RSE Tips

Coding: Collaboration, Quality, Architecture, and Performance

Chad Baker, ETHZ/EAPS/GEG chad.baker@eaps.ethz.ch

Table of Contents

1. Collaborative Coding
2. Creating a New Python Project
3. Code Quality
4. Systems Modeling Architecture
5. Serialization/Deserialization
6. Assessing Performance
7. Rust Programming language

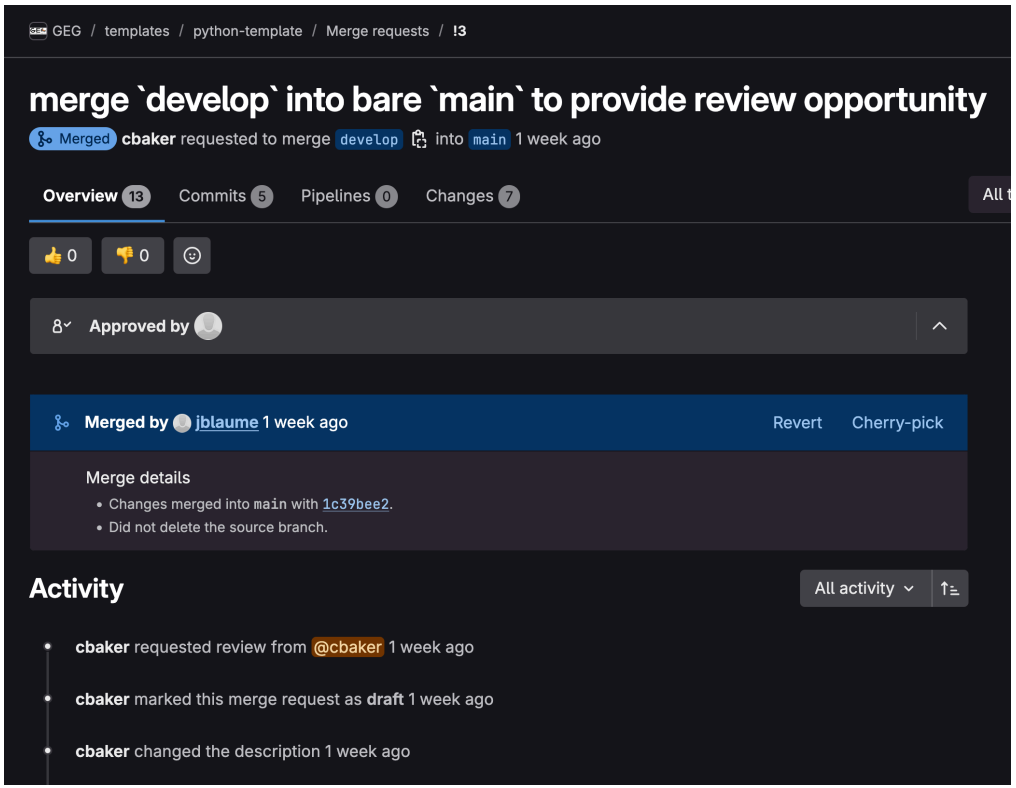
Overview

I will talk about things I learned from leading multiple teams to successfully develop scientific/ engineering software that is used by Google Maps, BNSF Railway, Wabtec (#1 US locomotive manufacturer), Toyota, Ford, Hyundai, ... I will also incorporate new concepts that we've implemented for the nGEO project.

This presentation was prepared with typst (source code), a new markup-based typesetting system for the sciences. It is designed to be an alternative both to advanced tools like LaTeX and simpler tools like Word and Google Docs. The goal of Typst is to build a typesetting tool that is highly capable and a pleasure to use.

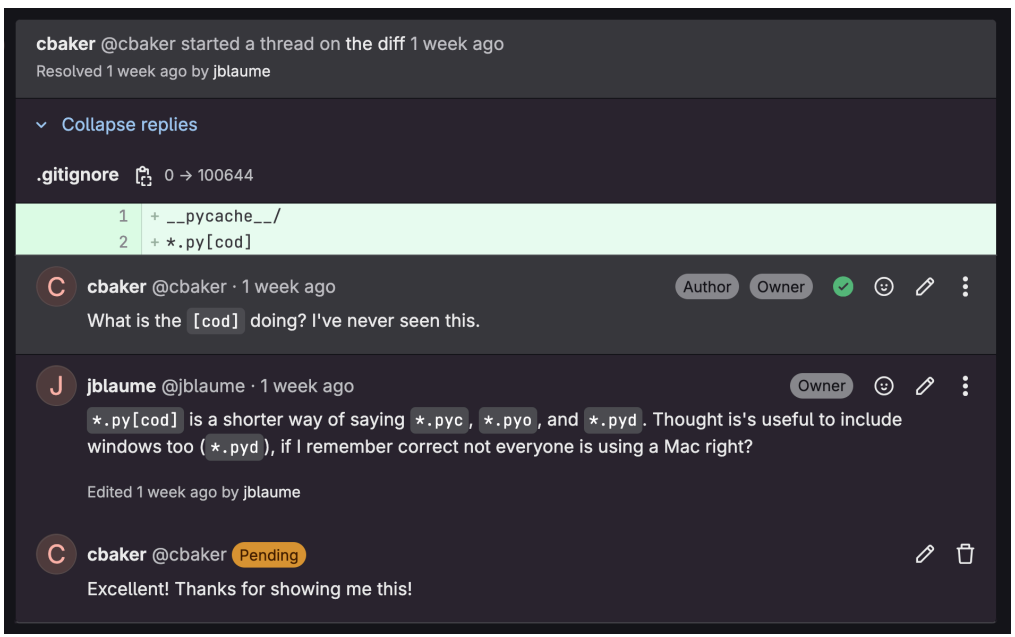
Collaborative Coding

- peer review is the most effective way to ensure high quality code!
- GitLab Merge Requests or GitHub Pull Requests are effective ways of creating a persistent record of peer review comments and responses



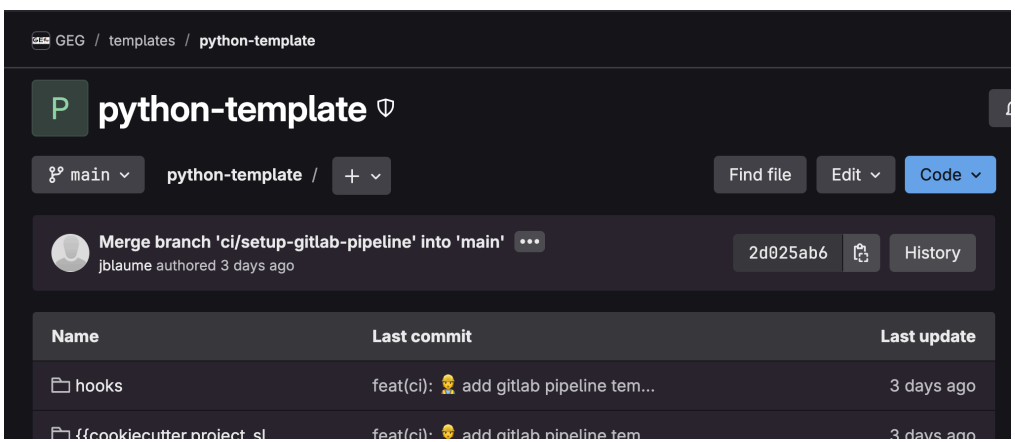
Collaborative Coding

- peer review is the most effective way to ensure high quality code!
- GitLab Merge Requests or GitHub Pull Requests are effective ways of creating a persistent record of peer review comments and responses



Creating a New Python Project

- the GEG Group maintains a cookiecutter python template on ETH-hosted GitLab, which provides a python project folder with reasonable, modifiable defaults for:
  - dependencies
  - testing
  - data pipelines
  - git hooks
- create a new Python project with the command `uvx --from cookiecutter cookiecutter git@gitlab.ethz.ch:geg/templates/package-template.git` (see README.md for other prerequisites)



Code Quality — Formatting

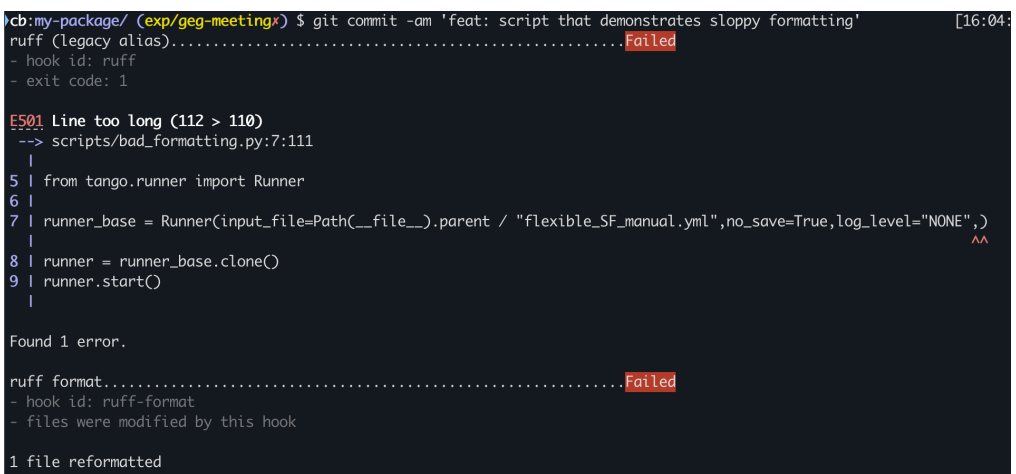
- code formatting and linting checks (e.g. with `ruff`) make it easier for reviewers (and yourself!) to read code
- this code block shows a line that is too long and has all the function args crammed together
- when we commit in our python-template repo, the commit fails and `ruff` fixes the formatting

Code Quality — Formatting

- code formatting and linting checks (e.g. with `ruff`) make it easier for reviewers to read code
- this code block shows a line that is too long and has all the function args crammed together
- when we commit in our python-template repo, the commit fails and `ruff` fixes the formatting

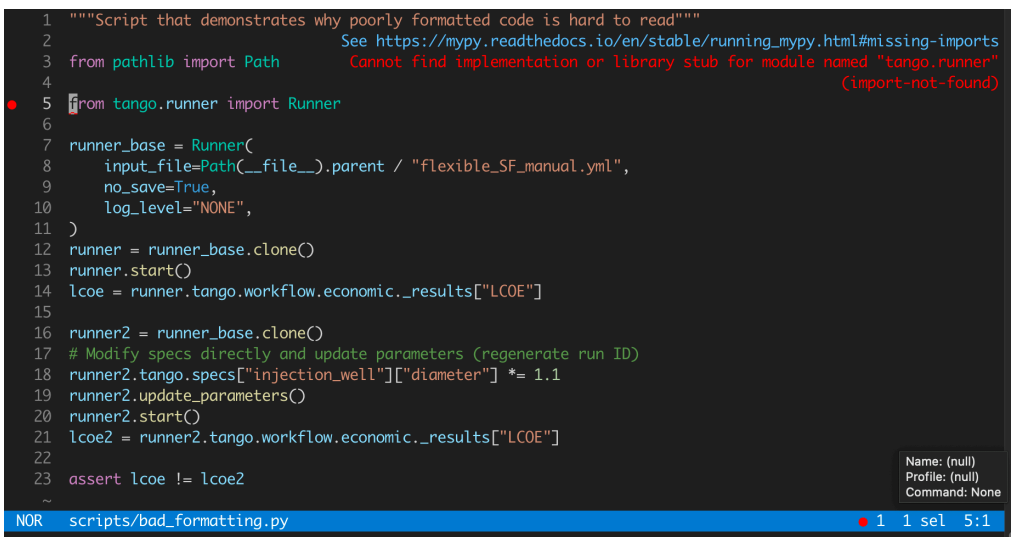
Code Quality — Formatting

- code formatting and linting checks (e.g. with `ruff`) make it easier for reviewers to read code
- this code block shows a line that is too long and has all the function args crammed together
- when we commit in our python-template repo, the commit fails and `ruff` fixes the formatting



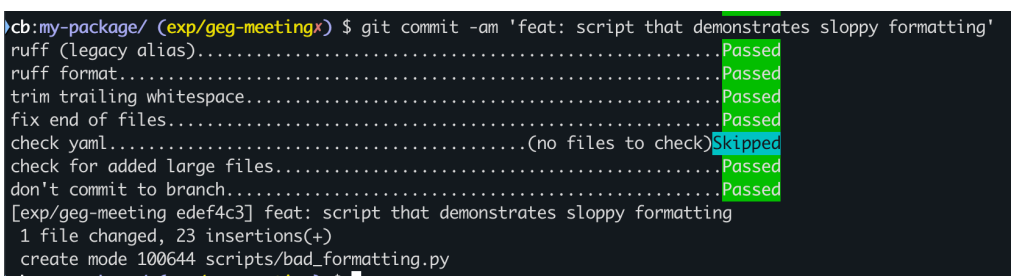
Code Quality — Formatting

- code formatting and linting checks (e.g. with `ruff`) make it easier for reviewers to read code
- this code block shows a line that is too long and has all the function args crammed together
- when we commit in our python-template repo, the commit fails and `ruff` fixes the formatting



Code Quality — Formatting

- code formatting and linting checks (e.g. with `ruff`) make it easier for reviewers to read code
- this code block shows a line that is too long and has all the function args crammed together
- when we commit in our python-template repo, the commit fails and `ruff` fixes the formatting



Code Quality — Type Checking

- type checking (e.g. with `mypy`) improves robustness, make it easier for other developers to contribute, makes it easier for users to understand expected attributes and function I/O
- this code block shows poorly typed class and function definitions
- this is the mypy output with `--strict` enabled in pyproject.toml
- this code block shows strongly typed class and function definitions

Code Quality — Type Checking

- type checking (e.g. with `mypy`) improves robustness, make it easier for other developers to contribute, makes it easier for users to understand expected attributes and function I/O
- this code block shows poorly typed class and function definitions
- this is the mypy output with `--strict` enabled in pyproject.toml
- this code block shows strongly typed class and function definitions

```
8
9
10  class FlowSolver(ABC):
11      """
12      Base class for flow solvers. Subclasses must implement solve_flow.
13
14      ... good luck figuring out what attributes you need or what goes in/out.
15      """
16
17      @abstractmethod
18      def solve_flow(self, conditions):
19          pass
20
21
22  class Wellbore(FlowSolver):
23
24      def __init__(self, params):
25          # Good luck knowing what keys are expected here without reading
26          # every line of solve_flow or asking the original author.
27          self.depth = params["depth"]
28          self.diameter = params["diameter"]
29          self.viscosity = params["viscosity"]
30          self.roughness = params["roughness"]
31
32      def solve_flow(self, conditions):
33          """
34          Solve for mass flow rate given inlet and outlet pressures.
35
36          Args:
37              conditions: dict with... some keys? check the implementation I guess
38
39          Returns:
40              dict with... something in it? run it and see
41          """
42          # Pull values out of the dict — no IDE completion, no type checking,
43          # a typo here silently returns None and blows up somewhere downstream
44          p_inlet = conditions["p_inlet"]
45          p_outlet = conditions["p_outlet"]
46          fluid_density = conditions["density"]
47
```

Code Quality — Type Checking

- type checking (e.g. with `mypy`) improves robustness, make it easier for other developers to contribute, makes it easier for users to understand expected attributes and function I/O
- this code block shows poorly typed class and function definitions
- this is the mypy output with `--strict` enabled in pyproject.toml
- this code block shows strongly typed class and function definitions

```
cb:my-package/ (exp/geg-meeting) $ uv run mypy src/my_package/bad_typing.py
src/my_package/bad_typing.py:16: error: Function is missing a type annotation  [no-untyped-def]
src/my_package/bad_typing.py:21: error: Function is missing a type annotation  [no-untyped-def]
src/my_package/bad_typing.py:29: error: Function is missing a type annotation  [no-untyped-def]
src/my_package/bad_typing.py:62: error: Call to untyped function "Wellbore" in typed context  [no-untyped-call]
src/my_package/bad_typing.py:69: error: Call to untyped function "solve_flow" in typed context  [no-untyped-call]
src/my_package/bad_typing.py:79: error: Function is missing a type annotation  [no-untyped-def]
src/my_package/bad_typing.py:82: error: Function is missing a type annotation  [no-untyped-def]
Found 7 errors in 1 file (checked 1 source file)
```

Code Quality — Type Checking

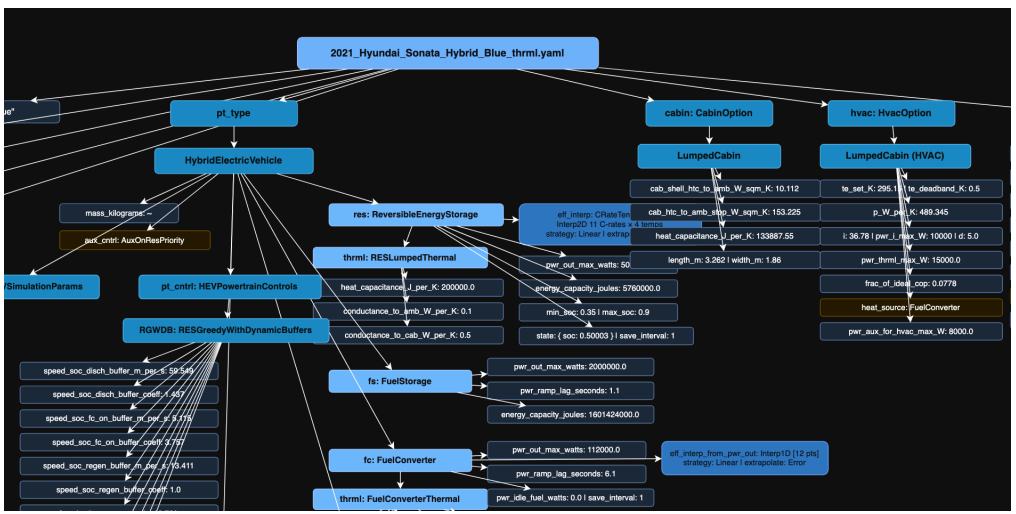
- type checking (e.g. with `mypy`) improves robustness, make it easier for other developers to contribute, makes it easier for users to understand expected attributes and function I/O
- this code block shows poorly typed class and function definitions
- this is the mypy output with `--strict` enabled in pyproject.toml
- this code block shows strongly typed class and function definitions

```python
@dataclass
class FlowBCs:
    """Boundary conditions for a flow solve."""
    p_inlet_pa: float       # Inlet pressure [Pa]
    p_outlet_pa: float      # Outlet pressure [Pa]
    fluid_density_kg_m3: float  # Fluid density [kg/m³]


@dataclass
class FlowResult:
    """Result of a flow solve."""
    mass_flow_rate_kg_s: float  # Mass flow rate [kg/s]


# ---------------------------------------------------------------------------
# Base class with explicit, validated attributes
# ---------------------------------------------------------------------------

class FlowSolver(BaseModel):
    """Abstract base for wellbore/pipe flow solvers.

    Any subclass needs at minimum these geometric and fluid properties.
    Pydantic enforces types and validators at construction time — you find
    out immediately if you passed depth in mm instead of m, not at hour 3
    of a debugging session.
    """
    depth_m: float = Field(..., gt=0, description="Wellbore/pipe length [m]")
    diameter_m: float = Field(..., gt=0, description="Inner diameter [m]")
    viscosity_pa_s: float = Field(..., gt=0, description="Dynamic viscosity [Pa·s]")
    roughness_m: float = Field(
        default=5e-5,
        ge=0,
        description="Absolute wall roughness [m], default ~commercial steel"
    )

    @field_validator("diameter_m")
    @classmethod
    def diameter_must_be_plausible(cls, v: float) -> float:
        """Verifies that diameter is within reasonable range"""
        if v > 1.0:
            raise ValueError(f"Diameter {v} m seems implausible for a wellbore — check units")
        return v

    @abstractmethod
    def solve_flow(self, conditions: FlowBCs) -> FlowResult:
        """Compute mass flow rate from inlet/outlet boundary conditions."""
        ...

```

Systems Modeling Architecture

At NREL, I used the following principles in developing models of physical systems:

- input files, code objects, module structure, and physical systems should have approximate congruence
- state variables should be stored in containers and locked to ensure updates once and only once per time step



Systems Modeling Architecture

At NREL, I used the following principles in developing models of physical systems:

- input files, code objects, module structure, and physical systems should have approximate congruence
- state variables should be stored in containers and locked to ensure updates once and only once per time step

```rust
#[derive(Default, PartialEq, Clone, Debug)]
/// Struct for storing state variable and ensuring one mutation per
/// initialization or reset -- i.e. one mutation per time step
pub struct TrackedState<T>(
    /// Value
    T,
    /// Update status
    StateStatus,
);

/// Provides methods to guarantee that states are updated once and only once per time step
impl<T> TrackedState<T>
where
    T: std::fmt::Debug + Clone + PartialEq + Default,
{
    pub fn new(value: T) -> Self {
        Self(value, Default::default())
    }

    fn is_fresh(&self) -> bool {
        self.1.is_fresh()
    }

    fn is_stale(&self) -> bool {
        self.1.is_stale()
    }

    /// # Arguments
    /// - `loc`: closure that returns file and line number where called
    fn ensure_fresh<F: Fn() -> String>(&self, loc: F) -> anyhow::Result<()> {
        ensure!(
            self.is_fresh(),
            format!(
                "{}\nState variable has not been updated. This is a bug in `fastsim-core`",
                loc()
            )
        );
        Ok(())
    }
```

Serialization/Deserialization

- by following the previously mentioned congruence principal, one is able to easily use off-the-shelf solutions for serialization/ deserialization (e.g. yaml, json, messagepack)
- provable round-trip equality
- easier to store recoverable, human-readable code objects

live demo in fastsim showing:

- round-trip equality
- yaml, json, and messagepack performance

Assessing Performance

- code profiling is a great way to check for bottlenecks!
- in `tango-core`, I ran `uv run python -m cProfile -o output.prof examples/run_flexible_SF_manual.py`
- then I ran `snakeviz output.prof` (after `brew install snakeviz`) to generate an interactive graphic

```python
"""Run flexible_SF_manual.py

This script was created for debuggging but also serves as an example of how to run Tango via script
"""

from pathlib import Path

from tango.runner import Runner

input_file = Path(__file__).parent / "flexible_SF_manual.yml"
runner_base = Runner(
    input_file=input_file,
    # save_dir=Path(__file__).parent / input_file.stem,
    no_save=True,
    log_level="NONE",
)
runner = runner_base.clone()
runner.start()
lcoe = runner.tango.workflow.economic._results["LCOE"]

runner2 = runner_base.clone()
# Modify specs directly and update parameters (regenerate run ID)
runner2.tango.specs["injection_well"]["diameter"] *= 1.1
runner2.update_parameters()
runner2.start()
lcoe2 = runner2.tango.workflow.economic._results["LCOE"]

assert lcoe != lcoe2
```

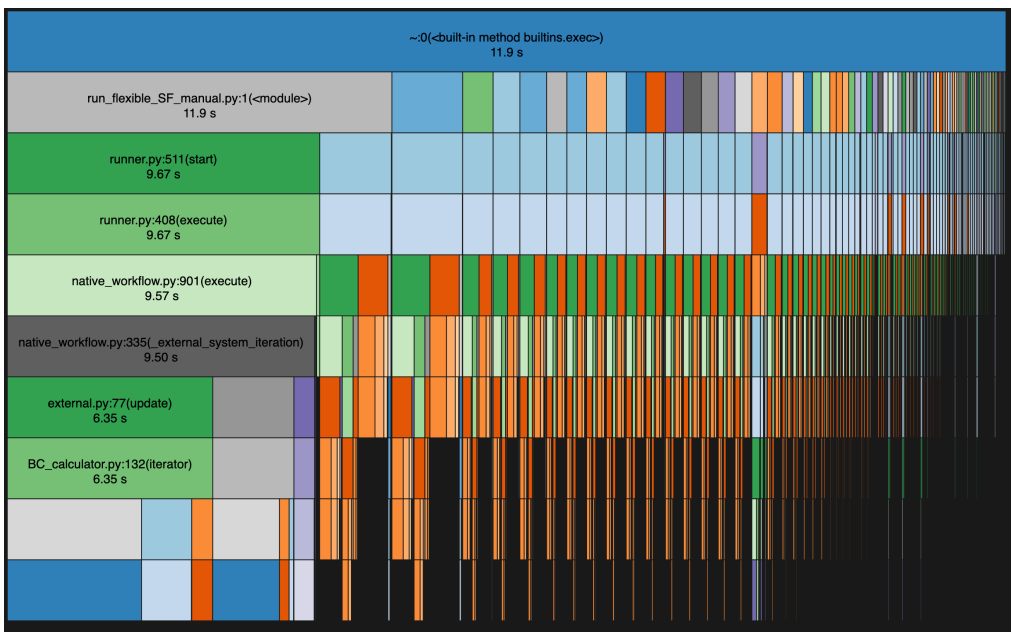
Assessing Performance

- code profiling is a great way to check for bottlenecks!
- in `tango-core`, I ran `uv run python -m cProfile -o output.prof examples/run_flexible_SF_manual.py`
- then I ran `snakeviz output.prof` (after `brew install snakeviz`) to generate an interactive graphic

```python
"""Run flexible_SF_manual.py

This script was created for debuggging but also serves as an example of how to run Tango via script
"""

from pathlib import Path

from tango.runner import Runner

input_file = Path(__file__).parent / "flexible_SF_manual.yml"
runner_base = Runner(
    input_file=input_file,
    # save_dir=Path(__file__).parent / input_file.stem,
    no_save=True,
    log_level="NONE",
)
runner = runner_base.clone()
runner.start()
lcoe = runner.tango.workflow.economic._results["LCOE"]

runner2 = runner_base.clone()
# Modify specs directly and update parameters (regenerate run ID)
runner2.tango.specs["injection_well"]["diameter"] *= 1.1
runner2.update_parameters()
runner2.start()
lcoe2 = runner2.tango.workflow.economic._results["LCOE"]

assert lcoe != lcoe2
```

Assessing Performance

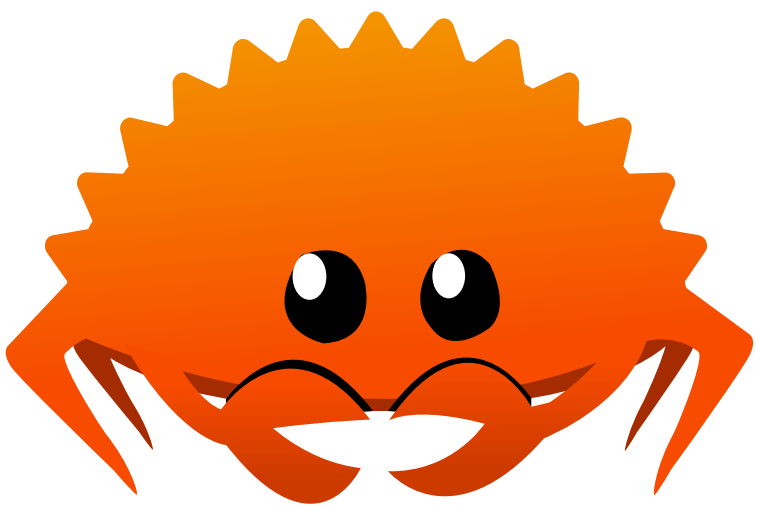
- code profiling is a great way to check for bottlenecks!
- in `tango-core`, I ran `uv run python -m cProfile -o output.prof examples/run_flexible_SF_manual.py`
- then I ran `snakeviz output.prof` (after `brew install snakeviz`) to generate an interactive graphic



Rust Programming language

Rust is a compiled language that:

- has a de facto compiler
- has a de facto package manager
- has great documentation
- has built in memory safety
- is, in my experience, relatively easy for python devs to learn
- is fun!



Using Rust Under the Hood for Python

Oxidize (Rust nerd jargon for port to Rust) your code when:

- robustness is important
  - your code will need to be persistent longer than a few months.

    Python tends to break when you don't actively touch it regularly.
  - you have more than 1,000 lines of code
  - lots of other people will be using your code
  - the compiler verifies that all code in a rust package (aka "crate") is valid
- your code needs to be performant

live demo showing performance boost in Rust in FASTSim