Tooling

Tooling

Tooling refers to software that helps one to develop other software. These include:

- tools for checking code style
- tools for *enforcing* code style
- tools for static type checking

The tools I'll talk about are all third-party tools available from Conda or PyPI (i.e., what `pip` downloads from). You can install all four with the following command:

```
conda install black isort flake8 mypy
```

Excursus: why do we care about code style?

- Code is written primarily to be read by humans, and only incidentally to be executed by machines.
  - Human development time is, in general, far more costly than computer run time.
- Your code may be read by not just "current day" you, but also:
  - "the man": managers, advisors, recruiters, funders, investors, lawyers
  - colleagues and collaborators
  - you, in the future
- "Good" style can be inscrutable; what we *really* want is a sensible, "consistent" style that minimizes the cognitive load of human readers (including future you).

Two types of tooling

- Some tools fix style violations
- Other tools simply point out style violations that we have to fix ourselves.

Usually we run the first kind of tool first, then the second.

PEP 8

PEP 8 is the official Python style guide.*

PEP 8 leaves some issues up to the developer, but more stringent style guides exist (e.g., the Google Python Style Guide).

Various tools enforce or check PEP 8 compliance:

- *reflowers* ([[re[flow]]er]s not [re[flower]]s) wrap long (>80 character) lines
- *linters* and *flakers* check for style violations

*PEP stands for "Python Enhancement Proposal". Anyone can write a PEP, but they have to be implemented, and then approved by a 2/3rd majority of the Python core developers before they're actually added to the language.

`black`

PEP 8 mandates that no line should be longer than 79 characters.*

`black` is a command-line tool that automatically wraps lines, in an opinionated (though, I think, quite nice looking) way.

```
$ black -l79 foo.py
reformatted foo.py
All done! ✨ 🍰 ✨
1 file reformatted.
```

`black` works "in-place": it modifies the files it is run on.

*This allows us to have multiple terminals or text editor windows side by side.

Imports formatting

PEP 8 mandates the following form for imports:

- Wildcard imports (e.g., `from foo import *`) should be avoided, as they make it unclear which names are present in the namespace.

```
import logging
import math
```

- Imports should be grouped in the following order:
  a. Standard library imports (e.g., `import math`)
  b. Related third party imports (e.g., `import pynini`, `from scipy import stats`)
  c. Local application/library specific imports (e.g., `from foo import bar`)

`isort`

`isort` is a command-line tool that automatically sorts imports according to the PEP 8 recommendation.

```
$ isort split.py
Fixing /home/kbg/split.py
```

Like `black`, `isort` works "in-place": it modifies the files it is run on.

`flake8`

`flake8` is a third-party command line tool that checks for, but does not fix, various style issues (including those proscribed by PEP 8):

```
$ flake8 foo.py
foo.py:2:5: F841 local variable 'y' is assigned to but
never used
```

I find that nearly all `flake8` issues are worth addressing. For instance, an assigned but unused variable (as above) is usually indicative of a bug.

Excursus: Python typing (1/)

PEP 484 added the ability to decorate Python code with type signatures to support

- (human-readable) documentation
- static type checking

though these may ultimately be used to accelerate Python code execution someday.

Excursus: Python typing (2/)

Types for ordinary variables, arguments to functions, and arguments to methods, are given after the identifier name, with a preceding colon:

```
bar: int = 3  # This is unnecessary but harmless.

def foo(bar: int): ...

def bar(fast: bool = False): ...
```

Excursus: Python typing (3/)

The return type of a function or method is given after the signature, preceded by an ASCII arrow (`->`) and followed by a colon:

```
def foo(bar: int) -> bool: ...
```

Excursus: Python typing (4/)

Types for instance variables (i.e., data stored within instances of a class) are given at the top of the class declaration:

```
class Puppy:
    name: str
    wet_nose: bool

    def __init__(self, name: str, wet_nose: bool = True):
        self.name = name
        self.wet_nose = wet_nose

    ...
```

Excursus: Python typing (5/)

Major types include:

- The placeholder type: `Any`
- Plain ole' data (POD) types: `bool`, `int`, `float`, `str`, `bytes`, `None`, etc.
- Polymorphic types: `Union[T, U]`, `Optional[T]` (= `Union[T, None]`)
- Containers: `Counter[K]`, `Dict[K, V]`, `List[T]`, `Tuple[T, U, ...]`, etc.
- Return type of generators: `Iterator[T]`
- Functions passed as arguments to other functions: `Callable`

Excursus: Python typing (6/)

As of Python 3.9, one no longer needs to write

```
from typing import List

def product(x: List[T]) -> T: ...
```

as one one can instead write

```
def product(x: list[T]) -> T: ...
```

`mypy`

Static type checking tools inspect code and confirm that it is consistent with all declared and/or inferred type signatures.

PEP 484 does not specify a static type checking tool, but one of the most widely used ones is the command-line tool `mypy`:

```
$ mypy foo.py
Success: no issues found in 1 source file
```

```
# Contents of: bar.py
def halve(x: int) -> int:
    return x / 2
```

```
$ mypy bar.py
bar.py:2: error: Incompatible return value type (got "float",
expected "int")
Found 1 error in 1 file (checked 1 source file)
```

Typing tips

- Add type signatures to all interfaces (functions, classes, and methods), but don't bother with simple variables unless `mypy` asks you to.
- Some third-party libraries do not yet have typing signatures. Add:

```
import pandas  # type: ignore
```

to silence `mypy` (etc.) warnings for that entire module.

Questions?