Rust Macro

4gboframram

May 16, 2022

LIBRARY

CHAPTER

ONE

GETTING STARTED

1.1 Introduction

Many low-level languages have a concept of macros that allow users to not have to write the same code over and over again. In Python, however, there isn't such a feature.

Why would anyone want macros in Python?

- Python is a very dynamic language and can be slow at times because of the overhead of calling the same function over and over
- You might want to add inline parsing of a language like SQL to remove runtime overhead of parsing a literal
- Times you want to call a function before a script is run
- When you want to have access to an expression and its result at the same time in a safe way that doesn't have massive runtime overhead (eg. a testing library)

1.2 Library Features

- A convenient way to provide import-time token generation and substitution through macros inspired by the Rust programming language
- An import hook that uses no external or platform-specific dependencies that can provide said functionality on any modern and compliant Python implementation running on any system
- Tools for manipulating tokens and creating usable macros in a way that is more clear than the Python standard library
- A commandline utility that automatically runs the import hook

1.3 Usage

With `Rust Macro`, it is easy to use macros from another module. Just put `# __use_macros__({modules})` on the first line of the file, where `modules` is a comma-separated list of string literals that contain the names of the modules you want to import from.

Note: The macro modules are imported in the exact same way as the standard Python import system except for a major difference; all macros are automatically brought into the module's macro namespace.

Macros can be then called within that same file with `{name}!({tokens})`.

Example:

Listing 1: (file hello.py)

```python
# __use_macros__('rust_macro.builtins')

print(stringify!(Hello, World))
```

To run this file, there are 2 options. It can be run with `python3 -m rust_macro hello.py` or you can create another file to import the module from like so:

Listing 2: (file main.py)

```python
from rust_macro import ExpandMacros

with ExpandMacros():
    import hello
```

Then you can simply run this main file by running `python3 main.py`

Note: The commandline utility approach sets the module's name to `__main__` so that scripts can work properly.

Both approaches are equally valid and usable, but the direct import approach is more embedable

Within the `ExpandMacros` context manager, all imported modules that have `# __use_macros__(args)` at the beginning of the file will have all macros expanded. This goes until all children modules' macros are expanded or until an exception is raised.

1.3.1 Creating Your Own Macros

In a module that defines macros, a macro is nothing but a function that takes an `List[Token]` as a single parameter and either returns an `Iterable[Token]` or a `str`.

When a `str` is returned, that string is then tokenized. **It is not converted into a string literal.**

To be able to export macros, define a variable named `__macros__` in the module's namespace that contains a mapping of names to a callable.

Example:

```python
from rust_macro.util import Token
from typing import List

def macro(tokens: List[Token]) -> str:
    return "print('Hello, World!')"

__macros__ = {'macro': macro}
```

This file can then be used in an `# __use_macros__` statement and gives access to a macro named `macro`

1.4 Where to go Next?

- For advanced usage, check out the api documentation.

CHAPTER

TWO

HOW IT WORKS

Warning: This section contains advanced information about how the library works internally that the average user might not care about. If you don't care how the library worls, you can skip this section and read the API documentation.

2.1 `ExpandMacros`- The Main Entry Point

`ExpandMacros` is a context manager that has a single purpose - add the `MacroExpander` class to `sys.path_hooks` to hook imports and remove `MacroExpander` from`sys.path_hooks` when the user doesn't want the hook anymore. `ExpandMacros` does not contain any logic for hooking imports; it is all handled by `MacroExpander`

2.2 `MacroExpander` - The Import Hook Itself

Warning: It is not recommended to use this class directly unless you know exactly what you're doing.

- `MacroExpander` is a subclass of `importlib.abc.SourceLoader`. `SourceLoader` provides sensible default methods to load data from source code, but the method `get_data(path)` from `imporlib.abc.ResourceLoader` is the most important.
- `self.get_data(path)` checks the file if the file contains `# __use_imports__` and imports all of the modules. Then, it gathers all of the macros that are defined in each module's `__macros__`. If a module doesn't have `__macros__`, an Exception is raised. Then it calls `self.recursive_expand` on a tokenized version of the file's contents, which recursively calls `self.expand_macros`
- `self.expand_macros(code: MutableSequence[Token])` expands all of the macros in the file based on the keys in `self.macros`, calling the macro that processes the tokens, replacing the entire macro invokation with the result.
- And then some `importlib` magic then turns the code from `self.get_data()` into the final module

CHAPTER

THREE

API REFERENCE

The API is currently made up of 3 modules:

`rust_macro.hook` - The place where the magic happens

`rust_macro.util` - Utilities for working with tokens and creating macros

`rust_macro.builtins` - A bunch of useful default macros

Note: `rust_macro` exports all names from `rust_macro.util` and `rust_macro.hook`

3.1 rust_macro.hook

`class rust_macro.hook.`**`ExpandMacros`**

The main class that provides a context manager interface to the main import hook.

`__enter__``()` → rust_macro.ExpandMacros

Enables macro expansion on import

Returns self

`__exit__``(`*exception_type*, *exception_value*, *exception_traceback*, `/`) → None

Disables macro expansion on import

`exception rust_macro.hook.`**`MacroFindError`**`(`*msg: str*`)`

An Exception that is raised when a module does not define macros when another module expects it.

`exception rust_macro.hook.`**`MacroNotFoundError`**`(`*name: str*`)`

A subclass of `NameError` that is raised when a macro cannot be found in the current scope.

`class rust_macro.hook.`**`MacroExpander`**`(`*fullname: str*, *path: str*`)`

A subclass of `importlib.abc.SourceLoader` that is responsible for processing macros in modules and loading processed modules

Warning: Do not use this class unless you know exactly what you are doing. If you do use this class, then do not call any of its methods directly. **This class's interface may change at any time and without warning.** The only guarantee is the existence of the methods `MacroExpander.get_data` and `MacroExpander.get_filename` and the class being a subclass of `importlib.abc.SourceLoader`. This class may also be deprecated in the future.

`fullname: str = fullname`

The full name of the module that `self` is responsible for loading

`path: str = path`

The path of the module that `self` is responsible for loading

`macros: dict[str, Callable[[Iterable[Token]], Union[Iterable[Token], str]]] = {}`

The mapping of names to macros that `self` uses to expand macros

`add_macros`(*fullname: str*) → None:

Update `self`'s macro mapping with the contents of the module `{fullname}.__macros__`.

Parameters **`fullname`** (`str`) – The full name of the module to import macros from

Raises

- *`MacroFindError`* – if the module at `fullname` does not have a `__macros__` atribute
- **`ModuleNotFoundError`** – when the module path doesn't exist

`get_filename`(*fullname: str*) → str

Gets the path of the file that `self` is responsible for loading.

Returns `self.path`

`expand_macros`(*self*, *tokens: MutableSequence[*Token*]*) → MutableSequence[*Token*]

Expands all registered macros in the token list

raises MacroNotFoundError when there is an attempt to expand a macro that isn't defined in the current scope

`recursive_expand`(*self*, *code: MutableSequence[*Token*]*, *, *depth_limit: int = 50*) → MutableSequence[*Token*]

Recursively expands macros that are in the token list.

Raises

- *`MacroNotFoundError`* – when there is an attempt to expand a macro that isn't defined in the current scope
- **`MacroError`** – when the `depth_limit` is exceeded

`get_data`(*filename: str*) → str

Gets the source code for the final processed module.

Parameters **`filename`** (`str`) – the file path that is opened

3.2 rust_macro.util

class `rust_macro.util.`**`Token`**

Canonical tokenize.TokenInfo

A class that represents a token from the lexer. Iterables that yield these are taken in and returned by macros. This class is the same as `tokenize.TokenInfo` in the standard library.

`type`

Type int

The type of token. See the Python token module for all of the different options.

`string`

Type str

The text that the token contains

`start`

Type int

`end`

Type int

`line`

Type str

The line the token is located in

`property exact_type`

Type int

The exact type of token. See the Python token module for all of the different options.

`rust_macro.util.`**`splitargs`**(*tokens: Iterable[*Token*]*, *, *delimiter: str* = *','*) → List[List[*Token*]]

Splits a group of tokens into parts by a delimiter string

Example:

```
from rust_macro.util import tokenize_string, splitargs

tokens = tokenize_string("'Hello, World', Hello There")

args = splitargs(tokens, delimiter=',')

assert len(args) == 2
assert args[0][0].string == "'Hello, World'"
assert [i.string for i in args[1]] == ['Hello', 'There']
```

`rust_macro.util.`**`fix`**(*text: str*) → str

Fixes a some wonky text created by `tokenize.untokenize`

`rust_macro.util.`**`untokenize`**(*tokens: Iterable[*Token*]*) → str

Converts an interable of Tokens back into a string.

`rust_macro.util.`**`tokenize_string`**(*s: str*) → List[Token]:

Converts a string into its tokens

More may come soon!

3.3 rust_macro.builtins

CHAPTER FOUR

UTIL

- genindex
- modindex
- search

PYTHON MODULE INDEX

INDEX