TOOLS FOR WRITING CONSISTENT AND RELIABLE
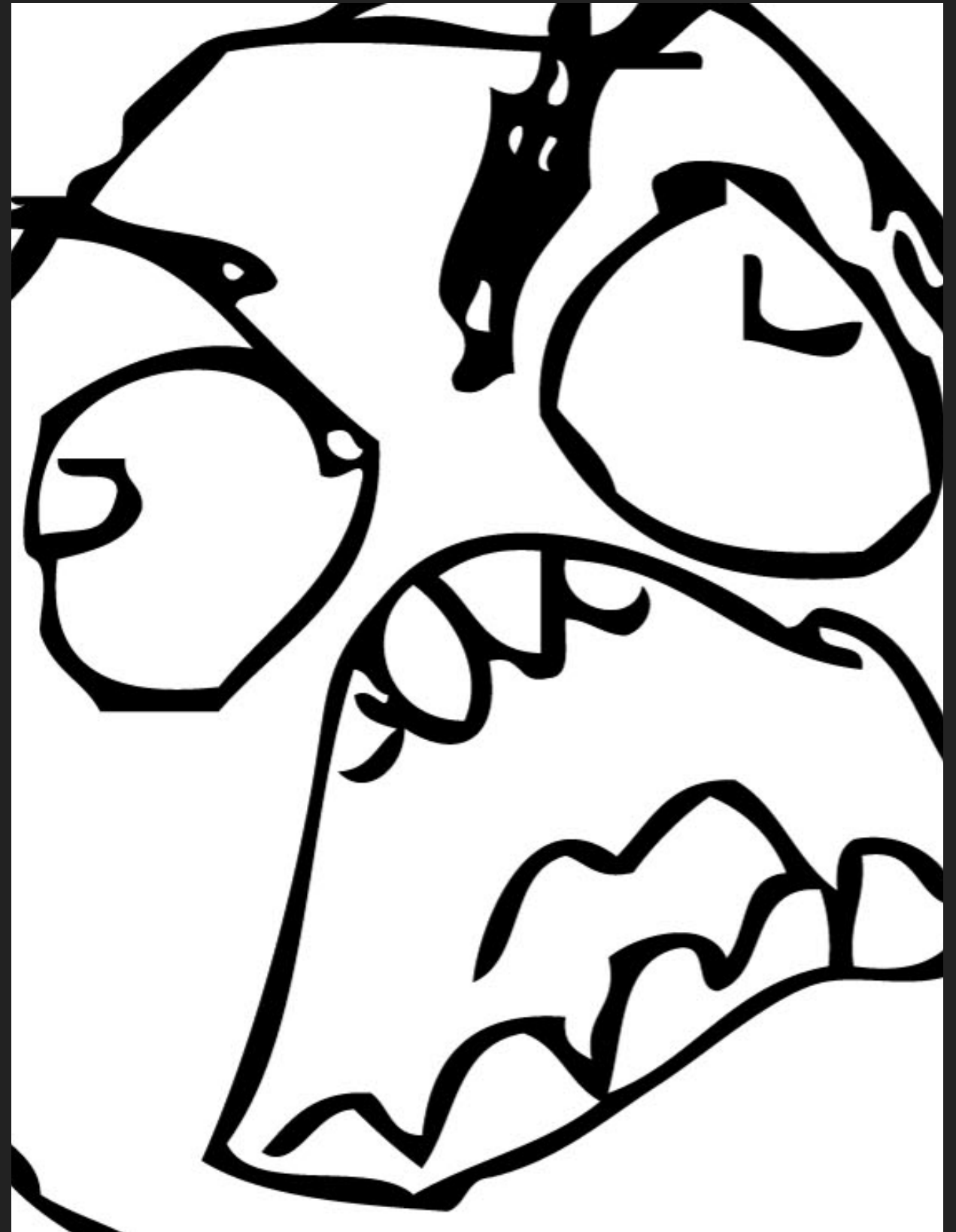
PYTHON CODE

TODAY WE'LL TALK ABOUT …

▸ Writing consistent Python code.

▸ Tools for enforcing code consistency.

▸ Type hints and `mypy` (trigger warning).

▸ Writing consistent docstrings.

▸ … will not talk about unit testing.

MOTIVATION

▸ Make your programming life easier.

▸ Inefficient workflow, wasted time on individual coding quirks.

▸ Simple tools can help you become a better programmer.

DISCLAIMER

▸ I am not an expert, just very interested.

▸ Which tools make sense in a data science context?

INCONSISTENCY IS THE HOBGOBLIN OF LITTLE MINDS.

PEP 8: Style Guide for Python Code

WHY SHOULD YOU CARE ABOUT CODING STYLE?

▸ The mental health of your future self.

▸ The mental health of your coworkers.

▸ Spend less time on formatting, more on logic.

PEP 8 – THE OFFICIAL STYLE GUIDE

▸ **P**ython **E**nhancement **P**roposal.

▸ PEP 8 formalises a recommended programming style.

▸ PEP 8 covers:

  ▸ Naming styles and conventions.

  ▸ Whitespace, maximum line lengths, indentations, trailing commas, and much more.

EXAMPLE: NAMING STYLES

Type	Example
Function/method	`function, my_function`
Variable	`variable, my_variable`
Class	`Class, MyClass`
Constant	`CONSTANT, MY_CONSTANT`
Module	`module.py, my_module.py`
Package	`package, mypackage`

PLEASE, PLEASE, PLEASE READ PEP 8

ENFORCING CONSISTENCY

▸ Memorise PEP 8? Ain't nobody got time for dat.

▸ Use linters!

▸ Linters check your code for logical errors and compliance with PEP 8.

▸ Most IDEs already use linting tools.

POPULAR PYTHON LINTERS

▸ `pylint`

  ▸ Very powerful.

  ▸ User-unfriendly defaults.

▸ `flake8`

  ▸ Wrapper around `pyflakes`, `pycodestyle` and `mccabe`.

  ▸ User-friendly defaults.

FLAKE8

▸ `pyflakes` checks your source files for errors. Does not complain about style.

▸ `pycodestyle` (formerly `pep8`) checks PEP 8 compliance.

▸ `mccabe` checks the complexity of your code (switched off by default).

EXAMPLE: USING FLAKE8

▸ Run `flake8` on `code_with_lint.py` through command line interface.

▸ Clone repository* to play with this example.

https://github.com/smu095/presentations/tree/master/python101

FLAKE8 OUTPUT

`{filename}:{line}:{column}:{error code} {message}`

‣ `W***`/`E***` are PEP 8 warnings (`pycodestyle`).

‣ `F***` are syntax errors (`pyflakes`).

‣ `C9**` are complexity errors (`mccabe`).

‣ Customisable, error messages can be ignored.

AUTOFORMATTING CODE

▸ Linters check your code, doesn't fix it.

▸ Autoformatters enforce consistency by <u>refactoring</u> your code.

▸ **Idea:** Avoid formatting arguments in code review, focus on logic.

BLACK – THE UNCOMPROMISING CODE FORMATTER

▸ `black` is an opinionated autoformatter that enforces a superset of PEP 8.

▸ Produces smallest diffs possible to make code review faster.

▸ Some stylistic deviations from PEP 8, e.g.

  ▸ maximum line length is 88 characters.

  ▸ all single quotes replaced by double quotes.

EXAMPLE: USING BLACK

▸ Run `black` on unformatted_code.`py` through command line interface.

▸ Clone repository* to play with this example.

https://github.com/smu095/presentations/tree/master/python101

SO FAR WE HAVE TALKED ABOUT…

▸ Community guidelines for coding style.

▸ Linters that check your code for errors.

▸ Autoformatters that refactor your code so you don't have to.

▸ There is a third linting tool, which relies on *type hints*.

STATIC VS. DYNAMIC TYPING

▸ Statically typed languages:

  ▸ Variables bound to both type and object.

  ▸ Types checked <u>before</u> code execution.

▸ Dynamically typed languages:

  ▸ Variables bound object.

  ▸ Types checked <u>during</u> code execution.

PYTHON IS DYNAMICALLY TYPED

▸ Objects are allowed to change types.

▸ Types are correctly inferred at runtime.

EXAMPLE: TYPES IN PYTHON

```
1  >>> if False:
2  ...     1 + "two"
3  ... else:
4  ...     1 + 2
5  3
6
7  >>> 1 + "two" # This will throw a TypeError
```

```
1  >>> thing = "Hello"
2  >>> type(thing)
3  <class 'str'>
4  >>> thing = 42
5  >>> type(thing)
6  <class 'int'>
```

STATIC TYPE CHECKING WITH MYPY

▸ `mypy` performs static type checking, i.e. code doesn't need to run to catch type errors.

▸ Needs your help, in the form of *type annotations*.

▸ Introduced in PEP 484, also called *type hints*.

▸ <u>Completely optional</u>, types are suggested but not enforced.

FUNCTION ANNOTATION

```python
def some_function(argument: type = default) -> type:
    do_something()
```

EXAMPLE: USING TYPE HINTS

▸ Run `mypy` on `type_hinting_example.py` through command line interface.

▸ Run `mypy` on `variable_hints.py` through command line interface.

▸ Clone repository* to play with this example.

https://github.com/smu095/presentations/tree/master/python101

SHOULD WE USE TYPE HINTING?

▸ Adds little value in short, throw-away scripts.

▸ Potential for a lot of value in bigger, collaborative projects.

TYPE HINTING YES-MEN

▸ More understandable, helps document code.

▸ More maintainable, easier to refactor.

▸ More reliable, catches bugs early.

TYPE HINTING NAY-SAYERS

▸ Requires time and effort.

▸ Introduces some overhead.

▸ Introduces unnecessary verbosity.

▸ Only available in Python 3.5+*.

* Works in earlier versions too, but with different syntax.

TYPE HINTING TAKEAWAYS

▸ Mildy controversial.

▸ <u>Completely optional</u>, Python will remain dynamically typed.

▸ More and more projects use type hinting and `mypy`, worth familiarising yourself with.

▸ Use your best judgement!
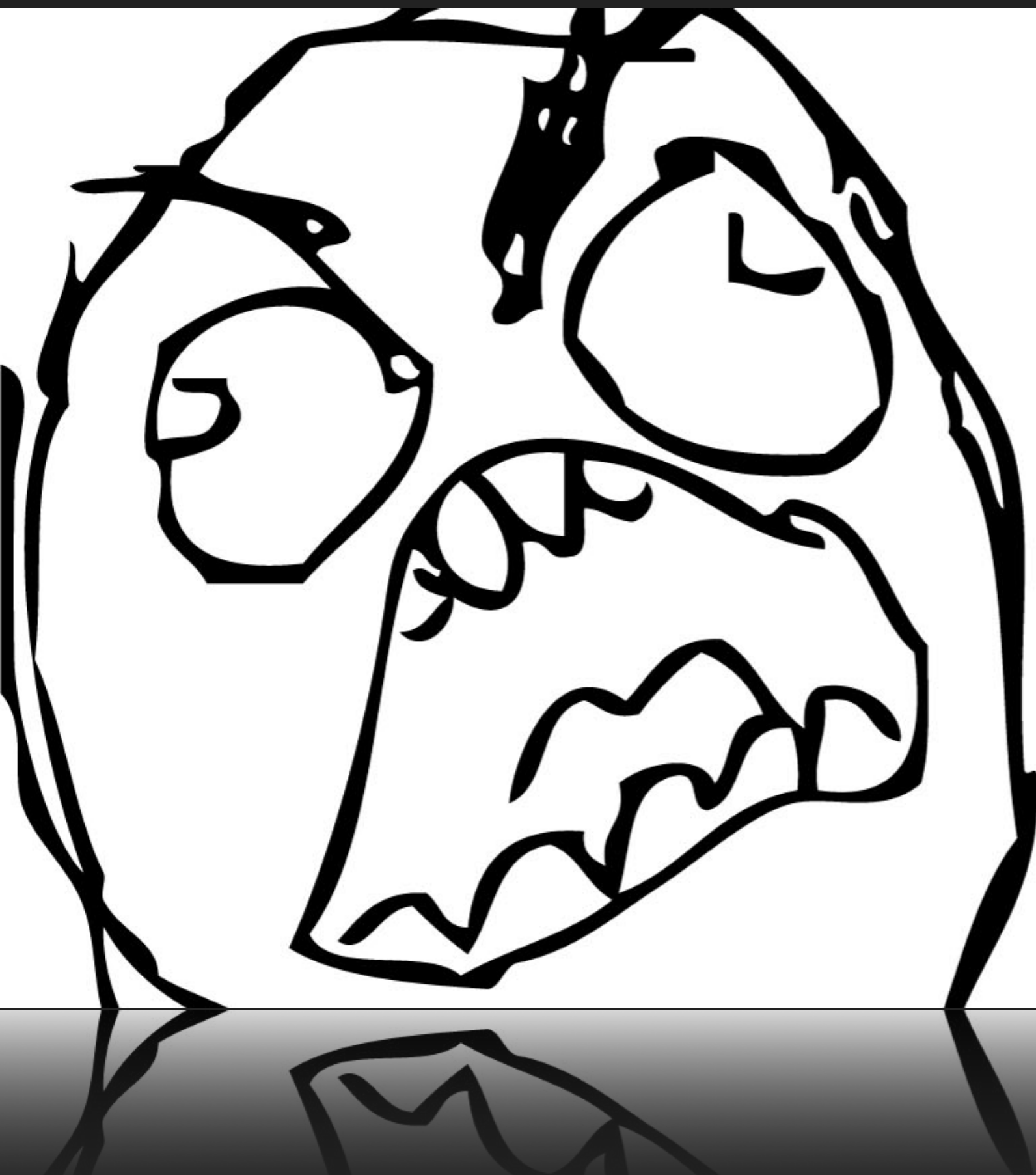
DOCSTRINGS

▸ **Problem:** Terribly documented code, took forever to figure out how it worked.

▸ PEP 257 formalises a style guide for docstrings.

▸ Docstrings come in different formats.

▸ Up to you, but <u>stay consistent</u> within your project.

A MINIMAL FUNCTION DOCSTRING

▸ Every docstring should at the very least contain:

  ▸ A one-liner summarising the function.

  ▸ A description of the parameters.

  ▸ A description of the return value.

▸ Surround by triple quotes, place immediately after function declaration.

EXAMPLE: UNDOCUMENTED FUNCTION

```python
1  def get_spreadsheet_cols(file_loc, print_cols=False):
2      file_data = pd.read_excel(file_loc)
3      col_headers = list(file_data.columns.values)
4
5      if print_cols:
6          print("\n".join(col_headers))
7
8      return col_headers
```

EXAMPLE: RESTRUCTURED TEXT

```
 1   """Gets and prints the spreadsheet's header columns
 2
 3   :param file_loc: The file location of the spreadsheet
 4   :type file_loc: str
 5   :param print_cols: A flag used to print the columns to the console
 6       (default is False)
 7   :type print_cols: bool
 8   :returns: a list of strings representing the header columns
 9   :rtype: list
10   """
```

EXAMPLE: GOOGLE DOCSTRINGS

```
1    """Gets and prints the spreadsheet's header columns
2
3    Parameters:
4        file_loc (str): The file location of the spreadsheet
5        print_cols (bool): A flag used to print the columns to the console
6            (default is False)
7
8    Returns:
9        list: a list of strings representing the header columns
10   """
```

EXAMPLE: NUMPY/SCIPY DOCSTRINGS

```
 1   """Gets and prints the spreadsheet's header columns
 2
 3   Parameters
 4   ----------
 5   file_loc : str
 6       The file location of the spreadsheet
 7   print_cols : bool, optional
 8       A flag used to print the columns to the console (default is False)
 9
10   Returns
11   -------
12   list
13       a list of strings representing the header columns
14   """
```

INCLUDING EXAMPLES

▸ The `numpy` docstring format strongly encourages examples in docstrings.

▸ Examples use the `docstring` module to parse docstrings for runnable examples of code.

▸ **Idea:** Make sure examples are running as shown, <u>not</u> intended as tests.

BUILDING DOCUMENTATION

▸ reStructuredText, Google and Numpy docstrings are widely used.

▸ Can produce well-formatted reference guides using tools like `sphinx` and `autodoc`.

PRACTICAL USE OF LINTING TOOLS

▸ Typically used in continuous integration pipeline.

```
37    # -------------------- script: run the build script --------------------------
38    before_script:
39      - pip install . # Install the package
40      - pip show treedoc # Show information about the package
41      - black . --check # Check that code is formatted correctly w.r.t. black
42      - flake8 treedoc --select=F401 --exclude=__init__.py # Unused imports
43      - bash ./linting.sh # Spelling errors
44      - mypy treedoc/*.py --ignore-missing-imports --show-error-context # Static type analysis
```

SUMMARY

▸ Read PEP 8.

▸ Follow the community's recommended style guide.

▸ Use linting and autoformatting tools when appropriate.

▸ Probably a good idea to familiarise yourself with type hints.

▸ Make a habit of writing good docstrings.

THE END