems with online editors. Often they require users to sign up, thus excluding a large number of children who, for legal (child protection) reasons, are not allowed to sign up because they have not reached the minimum age (usually around 14 years old) for them to be allowed to create their own accounts. Online editors introduce bureaucratic problems too: often schools use a "whitelist" system with their firewalls - they block everything except those sites on the whitelist. Getting a site onto the school's whitelist is often an onerously bureaucratic and slow task. Furthermore, assuming the website is available, many online editors expect their users to have access to modern hardware and browsers. This is often not the case and intractable technical problems result. Finally, a significant minority of children still don't have access to the internet, even in relatively advanced countries like the UK. For these reasons, a native developmnet environment is preferred.

Carrie Anne then explores two offerings for students to use as native code editors.

PyCharm has an educational edition that is both free and open. However, Carrie Anne claims it's not very obvious for either beginner developers or teachers how best to use the application. She mentions there are too many opportunities for users to fail because of the plethora of buttons and menus. As a teacher, she wants something simple and obvious.

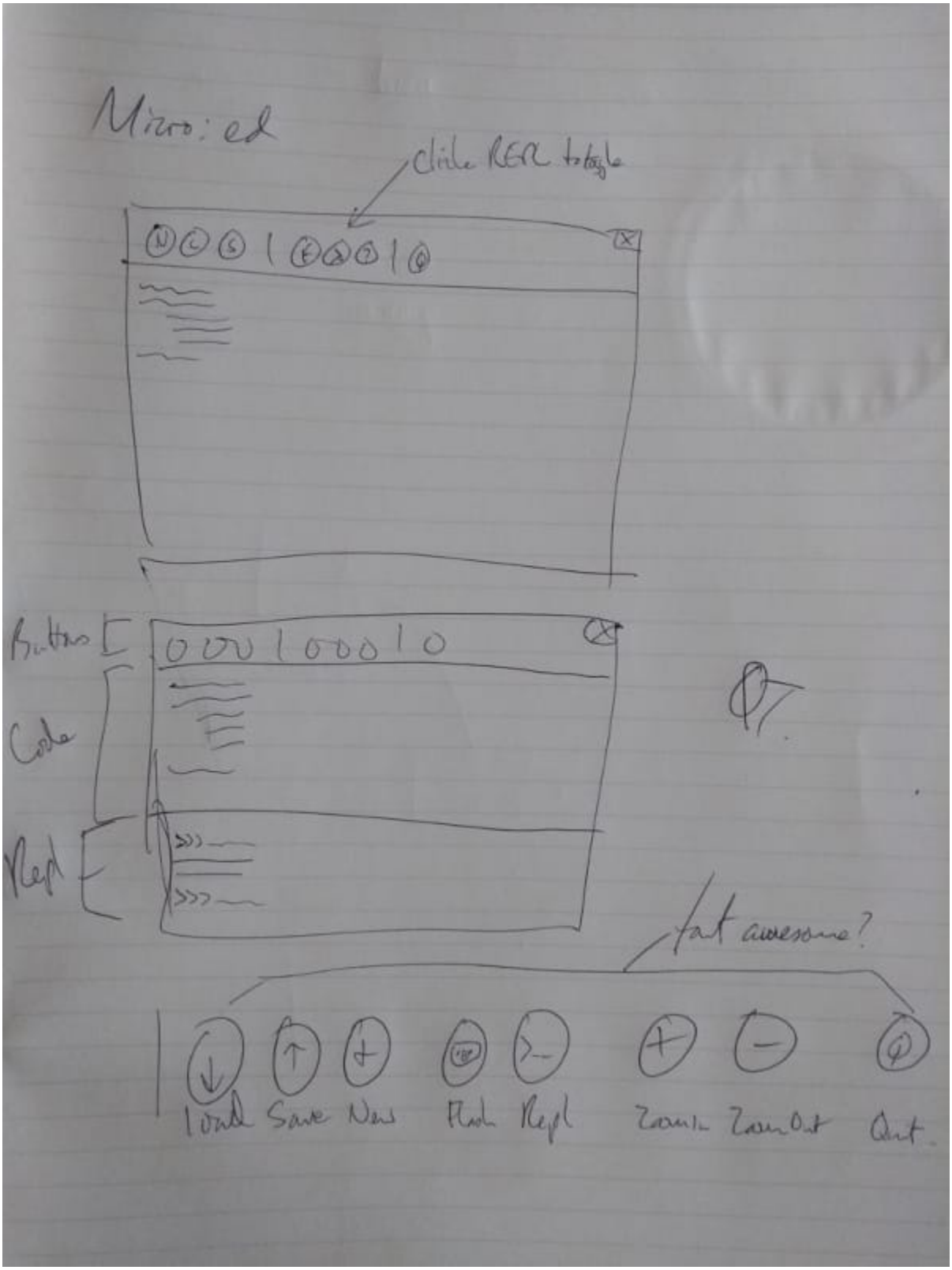
Next, she turns her attention to Idle - the editor that comes bundled with Python. It's good that Idle is free, has some syntax highlighting, auto-indents Python code, is cross platform, leightweight and simple. However, there are no line numbers, error reporting is incomrehensible to beginners and, most importantly, there are two separate windows that often get lost or confused with each other (one for code, the other for a sort of REPL).

She suggests we turn out attention to a project called SonicPi, a sort of programmable music tool for the Raspberry Pi, as an example of the sorts of features teachers and learners desire in a coding environment. She enumerates features that may not immediately seem important for beginner programmers and teachers.

- All the panels are in the same window and it's obvious what each one does.
- There's built-in help.
- There are a limited number of obviously named buttons that encompass the core tasks required of the user.
- Zooming in and out is a killer feature for teachers.
- Simple things like line numbers and help for aligning code make a huge difference.

Finally, she challenges the audience by asking, "Why can't we have something like that for Python?" Being of a teacher-ish disposition she sets the assembled conference homework to be due in 2016.

When I started work on Mu I watched the video mentioned above and sketched a rough outline of how Mu might work in terms of usability, reproduced below.



Notice that while the details are obviously different, the core interface looks like Mu (if you're wondering what "micro:ed" refers to - it was Mu's original name until the BBC got shirty about it and I changed it to Mu). I simply took Carrie Anne's suggestions and made the simplest thing possible.

Since then I've interviewed many teachers, observed lots of lessons and workshops and gathered feedback from users online. Changes to the usability of Mu generally follow a pattern:

- We find evidence of several people wanting a change to make their lives easier (we tend to ignore single case exacmples of desired changes).
- We use our issue tracking system built into GitHub as a way to come up with a tangible plan.
- We create the simplest possible solution and ask for feedback.
- Iterate!

6.5.3 Resources for UX

In addition to providing answers about UX, Ann very kindly pointed me to various resources on the web that helped me to understand the challenges and work needed to do actionable UX research.

Andrew Travers has blogged about a free pocket guide he has written on interviewing for research. I found this invaluable reading and helped me to prepare for the observations and interviews I conducted as part of the process of developing Mu. This is where I would start if I were new to UX research and wanted to get a quick overview of things to do.

The Government Digital Service of the UK Government has an international reputation for software development greatness. The foundation of this reputation are the documents it releases, for free, that outline the "best practices" and expectations about process that GDS have about various aspects of the software development process. Their service manual on user research is a comprehensive outline of the various tasks, processes and outcomes needed to do effective UX research. I particularly found the section on analysis of UX research helpful.

Finally, it's good to read the suggestions, heuristics and best practice for working with users who have additional requirements when using software. Again, the UK government's GDS has a number of resources, although I found this blog post on the Dos and don'ts on designing for accessibility (and the associated posters) to be a rich seam of useful advice. All their resources in this context can be found on their page about accessibility and assisted digital.

Mu has a long way to go on its path to being an inclusive and accessible code editor, but what is certain is that UX is a core driver of this journey.

6.6 Mu's Architecture

This section provides a high level overview of how the various parts of Mu fit together.

6.6.1 Key Concepts

The key concepts you should know are:

- Mu uses the PyQT5 framework (that makes the Qt GUI toolkit available to Python) for making its user interface.
- Mu is a modal editor: the behaviour of Mu changes, depending on mode.
- There are a number of core features and behaviours that are always available and never vary, no matter the mode.
- The text area into which users type code is based on a Scintilla based widget.
- Mu is easy to internationalise using Python's standard `gettext` based modules and tools.
- Mu's code base is small, well documented and has 100% unit test coverage.

6.6.2 Code Structure

The code is found in the `mu` directory and organised in the following way:

- The application is created and configured in `app.py`.
- Most of the fundamental logic for Mu is in `logic.py`.
- Un-packaged third party code used by Mu is found in `contrib`.
- The Python3 debugger consists of a debug client and debug runner found in the `debugger` namespace. A description of how the debugger works can be found in *Python Runner/Debugger*.
- Interacting with the UI layer is done via the `Window` class in the `interface.main` module. Mu specific UI code used by the `Window` class found in the other modules in the namespace.
- Internationalization (I18n) related assets are found under `locale`. Learn how this works via *Internationalisation of Mu*.
- Modes are found under the `modes` namespace. They all inherit from a `BaseMode` class and there's a tutorial for *Modes in Mu*.
- Graphical assets, fonts and CSS descriptions for the themes are all found under `resources`.

All classes, methods and functions have documentation *written for humans*. These are extracted into the *Mu API Reference*.

Mu's Test Suite is in the `test` directory and filenames for tests relate directly to the file they test in the Mu code base. The module / directory structure mirrors the organisation of the Mu code base. We use PyTest's assert based unit testing. All tests have a comment describing their intent.

The documentation you're reading right now (i.e. that written for developers) is found in the `docs` directory. We use Sphinx to write our docs and host them on ReadTheDocs. Other documentation (tutorials, user help and so on) is on *Developing Mu's Website*.

The `utils` directory contains various scripts used to scrape and / or build the API documentation used by Mu's autocomplete and call tip functionality.

The other assets in the root directory of the project are mainly for documentation (such as our Code of Conduct), configuration (for testing) or packaging for various platforms (see *Packaging Mu*).

If you want to make changes please read *Contributing to Mu*.

6.7 Modes in Mu

Mu is a modal editor: it behaves differently depending on the currently selected mode. The name of the current mode is always displayed in the bottom right hand corner of Mu's window. Clicking on the mode button opens up a dialog box to allow users to select a new mode.

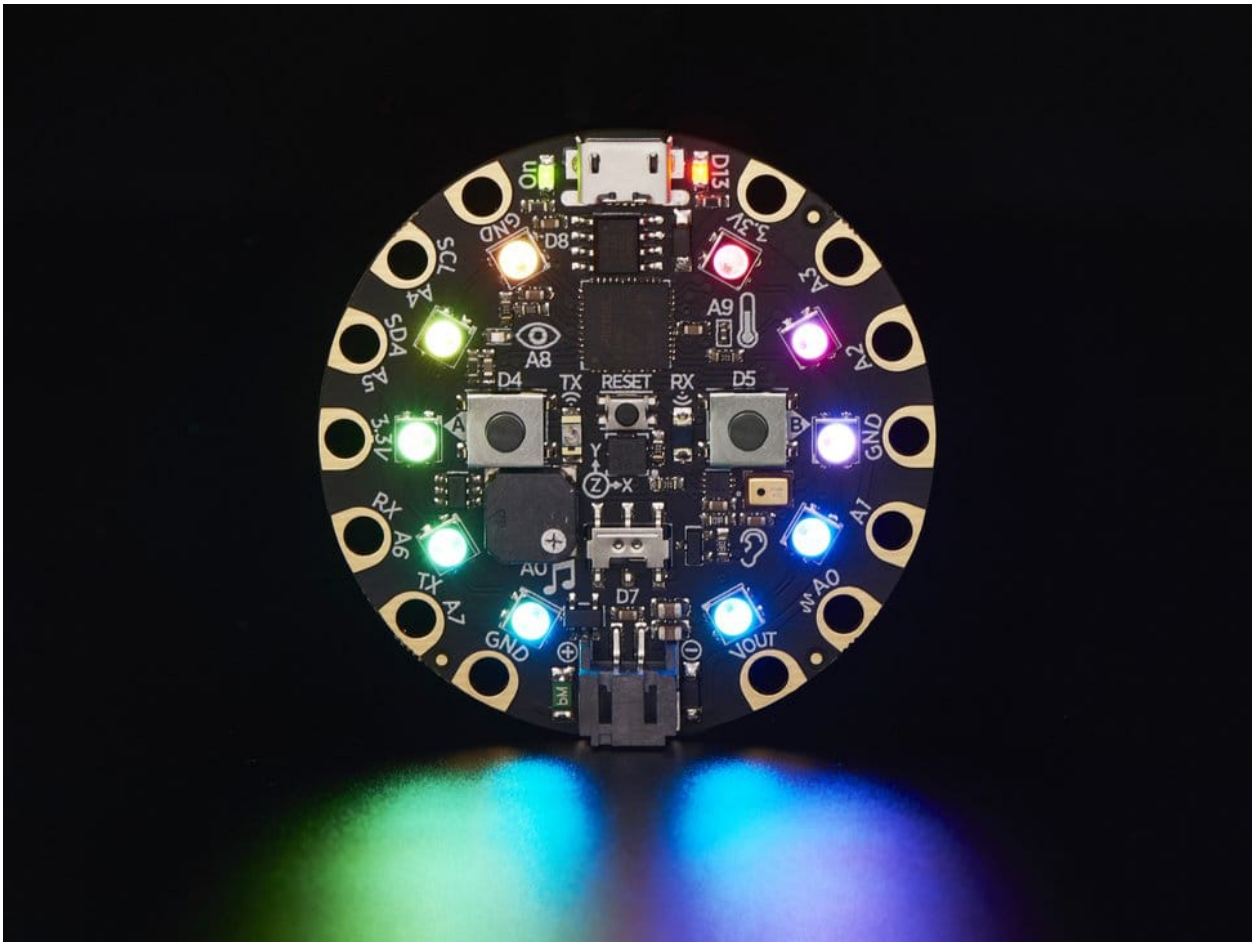
6.7.1 What Are Modes?

Modes are a way to customise how Mu should behave. This simplifies Mu: rather than trying to provide every possible feature at once (and thus become a nightmare of complexity for the user), modes bring related features together in a simple and easy to use manner.

Modes are able to add buttons to the user interface, handle certain events (such as when one of the mode's buttons is clicked) and provide contextual information for Mu (such as where files should be saved or what API metadata is available). It's also possible for one mode to transition to another and some modes are only available as transitional modes (i.e. they may not be selected by the user). A good example of such a "transitional" mode is the Python 3 debugger, which can only be accessed from the standard Python 3 mode.

Mu contains the following modes, although it is very easy to add more (the images below are used with permission, see *Copyright Information*).

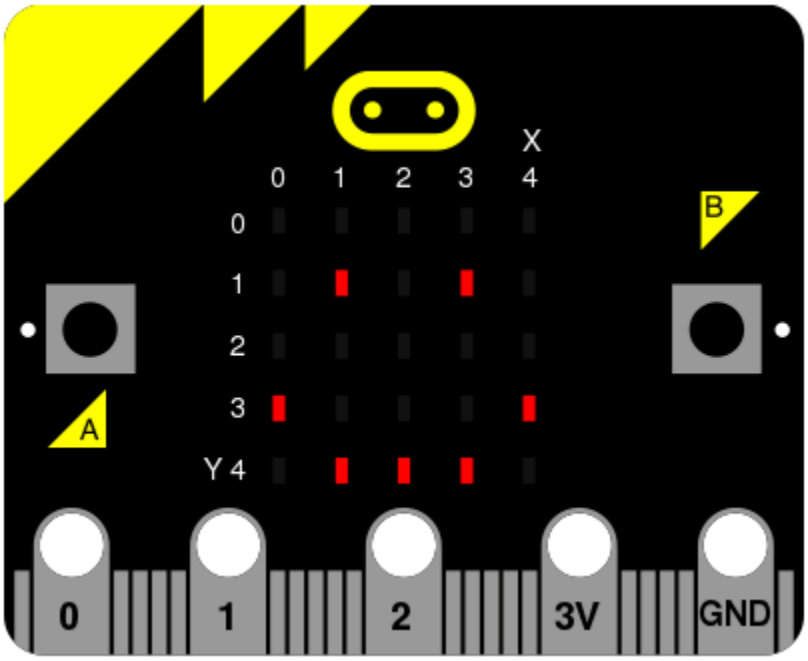
Adafruit Mode



Adafruit make extraordinarily awesome boards for embedded development. Many of these boards run Adafruit's own flavour of MicroPython called CircuitPython.

The Adafruit mode inherits from a base MicroPython mode that provides USB/serial connectivity to the board. Because source code is stored directly on the Adafruit boards, this mode ensures that filesystem based operations actually happen on the connected device. If no such device is found, the mode will warn you.

BBC micro:bit Mode



The BBC micro:bit is a small computing device for young coders that is capable of running MicroPython. Mu was originally created as volunteer led effort as part of the Python Software Foundation's contribution to the project.

Just like the Adafruit mode, micro:bit mode inherits from a base MicroPython mode so there's a REPL based interface to the device. It also provides functionality to "flash" (i.e. copy) your code onto the device and a simple user interface to the simple file system on the device.

Pygame Zero / PyGame Mode

PyGame (or, more correctly: "pygame") is a cross platform set of Python libraries for writing games. Pygame Zero is a wrapper for pygame that makes it easy for beginners to make games. If both pygame and Pygame Zero are installed (as they are if you used the official Windows installer), Mu's Pygame Zero mode makes it easy for beginner programmers to create games.

This mode provides a "Play" button that uses Pygame Zero's game-runner to launch the user's games. Two further buttons open the operating system's file system explorer for the directories containing images and sounds used in the user's games. This makes it easy for the user to copy and paste new game assets into the right place.

The standard Python3 mode (see below) is probably a better environment for more advanced pygame-only development. Mu ensures that all the game assets required by the Pygame Zero introductory tutorial are available by default.

Standard Python3 Mode

This mode is for creating simple Python 3 programs. As with the other modes, there is a REPL for live programming, but in this case it is an iPython based REPL that uses project Jupyter. As with other Jupyter notebooks, it's possible to embed graphics and charts into the REPL so it becomes a interesting to read and work with.

There are two ways to run your script in this mode:

1. Click the "Run" button: will launch the script using Python's interactive mode (so you'll be dropped into a basic interactive Python shell upon the script's completion).
2. Click the "Debug" button: Python mode transitions to the debug mode - a graphical way to inspect and watch your code execute.

Because of the overhead needed to start the graphical debugger it takes longer to start running your script. This is especially noticable on the Raspberry Pi.

Python 2 isn't supported by Mu and never will be.

Debug Mode

It's only possible to enter debug mode from standard Python mode. It's purpose is to manage the execution and inspection of your code.

Clicking the margin of the editor toggles "break points" that tell the debugger where to pause. Once paused it's possible to inspect the state of various objects at that moment in the code's execution and step over, into and out of lines of code. You're able to watch Python execute your code, allowing you to discover where there may be bugs.

Once the code has finished the debug mode transitions back to standard Python mode.

6.7.2 Create a New Mode

It's very easy to create new mode for Mu. The following tutorial explains how we created the Pygame Zero mode.

Create a Class

The most important aspects of a mode are encapsulated in a class that represents the mode. These classes live in the `mu.modes` namespace and **must** inherit from the `mu.modes.base.BaseMode` class. If your new mode is for a MicroPython based device, you should inherit from the `mu.modes.base.MicroPythonMode` class, since this includes various helpful utilities for such things as finding a connected device and running a REPL over a USB-serial connection.

The naming convention is to create a new module in which is found the class representing the mode. For example, for Pygame Zero, the new module is `mu.modes.pygamezero` in which is found the `PyGameZeroMode` class that inherits form `BaseMode`.

Integrate the Mode

Mu needs to know that the new mode is available to use. This is fulfilled by a couple of relatively simple steps:

- Add the mode's class to the `__all__` list in the `__init__.py` file for the `mu.modes` namespace.
- In `mu.app.py` import the new mode from `mu.modes` and add an instance of the mode's class to the dictionary returned by the `setup_modes` function. (All modes are instantiated with the available `editor` and `view` objects that represnt the editor's logic and UI layer respectively.)

Update the Class's Behaviour

The core elements of your new mode's class that need updating include some attributes and three methods.

The attributes that must be changed are:

- `name` – the full name of the mode, for example, "PyGame Zero".
- `description` – a short description of the mode to be displayed in the mode picker. For example, "Make games with Pygame Zero".
- `icon` – an icon used to represent the mode in the mode picker. This must be a `.png` image file found in the `mu/resources/images` directory.

Additional attritbutes with safe default values set in the `BaseMode` class which may be of value for you to change are:

- `save_timeout` – the number of seconds to wait before auto-saving work. If this value is 0 (zero) Mu will not auto-save changed files when in this mode.
- `builtins` – a list of strings defining symbols that Mu's code checker must assume are builtins (above and beyond Python's standard builtins).

Note: When creating strings that will be seen by users please remember to use the conventions for internationalization (i18n). Put simply, enclose your strings in a call to _ like this:

```
_('This string will be translated automatically')
```

Please see *Internationalisation of Mu* for more details.

You should pay attention to three methods of your class: `actions`, `api` and `workspace_dir`. You must override `actions` and `api` (see below) and *may* want to override `workspace_dir`.

The purpose of the `workspace_dir` method is to return a string representation of the path to the directory containing the code created with this mode. The default implementation in `BaseMode` is generally safe to use although some CircuitPython based boards may want to use this method to point to a connected device (if attached) or a safe default on the user's filesystem (if no device is attached). See how it's done in the `AdafruitMode` class. If in doubt, just use the method inherited from `BaseMode`.

However, you **must** override the `actions` method. It must return a list of dictionaries that describe the buttons to be added to Mu's user interface. Each dictionary must contain the following key/value pairs:

- `name` – the name of the button which doubles as the name of the icon found in `mu/resources/images` used as the visual representation of the button. To create a new button start with the blank `button.png` image and use either an icon from the FontAwesome set of icons, or some other graphical device that looks visually similar. Make sure that the colour of the image is correct blue of (hex value) #336699. Please remember to centre it within the button and make sure it has the same sort of scale as the existing buttons.
- `display_name` – the string displayed immediately underneath the button in Mu's user interface.
- `description` – the string displayed as a tool-top when the mouse pointer hovers over the button, but the button remains unclicked.
- `handler` – a reference to a method you have created in your mode's class that is called, with an event object, when the button is clicked.
- `shortcut` – a string representation of the keyboard shortcut for the button. Valid examples include, `'F5'` (for function key 5) or, `'Ctrl+Shift+I'` (for control-shift-I).

By way of illustration, here's the list of dictionaries returned in the Pygame Zero mode:

```
[
    {
        'name': 'play',
        'display_name': _('Play'),
        'description': _('Play your PyGame Zero game.'),
        'handler': self.play_toggle,
        'shortcut': 'F5',
    },
    {
        'name': 'images',
        'display_name': _('Images'),
        'description': _('Show the images used by PyGame Zero.'),
        'handler': self.show_images,
        'shortcut': 'Ctrl+Shift+I',
    },
    {
        'name': 'sounds',
        'display_name': _('Sounds'),
        'description': _('Show the sounds used by PyGame Zero.'),
        'handler': self.show_sounds,
        'shortcut': 'Ctrl+Shift+S',
    },
]
```

Notice how the handlers are references to methods of the `PyGameZeroMode` class, the details of which are left to the creator of the mode. Mu simply calls the handler and expects the author of the mode to know what they're doing.

Interactions with the Mu editor are via two objects referenced within the class:

- `self.editor` – represents an object containing the core logic of the editor (an instance of `mu.logic.Editor`).

- `self.view` – references the main GUI object through which all display and user interface related operations should pass (an instance of `mu.interface.main.Window`).

Please see the *Mu API Reference* for specific details of what these two objects offer.

Finally, you **must** also override the `api` method, whose role is to provide a list of strings that conform to Scintilla's protocol for defining and documenting API's to be used with autocomplete and call-tips. The protocol is:

```
'foo.bar(arg1, args2="baz") \nMulti line\n\nEnglish description.`
```

Happily, various scripts in the `utils` directory can be used, cloned and modified to autogenerate this documentation from source code. The reason the extraction of such API related information is automated is so it makes it very quick and easy to revise such data as APIs change in the future.

Take a look at the `pgzero_api.py` file and you'll find a simple recipe for extracting such information from Python modules. Three modules for Python's standard library (`json`, `inspect` and `importlib`) are used to import the modules we're interested in, inspect the signatures of the callable objects found therein and emit a JSON based output (called `pgzero_api.json`).

The resulting JSON is a list of JSON objects containing three attributes:

- `name` – the module name + object name.
- `args` – a list of the arguments taken by the callable Python object being described.
- `description` – the docstring associated with the Python object.

Here's an example of such an object from the emitted `pgzero_api.json` file:

```
{
    "description": "Interface to the screen.",
    "name": "screen.Screen",
    "args": [
        "surface"
    ]
}
```

Given such JSON serialised data, the `mkapi.py` command will take such a file as input and emit to stdout a list of strings for the API that conform to Scintilla's protocol to be used by autocomplete and call-tips.

In the case of the Pygame Zero mode, the output from the `mkapi.py` command ended up in `mu.modes.api.PYGAMEZERO_APIS`. The list itself is in the `pygamezero.py` file in the `mu/modes/api` directory, and the `__init__.py` found therein exposes it via the `__all__` list.

Back in the `PyGameZeroMode` class the `api` method simply returns a concatenated list of the APIs that a user of the mode may use:

```
from mu.modes.api import (PYTHON3_APIS, SHARED_APIS, PI_APIS,
                          PYGAMEZERO_APIS)

... later in the PyGameZeroMode class ...

def api(self):
    return SHARED_APIS + PYTHON3_APIS + PI_APIS + PYGAMEZERO_APIS
```

With these relatively simple steps, it's possible to create quite powerful modes. Most importantly, taking a look at the existing modes in the `mu.modes` namespace will reveal how to do most of the things you'll need.

However, there is one final aspect of creating a mode that we need to address.

Unit Test the Mode

We will not accept any new modes without 100% unit test coverage.

Please read the guide about *Mu's Test Suite* for how Mu is tested and the various expectations we have when it comes to writing tests.

If you are unsure about the best way to go about testing your mode please feel free to ask for help. We would much rather get a pull request for a "spike" (draft) version of a new mode and work with the original author on testing the code, than have no pull request at all.

If in doubt, ask. We're a friendly bunch and *Contributing to Mu* is easy.

6.8 Internationalisation of Mu

A really useful and relatively simple way to contribute to Mu is to translate the user interface into a different language. The steps to do this are very simple and there exist plenty of tools to help you.

You can contribute in three ways:

- Improve or extend an existing translation.
- Create a completely new translation for a new language.
- Make a translation of Mu's website (see the *Developing Mu's Website* guide for how to do this).

In both cases you'll be using assets found in the `mu/locale` directory.

Mu uses Python's standard gettext based internationalization API so we can make use of standard tools to help translators, such as babel or Poedit.

Non-technical users

If you are not a technical user and you are not familiar with the tools and jargon we use in this guide, please reach out to us by creating a new issue in GitHub.

We will help you set up a user-friendly tool that you can use to contribute new or improved translations, and integrate them into the next Mu release.

We welcome translations from all users!

6.8.1 How To

Updating or creating a new translation for Mu's user interface requires *setting up a development environment* beforehand and, from there, is a four-step process:

1. Produce an up to date `mu.po` file

Open a CLI shell, change the working directory to Mu's repository root, and run:

```
$ make translate_begin LANG=xx_XX
```

Where `xx_XX` is the identifier for the target language.

This creates (or updates, if it already exists) the `mu.po` file under the `mu/locale/xx_XX/LC_MESSAGES/` directory – this is where the original British English messages are associated with their localized translations.

2. Translate Mu user interface strings

Use a tool of your choice to edit the `mu.po` file:

- Those looking for a GUI based tool can try out Poedit.
- Others might prefer a plain text editor, which will be sufficient.

3. Check the translation result

As you progress, check the translation results by launching Mu with:

```
$ make translate_test LANG=xx_XX
```

As before, `xx_XX` is the identifier for the target language.

When done checking, quit Mu, and go back to step 2. as many times as needed.

4. Submit your translation work

This process produced two new or updated files, both under the `mu/locale/xx_XX/LC_MESSAGES/` directory:

- `mu.po` containing the text based source of the translation strings.
- `mu.mo` containing a compiled version of the above, used by Mu at runtime.

Commit your changes and create a pull request via GitHub.

Thanks!

6.9 Python Runner/Debugger

An obvious requirement for a Python editor is to run your Python code. For standard Python, Mu does this in two ways:

- With the Python runner (press the "Run" button).
- With the graphical debugger (click the "Debug" button).

Note: For MicroPython based modes, the code is run on the attached embedded device and not directly by Mu. For example, saving your code on an Adafruit board restarts the device and Circuit Python evaluates your code.

Both the Python runner and grapical debugger were created with the financial support of the Raspberry Pi Foundation.

If you are creating a new standard Python mode for Mu, you should *at least* make available the Python runner (please see *Modes in Mu* for more information about how to do this).

Both methods of running Python code essentially work in the same way: they fire up a new child process and connect its stdin, stdout, stderr to the `PythonProcessPane` found in the `mu.interface.panes` namespace so you're able to interact with it in a terminal like environment.

However, the Python runner starts immediately whereas the debugger has to set up a bunch of debug-related scaffolding, which makes it start slower. This is especially noticeable on the less powerful Raspberry Pi machine. Basically, if you just want to run your script, use the Python runner.

6.9.1 Python Runner

The essentials of the Python runner are in the afore mentioned `PythonProcessPane` class. The `start_process` method is used to create the new child process. The resulting process becomes a `process` attribute on the instance of the `PythonProcessPane`.

You have some control over how the child process behaves.

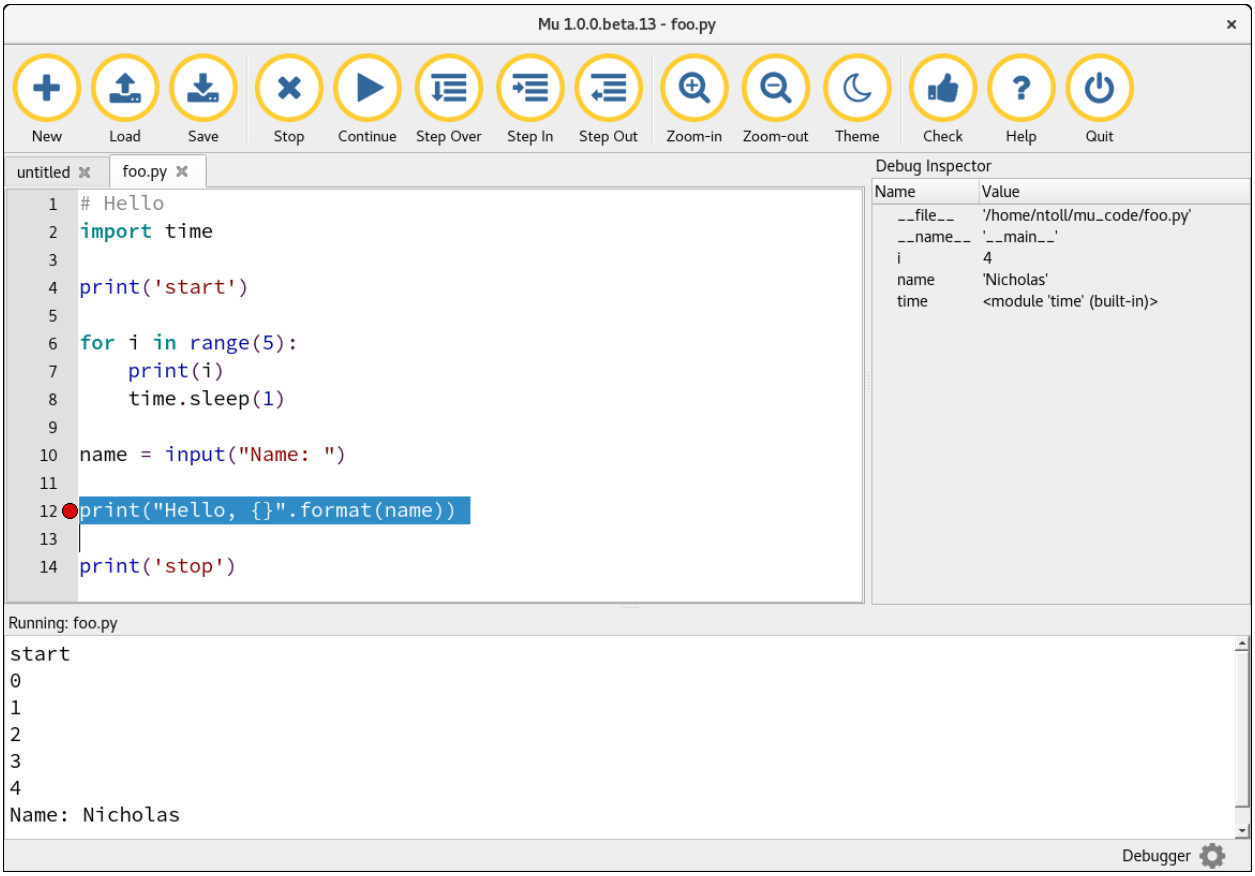
- You should supply the `script_name` to run.
- You must also provide a `working_directory` within which the script will run (this is usually the user's `mu_code` directory).
- The `interactive` flag (which defaults to `True`) will mean the user will drop into a simple Python REPL when the script completes. The default is at the request of the Raspberry Pi Foundation who explain that it is often handy for beginner developers to run their script and then explore the resulting context interactively.
- If the `debugger` flag is set to `True` (the default is `False`) then the debug runner (see below) is started in a child process for the referenced script. This overrides the `interactive` flag to being `False`.
- Any `command_args` for the referenced script should be a list of strings. The default is no `command_args` (i.e. None).

Handlers are configured to handle various events, such as when the process finishes or when a user type a character.

The `PythonProcessPane` includes basic command history and input editing features. It'll also respond to CTRL-C and CTRL-D. Copy and paste can be accessed via a context menu.

6.9.2 Graphical Debugger

The graphical debugger exists to give beginner programmers an easy way to observe their code while it is running and allows you to use breakpoints, step over and into code as well as use a simple object inspector to view the status of objects in scope.



When a user clicks the "Debug" button Mu transitions to "debug" mode which exposes the functionality of the debugger client which, in turn, communicates with the debug runner process which is actually driving the user's script.

The debugger is designed to be as simple as possible in order to introduce beginner programmers to the basic concepts of a debugger in the easiest way. It does **NOT** strive to be extensive or particularly powerful. Rather, its aim is to encourage beginner programmers to explore their code while it is running.

In this sense if conforms to the Mu outlook of providing the first steps for a beginner programmer with a view to them quickly graduating to a "proper" development environment once they've found their feet.

Most of the debugger's functionality can be found in the `mu.debugger` namespace. Coordination is done in the `mu.modes.debugger.DebugMode` class.

Debug Client

The debug client exists within the Mu process. It spins up an instance of the `mu.debugger.client.CommandBufferHandler` class in a separate thread to handle inter-process communication in a non-blocking manner, so the UI thread is never blocked.

The `mu.debugger.client.Debugger` class is used to react to incoming events from, and as an API for Mu to issue commands to the debug runner. It uses a reference to a `view` object to update the user inteface as events are detected.

Debug Runner

The debug runner exists on a new child process and makes use of Python's bdb debugger framework. It spins up a new thread to run the `command_buffer` function that listens for incoming commands.

The most interesting aspects of the runner are found in the `mu.debugger.runner.Debugger` class which inherits from the `bdb.Bdb` class found in Python's standard library. It responds to commands from the client and sends messages when various events occur during the debugging process. These messages are picked up by the debug client and reflected in Mu's UI.

The `mu.debugger.runner.run` function is the entry point for the debug runner and, as specified in Mu's `setup.py`, is accessed via the `mu-debug` command. This command expects at least one argument: the name of the script to be debugged. Any further arguments are passed on to the script to be debugged.

6.10 Mu's Test Suite

We have tests so we can make changes with confidence.

We use several different sorts of test:

- PyFlakes for checking for errors in our code.
- pycodestyle for making sure our coding style conforms with most of the conventions of PEP8.
- PyTest as a framework for writing our unit tests.
- Coverage for checking the coverage of our unit tests.

> **Warning:** We currently have 100% test coverage.
>
> It means **every line of code in Mu has been exercised by at least one unit test**. We would like to keep it this way!
>
> We can't claim that Mu is bug-free, but we can claim that we've expressed an opinion about how every line of code should behave. Furthermore, our opinion of how such code behaves may **NOT** be accurate or even desirable. ;-)

In addition, we regularly make use of the excellent LGTM online code quality service written, in part, by friend-of-Mu, Dr.Mark Shannon.

6.10.1 Running the Tests

Running the tests couldn't be simpler: just use the `make` command:

```
$ make check
```

This will run **ALL** the tests of each type.

To run specific types of test please try: `make pyflakes`, `make pycodestyle`, `make test` or `make coverage`.

> **Warning:** The test suite will only work if you have installed all the requirements for developing Mu.
>
> Please see *Developer Setup* for more information on how to achieve this.

6.10.2 Writing a New Test

All the unit tests are in the `tests` subdirectory in the root of Mu's repository. The tests are organised to mirror the code structure of the application itself. For example, the tests for the `mu.modes.base` namespace are in the `tests.modes.test_base.py` file.

As mentioned above, we use PyTest as a framework for writing our unit tests. Please refer to their extensive documentation for more details.

In terms of our expectation for writing a test, we expect it to look something like the following:

```
def test_MyClass_function_name_extra_info():
    """
    This is a description of the INTENTION of the test. For
    example, we may want to know why this test is important,
    any special context information and even a reference to a
    bug report if required.
    """
    assert True  # As per PyTest conventions, use simple asserts.
```

We also expect your test code to pass PyFlakes and PEP checks. If in doubt, don't hesitate to get in touch and ask.

6.11 Packaging Mu

Because our target users (beginner programmers and those who support them) may not be confident with the technical requirements for installing packages, we need to make obtaining and setting up Mu as simple and easy as possible.

Furthermore, we aim to make the creation of packages automatic and as simple as possible. By automating this process we ensure that the knowledge and steps needed to package Mu is stored in software (so everyone can see how we do it) and we don't rely on a volunteer to take time and effort to make things happen. If you submit code and it is accepted into our master branch, within minutes you should have a set of packages for different platforms that includes your changes. Such builds can be found here.

Of course, such builds are not "official" releases. We'll only do that every so often when major updates land. These will take the form of releases found in our GitHub repository. Such releases will include the "official" installers for supported platforms. The installers referenced on Mu's website will always be the latest stable release of Mu on GitHub.

Note: Huge thanks to Carlos Pereira Atencio who made considerable efforts to automate and configure the packaging of Mu. Without the contributions of volunteers like Carlos, projects like Mu simply wouldn't exist. If you find Mu useful why not say thank you to Carlos via Twitter..?

Thank you Carlos! :-)

We package Mu in various different ways so it is as widely available as possible. What follows is a brief description of how each package is generated (some of them require the manual intervention of others outside the Mu project).

6.11.1 Python Package

If you have Python 3.5 or later installed on Windows, OSX or 64-bit Linux and you are familiar with Python's built-in packaging system, you can install Mu into a virtual environment with `pip`:

```
$ pip install mu-editor
```

Note: By design, `pip` will not create any shortcuts for applications that it installs.

If you want to add a shortcut for Mu to your desktop/start menu you can use Martin O'Hanlon's amazingly useful Shortcut tool like this:

```
$ pip install shortcut
$ shortcut mu
```

As per conventions, the `setup.py` file contains all the details used by `pip` to install it. We use twine to push releases to PyPI and I (Nicholas - maintainer) simply use a Makefile to automate this:

```
$ make publish-test
$ make publish-live
```

The `make publish-live` command is what updates PyPI. The `make publish-test` command uses the test instance of PyPI so we can confirm the release looks, behaves and works as expected before pushing to live.

6.11.2 Raspberry Pi

Raspberry Pi OS (previously called Raspbian) is the official operating system for the Raspberry Pi and features Mu as Recommended Software. Raspberry Pi OS uses the Mu packages contributed to Debian by Nick Morrott.

To install Mu on Raspberry Pi OS from the command line, type:

```
$ sudo apt install mu-editor
```

Alternatively, Mu can be installed from the Recommended Software menu in the Programming section.

Warning: Since Mu for Raspberry Pi OS is packaged by a third party, our latest releases may not be immediately available.

6.11.3 Windows Installer

Packaging for Windows is essential for the widespread use of Mu since most computers in schools run this operating system. Furthermore, feedback from school network administrators tells us that they prefer installers since these are easier to install "in bulk" to computing labs.

There are two versions of the installer: one for 32bit Windows and the other for 64bit Windows. The 32bit version has been tested on Windows 7 and the 64bit version has been tested on Windows 10. Support for anything other than Windows 10 is important, but a "best effort" affair. If you find you're having problems please submit a bug report.

The latest *unsigned* builds for Mu on Windows can be found here.

Mu for Windows contains its own version of Python packaged in such a way that makes it only usable within the context of Mu (Python's so-called isolated mode). Of course, the version of Python in Mu will have as much or little access to computing resources as the host operating system will allow.

Packaging is automated using the Appveyor cloud based continuous integration solution for Windows. The .appveyor.yml file found in the root of Mu's repository, configures and describes this process. You can see the history of such builds here.

We use the NSIS tool to build the installers. This process if coordinated by the amazing pynsist utility.

Note: Pynsist is the creation of Thomas Kluyver, who has done an amazing job creating many useful tools and utilities for the wider Python community (for example, Thomas is also responsible for the Jupyter widget Mu uses for the REPL in Python 3 mode).

On several occasions Thomas has volunteered his time to help Mu. Like Carlos, Thomas is another example of the invaluable efforts that go into making Mu. Once again, if you find Mu useful, please don't hesitate to thank Thomas via Twitter.

Thank you Thomas!

The required configuration file for `pynsist` is automatically generated at packaging time, under a temporary working directory. The motive for that arises from the need to ensure that Mu's dependencies are sourced from a single place, which is `setup.py`. The `win_installer.py` script handles that, runs `pynsist`, moves the resulting installer executable to the `dist` directory, and cleans up. If you're interested in learning more, the script includes comments with detailed notes (also, check out the `pynsist` specification for configuration files).

The automated builds are unsigned, so Windows will complain about the software coming from an untrusted source. The official releases will be signed by me (Nicholas Tollervey - the current maintainer) on my local machine using a private key and uploaded to GitHub and associated with the relevant release. The instructions for cryptographically signing installers explain this process more fully (the details of which are described by Mozilla).

Use the `make` command to build your own installers:

```
$ make win32
$ make win64
```

This will clean the repository before running the `win_installer.py` command for the requested bitness.

Because Mu depends on the availability of tkinter, part of the build process is to download the appropriate tkinter-related resources from Mu's tkinter assets repository.

If asked, the command for automatically installing Mu, system wide, should use the following flags:

```
mu-editor_win64.exe /S /AllUsers
```

The `/S` flag tells the installer to work in "silent" mode (i.e. you won't see the windows shown in the screenshots above) and the `/AllUsers` flag makes Mu available to all users of the system (i.e. it's installed "system wide").

6.11.4 OSX App Installer

We use Travis to automate the building of the .app and .dmg installer (see the `.travis` file in the root of Mu's GIT repository for the steps involved). This process is controlled by Briefcase (part of the BeeWare suite of tools) which piggy-backs onto the `setup.py` script to build the necessary assets. To ensure Mu has Python 3 available for it to both run and use for evaluating users' scripts, we have created a portable/embeddable Python runtime whose automated build scripts can be found in this repository. This is the Python version used by Mu (not the one on the user's machine).

The end result of submitting a commit to Mu's master branch is an automatically generated installable for OSX. These assets are un-signed, so OSX will complain about Mu coming from an unknown developer. However, for full releases we sign the .app with our Apple developer key (a manual process).

6.11.5 Linux Packages

We don't automatically create packages for Linux distros. However, we liaise with upstream developers to ensure that Mu finds its way into both Debian and Fedora based distributions.

Debian

Mu (and the MicroPython runtime) were packaged for Debian and Ubuntu by Nick Morrott and have been available to install since the releases of Debian 10 "buster" and Ubuntu 19.04 "Disco Dingo".

To install Mu on Debian/Ubuntu from the command line, type:

```
$ sudo apt install mu-editor
```

Warning: Since Mu for Debian/Ubuntu is packaged by a third party, our latest releases may not be immediately available.

Fedora

Mu was packaged by Kushal Das for Fedora. However this is an old version of Mu and, as with the Raspberry Pi version, relies on a third party to package it so may lag behind the latest version.

Note: Last, but not least, Kushal does a huge amount of work for both the Fedora and Python communities and is passionate about sustaining our Python community through education outreach. With people like Kushal putting in the time and effort to package tools like Mu and mentor beginner programmers who use Mu our community would flourish less. If you find Mu useful, please don't hesitate to thank Kushal via Twitter.

Thank you Kushal.

6.12 Developing Mu's Website

The purpose of Mu's main website https://codewith.mu/ is to provide four things:

- Instructions for getting Mu.
- Learning oriented tutorials to show users how to get started with Mu.
- Goal oriented "how-to" guides that show how to solve a specific problems or achieve particular tasks.
- Links to other community-related resources such as the developer documentation you're reading right now, and online community discussions.

The site itself is hosted for free on GitHub Pages as a Jekyll created static site. The source code is found in the mu-editor.github.io repository. As soon as a new change lands in the master branch of the site's repository, GitHub automatically rebuilds the site and deploys it. This means everything is simple and automated.

We expect everyone participating in the development of the website to act in accordance with the PSF's *Code of Conduct*.

6.12.1 Developer Setup

1. Follow the instructions for your operating system to install the Jekyll static site generator.
2. Get the source code from GitHub:

```
git clone https://github.com/mu-editor/mu-editor.github.io.git
```

3. From within the root directory of the website's source code, use Jekyll to build and serve the site locally:

```
jekyll serve
```

4. Point your browser to http://127.0.0.1:4000 to see the locally running version.

As you make changes to the website's source, Jekyll will automatically update the locally running version so you'll immediately see your updates.

Warning: If the instructions above don't work, and since Jekyll isn't supported for all environments, a Vagrant image can be used for instead. Assuming you have Vagrant installed:

```
git clone https://github.com/lcreid/rails-5-jade.git
cd rails-5-jade
vagrant up
vagrant ssh
git clone https://github.com/mu-editor/mu-editor.github.io.git
cd mu-editor.github.io
bundle install
jekyll serve --host 0.0.0.0 --force_polling
```

You may need to restart your VM to ensure the port forwarding works properly.

The source code is arranged as a typical Jekyll website except it's not a blog, so there are no articles in the `_posts` directory.

Since we need our website to be easily translatable all the content will be in a directory named after the ISO language code of the translation. For example, all the original English content is in the `en` directory in the root of the repository.

All images should be in the `img` directory. If an image is for a specific translation of the website, it should be in a subdirectory of `img` which is named after the ISO language code (for example, as there is for `img/en`).

We use GIF based screen captures throughout the site (such as on the front page). The dimensions for such captures of Mu are 1140x660 pixels and must not include the window title bar (provided by the operating system). So far, we have found the peek utility on Linux an excellent choice for making such GIF based screen captures.

When adding such animated screen grabs please ensure the `img` element has the following classes (for the sake of visual consistency): `img-responsive center-block img-rounded movie`.

6.12.2 Internationalisation of the Website

There are two ways to contribute to the translation of Mu's website:

- Add / update existing content for your target language.
- Start a completely new translation for your target language.

When adding content to an existing translation of the website please remember that files can be either HTML or Markdown. At the top of each file is a YAML based header that must contain three entries: `layout` which must always be `default`, `title` which should be the title of the page you're creating and `i18n` which much be the ISO language code for your translation (this is used so the correctly translated version of the site's menu is displayed).

For example, the YAML header for the `index.html` site in the `en` sub-directory looks like this:

```
---
layout: default
title: Code With Mu
i18n: en
---
```

The workflow for creating a new translation of the website is:

1. Create a new directory named after the ISO language code for the new translation. For example, if we were creating a new French translation of the site, we'd create a `fr` directory in the root of the repository.
2. Ensure there's a version of the `index.html` file found in the root of the repository, translated into the target language in the new directory you created in step 1. Also ensure you copy the structure of the main sections of the website found in the `en` version of the site.
3. In the `_includes` directory found in the root of the repository, you must add the new language as a list item in the `lang_list.html` template. Ensure that the href for the link points to the new directory, and the name of the translation is in the target language. For example, this is how an entry for French would look (note the use of the French word for "French"):

   ```
   <li><a href="/fr/">Français</a></li>
   ```

4. In the same `_includes` directory, create a copy of the `nav_en.html` but with the `en` section of the name replaced with the ISO code for the new target language. For example, if we were to do this for a French translation, our new file would be called `nav_fr.html`. This file defines how the site's navigation bar should look. Make sure you translate the English version into your target language and remember to update the href values to use the new directory created in step 1.
5. Remember that the YAML headers for your new translation should have an `i18n` value with the expected ISO language code for the new target language. For example, if we were writing a new page for the French translation, the `i18n` entry would have the value `fr`.

Assuming you followed all the steps above, you should see your new language in the "language" dropdown in the site navigation. Clicking on it should take you to the `index.html` page in the new directory you created for the target language, and the site navigation should reflect the newly translated navigation template.

From this point on, it's just a case of adding content to the newly translated version of the site in much the same way as it is done in the "default" `en` directory.

6.13 Mu API Reference

This API reference is automatically generated from the docstrings found within the source code. It's meant as an easy to use and easy to share window into the code base.

Take a look around! The code is simple and short.

6.13.1 mu.app

The Mu application is created and configured in this module.

Mu - a "micro" Python editor for beginner programmers.

Copyright (c) 2015-2017 Nicholas H.Tollervey and others (see the AUTHORS file).

Based upon work done for Puppy IDE by Dan Pope, Nicholas Tollervey and Damien George.

This program is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program. If not, see <http://www.gnu.org/licenses/>.

class mu.app.**AnimatedSplash**(*animation*, *parent=None*)

An animated splash screen for gifs. Includes a text area for logging output.

draw_log(*text*)

Draw the log entries onto the splash screen. Will only display the last self.log_lines number of log entries. The logs will be displayed at the bottom of the splash screen, justified left.

draw_text(*text*)

Draw text into splash screen.

failed(*text*)

Something has gone wrong during start-up, so signal this, display a helpful message along with instructions for what to do.

set_frame()

Update the splash screen with the next frame of the animation.

class mu.app.**StartupWorker**

A worker class for running blocking tasks on a separate thread during application start-up.

The animated splash screen will be shown until this thread is finished.

run()

Blocking and long running tasks for application startup should be called from here.

mu.app.**excepthook**(**exc_args*)
Log exception and exit cleanly.

mu.app.**is_linux_wayland**()
Checks environmental variables to try to determine if Mu is running on wayland.

mu.app.**run**()
Creates all the top-level assets for the application, sets things up and then runs the application. Specific tasks include:

- set up logging
- create an application object
- create an editor window and status bar
- display a splash screen while starting
- close the splash screen after startup timer ends

mu.app.**setup_logging**()
Configure logging.

mu.app.**setup_modes**(*editor*, *view*)
Create a simple dictionary to hold instances of the available modes.

PREMATURE OPTIMIZATION ALERT This may become more complex in future so splitting things out here to contain the mess. ;-)

6.13.2 mu.logic

Most of the fundamental logic for Mu is in this module.

Copyright (c) 2015-2017 Nicholas H.Tollervey and others (see the AUTHORS file).

Based upon work done for Puppy IDE by Dan Pope, Nicholas Tollervey and Damien George.

This program is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program. If not, see <http://www.gnu.org/licenses/>.

class mu.logic.**Device**(*vid*, *pid*, *port*, *serial_number*, *manufacturer*, *long_mode_name*, *short_mode_name*, *board_name=None*)
Device object, containing both information about the connected device, the port it's connected through and the mode it works with.

property name
Returns the device name.

class mu.logic.**DeviceList**(*modes*, *parent=None*)

add_device(*new_device*)
Add a new device to the device list, maintains alphabetical ordering

check_usb()
Ensure connected USB devices are polled. If there's a change and a new recognised device is attached, inform the user via a status message. If a single device is found and Mu is in a different mode ask the user if they'd like to change mode.

data(*index*, *role*)
Reimplements QAbstractListModel.data(): returns data for the specified index and role. In this case only implmented for ToolTipRole and DisplayRole

remove_device(*device*)
Remove the given device from the device list

rowCount(*parent*)
Number of devices

class mu.logic.**Editor**(*view*)
Application logic for the editor itself.

ask_to_change_mode(*new_mode*, *mode_name*, *heading*)
Open a dialog asking the user, whether to change mode from mode_name to new_mode. The dialog can be customized by the heading-parameter.

autosave()
Cycles through each tab and, if changed, saves it to the filesystem.

change_mode(*mode*)
Given the name of a mode, will make the necessary changes to put the editor into the new mode.

check_code()
Uses PyFlakes and PyCodeStyle to gather information about potential problems with the code in the current tab.

check_for_shadow_module(*path*)
Check if the filename in the path is a shadow of a module already in the Python path. For example, many learners will save their first turtle based script as turtle.py, thus causing Python to never find the built in turtle module because of the name conflict.

If the filename shadows an existing module, return True, otherwise, return False.

connect_to_status_bar(*status_bar*)
Connect the editor with the Window-statusbar. Should be called after Editor.setup(), to ensure modes are initialized

debug_toggle_breakpoint(*margin*, *line*, *modifiers*)
How to handle the toggling of a breakpoint.

device_changed(*device*)
Slot for receiving signals that the current device has changed. If the device change requires mode change, the user will be asked through a dialog.

direct_load(*path*)
For loading files passed from command line or the OS launch.

find_again(*forward=True*)
Handle find again (F3 and Shift+F3) functionality.

find_again_backward(*forward=False*)
Handle find again backward (Shift+F3) functionality.

find_replace()
Handle find / replace functionality.

If find/replace dialog is dismissed, do nothing.

Otherwise, check there's something to find, warn if there isn't.

If there is, find (and, optionally, replace) then confirm outcome with a status message.

get_dialog_directory(*default=None*)
Return the directory folder which a load/save dialog box should open into. In order of precedence this function will return:

0) If not None, the value of default.

1) The last location used by a load/save dialog.

2) The directory containing the current file.

3) The mode's reported workspace directory.

get_tab(*path*)
Given a path, returns either an existing tab for the path or creates / loads a new tab for the path.

has_python_extension(*filename*)
Check whether the given filename matches recognized Python extensions.

load(**args*, *default_path=None*)
Loads a Python (or other supported) file from the file system or extracts a Python script from a hex file.

load_cli(*paths*)
Given a set of paths, passed in by the user when Mu starts, this method will attempt to load them and log / report a problem if Mu is unable to open a passed in path.

new()
Adds a new tab to the editor.

quit(**args*, ***kwargs*)
Exit the application.

rename_tab(*tab_id=None*)
How to handle double-clicking a tab in order to rename the file. If activated by the shortcut, activate against the current tab.

restore_session(*paths=None*)
Attempts to recreate the tab state from the last time the editor was run. If paths contains a collection of additional paths specified by the user, they are also "restored" at the same time (duplicates will be ignored).

save(**args*, *default=None*)
Save the content of the currently active editor tab.

save_tab_to_file(*tab*, *show_error_messages=True*)
Given a tab, will attempt to save the script in the tab to the path associated with the tab. If there's a problem this will be logged and reported and the tab status will continue to show as Modified.

select_mode(*event=None*)
Select the mode that editor is supposed to be in.

setup(*modes*)
Define the available modes and ensure there's a default working directory.

show_admin(*event=None*)
Cause the editor's admin dialog to be displayed to the user.

Ensure any changes to the envars is updated.

show_help()
Display browser based help about Mu.

show_status_message(*message*, *duration=5*)
Displays the referenced message for duration seconds.

sync_package_state(*old_packages*, *new_packages*)
Given the state of the old third party packages, compared to the new third party packages, ensure that pip uninstalls and installs the packages so the currently available third party packages reflects the new state.

tidy_code()
Prettify code with Black.

toggle_comments()
Ensure all highlighted lines are toggled between comments/uncommented.

toggle_theme()
Switches between themes (night, day or high-contrast).

zoom_in()
Make the editor's text bigger

zoom_out()
Make the editor's text smaller.

class mu.logic.**MuFlakeCodeReporter**
The class instantiates a reporter that creates structured data about code quality for Mu. Used by the PyFlakes module.

flake(*message*)
PyFlakes found something wrong with the code.

syntaxError(*filename*, *message*, *line_no*, *column*, *source*)
Records a syntax error in the file called filename.

The message argument contains an explanation of the syntax error, line_no indicates the line where the syntax error occurred, column indicates the column on which the error occurred and source is the source code containing the syntax error.

unexpectedError(*filename*, *message*)
Called if an unexpected error occured while trying to process the file called filename. The message parameter contains a description of the problem.

mu.logic.**check_flake**(*filename*, *code*, *builtins=None*)
Given a filename and some code to be checked, uses the PyFlakesmodule to return a dictionary describing issues of code quality per line. See:

https://github.com/PyCQA/pyflakes

If a list symbols is passed in as "builtins" these are assumed to be additional builtins available when run by Mu.

mu.logic.**check_pycodestyle**(*code*, *config_file=False*)
Given some code, uses the PyCodeStyle module (was PEP8) to return a list of items describing issues of coding style. See:

https://pycodestyle.readthedocs.io/en/latest/intro.html

mu.logic.**extract_envars**(*raw*)
Returns a list of environment variables given a string containing NAME=VALUE definitions on separate lines.

mu.logic.**read_and_decode**(*filepath*)
Read the contents of a file,

mu.logic.**save_and_encode**(*text*, *filepath*, *newline='\n'*)
Detect the presence of an encoding cookie and use that encoding; if none is present, do not add one and use the Mu default encoding. If the codec is invalid, log a warning and fall back to the default.

mu.logic.**sniff_encoding**(*filepath*)
Determine the encoding of a file:

- If there is a BOM, return the appropriate encoding
- If there is a PEP 263 encoding cookie, return the appropriate encoding
- Otherwise return None for read_and_decode to attempt several defaults

mu.logic.**sniff_newline_convention**(*text*)
Determine which line-ending convention predominates in the text.

Windows usually has U+000D U+000A Posix usually has U+000A But editors can produce either convention from either platform. And a file which has been copied and edited around might even have both!

mu.logic.**write_and_flush**(*fileobj*, *content*)
Write content to the fileobj then flush and fsync to ensure the data is, in fact, written.

This is especially necessary for USB-attached devices

6.13.3 mu.debugger

The debugger consists of two parts:

- Client - used by Mu to process messages from the process being debugged.
- Runner - created in a new process to run the code to be debugged.

Messages are passed via inter-process communication.

mu.debugger.client

Code used by the Mu application to communicate with the process being debugged.

A debug client for the Mu editor.

class mu.debugger.client.**Breakpoint**(*bpnum*, *filename*, *line*, *enabled=True*, *temporary=False*, *funcname=None*)
Represents a breakpoint, identified by a breakpoint number (bpnum). Users set breakpoints to stop the debugger at a certain line (potentially in a named function) in a file.

class mu.debugger.client.**CommandBufferHandler**(*debugger*)
Represents the work to be done on a separate thread for connecting and processing incoming messages.

Emits signals to indicate when messages are receievd or the connection fails at appropriate moments during the lifetime of a debug session.

on_command
Signal emitted when a command is received.

on_fail
Emitted when there was a connection failure.

worker()
Buffer input from a socket, emit complete debugger commands as signals.

exception mu.debugger.client.**ConnectionNotBootstrapped**
The connection to the runner hasn't been completed.

class mu.debugger.client.**Debugger**(*host*, *port*, *proc=None*)
Represents the networked debugger client.

breakpoint(*breakpoint*)
Given a breakpoint number or (filename, line), return an object representing the referenced breakpoint.

breakpoints(*filename*)
Return all the breakpoints associated with the referenced file.

clear_breakpoint(*breakpoint*)
Clear an existing breakpoint.

create_breakpoint(*filename*, *line*, *temporary=False*)
Create a new, enabled breakpoint at the specified line of the given file.

disable_breakpoint(*breakpoint*)
Disable an existing breakpoint.

do_next()
Go to the next line in the current stack frame.

do_return()
Return to the previous stack frame.

do_run()
Run the debugger until the next breakpoint.

do_step()
Step through one stack frame.

enable_breakpoint(*breakpoint*)
Enable an existing breakpoint.

ignore_breakpoint(*breakpoint*, *count*)
Ignore an existing breakpoint for "count" iterations.

(N.B. Use a count of 0 to restore the breakpoint.

on_bootstrap(*breakpoints*)
The runner has finished setting up.

on_breakpoint_clear(*bpnum*)
The runner has cleared the referenced breakpoint.

on_breakpoint_create(***bp_data*)
The runner has created a breakpoint.

on_breakpoint_disable(*bpnum*)
The runner has disabled a breakpoint referenced by breakpoint number.

on_breakpoint_enable(*bpnum*)
The runner has enabled the breakpoint referenced by breakpoint number.

on_breakpoint_ignore(*bpnum*, *count*)
The runner will ignore the referenced breakpoint "count" iterations.

on_call(*args*)
The runner has called a function with the specified arguments.

on_command(*command*)
Handle a command emitted by the client thread.

on_error(*message*)
The runner has sent an error message.

on_exception(*name*, *value*)
The runner has encountered a named exception with an associated value.

on_fail(*message*)
Handle if there's a connection failure with the debug runner.

on_finished()
The debug runner has finished running the script to be debugged.

on_info(*message*)
The runner has sent an informative message.

on_line(*filename*, *line*)
The runner has moved to the specified line in the referenced file.

on_postmortem(**args*, ***kwargs*)
The runner encountered a fatal error and has died.

on_restart()
The runner has restarted.

on_return(*retval*)
The runner has returned from a function with the specified return value.

on_stack(*stack*)
The runner has sent an update to the stack.

on_warning(*message*)
The runner has sent a warning message.

output(*event*, ***data*)
Send a command to the debug runner.

start()
Start the debugger session.

stop()
Shut down the debugger session.

exception mu.debugger.client.**UnknownBreakpoint**
The client encountered an unknown breakpoint.

mu.debugger.runner

The runner code controls the debug process.

A debug runner for the Mu editor.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program. If not, see <http://www.gnu.org/licenses/>.

exception mu.debugger.runner.**ClientClose**
Cause the debugger to wait for a new client to connect.

class mu.debugger.runner.**DebugState**(*value*)
Enumerates the three possible states of a debugging session.

class mu.debugger.runner.**Debugger**(*socket*, *host*, *port*, *skip=None*)
Instances of this class represent and drive the debugging process.

do_break(*filename*, *line*, *temporary=False*)
Set a breakpoint.

do_clear(*bpnum*)
Handle how a breakpoint must be removed when it is a temporary one.

do_close()
Respond to a closed socket (not a user commend, but needs handling).

do_continue()
Stop only at breakpoints or when finished. If there are no breakpoints on script start, do a set_trace to stop at the first available line. However, use the continue_flag to ensure set_continue is always called thereafter.

do_disable(*bpnum*)
Disable the breakpoint referenced by its breakpoint number (bpnum).

do_enable(*bpnum*)
Enables the breakpoint referenced by its breakpoint number (bpnum).

do_ignore(*bpnum*, *count*)
Ignore the breakpoint referenced by its breakpoint number (bpnum), count number of times.

do_next()
Stop on the next line in or below the given frame.

do_quit()
Set the quitting attribute to True. This raises BdbQuit in the next call to one of the dispatch_*() methods.

do_restart()
Restart the program by raising an exception to be caught by the debugger.

do_return()
Stop when returning from the current frame.

do_step()
Stop after one line of code.

interact(*frame*, *traceback*)
Contains the loop processing interactions with the debugger.

output(*event*, ***data*)
Dumps data related to a referenced event to the socket.

output_stack()
Dump the current stack.

If this is a normal situation, the top two frames are BDB and the runner executing the program. If there is an exception, there are two further extra frames. All these frames can be ignored.

reset()
Reset state.

setup(*frame*, *traceback*)
Start state should be set correctly.

user_call(*frame*, *argument_list*)
This method is called from dispatch_call() when there is the possibility that a break might be necessary anywhere inside the called function.

user_exception(*frame*, *exc_info*)
This method is called from dispatch_exception() when stop_here() yields True.

For when an exception occurs, but only if we are to stop at or just below this level.

user_line(*frame*)
This method is called from dispatch_line() when either stop_here() or break_here() yields True.

For when we stop or break at this line.

user_return(*frame*, *return_value*)
This method is called from dispatch_return() when stop_here() yields True.

For when a return trap is set here.

exception mu.debugger.runner.**Restart**
Cause the debugger to restart for the target Python program.

mu.debugger.runner.**command_buffer**(*debugger*)
Buffer input from a socket, yield complete debugger commands.

mu.debugger.runner.**run**(*hostname*, *port*, *filename*, *args*)
Run a Python script identified by "filename" with the specified arguments in a debugger session that's listening at hostname/port.

6.13.4 mu.interface

This module contains all the PyQt related code needed to create the user interface for Mu. All interaction with the user interface is done via the `Window` class in `mu.interface.main`.

All the other sub-modules contain different bespoke aspects of the user interface.

mu.interface.main

Contains the core user interface assets used by other parts of the application.

Contains the main Window definition for Mu's UI.

class mu.interface.main.**ButtonBar**(*parent*)
Represents the bar of buttons across the top of the editor and defines their behaviour.

addAction(*name*, *display_name*, *tool_text*)
Creates an action associated with an icon and name and adds it to the widget's slots.

connect(*name*, *handler*, *shortcut=None*)
Connects a named slot to a handler function and optional hot-key shortcuts.

reset()
Resets the button states.

set_responsive_mode(*width*, *height*)
Compact button bar for when window is very small.

class mu.interface.main.**FileTabs**
Extend the base class so we can override the removeTab behaviour.

addTab(*widget*, *title*)
Add a new tab to the switcher

change_tab(*tab_id*)
Update the application title to reflect the name of the file in the currently selected tab.

removeTab(*tab_id*)
Ask the user before closing the file.

class mu.interface.main.**StatusBar**(*parent=None*, *mode='python'*)
Defines the look and behaviour of the status bar along the bottom of the UI.

connect_logs(*handler*, *shortcut*)
Connect the mouse press event and keyboard shortcut for the log widget to the referenced handler function.

connect_mode(*handler*, *shortcut*)
Connect the mouse press event and keyboard shortcut for the mode widget to the referenced handler function.

device_connected(*device*)
Show a tooltip whenever a new device connects

set_message(*message*, *pause=5000*)
Displays a message in the status bar for a certain period of time.

set_mode(*mode*)
Updates the mode label to the new mode.

class mu.interface.main.**Window**(*parent=None*)
Defines the look and characteristics of the application's main window.

add_debug_inspector()
Display a debug inspector to view the call stack.

add_filesystem(*home*, *file_manager*, *board_name='board'*)
Adds the file system pane to the application.

add_jupyter_repl(*kernel_manager*, *kernel_client*)
Adds a Jupyter based REPL pane to the application.

add_micropython_plotter(*name*, *connection*, *data_flood_handler*)
Adds a plotter that reads data from a serial connection.

add_micropython_repl(*name*, *connection*)
Adds a MicroPython based REPL pane to the application.

add_plotter(*plotter_pane*, *name*)
Adds the referenced plotter pane to the application.

add_python3_plotter(*mode*)
Add a plotter that reads from either the REPL or a running script. Since this function will only be called when either the REPL or a running script are running (but not at the same time), it'll just grab data emitted by the REPL or script via data_received.

add_python3_runner(*interpreter*, *script_name*, *working_directory*, *interactive=False*, *debugger=False*, *command_args=None*, *envars=None*, *python_args=None*)
Display console output for the interpreter with the referenced pythonpath running the referenced script.

The script will be run within the workspace_path directory.

If interactive is True (default is False) the Python process will run in interactive mode (dropping the user into the REPL when the script completes).

If debugger is True (default is False) the script will be run within a debug runner session. The debugger overrides the interactive flag (you cannot run the debugger in interactive mode).

If there is a list of command_args (the default is None) then these will be passed as further arguments into the command run in the new process.

If envars is given, these will become part of the environment context of the new chlid process.

If python_args is given, these will be passed as arguments to the Python runtime used to launch the child process.

add_repl(*repl_pane*, *name*)
Adds the referenced REPL pane to the application.

add_snek_repl(*name*, *connection*, *force_interrupt=True*, *wait_input=False*)
Adds a Snek based REPL pane to the application.

add_tab(*path*, *text*, *api*, *newline*)
Adds a tab with the referenced path and text to the editor.

annotate_code(*feedback*, *annotation_type*)
Given a list of annotations about the code in the current tab, add the annotations to the editor window so the user can make appropriate changes.

change_mode(*mode*)
Given a an object representing a mode, recreates the button bar with the expected functionality.

connect_find_again(*handlers*, *shortcut*)
Create keyboard shortcuts and associate them with handlers for doing a find again in forward or backward direction. Any given shortcut will be used for forward find again, while Shift+shortcut will find again backwards.

connect_find_replace(*handler*, *shortcut*)
Create a keyboard shortcut and associate it with a handler for doing a find and replace.

connect_tab_rename(*handler*, *shortcut*)
Connect the double-click event on a tab and the keyboard shortcut to the referenced handler (causing the Save As dialog).

connect_toggle_comments(*handler*, *shortcut*)
Create a keyboard shortcut and associate it with a handler for toggling comments on highlighted lines.

connect_zoom(*widget*)
Connects a referenced widget to the zoom related signals and sets the zoom of the widget to the current zoom level.

copy_to_repl()
Copies currently selected text in the editor, into the active REPL widget and sets focus to the REPL widget. The final line pasted into the REPL waits for RETURN to be pressed by the user (this appears to be the default behaviour for pasting into the REPL widget).

property current_tab
Returns the currently focussed tab.

focus_tab(*tab*)
Force focus on the referenced tab.

get_load_path(*folder*, *extensions='*'*, *allow_previous=True*)
Displays a dialog for selecting a file to load. Returns the selected path. Defaults to start in the referenced folder unless a previous folder has been used and the allow_previous flag is True (the default behaviour)

get_microbit_path(*folder*)
Displays a dialog for locating the location of the BBC micro:bit in the host computer's filesystem. Returns the selected path. Defaults to start in the referenced folder.

get_save_path(*folder*)
Displays a dialog for selecting a file to save. Returns the selected path. Defaults to start in the referenced folder.

handle_python_anywhere_complete(*domain*)
Displays a confirmation that all the API calls completed OK and provides a link to the user's website.

handle_python_anywhere_error(*error_message*)
Display a friendly message to indicate a problem was encountered when uploading to PythonAnywhere.

hide_device_selector()
Hides the device selector in the status bar

highlight_text(*target_text*, *forward=True*)
Highlight the first match from the current position of the cursor in the current tab for the target_text. Returns True if there's a match.

property modified
Returns a boolean indication if there are any modified tabs in the editor.

on_context_menu()
Called when a user right-clicks on an editor pane.

If the REPL is active AND there is selected text in the current editor pane, modify the default context menu to include a paste to REPL option. Otherwise, just display the default context menu.

on_stdout_write(*data*)
Called when either a running script or the REPL write to STDOUT.

open_directory_from_os(*path*)
Given the path to a directory, open the OS's built in filesystem explorer for that path. Works with Windows, OSX and Linux.

remove_debug_inspector()
Removes the debug inspector pane from the application.

remove_filesystem()
Removes the file system pane from the application.

remove_plotter()
Removes the plotter pane from the application.

remove_python_runner()
Removes the runner pane from the application.

remove_repl()
Removes the REPL pane from the application.

replace_text(*target_text*, *replace*, *global_replace*)
Given target_text, replace the first instance after the cursor with "replace". If global_replace is true, replace all instances of "target". Returns the number of times replacement has occurred.

reset_annotations()
Resets the state of annotations on the current tab.

resizeEvent(*resizeEvent*)
Respond to window getting too small for the button bar to fit well.

screen_size()
Returns an (width, height) tuple with the screen geometry.

select_mode(*modes*, *current_mode*)
Display the mode selector dialog and return the result.

set_checker_icon(*icon*)
Set the status icon to use on the check button

set_read_only(*is_readonly*)
Set all tabs read-only.

set_theme(*theme*)
Sets the theme for the REPL and editor tabs.

set_timer(*duration*, *callback*)
Set a repeating timer to call "callback" every "duration" seconds.

set_usb_checker(*duration*, *callback*)
Sets up a timer that polls for USB changes via the "callback" every "duration" seconds.

set_zoom()
Sets the zoom to current zoom_position level.

setup(*breakpoint_toggle*, *theme*)
Sets up the window.

Defines the various attributes of the window and defines how the user interface is laid out.

show_admin(*log*, *settings*, *packages*, *mode*, *device_list*)
Display the administrative dialog with referenced content of the log and settings. Return a dictionary of the settings that may have been changed by the admin dialog.

show_annotations()
Show the annotations added to the current tab.

show_confirmation(*message*, *information=None*, *icon=None*)
Displays a modal message to the user to which they need to confirm or cancel.

If information is passed in this will be set as the additional informative text in the modal dialog.

Since this mechanism will be used mainly for warning users that something is awry the default icon is set to "Warning". It's possible to override the icon to one of the following settings: NoIcon, Question, Information, Warning or Critical.

show_device_selector()
Reveals the device selector in the status bar

show_find_replace(*find*, *replace*, *global_replace*)
Display the find/replace dialog. If the dialog's OK button was clicked return a tuple containing the find term, replace term and global replace flag.

show_message(*message*, *information=None*, *icon=None*)
Displays a modal message to the user.

If information is passed in this will be set as the additional informative text in the modal dialog.

Since this mechanism will be used mainly for warning users that something is awry the default icon is set to "Warning". It's possible to override the icon to one of the following settings: NoIcon, Question, Information, Warning or Critical.

size_window(*x=None*, *y=None*, *w=None*, *h=None*)
Makes the editor 80% of the width*height of the screen and centres it when none of x, y, w and h is passed in; otherwise uses the passed in values to position and size the editor window.

If the X or Y value will be off the screen, these are reset to None (thus stopping the window being drawn in a hard-to-reach place). See issue #1613 for context.

stop_timer()
Stop the repeating timer.

sync_packages(*to_remove*, *to_add*)
Display a modal dialog that indicates the status of the add/remove package management operation.

property tab_count
Returns the number of active tabs.

toggle_comments()
Toggle comments on/off for all selected line in the currently active tab.

update_debug_inspector(*locals_dict*)
Given the contents of a dict representation of the locals in the current stack frame, update the debug inspector with the new values.

update_title(*filename=None*)
Updates the title bar of the application. If a filename (representing the name of the file currently the focus of the editor) is supplied, append it to the end of the title.

upload_to_python_anywhere(*instance*, *username*, *token*, *app_name*, *files*)
Show a progress dialog as the files are uploaded to PythonAnywhere.

wheelEvent(*event*)
Trap a CTRL-scroll event so the user is able to zoom in and out.

property widgets
Returns a list of references to the widgets representing tabs in the editor.

zoom_in()
Handles zooming in.

zoom_out()
Handles zooming out.

mu.interface.dialogs

Bespoke modal dialogs required by Mu.

UI related code for dialogs used by Mu.

class mu.interface.dialogs.**AdminDialog**(*parent=None*)

Displays administrative related information and settings (logs, environment variables, third party packages etc…).

settings()

Return a dictionary representation of the raw settings information generated by this dialog. Such settings will need to be processed / checked in the "logic" layer of Mu.

class mu.interface.dialogs.**ESPFirmwareFlasherWidget**

Used for configuring how to interact with the ESP:

- Override MicroPython.

append_data(*msg*)

Add data to the end of the text area.

esptool_finished(*exitCode*, *exitStatus*)

Called when the subprocess that executes 'esptool.py is finished.

read_process()

Read data from the child process and append it to the text area. Try to keep reading until there's no more data from the process.

class mu.interface.dialogs.**EnvironmentVariablesWidget**

Used for editing and displaying environment variables used with Python 3 mode.

class mu.interface.dialogs.**FindReplaceDialog**(*parent=None*)

Display a dialog for getting:

- A term to find,
- An optional value to replace the search term,
- A flag to indicate if the user wishes to replace all.

find()

Return the value the user entered to find.

replace()

Return the value the user entered for replace.

replace_flag()

Return the value of the global replace flag.

class mu.interface.dialogs.**LocaleWidget**
Used for manually setting the locale (and thus the language) used by Mu.

get_locale()
Return the user-selected language code.

class mu.interface.dialogs.**LogWidget**
Used to display Mu's logs.

class mu.interface.dialogs.**MicrobitSettingsWidget**
Used for configuring how to interact with the micro:bit:

- Minification flag.
- Override runtime version to use.

class mu.interface.dialogs.**ModeItem**(*name*, *description*, *icon*, *parent=None*)
Represents an available mode listed for selection.

class mu.interface.dialogs.**ModeSelector**(*parent=None*)
Defines a UI for selecting the mode for Mu.

get_mode()
Return details of the newly selected mode.

select_and_accept()
Handler for when an item is double-clicked.

class mu.interface.dialogs.**PackageDialog**(*parent=None*)
Display the output of the pip commands needed to remove or install packages.

Because the QProcess mechanism we're using is asynchronous, we have to manage the pip requests via *pip_queue*. When one request is signalled as finished we start the next.

finish()
Set the UI to a valid end state.

next_pip_command()
Run the next pip command, finishing if there is none.

run_pip(*command*, *packages*)
Run a pip command in a subprocess and pipe the output to the dialog's text area.

setup(*to_remove*, *to_add*)
Create the UI for the dialog.

class mu.interface.dialogs.**PackagesWidget**
Used for editing and displaying 3rd party packages installed via pip to be used with Python 3 mode.

class mu.interface.dialogs.**PythonAnywhereWidget**
For configuring the user's username and API token for interacting with the PythonAnywhere API to deploy a website from web mode.

valid_instances = ['www', 'eu']
Valid server hosting instances for PythonAnywhere.

mu.interface.editor

Contains the customised Scintilla based editor used for textual display and entry.

UI related capabilities for the text editor widget embedded in each tab in Mu.

class mu.interface.editor.**CssLexer**

Fixes problems with comments in CSS.

description(*style*)

Ensures "Comment" is returned when the lexer encounters a comment (this is due to a bug in the base class, for which this is a work around).

class mu.interface.editor.**EditorPane**(*path*, *text*, *newline='\n'*)

Represents the text editor.

annotate_code(*feedback*, *annotation_type='error'*)

Given a list of annotations add them to the editor pane so the user can act upon them.

configure()

Set up the editor component.

connect_margin(*func*)

Connect clicking the margin to the passed in handler function, via a filtering handler that ignores clicks on margin 4.

contextMenuEvent(*event*)

A context menu (right click) has been actioned.

debugger_at_line(*line*)

Set the line to be highlighted with the DEBUG_INDICATOR.

dropEvent(*event*)

Run by Qt when *something* is dropped on this editor

find_next_match(*text*, *from_line=- 1*, *from_col=- 1*, *case_sensitive=True*, *wrap_around=True*)

Finds the next text match from the current cursor, or the given position, and selects it (the automatic selection is the only available QsciScintilla behaviour). Returns True if match found, False otherwise.

highlight_selected_matches()

Checks the current selection, if it is a single word it then searches and highlights all matches.

Since we're interested in exactly one word: * Ignore an empty selection * Ignore anything which spans more than one line * Ignore more than one word * Ignore anything less than one word

property label

The label associated with this editor widget (usually the filename of the script we're editing).

range_from_positions(*start_position*, *end_position*)
Given a start-end pair, such as are provided by a regex match, return the corresponding Scintilla line-offset pairs which are used for searches, indicators etc.

NOTE: Arguments must be byte offsets into the underlying text bytes.

reset_annotations()
Clears all the assets (indicators, annotations and markers).

reset_check_indicators()
Clears all the text indicators related to the check code functionality.

reset_debugger_highlight()
Reset all the lines so the DEBUG_INDICATOR is no longer displayed.

We need to check each line since there's no way to tell what the currently highlighted line is. This approach also has the advantage of resetting the *whole* editor pane.

reset_search_indicators()
Clears all the text indicators from the search functionality.

selection_change_listener()
Runs every time the text selection changes. This could get triggered multiple times while the mouse click is down, even if selection has not changed in itself. If there is a new selection is passes control to highlight_selected_matches.

set_api(*api_definitions*)
Sets the API entries for tooltips, calltips and the like.

set_theme(*theme=<class 'mu.interface.themes.DayTheme'>*)
Connect the theme to a lexer and return the lexer for the editor to apply to the script text.

set_zoom(*size='m'*)
Sets the font zoom to the specified base point size for all fonts given a t-shirt size.

show_annotations()
Display all the messages to be annotated to the code.

property title
The title associated with this editor widget (usually the filename of the script we're editing).

If the script has been modified since it was last saved, the label will end with an asterisk.

toggle_comments()
Iterate through the selected lines and toggle their comment/uncomment state. So, lines that are not comments become comments and vice versa.

toggle_line(*raw_line*)
Given a raw_line, will return the toggled version of it.

wheelEvent(*event*)
Stops QScintilla from doing the wrong sort of zoom handling.

class mu.interface.editor.**PythonLexer**(**args*, ***kwargs*)
A Python specific "lexer" that's used to identify keywords of the Python language so the editor can do syntax highlighting.

keywords(*flag*)
Returns a list of Python keywords.

mu.interface.panes

Contains code used to populate the various panes found in the user interface (REPL, file list, debug inspector etc...).

Contains the UI classes used to populate the various panes used by Mu.

class mu.interface.panes.**DebugInspector**

Presents a tree like representation of the current state of the call stack to the user.

record_collapsed(*index*)

Remove collapsed dicts from set, so they render collapsed.

record_expanded(*index*)

Keep track of expanded dicts for displaying in debugger.

set_font_size(*new_size=14*)

Sets the font size for all the textual elements in this pane.

set_zoom(*size*)

Set the current zoom level given the "t-shirt" size.

class mu.interface.panes.**DebugInspectorItem**(**args*)

class mu.interface.panes.**FileSystemPane**(*home*)

Contains two QListWidgets representing the micro:bit and the user's code directory. Users transfer files by dragging and dropping. Highlighted files can be selected for deletion.

disable()

Stops interaction with the list widgets.

enable()

Allows interaction with the list widgets.

on_delete_fail(*filename*)

Fired when a deletion on the device for the given file failed.

on_get_fail(*filename*)

Fired when getting the referenced file on the device failed.

on_ls(*microbit_files*)

Displays a list of the files on the micro:bit.

Since listing files is always the final event in any interaction between Mu and the micro:bit, this enables the controls again for further interactions to take place.

on_ls_fail()

Fired when listing files fails.

on_put_fail(*filename*)

Fired when the referenced file cannot be copied onto the device.

set_font_size(*new_size=14*)
Sets the font size for all the textual elements in this pane.

set_zoom(*size*)
Set the current zoom level given the "t-shirt" size.

show_message(*message*)
Emits the set_message signal.

show_warning(*message*)
Emits the set_warning signal.

class mu.interface.panes.**JupyterREPLPane**(**args*, ***kwargs*)
REPL = Read, Evaluate, Print, Loop.

Displays a Jupyter iPython session.

setFocus()
Override base setFocus so the focus happens to the embedded _control within this widget.

set_font_size(*new_size=14*)
Sets the font size for all the textual elements in this pane.

set_theme(*theme*)
Sets the theme / look for the REPL pane.

set_zoom(*size*)
Set the current zoom level given the "t-shirt" size.

class mu.interface.panes.**LocalFileList**(*home*)
Represents a list of files in the Mu directory on the local machine.

contextMenuEvent(*self*, *QContextMenuEvent*)

dropEvent(*self*, *QDropEvent*)

on_get(*microbit_file*)
Fired when the get event is completed for the given filename.

class mu.interface.panes.**MicroPythonDeviceFileList**(*home*)
Represents a list of files on a MicroPython device.

contextMenuEvent(*self*, *QContextMenuEvent*)

dropEvent(*self*, *QDropEvent*)

on_delete(*microbit_file*)
Fired when the delete event is completed for the given filename.

on_put(*microbit_file*)
Fired when the put event is completed for the given filename.

class mu.interface.panes.**MicroPythonREPLPane**(*connection*, *theme='day'*, *parent=None*)
REPL = Read, Evaluate, Print, Loop.

This widget represents a REPL client connected to a device running MicroPython.

The device MUST be flashed with MicroPython for this to work.

clear()
Clears the text of the REPL.

context_menu()
Creates custom context menu with just copy and paste.

delete_selection()
Returns true if deletion happened, returns false if there was no selection to delete.

insertFromMimeData(*source*)
Insert mime data by sending it to the REPL

keyPressEvent(*data*)
Called when the user types something in the REPL.

Correctly encodes it and sends it to the connected device.

mouseReleaseEvent(*mouseEvent*)
Called whenever a user have had a mouse button pressed, and releases it. We pass it through to the normal way Qt handles button pressed, but also sends as cursor movement signal to the device (except if a selection is made, for selections we first move the cursor on deselection)

move_cursor_to(*new_position*)
Move the cursor, by sending vt100 left/right signals through serial. The Qt cursor is first returned to the known location of the device cursor. Then the appropriate number of move left or right signals are send. The Qt cursor is not moved to the new_position here, but will be moved once receiving a response (in process_tty_data).

process_tty_data(*data*)
Given some incoming bytes of data, work out how to handle / display them in the REPL widget. If received input is incomplete, stores remainder in self.unprocessed_input.

Updates the self.device_cursor_position to match that of the device for every input received.

set_devicecursor_to_qtcursor()
Call this whenever the cursor has been moved by the user, to send the cursor movement to the device.

set_font_size(*new_size=14*)
Sets the font size for all the textual elements in this pane.

set_qtcursor_to_devicecursor()
Resets the Qt TextCursor to where we know the device has the cursor placed.

set_zoom(*size*)
Set the current zoom level given the "t-shirt" size.

class mu.interface.panes.**MuFileList**
Contains shared methods for the two types of file listing used in Mu.

show_confirm_overwrite_dialog()
Display a dialog to check if an existing file should be overwritten.

Returns a boolean indication of the user's decision.

class mu.interface.panes.**PlotterPane**(*parent=None*)
This plotter widget makes viewing sensor data easy!

This widget represents a chart that will look for tuple data from the MicroPython REPL, Python 3 REPL or Python 3 code runner and will auto-generate a graph.

add_data(*values*)
Given a tuple of values, ensures there are the required number of line series, add the data to the line series, update the range of the chart so the chart displays nicely.

process_tty_data(*data*)
Takes raw bytes and, if a valid tuple is detected, adds the data to the plotter.

The the length of the bytes data > 1024 then a data_flood signal is emitted to ensure Mu can take action to remain responsive.

set_theme(*theme*)
Sets the theme / look for the plotter pane.

class mu.interface.panes.**PythonProcessPane**(*parent=None*)
Handles / displays a Python process's stdin/out with working command history and simple buffer editing.

append(*msg*)
Append text to the text area.

backspace()
Removes a character from the current buffer – to the left of cursor.

clear_input_line()
Remove all the characters currently in the input buffer line.

context_menu()
Creates custom context menu with just copy and paste.

delete()
Removes a character from the current buffer – to the right of cursor.

finished(*code*, *status*)
Handle when the child process finishes.

history_back()
Replace the current input line with the next item BACK from the current history position.

history_forward()
Replace the current input line with the next item FORWARD from the current history position.

insert(*msg*)
Insert text to the text area at the current cursor position.

insertFromMimeData(*source*)
Insert mime data by sending it to the REPL

keyPressEvent(*data*)
Called when the user types something in the REPL.

on_process_halt()
Called when the the user has manually halted a running process. Ensures that the remaining data from the halted process's stdout is handled properly.

When the process is halted the user is dropped into the Python prompt and this method ensures the UI is updated in a clean, non-blocking way.

parse_input(*key*, *text*, *modifiers*)
Correctly encodes user input and sends it to the connected process.

The key is a Qt.Key_Something value, text is the textual representation of the input, and modifiers are the control keys (shift, CTRL, META, etc) also used.

parse_paste(*text*)
Recursively takes characters from text to be parsed as input. We do this so the event loop has time to respond to output from the process to which the characters are sent (for example, when a newline is sent).

Yes, this is a quick and dirty hack, but ensures the pasted input is also evaluated in an interactive manner rather than as a single-shot splurge of data. Essentially, it's simulating someone typing in the characters of the pasted text *really fast* but in such a way that the event loop cycles.

read_from_stdout()
Process incoming data from the process's stdout.

replace_input_line(*text*)
Replace the current input line with the passed in text.

set_font_size(*new_size=14*)
Sets the font size for all the textual elements in this pane.

set_start_of_current_line()
Set the flag to indicate the start of the current line (used before waiting for input).

This flag is used to discard the preceeding text in the text entry field when Mu parses new input from the user (i.e. any text beyond the self.start_of_current_line).

set_zoom(*size*)
Set the current zoom level given the "t-shirt" size.

start_process(*interpreter*, *script_name*, *working_directory*, *interactive=True*, *debugger=False*, *command_args=None*, *envars=None*, *python_args=None*)
Start the child Python process.

Will use the referenced interpreter to run the Python script_name within the context of the working directory.

If interactive is True (the default) the Python process will run in interactive mode (dropping the user into the REPL when the script completes).

If debugger is True (the default is False) then the script will run within a debug runner session.

If there is a list of command_args (the default is None), then these will be passed as further arguments into the script to be run.

If there is a list of environment variables, these will be part of the context of the new child process.

If python_args is given, these are passed as arguments to the Python interpreter used to launch the child process.

try_read_from_stdout()
Ensure reading from stdout only happens if there is NOT already current attempts to read from stdout.

write_to_stdin(*data*)
Writes data from the Qt application to the child process's stdin.

class mu.interface.panes.**SnekREPLPane**(*connection*, *theme='day'*, *parent=None*)
REPL = Read, Evaluate, Print, Loop.

This widget represents a REPL client connected to a device running Snek.

The device MUST be flashed with Snek for this to work.

execute(*commands*)
Execute a series of commands over a period of time (scheduling remaining commands to be run in the next iteration of the event loop).

insertFromMimeData(*source*)
Insert mime data by sending it to the REPL

keyPressEvent(*data*)
Called when the user types something in the REPL.

Correctly encodes it and sends it to the connected device.

process_bytes(*data*)
Given some incoming bytes of data, work out how to handle / display them in the REPL widget.

send_commands(*commands*)
Send commands to the REPL via raw mode.

set_devicecursor_to_qtcursor()
Call this whenever the cursor has been moved by the user, to send the cursor movement to the device.

mu.interface.themes

Theme related code so Qt changes for each pre-defined theme.

Theme and presentation related code for the Mu editor.

class mu.interface.themes.**ContrastTheme**
Defines a Python related theme including the various font colours for syntax highlighting.

This is the high contrast theme.

class mu.interface.themes.**DayTheme**
Defines a Python related theme including the various font colours for syntax highlighting.

This is a light theme.

class mu.interface.themes.**Font**(*color='#181818'*, *paper='#FEFEF7'*, *bold=False*, *italic=False*)
Utility class that makes it easy to set font related values within the editor.

classmethod get_database()
Create a font database and load the MU builtin fonts into it. This is a cached classmethod so the font files aren't re-loaded every time a font is refereced

load(*size=14*)
Load the font from the font database, using the correct size and style

property stylename
Map the bold and italic boolean flags here to a relevant font style name.

class mu.interface.themes.**NightTheme**
Defines a Python related theme including the various font colours for syntax highlighting.

This is the dark theme.

class mu.interface.themes.**Theme**
Defines a font and other theme specific related information.

6.13.5 mu.modes

Contains the definitions of the various modes Mu into which Mu can be put. All the core functionality is in the `mu.modes.base` module.

mu.modes.base

Core functionality and base classes for all Mu's modes. The definitions of API autocomplete and call tips can be found in the `mu.modes.api` namespace.

Contains the base classes for Mu editor modes.

class mu.modes.base.**BaseMode**(*editor*, *view*)

Represents the common aspects of a mode.

actions()

Return an ordered list of actions provided by this module. An action is a name (also used to identify the icon) , description, and handler.

activate()

Executed when the mode is activated

add_plotter()

Mode specific implementation of adding and connecting a plotter to incoming streams of data tuples.

api()

Return a list of API specifications to be used by auto-suggest and call tips.

assets_dir(*asset_type*)

Determine (and create) the directory for a set of assets

This supports the [Images] and [Sounds] &c. buttons in pygamezero mode and possibly other modes, too.

If a tab is current and has an active file, the assets directory is looked for under that path; otherwise the workspace directory is used.

If the assets directory does not exist it is created

builtins = None

Symbols to assume as builtins when checking code style.

deactivate()

Executed when the mode is activated

device_changed(*new_device*)

Invoked when the user changes device.

ensure_state()
Executed when the mode is finished setting up. Used to ensure button / UI state according to current state of settings.

on_data_flood()
Handle when the plotter is being flooded by data (which usually causes Mu to become unresponsive). In this case, remove the plotter and display a warning dialog to explain what's happened and how to fix things (usually, put a time.sleep(x) into the code generating the data).

open_file(*path*)
Some files are not plain text and each mode can attempt to decode them.

When overridden, should return the text and newline convention for the file.

remove_plotter()
If there's an active plotter, hide it.

Save any data captured while the plotter was active into a directory called 'data_capture' in the workspace directory. The file contains CSV data and is named with a timestamp for easy identification.

return_focus_to_current_tab()
After, eg, stopping the plotter or closing the REPL return the focus to the currently-active tab is there is one.

save_timeout = 5
Number of seconds to wait before saving work.

set_buttons(***kwargs*)
Given the names and boolean settings of buttons associated with actions for the current mode, toggles them into the boolean enabled state.

stop()
Called if/when the editor quits when in this mode. Override in child classes to clean up state, stop child processes etc.

workspace_dir()
Return the location on the filesystem for opening and closing files.

The default is to use a directory in the users home folder, however in some network systems this in inaccessible. This allows a key in the settings file to be used to set a custom path.

write_plotter_data_to_csv(*csv_filepath*)
Write any plotter data out to a CSV file when the plotter is closed

class mu.modes.base.**FileManager**(*port*)
Used to manage filesystem operations on connected MicroPython devices in a manner such that the UI remains responsive.

Provides an FTP-ish API. Emits signals on success or failure of different operations.

delete(*device_filename*)
Delete the referenced file on the device's filesystem. Emit the name of the file when complete, or emit a failure signal.

get(*device_filename*, *local_filename*)
Get the referenced device filename and save it to the local filename. Emit the name of the filename when complete or emit a failure signal.

ls()
List the files on the micro:bit. Emit the resulting tuple of filenames or emit a failure signal.

on_start()
Run when the thread containing this object's instance is started so it can emit the list of files found on the connected device.

put(*local_filename*, *target=None*)
Put the referenced local file onto the filesystem on the micro:bit. Emit the name of the file on the micro:bit when complete, or emit a failure signal.

class mu.modes.base.MicroPythonMode(*editor*, *view*)
Includes functionality that works with a USB serial based REPL.

activate()
Invoked whenever the mode is activated.

add_plotter()
Check if REPL exists, and if so, enable the plotter pane!

add_repl()
Detect a connected MicroPython based device and, if found, connect to the REPL and display it to the user.

compatible_board(*port*)
A compatible board must match on vendor ID, but only needs to match on product ID or manufacturer ID, if they are supplied in the list of valid boards (aren't None).

deactivate()
Invoked whenever the mode is deactivated.

device_changed(*new_device*)
Invoked when the user changes device.

find_devices(*with_logging=True*)
Returns the port and serial number, and name for the first MicroPython-ish device found connected to the host computer. If no device is found, returns the tuple (None, None, None).

on_data_flood()
Ensure the REPL is stopped if there is data flooding of the plotter.

remove_plotter()
Remove plotter pane. Disconnects serial connection to device.

remove_repl()
If there's an active REPL, disconnect and hide it.

toggle_plotter(*event*)
Toggles the plotter on and off.

toggle_repl(*event*)
Toggles the REPL on and off.

class mu.modes.base.REPLConnection(*port*, *baudrate=115200*)

close()
Close and clean up the currently open serial link.

execute(*commands*)
Execute a series of commands over a period of time (scheduling remaining commands to be run in the next iteration of the event loop).

open()
Open the serial link

send_commands(*commands*)
Send commands to the REPL via raw mode.

mu.modes.base.**get_default_workspace**()
Return the location on the filesystem for opening and closing files.

The default is to use a directory in the users home folder, however in some network systems this in inaccessible. This allows a key in the settings file to be used to set a custom path.

mu.modes.circuitpython

CircuitPython mode for Adafruit boards (and others).

A mode for working with Circuit Python boards.

class mu.modes.circuitpython.**CircuitPythonMode**(*editor*, *view*)
Represents the functionality required by the CircuitPython mode.

actions()
Return an ordered list of actions provided by this module. An action is a name (also used to identify the icon) , description, and handler.

api()
Return a list of API specifications to be used by auto-suggest and call tips.

compatible_board(*port*)
Use adafruit_board_toolkit to find out whether a board is running CircuitPython. The toolkit sees if the CDC Interface name is appropriate.

connected = True
is the board connected.

force_interrupt = False
NO keyboard interrupt on serial connection.

save_timeout = 0
No auto-save on CP boards. Will restart.

workspace_dir()
Return the default location on the filesystem for opening and closing files.

mu.modes.debugger

The Python 3 debugger mode.

The mode Mu is is when it's debugging a Python 3 script.

class mu.modes.debugger.**DebugMode**(*editor*, *view*)

Represents the functionality required by the Python 3 visual debugger.

actions()

Return an ordered list of actions provided by this module. An action is a name (also used to identify the icon) , description, and handler.

api()

Return a list of API specifications to be used by auto-suggest and call tips.

button_continue(*event*)

Button clicked to continue running the script.

button_step_in(*event*)

Button clicked to step into the current block of code.

button_step_out(*event*)

Button clicked to step out of the current block of code.

button_step_over(*event*)

Button clicked to step over the current line of code.

button_stop(*event*)

Button clicked to stop the current script and return to Python3 mode.

debug_on_bootstrap()

Once the debugger is bootstrapped ensure all the current breakpoints are set. Do not set breakpoints (and remove the marker) if:

- The marker is not visible (the line is -1)
- The marker is not a duplicate of an existing line.
- The line with the marker is not a valid breakpoint line.

debug_on_breakpoint_clear(*breakpoint*)

Handle the clearing of the referenced breakpoint. Currently an unimplemented extra feature.

debug_on_breakpoint_disable(*breakpoint*)

Handle when a breakpoint is disabled.

debug_on_breakpoint_enable(*breakpoint*)

Handle when a breakpoint is enabled.

debug_on_breakpoint_ignore(*breakpoint*, *count*)
Handle when a breakpoint is to be ignored by the debugger. Currently an unimplemented extra feature.

debug_on_call(*args*)
Handle when the debugger has called a function with the referenced args. Make sure the debugger steps into the function.

debug_on_error(*message*)
Handle when the debugger sends an error message.

debug_on_exception(*name*, *value*)
Handle when the debugger encounters a named exception with an associated value. Clear the highlighted line and allow the script to run until the end so the error message is printed to stdout.

debug_on_fail(*message*)
Called when, for any reason, the debug client was unable to connect to the debug runner. On a Raspberry Pi this is usually because it's an underpowereed machine and it takes time to start the debug runner process. (However, the debug client waits for 10 seconds for the runner to start.)

debug_on_finished()
Called when the runner has completed running the script to be debugged.

debug_on_info(*message*)
Handle when the debugger sends an informative textual message.

debug_on_line(*filename*, *line*)
Handle when the debugger has moved to the referenced line in the file.

debug_on_postmortem(*args*, *kwargs*)
Handle when something catastrophic happens to the debugger.

debug_on_restart()
Handle when the debugger restarts. Currenty an unimplemented extra feature.

debug_on_return(*return_value*)
Handle when the debugger returns from a function call with the referenced return value. Make sure the debugger steps out of the function to the caller.

debug_on_stack(*stack*)
Handle when the debugger sends an updated stack.

debug_on_warning(*message*)
Handle when the debugger sends a warning message.

disable_buttons()
Disable all debug control buttons except 'stop'.

disable_buttons_later(*, *milliseconds=100*)
Set a timer to disable all debug control buttons except 'stop'.

enable_buttons()
Enable all debug control buttons except 'stop': if the timer started in *disable_buttons_later* is active, stops it and does nothing else.

finished()
Called when the debugged Python process is finished.

start()
Start debugging the current script.

stop()
Stop the debug runner and reset the UI.

toggle_breakpoint(*line*, *tab*)
Toggle a breakpoint in the debugger.

mu.modes.microbit

The original BBC micro:bit mode.

The mode for working with the BBC micro:bit. Contains most of the origial functionality from Mu when it was only a micro:bit related editor.

class mu.modes.microbit.**DeviceFlasher**(*path_to_microbit*, *python_script=None*, *path_to_runtime=None*)
Used to flash the micro:bit in a non-blocking manner.

run()
Flash the device. If we are sending a custom hex we need to manually read it and copy it into the micro:bit drive otherwise use uFlash.

class mu.modes.microbit.**MicrobitMode**(*editor*, *view*)
Represents the functionality required by the micro:bit mode.

actions()
Return an ordered list of actions provided by this module. An action is a name (also used to identify the icon) , description, and handler.

add_fs()
Add the file system navigator to the UI.

api()
Return a list of API specifications to be used by auto-suggest and call tips.

copy_main(*script*)
If script argument contains any code, copy it onto the connected micro:bit as main.py, then restart the board (CTRL-D).

deactivate()
Invoked whenever the mode is deactivated.

device_changed(*new_device*)
Invoked when the user changes device.

find_microbit()
Finds a micro:bit path, serial port and board ID.

flash()
Performs multiple checks to see if it needs to flash MicroPython into the micro:bit and then sends via serial the Python script from the currently active tab. In some error cases it attaches the code directly into the MicroPython hex and flashes that (this method is much slower and deprecated).

WARNING: This method is getting more complex due to several edge cases. Ergo, it's a target for refactoring.

flash_and_send(*script*, *microbit_path*, *rt_path=None*)
Start the MicroPython hex flashing process in a new thread with a custom hex file, or the one provided by uFlash. Then send the user script via serial.

flash_attached(*script*, *microbit_path*)
Start the MicroPython hex flashing process in a new thread with the hex file provided by uFlash and the script added to the filesystem in the hex.

flash_failed(*error*)
Called when the thread used to flash the micro:bit encounters a problem.

flash_finished()
Called when the thread used to flash the micro:bit has finished.

fs = None
Reference to filesystem navigator.

get_device_micropython_version()
Retrieves the MicroPython version from a micro:bit board. Errors bubble up, so caller must catch them.

minify_if_needed(*python_script_bytes*)
Minify the script if is too large to fit in flash via uFlash appended method. Raises exceptions if minification fails or cannot be performed.

on_data_flood()
Ensure the Files button is active before the REPL is killed off when a data flood of the plotter is detected.

open_file(*path*)
Tries to open a MicroPython hex file with an embedded Python script.

Returns the embedded Python script and newline convention.

remove_fs()
Remove the file system navigator from the UI.

toggle_files(*event*)
Check for the existence of the REPL or plotter before toggling the file system navigator for the micro:bit on or off.

toggle_plotter(*event*)
Check for the existence of the file pane before toggling plotter.

toggle_repl(*event*)
Check for the existence of the file pane before toggling REPL.

mu.modes.pygamezero

The Pygame Zero / pygame mode.

The Pygame Zero mode for the Mu editor.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program. If not, see <http://www.gnu.org/licenses/>.

class mu.modes.pygamezero.**PyGameZeroMode**(*editor*, *view*)

Represents the functionality required by the PyGameZero mode.

actions()

Return an ordered list of actions provided by this module. An action is a name (also used to identify the icon) , description, and handler.

api()

Return a list of API specifications to be used by auto-suggest and call tips.

play_toggle(*event*)

Handles the toggling of the play button to start/stop a script.

run_game()

Run the current game.

show_fonts(*event*)

Open the directory containing the font assets used by Pygame Zero.

This should open the host OS's file system explorer so users can drag new files into the opened folder.

show_images(*event*)

Open the directory containing the image assets used by Pygame Zero.

This should open the host OS's file system explorer so users can drag new files into the opened folder.

show_music(*event*)

Open the directory containing the music assets used by Pygame Zero.

This should open the host OS's file system explorer so users can drag new files into the opened folder.

show_sounds(*event*)

Open the directory containing the sound assets used by Pygame Zero.

This should open the host OS's file system explorer so users can drag new files into the opened folder.

stop_game()

Stop the currently running game.

mu.modes.python3

The Python 3 editing mode.

The Python3 mode for the Mu editor.

Copyright (c) 2015-2017 Nicholas H.Tollervey and others (see the AUTHORS file).

This program is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program. If not, see <http://www.gnu.org/licenses/>.

class mu.modes.python3.**KernelRunner**(*kernel_name*, *cwd*, *envars*)
Used to control the iPython kernel in a non-blocking manner so the UI remains responsive.

start_kernel()
Create the expected context, start the kernel, obtain a client and emit a signal when both are started.

stop_kernel()
Clean up the context, stop the client connections to the kernel, affect an immediate shutdown of the kernel and emit a "finished" signal.

class mu.modes.python3.**MuKernelManager**(**args*, ***kwargs*)

start_kernel(***kw*)
Starts a kernel on this host in a separate process.

Subclassed to allow checking that the kernel uses the same Python as Mu itself.

class mu.modes.python3.**PythonMode**(*editor*, *view*)
Represents the functionality required by the Python 3 mode.

actions()
Return an ordered list of actions provided by this module. An action is a name (also used to identify the icon) , description, and handler.

add_plotter()
Add a plotter pane.

add_repl()
Create a new Jupyter REPL session in a non-blocking way.

api()
Return a list of API specifications to be used by auto-suggest and call tips.

debug(*event*)
Debug the script using the debug mode.

on_data_flood()
Ensure the process (REPL or runner) causing the data flood is stopped *before* the base on_data_flood is called to turn off the plotter and tell the user what to fix.

on_kernel_start(*kernel_manager*, *kernel_client*)
Handles UI update when the kernel runner has started the iPython kernel.

on_kernel_stop()
Handles UI updates for when the kernel runner has shut down the running iPython kernel.

remove_plotter()
Remove the plotter pane, dump data and clean things up.

remove_repl()
Remove the Jupyter REPL session.

run_script()
Run the current script.

run_toggle(*event*)
Handles the toggling of the run button to start/stop a script.

stop_script()
Stop the currently running script.

toggle_plotter()
Toggles the plotter on and off.

toggle_repl(*event*)
Toggles the REPL on and off

6.13.6 mu.resources

Contains utility functions for working with binary assets used by Mu (mainly images).

Based upon work done for Puppy IDE by Dan Pope, Nicholas Tollervey and Damien George.

mu.resources.**load_font_data**(*name*)
Load the (binary) content of a font as bytes

mu.resources.**load_icon**(*name*)
Load an icon from the resources directory.

mu.resources.**load_movie**(*name*)
Load an animated GIF from the resources directory.

mu.resources.**load_pixmap**(*name*, *size=None*)
Load a pixmap from the resources directory.

mu.resources.**load_stylesheet**(*name*)
Load a CSS stylesheet from the resources directory.

mu.resources.**path**(*name*, *resource_dir='images/'*, *ext=''*)
Return the filename for the referenced image.

6.14 Design Decisions

The following documents concern the decision making aspects behind various aspects of Mu. This is a recent practice (started by Tim Golden) so these documents do not cover many aspects of Mu. However, moving forward newer technical decisions will be documented in this way.

6.14.1 Reading and writing code files

Decision

By default Mu will save files encoded as UTF-8 without a PEP 263 encoding cookie. However, if the file as loaded started with an encoding cookie, then the file will be saved again with that encoding.

When reading files, Mu will detect UTF8/16 BOMs and encoding cookies. In their absence, UTF-8 will be attempted. If that fails, the OS default will be used (ie locale.getpreferredencoding()).

If the file cannot be decoded according to these rules, refuse to guess. Instead, produce an informative error popup.

Background

Originally Mu used the built-in open() function for reading and writing its files without specifying an encoding. In that situation Python would request the preferred encoding for the locale and use that. If the user then used a character in their code which had no mapping in that encoding, the save/autosave functionality would raise an uncaught exception and the user would lose their code.

Discussion and Implementation

It was initially suggested that we simply read/write everything as UTF-8 which can encode the entire universe of Unicode codepoints. However, files which had previously been saved by Mu under a different encoding might produce mojibake or simply raise UnicodeDecodeError.

To overcome the difficulty of using UTF-8 going forwards without losing backwards compatibility, the compromise was adopted of *writing* UTF-8 with an encoding cookie, while *reading* according to the rules above.

It will still possible for a file to fail decoding on the way in (eg because the locale-default encoding is used, but the file is encoded otherwise). In that situation we might have attempted to reload using, eg, latin-1 which can decode every byte to *something*. But the result would have been mojibake and – crucially – the autosave mechanism would have kicked in 30 seconds later, overwriting the user's original file for good.

Instead it was decided to offer an informative message box which could explain the situation in enough terms to offer the user a way forward without risking the integrity of their code.

UPDATE: After initial implementation of the encoding cookie, it was thought that it was too arcane for beginner coders. It was decided then to save as UTF-8 by default, although without a cookie. But if a file already has an encoding cookie, that should be preserved and the encoding honoured.

Implemented via:

- https://github.com/mu-editor/mu/pull/390
- https://github.com/mu-editor/mu/pull/399
- https://github.com/mu-editor/mu/pull/418

Discussion in:

(Initial)

- https://github.com/mu-editor/mu/issues/370
- https://github.com/mu-editor/mu/pull/364
- https://github.com/mu-editor/mu/pull/371

(Later)

- https://github.com/mu-editor/mu/issues/402
- https://github.com/mu-editor/mu/issues/696

6.14.2 Line-endings

Decision

Use n internally in Mu. Detect the majority line-ending when loading a file and store it on the tab object within the editor. Then use that newline convention when saving. By default, eg for a new / empty file, use the platform line-ending.

Background

Mu is designed to run on any platform which supports Python / Qt. This includes Windows and any *nix variant, including OS/X. Windows traditionally uses rn (ASCII 13 + ASCII 10) for line-endings while *nix usually recognises a single n (ASCII 10). Although many editors now detect and adapt to either convention, it's common enough for beginners to use, eg, Windows notepad which only honours and only generates the rn convention.

When reading / writing files, Python offers several forms of line-ending manipulation via the newline= parameter in the built-in open() function. Mu originally used Universal newlines (newline=None; the default), but then switched to retaining newlines (newline="") in PR #133

The effect of this last change is to retain whatever convention or mix of conventions is present in the source file. In effect it is overriding any newline manipulation to present to the editor control the characters originally present in the file. When the file is saved, the same characters are written out.

However this creates a quandary when programatically manipulating the editor text: do we use the most widespread n as a line-ending; or do we use the platform convention *os.linesep*; or do we use the convention used in the file itself, which may or may not follow the platform convention?

Discussion and Implementation

My proposal here is that Mu operate its own line-ending manipulation.

When reading the file, note the majority line-ending convention but convert wholly to n. When writing the file, use the convention noted on the way in. This is essentially the same as we would do when reading encoded Unicode from a file.

This way the line-endings are honoured so that, eg, a file can be read/written in Notepad without problems. And the Mu code can be sure of using n as a line-ending convention when manipulating the text.

In terms of the current implementation, the convention from the incoming file could presumably be stored on the tab object.

Implemented via:

- https://github.com/mu-editor/mu/pull/390
- https://github.com/mu-editor/mu/pull/399

Discussion in:

- (original change) https://github.com/mu-editor/mu/pull/133
- https://github.com/mu-editor/mu/pull/371
- https://github.com/mu-editor/mu/issues/380

6.14.3 Run the user's code inside its own virtual environment

Decision

Mu, whether pip-installed or via installer will maintain a separate virtual environment for running the user's code. Initially this will contain all the dependencies used by any of our modes, plus any packages the user installs in addition.

The dependencies needed for the running of the editor itself (mostly PyQt stuff but currently including some serial packages which have "leaked" from the modes) will be kept in a separate environment, virtual or otherwise.

The installers will have to support this approach by bundling wheels (as our assumption is that some schools at least will have restricted or no internet access).

For now we're not breaking out modes into plugins or separate environments although that's a natural extension of this work. If that were done later, each mode could specify its own dependencies which could be installed on demand.

Background

- Getting Mu to unpack and run out of the box on the three main platforms: Windows, OS X & Linux has always proven challenging.
- In addition, within the Mu codebase, the code to run the user's code is scattered and contains a fragile re-implementation of a virtual environment.
- Installing 3rd-party modules was also a little fragile as we had to run *pip* with a *–target* parameter
- There are other issues, especially around the Jupyter console and, on Windows, its use of the pywin32 packages which have slightly odd path handling.

Discussion and Implementation

This started off with work by @ntoll to have the 3rd-party apps installed into a virtual environment rather than with the *–target* parameter which would try to force them into a directory where Mu could find them. This stalled somewhat, especially on Windows, and I (@tjguk) took over that branch.

Having focused on the getting a virtual environment running for the 3rd-party installs, I realised that having a venv for the whole of the code runtime might help solve some of the other issues. After a few merges to get some changes in, especially those by @tmontes to the installer, we started PR#1072 https://github.com/mu-editor/mu/pull/1072.

There is now a *virtual_environment.py* module which initially brings together various pieces of code which were scattered around the codebase and adds support for creating and installing into a virtual environment. The various places

where the user code is run (mostly within the *modes* package, but including inside *panes.py*) have been updated to use this virtual environment logic. Possible future work might involve adding a "run process" method to the class itself.

As far as possible this should remove the need to hack up special *PYTHONPATH*, **.pth* and *site.py* logic.

Implemented via:

- https://github.com/mu-editor/mu/pull/1068
- https://github.com/mu-editor/mu/pull/1072
- https://github.com/mu-editor/mu/pull/1056
- https://github.com/mu-editor/mu/pull/1058

Discussion in:

- https://github.com/mu-editor/mu/issues/1061
- https://github.com/mu-editor/mu/issues/1070

6.14.4 Session & Settings Data

Decision

Centralise access to settings inside a standalone module offering a dictalike-interface. The settings can be loaded from and saved to files. This currently uses JSON (as we have historically) but https://github.com/mu-editor/mu/issues/1203 is tracking the possibility of using TOML or some other format.

Settings objects have defaults which are overridden by values loaded from file or set programatically. When the settings are saved, only values overriding the defaults are saved.

The `load` method can be called several times for the same settings; values in each one override any corresponding existing values. The last loaded filename is the file which the settings will be saved to. Both load and save attempt to be robust, carrying on with warnings in the log if files can't be found, open, read etc.

The existing files (session.json, settings.json) are implemented as singletons in the settings module, and settings.json is autosaved. New settings to support venv functionality – in particular, baseline packages – is also added.

At its simplest https://github.com/mu-editor/mu/pull/1200 does no more than implement this set of functionality. The few places in existing code where settings were used or altered have been updated to use the new objects and functionality.

Not Implemented / Hooks

During the design and/or based on previous discussions, several ideas were floated which are at least supported by the new implementation.

- Safe mode / Readonly mode / Reset mode

  As described below, there are situations where teachers or admins would like to reset settings for use in a club or classroom setting. The new implementation supports this idea via the `reset` method and `readonly` flags without actually implementing it as such.

  Such functionality might, in the future, be managed by means of command-line switches or some other flag.

- File format: JSON, YAML, TOML…

  The implementation tries to be agnostic as to file format. At present it uses the historically-implemented JSON format. But the choice of serialiser is centralised towards the top of the module and shouldn't be hard to change, especially for any serialiser which uses the conventional `.dumps`, `.loads` API.

  cf https://github.com/mu-editor/mu/issues/1203

- One file / Two files?

  The new settings implementation facilitates any number of files each of which can have an arbitrary hierarchy. Whether we end up with one settings file containing, eg, session settings and board settings, or several files each specific to an area can be decided later. Nothing in this implementation precludes either approach.

- Interpolation

  Because it is easy to implement and doesn't seem risky, this implementation applies `os.path.expandvars` to any values retrieved. This will do platform-sensitive env var expansion so admins can specify, eg, a workspace directory of %USERPROFILE%mu_code or $HOME/.mu/mu_code.

  Value interpolation (where one settings value can rely on another) has *not* been implemented. It's potentially quite an involved piece of work, and the benefit is not so clear.

- Indicating failure to users

  This is obviously a wider issue, but while this implementation tries to be robust when loading / saving settings, it only writes to the standard logs and then fails quietly. The problem here is that we're possibly not operating within the UI. At the least, we don't have a good overall story for a UI which isn't part of the central editor itself.

Background

Mu maintains two files, automatically saved on exit, to hold user settings and session data. The former contains critical parameters without which the editor probably won't function. The latter contains more or less cosmetic items which can be cleared (eg by a "Reset" button) without losing functionality.

Historically access to these files has been somewhat scattered around the codebase, making it difficult for modules to access them coherently. The first aim of the re-implementation is to create globally-accessible singletons, much as is conventional for logging. Those "settings" objects would offer a dictionary-like interface so that code could easily do:

```
import settings

def set_new_theme(theme):
    ...
    settings.session['theme'] = theme.name
```

The second aim is, possibly, to reconsider the use of the settings, or their structures, or which / how many files they are and where they're situated. Any such refactoring or restructuring should be a lot easier with a newer implementation.

Discussion and Implementation

Open questions:

- How many / which files do we need?
- Should we combine both settings / sessions into one file? Is there a meaningful difference which we want to maintain? [+1]
- Should we register exit handlers so the files are always saved on closedown? [+1]
- Should we write files to disc as soon as they are updated? [-0]
- Should we re-read files to allow users to update them mid-session? [-1]
- Should we implement read-only mode (ie the existing file is loaded but not written back)? [+0]
- Should we implement safe mode (ie the file is neither loaded nor written back)? [+1]
- Should we implement reset mode (ie the file is not loaded but is written back)? [+0]
- Should we break out the virtual environment settings (venv location, baseline packages) into its own file? [+1]
- Could we add a boards.json file to allow users to add new/variant configurations? [+0]
- What levels of config do we need? Defaults? One/multiple settings files? Override at instance level?
- Do we still need to look in the application directory as well as the data directory? [-0]
- What format should the files use? [cf https://github.com/mu-editor/mu/issues/1203]
- Should we save everything every time? [-0.5]
- Do we need interpolation of other settings? (eg ROOT_DIR = abc; WORK_DIR = %(ROOT_DIR)/xyz)
- Do we need interpolation of env vars? (eg ROOT_DIR = %USERPROFILE%mu_code) [+0.5]
- Should we merge `settings.py` into `config.py` [+0]
- Should settings (as opposed to sessions) be read-only? [+1]

Exit Handlers

Registered exit handlers to ensure that files are saved when Mu exits. (This could probably be alternatively achieved within the Qt app). The advantage of this is that the save is automatic; the disadvantage is that it's a little hidden.

Not currently writing to disc as soon as updated: having an exit handler ensures the settings will be written, even in the event of an unhandled exception. And it's not clear what advantage an "autosave" would offer.

Levels of Config

Allowing three levels of data: the defaults for each setting type, held in a class dictionary; possible overrides at class instantiation [I'm not clear where this would be used; it can probably go]; and the .json files.

The `load` function merges into the existing settings. Most commonly this means it'll be preceded by a call to `reset`. But it could be used to implement a cascade of settings, eg where an admin sets site-wide settings which are then overridden by user settings.

Amnesia / Read-only / Reset modes

To support the possible "modes" above – amnesia, read-only etc. there is a `readonly` flag on each settings object, preventing it from being written to disc; and a `reset` method which will return to default settings. This last can be used either to "forget" any loaded or set settings; or before reloading from a different file.

So *Safe mode* is implemented by calling `reset` without `load` and setting `readonly`. *Read-only mode* is implemented by calling `reset` followed by `load` and setting `readonly` And *Reset mode* is implemented by calling `reset` without `load` and *not* setting `readonly`

The use cases here would be mostly for admins or leaders who needed, eg, to ensure that new sessions were started for every user, or who needed to debug or recover from a corrupt settings file.

Failure modes

It's critical that we should recover well from not being able to read or to write settings files, whether that's a file system failure or invalid JSON. Regardless of the approach we should definitely log any exception, or log a warning where there's no exception as such but, say, a missing file.

Reading

- A failure to find/open a settings file is considered usual: it's expected that, the first time around, a user settings file won't exist to be read. The loader will log a warning and carry on as though it had found it empty
- A failure to read the JSON from a settings file is more complicated. For pragmatic purposes, the intention is here is: log a warning; quarantine the file; and carry on as though it had been found empty. That way the editor continues to work, albeit in "reset" mode, and the failing file is available for debugging.

  Not quite clear: should we automatically enter read-only mode in this situation?

Writing

- A failure to open a settings file to write to is more problematic, and there's not very much we can do. Log the exception (eg AccessDenied or whatever). Perhaps – given that the text won't be great – pushign the JSON output to the logs as debug might give some manual fallback.
- A failure to *write* JSON is less probable – although it does happen during testing where the JSON lib attempts to serialise a Mock object. Here, we can't really do more than log the exception and fail gracefully.

Levels of Config & Defaults

The thrust of this proposal expects the *Settings* subclass to hold a dictionary of defaults at class level. These are applied first before any file is loaded. Any information from a loaded file is overlaid, so the file data "wins". Any values not present in the file remain per the default.

Although not implemented in any way at present, the mechanism allows for several files to be loaded in succession, typically for a site-wide file, set up by an administrator, followed by a user-specific file. In this scenario, the data would be read: Defaults < Site settings < User settings with later data replacing earlier data.

The presence of the defaults in the *Settings* subclass should also make for a more consistent use of defaults across the codebase. Eg if in general device timeouts should be 2 seconds but can be changed, one piece of code might do:

```
timeout_s = settings.user.get('timeout_s', 2)
```

while another piece elsewhere might do:

```
timeout_s = settings.user.get('timeout_s', 3)
```

If the defaults are present in the class, the *.get* method could be implemented so the default, instead of *None* as conventional, returns the class default:

```
timeout_s = settings.user.get('timeout_s')
# with no explicit timeout_s setting, timeout_s is now the default value
```

Taking this further, it's not clear that we even need to load the defaults as such; we could always just fall back to them in the event of a .get KeyError or even a __getitem__ KeyError. Taking that approach would also means we wouldn't need the "dirty data" mechanism because anything in the `_Settings` object's own `_dict` should be saved out at the end.

Saving Everything?

Implicit in the new design is the idea that settings are saved out to file(s) at the end of every session.

Originally, the effect of the defaults was that, say, a workspace directory would inherit the default which will then be written out to the settings file at the end of the session. Even if that file had not originally had a settings for the workspace directory.

On reflection, I've re-implemented for now a "dirty" setting for each attribute. Only "dirty" attributes are written out to file. Anything loaded from a file is considered "dirty" even if it remains unchanged for the duration of the session. Anything updated during the session – and this will typically be user-configurable items like Zoom level, Theme &c. – is also tagged as "dirty" and will be written out to file.

Implemented via:

- https://github.com/mu-editor/mu/pull/1200

Discussion in:

- https://github.com/mu-editor/mu/issues/1184
- https://github.com/mu-editor/mu/issues/1203

6.15 Release Process

Our continuous integration setup provides the following automation:

- Running of the unit test suite on Windows, OSX and Linux for each commit.
- Code quality checks via LGTM.com. Mu has an A+ rating for code quality.
- Generation of installables for Windows 32bit and Windows 64bit for each commit on our master branch.
- Creation of a stand-alone .app for Mac OSX for each commit on our master branch.

However, such automation does not make a release. What follows is a check-list of steps needed to cut a release.

6.15.1 User Activity Checks

To ensure nothing is broken from the user's point of view the following key user activities should be completed on Windows, OSX and Linux (to ensure the cross platform nature of Mu is consistent):

- Start Mu from a clean state (delete your Mu configuration file, and mu_code directory). *Outcome: Mu should ask for an initial mode and a fresh mu_code directory is created. Upon restart, Mu doesn't repeat this process.*
- Click the "Mode" button, select a new mode. *Outcome: the mode selection dialog should appear and you'll find yourself in a new mode.*
- Click the "New" button. *Outcome: a new blank tab will appear.*
- Click the "Load" button. *Outcome: the operating system's file selector dialog should appear. The selected file should open in a new tab.*
- With a new tab, click the "Save" button. *Outcome: the operating system's file naming dialog should appear, and the tab will be updated with the newly named filename.*
- While in Python mode, plug in an Adafruit board. *Outcome: Mu should suggest switching to Adafruit mode.*
- In Adafruit mode, load, edit and save a file on the device. *Outcome: upon saving the file, the device will reboot and run your code.*
- In Adafruit mode, click the "Serial" button. *Outcome: a serial connection to the attached device is shown in a pane at the bottom of Mu's window. Pressing Ctrl-C should drop you to the CircuitPython REPL.*
- In Adafruit mode, use some code like this on the Adafruit device (in the case of this example, a Circuit Playground Express) to emit tuple based data. Click the "Plotter" button while the code is running. *Outcome: the plotter should display the output as a graph.*
- While in Python mode, plug in a micro:bit board. *Outcome: Mu should suggest switching to micro:bit mode.*
- In micro:bit mode (from now on, assuming you have a micro:bit device connected), while the current tab is completely empty, click the "Flash" button. *Outcome: Mu should do a complete fresh flash of "vanilla" MicroPython.*
- In micro:bit mode, write some simple working code and click the "Flash" button. *Outcome: since MicroPython is already flashed on the device, only the file will be copied over and the device will soft-reboot.*
- In micro:bit mode, click on the "Files" button. *Outcome: the files pane will appear and contain a true reflection of the current state of the file system on the device and in your mu_code directory.*
- In micro:bit mode, while the "Files" pane is active, copy to/from the device, delete a file on the device and open files listed on your computer by right-clicking them. *Outcome: the file pane state should update and no error message appear.*
- In micro:bit mode, click on the "REPL" button. *Outcome: you should see and be able to interact with the REPL of MicroPython running on the connected device.*
- In micro:bit mode, use some code to emit tuple based data. Click on the "Plotter" button while the code is running. *Outcome: the plotter should display the output as a graph.*
- In PyGameZero mode, with an empty file, click the "Play" button. *Outcome: a blank Pygame window will appear.*
- In PyGameZero mode, with correctly working code, click the "Play" button. *Outcome: the game runs.*
- In PyGameZero mode, click each of the "Images", "Fonts", "Sounds" and "Music" buttons. *Outcome: the operating system's file manager should open in the correct directory for each of these types of game asset.*
- In Python mode, enter a simple script and click "Run". *Outcome: the script should run with input/output being handled by a pane at the bottom of the Mu window.*

- In Python mode, add a new breakpoint to your code, click "Debug". *Outcome: the visual debugger should start and stop at your breakpoint.*
- In Python mode, with no breakpoints present, click "Debug". *Outcome: the visual debugger will start and stop at the first valid line of code.*
- While the visual debugger is active, add and remove breakpoints. *Outcome: the UI will update (red dots will appear etc) and the debugger will respect such changes (stopping at new breakpoint, ignoring removed breakpoints).*
- While in the visual debugger click the "Stop", "Continue", "Step Over", "Step In", "Step Out" buttons. *Outcome: the conventional behaviour for each button should happen. "Stop" will stop the script. "Continue" will run to the next break or end of script. "Step Over" will move to the next valid line of code. "Step In" will move into the called funtion. "Step Out" will move out of the current function. As all this happens, the input/output pane and object inspector should update as the code progresses.*
- In Python mode, click on the "REPL" button. *Outcome: an iPython based REPL should appear in a new pane at the bottom of Mu's window. Clicking the button again toggles the REPL off.*
- In Python mode, use some code to emit tuple based data. Click on the "Plotter" button while the code is running. *Outcome: the plotter should display the output as a graph.*
- Click "Zoom-In". *Outcome: the font size should increase.*
- Click "Zoom-out". *Outcome: the font size should decrease.*
- Click "Theme" several times. *Outcome: the theme/look should toggle.*
- With incorrect code in the current tab, click "Check". *Outcome: problems like syntax errors or undefined names should be highlighted with annotations on the correct line. If appropriate, they will be underlined.*
- Click the "Help" button. *Outcome: the operating system's default browser should open at the help page for the current version of Mu.*
- With unsaved code in the current tab, click "Quit". *Outcome: Mu should warn you may lose unsaved work and prompt you to confirm.*
- With all work saved, click "Quit". *Outcome: Mu should quit.*
- Click on the "cog" icon in the bottom right of Mu's Window. *Outcome: the "admin" dialog should open with the "logs" tab in focus.*
- In the editor panel, type CTRL-K while code is selected. *Outcome: the selected code should toggle between commented and uncommented.*
- Type CTRL-F. *Outcome: the find/replace dialog should appear.*

6.15.2 Pre-Packaging Checklist

- All autogenerated API information used by Mu for auto-completion and call tips should be regenerated.
- The developer documentation should be checked, re-read and regenerated locally to ensure everything is presented correctly.
- The CHANGELOG.rst file should be updated to reflect the differences since the last officially packaged release.
- If this is a major release make sure the resources for the old version of Mu on the project website are archived under the correctly versioned URL scheme.
- Make sure the current resources in the source for the project website reference the new version of Mu.

6.15.3 Packaging Processes

Official final releases will be signed by Nicholas H.Tollervey (the creator and current maintainer of Mu). This is a manual step only Nicholas can do (since only he has the cryptographic keys to make this work). Once the release packages for Windows (32bit and 64bit) and OSX have been created and signed they should be checked so no warning messages appear about untrusted sources during the installation process.

The instructions for signing the Windows installers are explain in this wonderful article on Adafruit's website. But the essence is that the command issued should look something like:

```
"C:\Program Files (x86)\Windows Kits\10\bin\10.0.17134.0\x86\signtool" sign /v /n
↪"Nicholas H.Tollervey" /tr http://timestamp.globalsign.com/?signature=sha2 /td sha256␣
↪mu-editor_1.0.1_win32.exe
```

Signing the Mac app involves issuing the following two commands:

```
codesign --deep --force --verbose --sign "CERT_ID" mu-editor.app
dmgbuild -s package/dmg_settings.py "Mu Editor" dist/mu-editor.dmg
```

The appropriate installer should be checked on the following operating systems:

- Windows 7 (32bit)
- Windows 10 (64bit)
- Latest OSX.

For native Python packaging, ensure Mu is installable via `pip install .` run in the root of the source repository in a virtualenv.

6.15.4 Pre-Release Checklist

- Create an announcement blog post for MadeWithMu.
- Tweet an announcement for the timing of the upcoming release.
- Compose (but do not publish) a tweet to announce Mu's release.
- Ensure the source code for the developer docs, the project website and MadeWithMu is all ready to be published.
- Prepare a press release and circulation list.
- Check other possible channels for announcements, community websites etc.

6.15.5 Release Process

- Build the developer documentation on ReadTheDocs. Make a note of the link to the latest release in the resulting page on the CHANGELOG.
- Create a new release on GitHub and attach the signed 32bit and 64bit Windows installers and OSX dmg. Reference the changelog from step 1 in the release notes.
- Update the download page on the project website to the URLs for the installers added to the release in step 2. Update the live version of the website.
- Push the latest version to PyPI (`make publish-test` then `make publish-live`).
- Publish the blog post announcement to MadeWithMu.
- Tweet with link to the announcement blog post and changelog.

- Mention release in Gitter, Adafruit's CircuitPython Discord.
- Send out press release / news item to circulation list / friends.
- Hit other possible announcement channels.

6.15.6 Post-Release Tasks

- Monitor Gitter chat channel for problems.
- Clean up fixed issues in GitHub.
- Update Roadmap.rst with reference to the next release.
- Send out thanks / gifts where appropriate.

6.16 Roadmap (Mappa MUndi)

(Apologies for the pun: https://en.wikipedia.org/wiki/Mappa_mundi)

Mu started as a shonky hack. Now many people are interested in our small editor for educational use and we owe it to them to be clear what our plans are, how we work together and how *anyone* can get involved (see `CONTRIBUTING.rst`).

I believe it worth repeating the Mu philosophy we have followed so far:

- Less is More: Mu has only the most essential features, so users are not intimidated by a baffling interface.
- Path of Least Resistance: Whatever the task, there is always only one obvious way to do it with Mu.
- Keep it Simple: It's quick and easy to learn Mu ~ complexity impedes a novice programmer's first steps.
- Have fun: Learning should inspire fun ~ Mu helps learners quickly create and test working code.

Python aims to make code readable, Mu aims to make it writeable.

With this in mind:

6.16.1 Next Point Release

1.0.1 is a bug fix / translations release and will only include:

- Update to Adafruit boards and future proofing for as-yet-unknown boards.
- Swedish translation.
- Updated and complete Chinese translation.
- Blocking IO from Python 3 sub-process flooding data is fixed.
- New MicroPython runtime for micro:bit (bug fixes).
- Improvements to the stability of micro:bit flashing.

Expected delivery: mid-September 2018.

6.16.2 Next Minor Release

1.1.0 will introduce some new "beta" modes:

- ESP mode for embedded devices from ESP.
- Web mode for creating simple dynamic websites.

It will also add some new features:

- Use of "black" for code style / quality checking.
- Configuration of UI for purposes of better presentation:
  - Change size of buttons.
  - Tool-tips and auto-complete toggle.
  - Colour configuration for "Custom" theme (help dyslexic users via colour).
  - Transparent background (makes screen-casting easier).
- Update minifier.
- More translations.
- Cleanups to the documentation.
- Bug fix release of MicroPython for micro:bit.

Expected delivery: late-November 2018.

6.17 Mu's Developers

Mu was created and mostly written by Nicholas H.Tollervey (ntoll@ntoll.org). Some of Nicholas's work has been magnificently supported by the Raspberry Pi Foundation.

Happily, many people have volunteered wonderful and varied contributions to Mu. These include (but are not limited to):

- Tim Golden (mail@timgolden.me.uk)
- Peter Inglesby
- Carlos Pereira Atencio (carlosperate@embeddedlog.com)
- Nick Sarbicki (nick.a.sarbicki@gmail.com)
- Kushal Das (mail@kushaldas.in)
- Tibs / Tony Ibbs (tibs@tonyibbs.co.uk)
- Zander Brown
- Alistair Broomhead (alistair.broomhead@gmail.com)
- Frank Morton (fmorton@mac.com)
- Keith Packard (keithp@keithp.com)

We welcome contributions from anyone! Please see *Contributing to Mu* for more information.

If you have made a contribution to Mu and would like to be recognised, please feel to add yourself to the list above.

6.18 Release History

6.18.1 1.2.0

This release fixes some minor bugs, addresses some usability gremlins and adjusts some capabilities to make things tidier. Much of this work was done over the summer at the code-sprints at EuroPython 2022 in Dublin. Kudos and thanks to all the new contributors to Mu.

Please provide bug reports or feedback via:

https://github.com/mu-editor/mu/issues/new

- Thanks to @keith-packard for Snek mode. [Snek](https://sneklang.org/) is a Python inspired language for processors too small even to run MicroPython.
- @tmontes contributed changes so Mu builds to Linux AppImages (an easy way to package application for Linux).
- Minor fixes by @stratakis in the Russian translation.
- @carlosperate fixed many minor glitches and gremlins.
- @carlosperate was on fire with fixes needed to ensure Mu continues to work with very old versions of OSX (as used in many educational institutions).
- Again, thanks to @carlosperate, AppImage with Wayland no longer the setting of an environment variable to make it work properly.
- The web mode includes simple and easy to use integration with beginner and education friendly web hosts, PythonAnywhere.
- @agdales, @Jeffrey04, @johannaengland and @AnjaVerboven contributed new messages of the day as part of their onboarding at EuroPython.
- @tonybaloney contributed several Windows based fixes and clean-ups.
- @johannaengland and @prcutler were on fire tidying up and fixing docs at EuroPython.
- A bug was fixed in the web mode relating to the resolution and/or recreation of the assets directory (in which images, css and templates were to be found).
- Or friend at Adafruit, @tennewt made the necessary changes so Mu handles OSC commands gracefully (see the [PR](https://github.com/mu-editor/mu/pull/2326) for more details).
- New contributor, @zigit ensured "Unexpected Maker" based ESP boards are correctly detected.
- Thanks to @Jayman2000, error messages are correctly capitalized (or not) to avoid potential confusion.

6.18.2 1.1.1

Inevitably, while testing 1.1.0 we found we'd missed something and caught a late breaking bug.

Please provide bug reports or feedback via:

https://github.com/mu-editor/mu/issues/new

- Thanks to @MinoruInachi (with feedback from @odaki) for a revised Japanese translation for Mu.
- Due to complicated dependency problems, we've updated the bundled version of Flask to 1.1.4. Thanks to @carlosperate for quickly resolving this problem.

6.18.3 1.1.0 (final)

What a journey to get to the 1.1.0 release of Mu. Many thanks to all the contributors who have made this version possible. All your efforts, no matter large or small, are really appreciated.

Please provide bug reports or feedback via:

https://github.com/mu-editor/mu/issues/new

- Minor clean ups in the Makefile.
- Thank you to @microbit-mark for updating the board IDs to support version 2.2 of the device.
- Updates to the Chinese translation by @CSharperMantle.
- Updates to the Slovak translation by @bletvaska. Ďakujem.
- The foundations of a brand new Russian translation of Mu by @grovz with contributions from @iamdbychkov. !

6.18.4 1.1.0-beta.7

This is a beta release and may contain bugs or unfinished features. Please provide bug reports or feedback via: https://github.com/mu-editor/mu/issues/new

- We expect this release to be the last beta before the final 1.1 release in the new year of 2022. Season's greetings to everyone using or contributing to Mu, and here's wishing you all a flourishing and fulfilling 2022.
- As always there have been the usual minor bug fixes and clean ups from the core team of maintainers. Thank you so much for all that you do to support the continued development of Mu.
- Thanks to the ever-green @keith-packard for his contribution to ensure icons on the button bar continuously scale based on the window width. This looks really smooth and slick.
- Tinsel laden @tmontes has made a number of contributions around tooling for internationalization (i18n) of Mu. These include using the Babel package for generating the required translation files from our source code, and updating the `Makefile` (and `make.py`) so the process can be automated.
- Xmassy @xbecas is a new joiner to the core team and we're very please to have him with us since he has done a **huge** amount of work on updating and curating the translation files needed for i18n. Thanks to his work, translators for all the other existing locales need not have to go through the string generation/update steps (he's done that for you already!).
- Both @xbecas and @tmontes have made extensive updates to our pt-PT (Portuguese) translation. Feliz Natal e Próspero Ano Novo.
- This was swiftly followed by a welcome contribution by @rffontenelle the red-nosed translator, who made extensive updates to the pt-BR (Brazilian Portuguese) translation. Many thanks Rafael, you continue to demonstrate why the Brazilian FLOSS community is such a vibrant place, and we hope your work will help beginner coders in Brazil take their first steps to join your community. Boas Festas!
- Now that the upstream PyGame / PyGameZero packages have been updated and repackaged, @tmontes has ensured we use these (rather than our own custom builds) in our installers for Windows and OSX. Many thanks to our friends and collaborators in those projects (cc/ @illume and @lordmauve).
- Once in royal @devdanzin's repos, stood some lowly bugs to fix. These include ensuring empty path handling is properly handled by `get_save_path`, correct highlighting of both f-strings and triple quoted strings in the editor widget, fixing a comment-toggling bug that deleted the first character of the next line under certain circumstances and more robust handling of environment variables. Wow, @devdanzin was on fire..! (...and has further work in development, thank you so much for your continued contributions.)

- Carolling @carlosperate has put a huge effort in. He has triaged various crash reports, administered our continuous integration pipeline, and reviewed and merged much of the work described above. He also ensured our version numbering for Mu is no longer odd, and meets the guidelines set out in PEP440.
- Good Tim Golden (@tjguk) fast typed out, a venv that's crisp and even. His outstanding work on making Python virtual environments work in some of the most inhospitable computing environments ever found is miraculous. Tim's genius is to know exactly the right intervention to make, and in this case his epic addition of `-I` to the Mu codebase will help ensure the user's virtual environments are properly isolated.
- @tiago has updated the pup packager we use to create our installer. This should fix a problem found on the new ARM based Macs. He has also made significant progress on a cross-distro Linux package which we hope will make an appearance in the not-too-distant future.
- Finally, Nicholas (@ntoll) promises never to do another Christmas themed changelog.

6.18.5 1.1.0-beta.6

This is a beta release and may contain bugs or unfinished features. Please provide bug reports or feedback via: https://github.com/mu-editor/mu/issues/new

- Another delayed release due to busy-ness of the volunteer team involved in Mu. Thank you for your patience, bug reports and code patches.
- There have been the usual minor bug fixes and clean ups from various regular contributors and some new ones too. Thank you for your careful and well targetted changes.
- Carlos (@carlosperate) fixed some packaging problems relating to the iPython kernel bundled with Mu.
- Martin (@dybber) fixed a couple of problems relating to the stopping of child processes (Flask and scripts stopped via KeyboardInterrupt in Linux).
- The web mode checks for the availability of templates in the local directory tree before starting up. If a template directory isn't found in the expected location, then the user sees a helpful message describing the problem and what they need to do to fix it.
- Mu's splash screen no longer always appears on top of everything else on the user's desktop. The splash screen now also logs the progress of installing the various packages needed on first install. Thanks to Carlos for these changes.
- A new admin/settings option has been added to allow users to manually change the translation Mu uses for its interface. Updating this setting requires a restart of Mu. Zander (@ZanderBrown) contributed the icon/glyph to indicate the relevant tab is for translation related settings (not entirely obvious if Mu's UI is using a language you don't understand and you're looking for the setting that relates to translations). The icon makes this clear.
- On some desktop windowing systems there is a bug that means windows re-open at a position higher up the screen, and so may appear off the top of the screen. We've ensured this never happens with Mu. If Mu starts with any part of the window off the screen, the window is moved to be within the dimensions of the screen. This was a weird one to track down and fix.
- Many thanks to Ethan Spoelstra (@espoelstra) who contributed a change so Crostini on ChromeOS is used to return the correct CIRCUITPY path if it exists.
- Huge thanks to Keith Packard (@keith-packard) for several contributions to this release of Mu. Keith refactored the way in which Mu handles pasting in the REPL window so it works correctly and more broadly across operating systems.
- Keith also fixed some font related issues in the REPL.
- Keith was on fire with a couple more contributions relating to SVG icons in the buttons in Mu. We're very grateful to Ben Williams (@Rybec) for putting in the work to make our button icons SVG files. Keith made the code changes to implement this.

- Thanks to Miro Hrončok (@hroncok) for pointing out a change in Python 10 which would break some of our UI calls into PyQt, and who provided a patch to fix things.
- Some minor clarifications in our developer documentation (https://mu.rtfd.io).

6.18.6 1.1.0-beta.5

This is a beta release and may contain bugs or unfinished features. Please provide bug reports or feedback via: https://github.com/mu-editor/mu/issues/new

- We had hoped for a regular (fortnightly) release tempo. Due to the voluntary nature of Mu's development and because some of the updates in this release were quite challenging (see below), this release is a LOT later than we had planned.
- Several of us made minor updates and fixes (such as ensuring various packages had explicit dependency versions listed, updating versions for Mu's own dependencies and so on).
- Right clicking on highlighted text in the editor, with the REPL active, now has an additional option added to the context menu: to correctly paste the text from the editor into the REPL. Thanks to Professor Chris Rogers of Tufts University for suggesting this feature.
- The multi-talented Dan Halbert of Adafruit very kindly fixed a bug in the Adafruit board handling when on run on new Apple M1 hardware. Thank you Dan for your valuable contribution.
- A huge amount of work by Tim and Carlos has gone into analysing the crash reports from recent beta releases of Mu. This has resulted in significant effort to address many of the bugs encountered, many of which related to edge cases encountered by the new virtual environment feature. Tim and Carlos have created many fixes and checks to ensure these bugs are either completely fixed or are, at least, mitigated in more helpful ways. This has been a challenging and "fiddly" bit of work, so kudos and thanks, as always, to both Tim and Carlos for their continued efforts.
- Carlos has also updated the version of MicroPython used in the BBC micro:bit mode to the latest 2.0.0-beta.5 version.
- In addition, Carlos has ensured that the micro:bit mode flashes files onto the micro:bit using the correctly memory aligned hex string.
- Github user ajs256 has ensured the crash reporter doesn't kick in when a `KeyboardInterrupt` is triggered in Mu (CTRL-C).
- Sometimes in Mu for Linux, the expected `.py` file extension wasn't added to new files. This depended on the user's graphical shell. Mu now checks the output from the shell and, if requires, will add `.py` itself.
- Various fixes to Mu's logging make it more robust, clear and useful.
- Tiago fixed a late breaking bug in packaging Mu for OSX. All fixed in a matter of hours. Amazing work!

There are perhaps a couple more features we want to land in the coming weeks, and then we will start the work of ensuring internationalization is fully up to date, the website reflects the new features and various changes, and PUP will be able to produce redistributable appimages for Linux. Then we will have reached 1.1.0-final. :-)

6.18.7 1.1.0-beta.4

This is a beta release and may contain bugs or unfinished features. Please provide bug reports or feedback via: https://github.com/mu-editor/mu/issues/new

- During the beta phase, we're moving to a fortnightly release cadence. Since this release is a week late, expect the next one in a week's time - 26th April.
- Carlos made many changes to clean up the specification for required modules used by the installer. This will make supporting and tracking Mu's dependencies MUCH easier. Thank you Carlos!
- Huge thanks to Dan Halbert of Adafruit who contributed a significant amount of refactoring to the CircuitPython mode. As a result Mu now uses the *adafruit-board-toolkit* module for device identification, among many other helpful changes [described in Dan's pull request](https://github.com/mu-editor/mu/pull/1371). Thank you Dan..!
- Carlos was on fire… he also fixed a bug in the file-copy dialog when the context menu was opened with an empty list of files.
- Carlos (again), fixed some outstanding documentation issues for supporting Raspbian Buster (and newer). These are now at https://mu.rtfd.io/.
- Carlos (again, again) tidied up various aspects of the Makefile so there is only a single source of truth for running various utilities and commands.
- Logging was another focus for this release. Now that we have a few weeks worth of crash reports we've been able to look at the parts of the application that cause most grief and add extra-logging in various locations. Tim put in a great effort to make sure the "first run" and other virtual environment based aspects of Mu now have clearer and more useful logging and throw more useful exceptions, closer to the source of the problem, for the resulting crash report. Carlos ensured the IPython kernel installation was properly logged.
- We ensured various key packages were pinned to particular versions to maximise compatibility with older versions of Python still found in schools.

There are many pull requests and work items currently in flight and they'll be landing very soon as the overall quality and robustness of Mu significantly improves. Many thanks to everyone who continues to help, support and contribute to the ongoing development of Mu.

6.18.8 1.1.0-beta.3

This is a beta release and may contain bugs or unfinished features. Please provide bug reports or feedback via: https://github.com/mu-editor/mu/issues/new

- During beta phase, we're moving to a fortnightly release cadence. Expect beta 4 on the 12th April.
- The final version of the Mu splash screen was delivered. Huge thanks to the extraordinarily talented Steve Hawkes ([@hawkz](https://github.com/hawkz)) of [The Developer Society](https://www.dev.ngo/) for his generous artistic support, patience and humorous approach.
- Thanks to a recent update in [PyGame Zero](https://pypi.org/project/pgzero/), we're back to using the official package from PyPI, rather than our patched fork, in the installer.
- Both Tim and Carlos have contributed updates, fixes and tests to address a bug affecting Windows users who may have a space in the file path upon which Mu is found. This was a difficult bug to reproduce but Tim did a lot of digging to isolate the cause with as much confidence as is possible when it comes to such things. Carlos did a bunch of thankless and fiddly test related work so testing with spaces in the path is part of our test suite. Work on this is ongoing so expect further improvements in upcoming releases. As always, many thanks for these efforts.
- Tim addressed a *wheel*/*sdist* related problem that was causing odd side effects with regard to dependancies.

- A strange bug, where it was not possible to install third-party packages on first run of Mu, opened up a deep rabbit hole of investigation. In the end Tim was able to fix this AND address the source of a warning message from Qt when Mu was starting for the first time.
- The splash screen code was rewritten in such a way that objects relating to the splash screen will always be garbage-collected by Python and destroyed by Qt5. Previously, they existed for the full duration of the application, not really causing any problems, but "in limbo" nonetheless.
- The crash reporting tool has had a minor update so the user is reminded to attach their log file to the bug report, along with an indication of where to find the log file.

6.18.9 1.1.0-beta.2

This is a beta release and may contain bugs or unfinished features. Please provide bug reports or feedback via: https://github.com/mu-editor/mu/issues/new

- This is the first public beta release (beta 1 was created for testing by the core development team).
- Many minor bug fixes to the existing new features found in beta 1 (see below).
- Many thanks to Martin Dybdal for his work on improving the admin panel.
- Carlos made significant changes so Mu can be packaged with very recent versions of Python. Carlos also made various changes relating to the status of Python packages contained within the official installer.
- Many thanks to Dan Pope for assistance with an upgraded version of PyGameZero (which uses the latest version of PyGame - kudos to René and the other developers of PyGame for the recent improvements).
- Various fixes to the UI so that panes are easier to resize and the themes are correctly applied to the REPL (thanks again to Martin for these fixes).
- Carlos also contributed fixes relating to the micro:bit mode (compatibility with versions 1 and 2).
- Tim has made herculean efforts to ensure the creation and checking of Mu's virtual environment is robust and easy to maintain.
- A new crash reporting feature has been added. If Mu breaks the user will be redirected to the endpoint code-with.mu/crash with details of the crash and an option to create a bug report. This ensures Mu crashes are handled more gracefully, and the user is able to see the error that caused the crash.
- A new animated splash screen has been added so the initial creation of Mu's virtual environment happens in such a way that the user can see progress is being made, and updates are logged on the splash screen for the user. If Mu encounters a problem at this early stage, the splash screen recovers and the new crash reporting feature kicks in. The current animation was created by Steve Hawkes (thank you) with a much more polished version promised very soon..!
- Behind the scenes, Tiago has continued to make outstanding work on the *pup* tool we use to create the installers for Windows 64/32 bit and MacOS X. This beta release will be the first to use installers created with *pup*.
- **Known bug** - on first ever start of Mu, if in Python3 mode the package manager will not work. Re-starting Mu fixes this (i.e. from second and subsequent starts). We're tracking this problem via [this issue](https://github.com/mu-editor/mu/issues/1358).

6.18.10 1.1.0-beta.1

This is a beta release and may contain bugs or unfinished features. Please provide bug reports or feedback via: https://github.com/mu-editor/mu/issues/new

- A new mode for ESP8266/ESP32 devices running MicroPython. This work and a significant amount of related refactoring was contributed with Viking like energy and efficiency by Martin Dybdal. This work has meant it was relatively easy to create two further new modes…
- New mode for Lego Spike devices (thanks to Chris and Ethan at Tufts University for the help and support).
- New mode for Raspberry Pi Pico (thanks to Zander, Martin and Carlos for the extensive testing).
- Updates to the Microbit mode made by Spanish source-code wrangler extraordinaire (and resident Microbit expert) Carlos Pereira Atencio. The Microbit mode now supports versions 1 and 2 of the board.
- Various bits of artwork used in the application have been updated (including a new [temporary] animated splash screen). Thanks to devdanzin for choreographing the initial work on the splash screen at short notice.
- A complete re-write of the virtualenv and third party package handlers by the hugely talented Tim Golden. This was a long term and difficult refactoring project which Tim has delivered with great aplomb. This should make package handling much smoother and simpler.
- Various smallish UI fixes, enhancements and smoothing by devdanzin. Thank you for these contributions - they really make a difference to the ease of use and friendly feel of Mu.
- This version of Mu is packaged with stand-alone installers for Windows and OSX by the wonder that is PUP - a new packaging tool by our very own Tiago Montes ~ Portugal's Premier Python Packager Par-excellence. We have big plans for PUP… watch this space. :-)
- Many many many minor bug fixes contributed by many many many people to whom we are eternally grateful.

We hope to release beta.2 very soon.

6.18.11 1.0.3

Bugfix.

- Updated to the latest version of Qt to fix syntax highlighting issues in OSX.
- Ensure CWD is set to the directory containing the script to be run in Python3 mode.
- Updated website with instructions in light of OSX changes.

6.18.12 1.1.0-alpha.2

The second alpha release of 1.1. This version may contain bugs and is unfinished (more new features will be arriving in alpha 3). Please provide bug reports or feedback via: https://github.com/mu-editor/mu/issues/new

- **NEW FEATURE** A brand new web mode for creating simple dynamic web applications with the Flask web framework. Currently users are able to edit Python, HTML and CSS files, run a local server and view their website in thier browser. We expect to add a deployment option thanks to PythonAnywhere by the time alpha 3 is released.
- **NEW FEATURE** A new Slovak translation of Mu thanks to Miroslav Biňas (GitHub user bletvaska).
- **ACHIEVEMENT UNLOCKED** Fixed a problematic bug where students got into a seemingly impossible loop because the auto-save feature encountered errors and got in the way of renaming a file. We are THRILLED TO BITS that the fix for this problem was contributed by Sean Tibor, a teacher from Fort Lauderdale, Florida.

Teachers coding the tools they use to teach has been a core aim for Mu, and Sean gets the gold medal (or perhaps a beer when I next see him) for unlocking this achievement.

- **RENAME** At the suggestion of Adafruit's Dan Halbert, the "Adafruit" mode has been renamed to "CircuitPython" mode to reflect the growing number of manufacturers who support CircuitPython. Many thanks to Benjamin Shockley for putting the work in to make this happen.
- **NEW DEVICES** Several new non-Adafruit boards have been added to the renamed CircuitPython mode. Many thanks to Shawn Hymel (SparkFun) and Gustavo Reynaga (Electronic Cats) for contributing these valuable changes.
- Add some new free-to-reuse image and sound assets for use in PyGameZero example games.
- Middle mouse wheel scrolling with the CTRL or CMD (on Mac) keys will zoom the UI in a consistent manner across all platforms.
- Minor documentation updates / corrections thanks to Luke Slevinsky.
- Refinement of the built-in educational libraries as we start to unbundle a slew of software from Mu's installer so users can install such packages from within Mu. Many thanks to the formidably talented Martin O'Hanlon for his help.
- PyGameZero mode will look for game assets relative to the location of the game file, rather than just within the user's workspace. Thanks to the evergreen Tim Golden for this helpful update.
- Minor corrections to the French localisation by GitHub user ogoletti.
- UI related convenience in the new ESP mode so that the current / most recent filesystem path is used when using the file copy pane. Many thanks (as always) to Martin Dybdal for his continued work on all things ESP related in Mu.
- A tidy up of the file save dialog so it uses Qt's built in dialog features. Thanks to Tiago Montes for being his usual awesome self.
- Tabs are restored on startup in the correct order. Once again, this is the work of Tiago Montes.
- The mechanism for generating the various installers and packages for Mu has been significantly refactored so that there is, if possible, always a single source for configuration information. The significant amount of effort to make this happen was, once again (again), contributed by Tiago Montes.
- Window size and location is also restored on startup. Tiago Montes, who implemented this change, has been **ON FIRE** during this development phase.
- A small (but important) change to the tool-tip for the sleep function found in MicroPython on the micro:bit has been submitted to the pedagogical legend and friend of Mu that is Dave Ames.
- A helpful message is now sent to the output pane when the graphical debugger starts in Python 3 mode. The Shakespeare like talents of long term Mu-tineer Steve Stagg are behind this Nobel-prize-worthy literary contribution.
- Re-add support for user defined syntax check overrides. Many thanks to Leroy Levin for making this happen..!
- Ensure that `pip` is updated while creating the Windows installers. Thanks to Yu Wang for making this change.
- Various minor updates and fixes to aid code readibility.

6.18.13 1.1.0-alpha.1

The first alpha release of 1.1. This version may contain bugs and is unfinished (more new features will be added in later alpha releases or, depending on feedback, we may change the behaviour of existing features). Please provide bug reports or feedback via: https://github.com/mu-editor/mu/issues/new

- **NEW FEATURE** Installation of third party packages from PyPI. Click on the cog icon to open the admin dialog and select the "Third Party Packages" tab.
- **NEW FEATURE** Code tidy via the wonderful code formatter Black. Click the new "Tidy" button to reformat and tidy your code so it looks more readable. If your code has errors, these will be pointed out. Many thanks to Black's creator and maintainer, Łukasz Langa, for this contribution.
- **NEW FEATURE** A new ESP8266 / ESP32 mode for working with these WiFi enabled cheap IoT boards. Many thanks to Martin Dybdal for driving this work forward and doing the heavy lifting. Thanks also to Murilo Polese for testing and very constructive input in the review stage of this feature.
- **OS CHANGE** Due to Qt's and Travis's lack of support, Mu will only run on Mac OS 10.12 and above.
- Ensure line-number margin is not too sensitive to inaccurate clicking from young coders trying to position the cursor at the beginning of the line. Thanks to Tiago Montes for this enhancement.
- Fix some typos in the French translation. Thank you to GitHub user @camillem.
- Fix a bug relating to Adafruit boards when a file on a board which is then unplugged is saved, Mu used to crash. Thanks to Melissa LeBlanc-Williams for the report of this problem.
- Fix problem with a missing newline at the end of a file. Thanks to Melissa LeBlanc-Williams for the eagle-eyes and fix.
- Fix for PYTHONPATH related problems on Windows (the current directory is now on the path when a script is run). Thanks to Tim Golden for this fix.
- Update to locale detection (use Qt's QLocale class). Thanks to Tiago Montes for making this happen.
- Fix bug relating to match selection of non-ASCII characters. Thank you to Tiago Montes for this work.
- Fixed various encoding related issues on OSX.
- Various minor / trivial bug fixes and tidy ups.

6.18.14 1.0.2

Another bugfix and translation release. No new features were added. Unless there are show-stoppers, the next release will be 1.1 with new features.

- Updated OSX to macOS, as per Apple's usage of the terms. Thanks Craig Steele.
- Updates and improvements to the Chinese translation. Thank John Guan.
- Improved locale detection on macOS. Many thanks to Tiago Montes.
- Cosmetic stripping of trailing spaces on save. Thanks to Tim Golden.
- Update PyQt version so pip installed Mu works with Python 3.5. Thanks to Carlos Pereira Atencio.
- Fix incorrect setting of dataTerminalReady flag. Thanks to GitHub user @wu6692776.
- Spanish language improvements and fixes by Juan Biondi, @yeyeto2788 and Carlos Pereira Atencio.
- Improvements and fixes to the German translation by Eberhard Fahle.
- Fix encoding bug on Windows which caused crashes and lost files. Many thanks to Tim Golden for this work.

- Keyboard focus loss when closing REPL is now fixed. Thanks again Tim Golden.
- More devices for Adafruit mode along with a capability to work with future devices which have the Adafruit vendor ID. Thanks to Limor Friend for this contribution.
- Fix a bug introduced in 1.0.1 where output from a child Python process was being truncated.
- Fix an off-by-one error when reading bytes from UART on MicroPython devices.
- Ensure zoom is consistent and remembered between panes and sessions.
- Ensure mu_code and/or current directory of current script are on Python path in Mu installed from the installer on Windows. Thanks to Tim Golden and Tim McCurrach for helping to test the fix.
- Added Argon, Boron and Xenon boards to Adafruit mode since they're also supported by Adafruit's CircuitPython.
- The directory used to start a load/save dialog is either what the user last selected, the current directory of the current file or the mode's working directory (in order of precedence). This is reset when the mode is changed.
- Various minor typo and bug fixes.

6.18.15 1.0.1

This is a bugfix and new translation release. No new features were added. The next release will be 1.1.0 with some new features.

- Added a German translation by René Raab.
- Added various new Adafruit boards, thanks Limor!
- Added a Vietnamese translation by GitHub user @doanminhdang.
- Fix bug in MicroPython REPL when dealing with colour escape sequences, thanks Martin Dybdal of Coding Pirates! Arrr.
- Ensured anyone trying to setup on an incompatible version of Python is given a friendly message explaining the problem. Thanks to the hugely talented René Dudfield for migrating this helpful function from PyGame!
- Added a Brasilian translation by Marco A L Barbosa.
- Added missing API docs for PyGameZero. Thanks to Justin Riley.
- Added a Swedish translation by Filip Korling.
- Fixes to various metadata configuration entries by Nick Morrott.
- Updated to a revised Chinese translation. Thanks to John Guan.
- Added the Mappa MUndi (roadmap) to the developer documentation.
- Added a Polish translation by Filip Kłębczyk.
- Fixes and enhancements to the UI to aid dyslexic users by Tim McCurrach.
- Updated to version 1.0.0.final for MicroPython on the BBC micro:bit. Many thanks to Damien George of the MicroPython project for his amazing work.
- Many other minor bugs caught and fixed by the likes of Zander and Carlos!

6.18.16 1.0.0

- Fix for font related issues in OSX Mojave. Thanks to Steve Stagg for spotting and fixing.
- Fix for encoding issue encountered during code checking. Thanks to Tim Golden for a swift fix.
- Fix for orphaned modal dialog. Thanks for spotting this Zander Brown.
- Minor revisions to hot-key sequences to avoid duplications. All documented at https://codewith.mu/en/tutorials/1.0/shortcuts.
- Update to latest version of uflash and MicroPython 1.0.0-rc.2 for micro:bit.
- Updated to latest GuiZero in Windows installers.
- Update third party API documentation used by QScintilla for code completion and call tips. Includes CircuitPython 3 and PyGame Zero 1.2.
- Added swag related graphics to the repository (non-functional change).

6.18.17 1.0.0.rc.1

- Various UI style clean ups to make sure the look of Mu is more consistent between platforms. Thanks to Zander Brown for this valuable work.
- Added French translation of the user interface. Thanks to Gerald Quintana.
- Added Japanese translation of the user interface. Thanks to @MinoruInachi.
- Added Spanish translation of the user interface. Thanks to Carlos Pereira Atencio with help from Oier Echaniz.
- Added Portuguese translation of the user interface. Thanks to Tiago Montes.
- Fixed various edge cases relating to the new-style flashing of micro:bits.
- Fixed off-by-one error in the visual debugger highlighting of code (caused by Windows newlines not correctly handled).
- Fixed shadow module related problem relating to Adafruit mode. It's now possible to save "code.py" files onto boards.
- Updated to latest version of uflash and MicroPython 1.0.0-rc.1 for micro:bit.
- Various minor bugs and niggles have been fixed.

6.18.18 1.0.0.beta.17

- Update to the latest version of uflash with the latest version of MicroPython for the BBC micro:bit.
- Change flashing the BBC micro:bit to become more efficient (based on the copying of files to the boards small "fake" filesystem, rather than re-flashing the whole device in one go).
- Ensure user agrees to GPL3 license when installing on OSX.
- Fix Windows "make" file to correctly report errors thanks to Tim Golden.
- The debugger in Python mode now correctly handles user-generated exceptions.
- The debugger in Python mode updates the stack when no breakpoints are set.
- Major update of the OSX based automated build system.
- Modal dialog boxes should behave better on GTK based desktops thanks to Zander Brown.

- Right click to access context menu in file panes in micro:bit mode so local files can be opened in Mu.
- Fix bug where REPL, Files and Plotter buttons got into a bad state on mode change.
- Update to use PyQt 5.11.
- On save, check for shadow modules (i.e. user's are not allowed to save code whose filename would override an existing module name).
- Automatic comment toggling via Ctrl-K shortcut.
- A simple find and replace diaolog is now available via the Ctrl-F shortcut.
- Various minor bugs and niggles have been squashed.

6.18.19 1.0.0.beta.16

- Updated flashing in micro:bit mode so it is more robust and doesn't block on Windows. Thank you to Carlos Pereira Atencio for issue #350 and the polite reminder.
- Updated the mu-debug runner so if the required filename for the target isn't passed into the command, a helpful message is displayed to the user.
- Developer documentation updates.
- Updated to the latest version of uflash, which contains the latest stable release of MicroPython for the micro:bit. Many thanks to Damien George for all his continuing hard work on MicroPython for the micro:bit.
- Inclusion of tkinter, turtle, gpiozero, guizero, pigpio, pillow and requests libraries as built-in modules.
- Update to latest version of Pygame Zero.
- Fix plotter axis label bug which wouldn't display numbers if value was a float.
- Separate session and settings into two different files. Session includes user defined changes to configuration whereas settings contains sys-admin-y configuration.
- Update the CSS for the three themes so they display consistently on all supported platforms. Thanks to Zander Brown for his efforts on this.
- Move the mode selection to the "Mode" button in the top left of the window.
- Support for different encodings and default to UTF-8 where possible. Many thanks to Tim Golden for all the hard work on this rather involved fix.
- Consistent end of line support on all platforms. Once again, many thanks to Tim Golden for his work on this difficult problem.
- Use `mu-editor` instead of `mu` to launch the editor from the command line.
- More sanity when dealing with cross platform paths and ensure filetypes are treated in a case insensitive manner.
- Add support for minification of Python scripts to be flashed onto a micro:bit thanks to Zander Brown's nudatus module.
- Clean up logging about device discovery (it's much less verbose).
- Drag and drop files onto Mu to open them. Thanks to Zander Brown for this *really useful* feature.
- The old logs dialog is now an admin dialog which allows users to inspect the logs, but also make various user defined configuration changes to Mu.
- Plotter now works in Python 3 mode.
- Fix problem in OSX with the `mount` command when detecting Circuit Python boards. Thanks to Frank Morton for finding and fixing this.

- Add data flood avoidance to the plotter.
- OSX automated packaging. Thanks to Russell Keith-Magee and the team at BeeWare for their invaluable help with this problematic task.
- Refactoring and bug fixing of the visual debugger's user interface. Thank you to Martin O'Hanlon and Carlos Pereira Atencio for their invaluable bug reports and testing.
- Various fixes to the way the UI and themes are displayed (crisper icons on HiDPI displays and various other fixes). Thanks to Steve Stagg for putting lipstick on the pig. ;-)
- A huge number of minor bug fixes, UI clean-ups and simplifications.

6.18.20 1.0.0.beta.15

- A new plotter works with CircuitPython and micro:bit modes. If you emit tuples of numbers via the serial connection (e.g. `print((1, 2, 3))` as three arbitrary values) over time these will be plotted as line graphs. Many thanks to Limor "ladyada" Fried for contributing code for this feature.
- Major refactoring of how Mu interacts with connected MicroPython based boards in order to enable the plotter and REPL to work independently.
- Mu has a new mode for Pygame Zero (version 1.1). Thanks to Dan Pope for Pygame Zero and Rene Dudfield for being Pygame maintainer.
- It's now possible to run mu "python3 -m mu". Thanks to Cefn Hoile for the contribution.
- Add support for pirkey Adafruit board. Thanks again Adafruit.
- Updated all the dependencies to the latest upstream versions.
- Various minor bug fixes and guards to make Mu more robust (although this will always be bugs!).

6.18.21 1.0.0.beta.14

- Add new PythonProcessPanel to better handle interactions with child Python3 processes. Includes basic command history and command editing.
- Move the old "run" functionality in Python3 mode into a new "Debug" button.
- Create a new "Run" button in Python3 mode that uses the new PythonProcessPanel.
- Automation of 32bit and 64bit Windows installers (thanks to Thomas Kluyver for his fantastic pynsist tool).
- Add / revise developer documentation in light of changes above.
- (All the changes mentioned above were supported by the Raspberry Pi Foundation – Thank you!)
- Update / add USB PIDs for Adafruit boards (thanks Adafruit for the heads up).
- Minor cosmetic changes.
- Additional test cases.

6.18.22 1.0.0.beta.13

- Fix to solve problem when restoring CircuitPython session when device is not connected.
- Fix to solve "data terminal ready" (DTR) problem when CircuitPython expects DTR to be set (and it isn't by default in Qt).
- Added initial work on developer documentation found here: http://mu.rtfd.io/
- Updates to USB PIDs for Adafruit boards.
- Added functionally equivalent "make.py" for Windows based developers.
- Major refactor of the micro:bit related "files" UI pane: it no longer blocks the main UI thread.

6.18.23 1.0.0.beta.12

- Update "save" related behaviour so "save as" pops up when the filename in the tab is double clicked.
- Update the debugger so the process stops at the end of the run.
- Ensure the current working directory for the REPL is set to mu_mode.
- Add additional documentation about Raspberry Pi related API.
- Update micro:bit runtime to lates MicroPython beta.
- Make a start on developer documentation.

6.18.24 1.0.0.beta.11

- Updated Python 3 REPL to make use of an out of process iPython kernel (to avoid problems with blocking Mu's UI).
- Reverted Save related functionality to prior behaviour.
- The "Save As" dialog for re-naming a file is launched when you click the filename in the tab associated with the code.

6.18.25 1.0.0.beta.10

- Ensured "Save" button prompts user to confirm (or replace) the filename of an existing file. Allows Mu to have something like "Save As".
- Updated to latest microfs library for working with the micro:bit's filesystem.
- Fixed three code quality warnings found by https://lgtm.com/projects/g/mu-editor/mu/alerts/?mode=list
- Updated API generation so the output is ordered (helps when diffing the generated files).
- Updated Makefile to create Python packages/wheels and deploy to PyPI.
- Explicit versions for packages found within install_requires in setup.py.
- Minor documentation changes.

6.18.26 1.0.0.beta.9

- Debian related packaging updates.
- Fixed a problem relating to how Windows stops the debug runner.
- Fixed a problem relating to how Windows paths are expressed that was stopping the debug runner from starting.

6.18.27 1.0.0.beta.8

- Updated splash image to reflect trademark usage of logos.
- Refactored the way the Python runner executes so that it drops into the Python shell when it completes.
- The debug runner now reports when it has finished running a script.

6.18.28 1.0.0.beta.7

- Update PyInstaller icons.
- Fix some tests that fail on older version of Python 3.
- Add scripts to extract API information from Adafruit and Python 3.
- Add generated API documentation to Mu so autosuggest and call tips have data.
- Ensure translation files are distributed.

6.18.29 1.0.0.beta.6

- Pip installable.
- Updated theme handling: day, night and high-contrast (as per user feedback).
- Keyboard shortcuts.

6.18.30 1.0.0.beta.*

- Added modes to allow Mu to be a general Python editor. (Python3, Adafruit and micro:bit.)
- Added simple visual debugger.
- Added iPython based REPL for Python3 mode.
- Many minor UI changes based on UX feedback.
- Many bug fixes.

6.18.31 0.9.13

- Add ability to change default Python directory in the settings file. Thanks to Zander Brown for the contribution. See #179.

6.18.32 0.9.12

- Change the default Python directory from `~/python` to `~/mu_code`. This fixes issue #126.
- Add instructions for installing PyQt5 and QScintilla on Mac OS.
- Update to latest version of uFlash.
- Add highlighting of search mathes.
- Check if the script produced is > 8k.
- Use a settings file local to the Mu executable if available.
- Fix bug with highlighting code errors in Windows.
- Check to overwrite an existing file on the micro:bit FS.
- Start changelog

6.19 GNU General Public License

Version 3, 29 June 2007 Copyright © 2007 Free Software Foundation, Inc <http://fsf.org>

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

6.19.1 Preamble

The GNU General Public License is a free, copyleft license for software and other kinds of works.

The licenses for most software and other practical works are designed to take away your freedom to share and change the works. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change all versions of a program–to make sure it remains free software for all its users. We, the Free Software Foundation, use the GNU General Public License for most of our software; it applies also to any other work released this way by its authors. You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for them if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs, and that you know you can do these things.

To protect your rights, we need to prevent others from denying you these rights or asking you to surrender the rights. Therefore, you have certain responsibilities if you distribute copies of the software, or if you modify it: responsibilities to respect the freedom of others.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must pass on to the recipients the same freedoms that you received. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

Developers that use the GNU GPL protect your rights with two steps: **(1)** assert copyright on the software, and **(2)** offer you this License giving you legal permission to copy, distribute and/or modify it.

For the developers' and authors' protection, the GPL clearly explains that there is no warranty for this free software. For both users' and authors' sake, the GPL requires that modified versions be marked as changed, so that their problems will not be attributed erroneously to authors of previous versions.

Some devices are designed to deny users access to install or run modified versions of the software inside them, although the manufacturer can do so. This is fundamentally incompatible with the aim of protecting users' freedom to change the software. The systematic pattern of such abuse occurs in the area of products for individuals to use, which is precisely where it is most unacceptable. Therefore, we have designed this version of the GPL to prohibit the practice for those products. If such problems arise substantially in other domains, we stand ready to extend this provision to those domains in future versions of the GPL, as needed to protect the freedom of users.

Finally, every program is threatened constantly by software patents. States should not allow patents to restrict development and use of software on general-purpose computers, but in those that do, we wish to avoid the special danger that patents applied to a free program could make it effectively proprietary. To prevent this, the GPL assures that patents cannot be used to render the program non-free.

The precise terms and conditions for copying, distribution and modification follow.

6.19.2 TERMS AND CONDITIONS

0. Definitions

"This License" refers to version 3 of the GNU General Public License.

"Copyright" also means copyright-like laws that apply to other kinds of works, such as semiconductor masks.

"The Program" refers to any copyrightable work licensed under this License. Each licensee is addressed as "you". "Licensees" and "recipients" may be individuals or organizations.

To "modify" a work means to copy from or adapt all or part of the work in a fashion requiring copyright permission, other than the making of an exact copy. The resulting work is called a "modified version" of the earlier work or a work "based on" the earlier work.

A "covered work" means either the unmodified Program or a work based on the Program.

To "propagate" a work means to do anything with it that, without permission, would make you directly or secondarily liable for infringement under applicable copyright law, except executing it on a computer or modifying a private copy. Propagation includes copying, distribution (with or without modification), making available to the public, and in some countries other activities as well.

To "convey" a work means any kind of propagation that enables other parties to make or receive copies. Mere interaction with a user through a computer network, with no transfer of a copy, is not conveying.

An interactive user interface displays "Appropriate Legal Notices" to the extent that it includes a convenient and prominently visible feature that **(1)** displays an appropriate copyright notice, and **(2)** tells the user that there is no warranty for the work (except to the extent that warranties are provided), that licensees may convey the work under this License, and how to view a copy of this License. If the interface presents a list of user commands or options, such as a menu, a prominent item in the list meets this criterion.

1. Source Code

The "source code" for a work means the preferred form of the work for making modifications to it. "Object code" means any non-source form of a work.

A "Standard Interface" means an interface that either is an official standard defined by a recognized standards body, or, in the case of interfaces specified for a particular programming language, one that is widely used among developers working in that language.

The "System Libraries" of an executable work include anything, other than the work as a whole, that **(a)** is included in the normal form of packaging a Major Component, but which is not part of that Major Component, and **(b)** serves only to enable use of the work with that Major Component, or to implement a Standard Interface for which an implementation is available to the public in source code form. A "Major Component", in this context, means a major essential component (kernel, window system, and so on) of the specific operating system (if any) on which the executable work runs, or a compiler used to produce the work, or an object code interpreter used to run it.

The "Corresponding Source" for a work in object code form means all the source code needed to generate, install, and (for an executable work) run the object code and to modify the work, including scripts to control those activities. However, it does not include the work's System Libraries, or general-purpose tools or generally available free programs which are used unmodified in performing those activities but which are not part of the work. For example, Corresponding Source includes interface definition files associated with source files for the work, and the source code for shared libraries and dynamically linked subprograms that the work is specifically designed to require, such as by intimate data communication or control flow between those subprograms and other parts of the work.

The Corresponding Source need not include anything that users can regenerate automatically from other parts of the Corresponding Source.

The Corresponding Source for a work in source code form is that same work.

2. Basic Permissions

All rights granted under this License are granted for the term of copyright on the Program, and are irrevocable provided the stated conditions are met. This License explicitly affirms your unlimited permission to run the unmodified Program. The output from running a covered work is covered by this License only if the output, given its content, constitutes a covered work. This License acknowledges your rights of fair use or other equivalent, as provided by copyright law.

You may make, run and propagate covered works that you do not convey, without conditions so long as your license otherwise remains in force. You may convey covered works to others for the sole purpose of having them make modifications exclusively for you, or provide you with facilities for running those works, provided that you comply with the terms of this License in conveying all material for which you do not control copyright. Those thus making or running the covered works for you must do so exclusively on your behalf, under your direction and control, on terms that prohibit them from making any copies of your copyrighted material outside their relationship with you.

Conveying under any other circumstances is permitted solely under the conditions stated below. Sublicensing is not allowed; section 10 makes it unnecessary.

3. Protecting Users' Legal Rights From Anti-Circumvention Law

No covered work shall be deemed part of an effective technological measure under any applicable law fulfilling obligations under article 11 of the WIPO copyright treaty adopted on 20 December 1996, or similar laws prohibiting or restricting circumvention of such measures.

When you convey a covered work, you waive any legal power to forbid circumvention of technological measures to the extent such circumvention is effected by exercising rights under this License with respect to the covered work, and you disclaim any intention to limit operation or modification of the work as a means of enforcing, against the work's users, your or third parties' legal rights to forbid circumvention of technological measures.

4. Conveying Verbatim Copies

You may convey verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice; keep intact all notices stating that this License and any non-permissive terms added in accord with section 7 apply to the code; keep intact all notices of the absence of any warranty; and give all recipients a copy of this License along with the Program.

You may charge any price or no price for each copy that you convey, and you may offer support or warranty protection for a fee.

5. Conveying Modified Source Versions

You may convey a work based on the Program, or the modifications to produce it from the Program, in the form of source code under the terms of section 4, provided that you also meet all of these conditions:

- **a)** The work must carry prominent notices stating that you modified it, and giving a relevant date.
- **b)** The work must carry prominent notices stating that it is released under this License and any conditions added under section 7. This requirement modifies the requirement in section 4 to "keep intact all notices".
- **c)** You must license the entire work, as a whole, under this License to anyone who comes into possession of a copy. This License will therefore apply, along with any applicable section 7 additional terms, to the whole of the work, and all its parts, regardless of how they are packaged. This License gives no permission to license the work in any other way, but it does not invalidate such permission if you have separately received it.
- **d)** If the work has interactive user interfaces, each must display Appropriate Legal Notices; however, if the Program has interactive interfaces that do not display Appropriate Legal Notices, your work need not make them do so.

A compilation of a covered work with other separate and independent works, which are not by their nature extensions of the covered work, and which are not combined with it such as to form a larger program, in or on a volume of a storage or distribution medium, is called an "aggregate" if the compilation and its resulting copyright are not used to limit the access or legal rights of the compilation's users beyond what the individual works permit. Inclusion of a covered work in an aggregate does not cause this License to apply to the other parts of the aggregate.

6. Conveying Non-Source Forms

You may convey a covered work in object code form under the terms of sections 4 and 5, provided that you also convey the machine-readable Corresponding Source under the terms of this License, in one of these ways:

- **a)** Convey the object code in, or embodied in, a physical product (including a physical distribution medium), accompanied by the Corresponding Source fixed on a durable physical medium customarily used for software interchange.
- **b)** Convey the object code in, or embodied in, a physical product (including a physical distribution medium), accompanied by a written offer, valid for at least three years and valid for as long as you offer spare parts or customer support for that product model, to give anyone who possesses the object code either **(1)** a copy of the Corresponding Source for all the software in the product that is covered by this License, on a durable physical medium customarily used for software interchange, for a price no more than your reasonable cost of physically performing this conveying of source, or **(2)** access to copy the Corresponding Source from a network server at no charge.
- **c)** Convey individual copies of the object code with a copy of the written offer to provide the Corresponding Source. This alternative is allowed only occasionally and noncommercially, and only if you received the object code with such an offer, in accord with subsection 6b.

- **d)** Convey the object code by offering access from a designated place (gratis or for a charge), and offer equivalent access to the Corresponding Source in the same way through the same place at no further charge. You need not require recipients to copy the Corresponding Source along with the object code. If the place to copy the object code is a network server, the Corresponding Source may be on a different server (operated by you or a third party) that supports equivalent copying facilities, provided you maintain clear directions next to the object code saying where to find the Corresponding Source. Regardless of what server hosts the Corresponding Source, you remain obligated to ensure that it is available for as long as needed to satisfy these requirements.
- **e)** Convey the object code using peer-to-peer transmission, provided you inform other peers where the object code and Corresponding Source of the work are being offered to the general public at no charge under subsection 6d.

A separable portion of the object code, whose source code is excluded from the Corresponding Source as a System Library, need not be included in conveying the object code work.

A "User Product" is either **(1)** a "consumer product", which means any tangible personal property which is normally used for personal, family, or household purposes, or **(2)** anything designed or sold for incorporation into a dwelling. In determining whether a product is a consumer product, doubtful cases shall be resolved in favor of coverage. For a particular product received by a particular user, "normally used" refers to a typical or common use of that class of product, regardless of the status of the particular user or of the way in which the particular user actually uses, or expects or is expected to use, the product. A product is a consumer product regardless of whether the product has substantial commercial, industrial or non-consumer uses, unless such uses represent the only significant mode of use of the product.

"Installation Information" for a User Product means any methods, procedures, authorization keys, or other information required to install and execute modified versions of a covered work in that User Product from a modified version of its Corresponding Source. The information must suffice to ensure that the continued functioning of the modified object code is in no case prevented or interfered with solely because modification has been made.

If you convey an object code work under this section in, or with, or specifically for use in, a User Product, and the conveying occurs as part of a transaction in which the right of possession and use of the User Product is transferred to the recipient in perpetuity or for a fixed term (regardless of how the transaction is characterized), the Corresponding Source conveyed under this section must be accompanied by the Installation Information. But this requirement does not apply if neither you nor any third party retains the ability to install modified object code on the User Product (for example, the work has been installed in ROM).

The requirement to provide Installation Information does not include a requirement to continue to provide support service, warranty, or updates for a work that has been modified or installed by the recipient, or for the User Product in which it has been modified or installed. Access to a network may be denied when the modification itself materially and adversely affects the operation of the network or violates the rules and protocols for communication across the network.

Corresponding Source conveyed, and Installation Information provided, in accord with this section must be in a format that is publicly documented (and with an implementation available to the public in source code form), and must require no special password or key for unpacking, reading or copying.

7. Additional Terms

"Additional permissions" are terms that supplement the terms of this License by making exceptions from one or more of its conditions. Additional permissions that are applicable to the entire Program shall be treated as though they were included in this License, to the extent that they are valid under applicable law. If additional permissions apply only to part of the Program, that part may be used separately under those permissions, but the entire Program remains governed by this License without regard to the additional permissions.

When you convey a copy of a covered work, you may at your option remove any additional permissions from that copy, or from any part of it. (Additional permissions may be written to require their own removal in certain cases when you modify the work.) You may place additional permissions on material, added by you to a covered work, for which you have or can give appropriate copyright permission.

Notwithstanding any other provision of this License, for material you add to a covered work, you may (if authorized by the copyright holders of that material) supplement the terms of this License with terms:

- **a)** Disclaiming warranty or limiting liability differently from the terms of sections 15 and 16 of this License; or
- **b)** Requiring preservation of specified reasonable legal notices or author attributions in that material or in the Appropriate Legal Notices displayed by works containing it; or
- **c)** Prohibiting misrepresentation of the origin of that material, or requiring that modified versions of such material be marked in reasonable ways as different from the original version; or
- **d)** Limiting the use for publicity purposes of names of licensors or authors of the material; or
- **e)** Declining to grant rights under trademark law for use of some trade names, trademarks, or service marks; or
- **f)** Requiring indemnification of licensors and authors of that material by anyone who conveys the material (or modified versions of it) with contractual assumptions of liability to the recipient, for any liability that these contractual assumptions directly impose on those licensors and authors.

All other non-permissive additional terms are considered "further restrictions" within the meaning of section 10. If the Program as you received it, or any part of it, contains a notice stating that it is governed by this License along with a term that is a further restriction, you may remove that term. If a license document contains a further restriction but permits relicensing or conveying under this License, you may add to a covered work material governed by the terms of that license document, provided that the further restriction does not survive such relicensing or conveying.

If you add terms to a covered work in accord with this section, you must place, in the relevant source files, a statement of the additional terms that apply to those files, or a notice indicating where to find the applicable terms.

Additional terms, permissive or non-permissive, may be stated in the form of a separately written license, or stated as exceptions; the above requirements apply either way.

8. Termination

You may not propagate or modify a covered work except as expressly provided under this License. Any attempt otherwise to propagate or modify it is void, and will automatically terminate your rights under this License (including any patent licenses granted under the third paragraph of section 11).

However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated **(a)** provisionally, unless and until the copyright holder explicitly and finally terminates your license, and **(b)** permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, you do not qualify to receive new licenses for the same material under section 10.

9. Acceptance Not Required for Having Copies

You are not required to accept this License in order to receive or run a copy of the Program. Ancillary propagation of a covered work occurring solely as a consequence of using peer-to-peer transmission to receive a copy likewise does not require acceptance. However, nothing other than this License grants you permission to propagate or modify any covered work. These actions infringe copyright if you do not accept this License. Therefore, by modifying or propagating a covered work, you indicate your acceptance of this License to do so.

10. Automatic Licensing of Downstream Recipients

Each time you convey a covered work, the recipient automatically receives a license from the original licensors, to run, modify and propagate that work, subject to this License. You are not responsible for enforcing compliance by third parties with this License.

An "entity transaction" is a transaction transferring control of an organization, or substantially all assets of one, or subdividing an organization, or merging organizations. If propagation of a covered work results from an entity transaction, each party to that transaction who receives a copy of the work also receives whatever licenses to the work the party's predecessor in interest had or could give under the previous paragraph, plus a right to possession of the Corresponding Source of the work from the predecessor in interest, if the predecessor has it or can get it with reasonable efforts.

You may not impose any further restrictions on the exercise of the rights granted or affirmed under this License. For example, you may not impose a license fee, royalty, or other charge for exercise of rights granted under this License, and you may not initiate litigation (including a cross-claim or counterclaim in a lawsuit) alleging that any patent claim is infringed by making, using, selling, offering for sale, or importing the Program or any portion of it.

11. Patents

A "contributor" is a copyright holder who authorizes use under this License of the Program or a work on which the Program is based. The work thus licensed is called the contributor's "contributor version".

A contributor's "essential patent claims" are all patent claims owned or controlled by the contributor, whether already acquired or hereafter acquired, that would be infringed by some manner, permitted by this License, of making, using, or selling its contributor version, but do not include claims that would be infringed only as a consequence of further modification of the contributor version. For purposes of this definition, "control" includes the right to grant patent sublicenses in a manner consistent with the requirements of this License.

Each contributor grants you a non-exclusive, worldwide, royalty-free patent license under the contributor's essential patent claims, to make, use, sell, offer for sale, import and otherwise run, modify and propagate the contents of its contributor version.

In the following three paragraphs, a "patent license" is any express agreement or commitment, however denominated, not to enforce a patent (such as an express permission to practice a patent or covenant not to sue for patent infringement). To "grant" such a patent license to a party means to make such an agreement or commitment not to enforce a patent against the party.

If you convey a covered work, knowingly relying on a patent license, and the Corresponding Source of the work is not available for anyone to copy, free of charge and under the terms of this License, through a publicly available network server or other readily accessible means, then you must either **(1)** cause the Corresponding Source to be so available, or **(2)** arrange to deprive yourself of the benefit of the patent license for this particular work, or **(3)** arrange, in a manner consistent with the requirements of this License, to extend the patent license to downstream recipients. "Knowingly relying" means you have actual knowledge that, but for the patent license, your conveying the covered work in a country, or your recipient's use of the covered work in a country, would infringe one or more identifiable patents in that country that you have reason to believe are valid.

If, pursuant to or in connection with a single transaction or arrangement, you convey, or propagate by procuring conveyance of, a covered work, and grant a patent license to some of the parties receiving the covered work authorizing

them to use, propagate, modify or convey a specific copy of the covered work, then the patent license you grant is automatically extended to all recipients of the covered work and works based on it.

A patent license is "discriminatory" if it does not include within the scope of its coverage, prohibits the exercise of, or is conditioned on the non-exercise of one or more of the rights that are specifically granted under this License. You may not convey a covered work if you are a party to an arrangement with a third party that is in the business of distributing software, under which you make payment to the third party based on the extent of your activity of conveying the work, and under which the third party grants, to any of the parties who would receive the covered work from you, a discriminatory patent license **(a)** in connection with copies of the covered work conveyed by you (or copies made from those copies), or **(b)** primarily for and in connection with specific products or compilations that contain the covered work, unless you entered into that arrangement, or that patent license was granted, prior to 28 March 2007.

Nothing in this License shall be construed as excluding or limiting any implied license or other defenses to infringement that may otherwise be available to you under applicable patent law.

12. No Surrender of Others' Freedom

If conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot convey a covered work so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not convey it at all. For example, if you agree to terms that obligate you to collect a royalty for further conveying from those to whom you convey the Program, the only way you could satisfy both those terms and this License would be to refrain entirely from conveying the Program.

13. Use with the GNU Affero General Public License

Notwithstanding any other provision of this License, you have permission to link or combine any covered work with a work licensed under version 3 of the GNU Affero General Public License into a single combined work, and to convey the resulting work. The terms of this License will continue to apply to the part which is the covered work, but the special requirements of the GNU Affero General Public License, section 13, concerning interaction through a network will apply to the combination as such.

14. Revised Versions of this License

The Free Software Foundation may publish revised and/or new versions of the GNU General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Program specifies that a certain numbered version of the GNU General Public License "or any later version" applies to it, you have the option of following the terms and conditions either of that numbered version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of the GNU General Public License, you may choose any version ever published by the Free Software Foundation.

If the Program specifies that a proxy can decide which future versions of the GNU General Public License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Program.

Later license versions may give you additional or different permissions. However, no additional obligations are imposed on any author or copyright holder as a result of your choosing to follow a later version.

15. Disclaimer of Warranty

THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

16. Limitation of Liability

IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MODIFIES AND/OR CONVEYS THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

17. Interpretation of Sections 15 and 16

If the disclaimer of warranty and limitation of liability provided above cannot be given local legal effect according to their terms, reviewing courts shall apply local law that most closely approximates an absolute waiver of all civil liability in connection with the Program, unless a warranty or assumption of liability accompanies a copy of the Program in return for a fee.

END OF TERMS AND CONDITIONS

6.19.3 How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively state the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

> <one line to give the program's name and a brief idea of what it does.> Copyright (C) <year> <name of author>
>
> This program is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.
>
> This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.
>
> You should have received a copy of the GNU General Public License along with this program. If not, see <http://www.gnu.org/licenses/>.

Also add information on how to contact you by electronic and paper mail.

If the program does terminal interaction, make it output a short notice like this when it starts in an interactive mode:

> <program> Copyright (C) <year> <name of author> This program comes with ABSOLUTELY NO WARRANTY; for details type 'show w'. This is free software, and you are welcome to redistribute it under certain conditions; type 'show c' for details.

The hypothetical commands *show w* and *show c* should show the appropriate parts of the General Public License. Of course, your program's commands might be different; for a GUI interface, you would use an "about box".

You should also get your employer (if you work as a programmer) or school, if any, to sign a "copyright disclaimer" for the program, if necessary. For more information on this, and how to apply and follow the GNU GPL, see <http://www.gnu.org/licenses/>.

The GNU General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Lesser General Public License instead of this License. But first, please read <http://www.gnu.org/philosophy/why-not-lgpl.html>.

6.20 Copyright Information

6.20.1 Mu Copyright

Mu, its source code and associated assets are copyright Nicholas H.Tollervey and others (those who have made significant contributions to Mu can be found in this list of *Mu's Developers*).

6.20.2 Notes on Image Copyright Status

All images used in Mu's developer documentation fall under Mu's copyright status, except for the following images sourced from third parties:

beautifully_useless.jpg

Permission was sought and obtained from Katerina Kamprani (the creator of the uncomfortable):

```
Hello Nicholas,

Thank you very much for asking and for your kind words.
As long as there is no commercial use of the image, and since you
mention my project, it is absolutely fine!
I am very happy the images help to prove a point!

Many greetings from Athens, Greece,
Katerina
```

circuit_playground.jpg

Permission was sought and obtained from Adafruit Industries, the source of the image:

```
yup! 100%! please use!

> Hi Folks,
>
> Is it OK to use a picture of a Circuit Playground Express taken
> from your website in the developer docs for Mu. Like this..?
>
> https://mu.readthedocs.io/en/latest/modes.html#adafruit-mode
```

pygame.png

Permission was sought and obtained from René Dudfield, the current core maintainer of Pygame:

```
public domain

Go for it! Feel free to do whatever weird(and not weird) things you
like with it.

It's a modification (by me) of a logo by Gareth Noyce, who also put
the logo files in public domain.

Gareth Noyce said of the logo files:
They're public domain but I'd like attribution if they're used
anywhere.  Just a "logo by Gareth Noyce" would do, but I won't be
complaining if people forget. :)'
```

python.png

This is a copy of the Python logo owned by the Python Software Foundation (PSF). Mu was originally written by a PSF Fellow on behalf of the PSF as part of the PSF's contribution to the BBC's micro:bit project. Furthermore, the PSF say of the use of the Python logo):

> "Projects and companies that use Python are encouraged to incorporate the Python logo on their websites, brochures, packaging, and elsewhere to indicate suitability for use with Python or implementation in Python. Use of the "two snakes" logo element alone, without the accompanying wordmark is permitted on the same terms as the combined logo.
>
> In general, we want the logo to be used as widely as possible to indicate use of Python or suitability for Python."

lego.png

This is a copped version of the Creative Commons licensed photograph taken by Jeff Eaton that can be found here.

pyboard.png

The MicroPython Logo is covered by the MIT license.

web.png

The Flask Logo has been released to the public domain.

PYTHON MODULE INDEX

m

INDEX

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

Z