On programming and code quality

Ask Hjorth Larsen

CAMD, DTU

December 1, 2022

In this talk

- Introduction to doing Python projects
- Testing and pytest
- Refactoring, antipatterns, and code smells
- Work of the Response Code Focus Group

Coding quality and practices

What is code quality about?

- Producing reliable code
- Producing code that it is easy to adapt and extend
- Producing code that can be maintained for many years
- Changing code to improve quality without changing functionality falls under the umbrella of refactoring

What is code quality not about?

- Code quality is *not at all* about “style” nor “looking good” (although those things can help)

How to run high-throughput projects

- Write code to do a computation
- Run some materials with cheap parameters
- Run a few materials with good parameteres
- Be ready to delete all data and redo everything at any time for any reason! Automate any steps necessary for this to be easy.
- Keep code in version control!
- Make sure the whole chain works: Code, database collection, web panels, ...
- Write tests which would fail if any of this did not work
- Get feedback on code (frequently)
- Iterate the all the steps above as necessary until everything works and everyone agrees that things are good
- Only at the end: Submit thousands of materials with production quality parameters

Doing projects in Python

Basic steps

- Create project on gitlab/github
- Create setup.py to allow installation with pip
- Write code inside importable modules
- Have a test suite

Effectively: Make sure the code (actually: the project) lives in a well-defined place with version control and a basic level of documentation; keep dangling scripts all over Niflheim to a minimum.

On testing

- Code becomes more complex over time as new functionality is added
- A test is a bit of code which would fail (raise an error) if something does not work as expected
- Murphy's law: Everthing that can go wrong will go wrong
- Specifically: The probability that an untested feature still works decreases exponentially as changes are made
- A test suite prevents exponential degeneration of features
- The goal of a test suite is to fail whenever something doesn't work

```
askhl@noether:~/src/gpaw/gpaw/test/response$ pytest --collectonly
============================= test session starts ==============================
platform linux -- Python 3.9.2, pytest-7.2.0, pluggy-1.0.0
rootdir: /home/askhl/src/gpaw, configfile: pytest.ini
plugins: forked-1.4.0, cov-3.0.0, anyio-3.6.2, xdist-3.0.2, instafail-0.4.2, xvfb-2.0.0, mock-3.8.2
collected 88 items

<Package response>
  <Module test_afm_hchain_sf_gssALDA.py>
    <Function test_response_afm_hchain_gssALDA>
  <Module test_aluminum_EELS_ALDA.py>
    <Function test_response_aluminum_EELS_ALDA>
  <Module test_aluminum_EELS_RPA.py>
    <Function test_response_aluminum_EELS_RPA>
  <Module test_au02_absorption.py>
    <Function test_response_au02_absorption>
  <Module test_bse_MoS2_cut.py>
    <Function test_response_bse_MoS2_cut>
  <Module test_bse_aluminum.py>
    <Function test_response_bse_aluminum>
  <Module test_bse_silicon.py>
    <Function test_response_bse_silicon>
  <Module test_chi0.py>
    <Function test_response_chi0>
  <Module test_chi0_intraband_test.py>
```

Writing a test

- Set up initial conditions
- Run code to be tested
- Verify that results are correct

A unit test with pytest

```
def test_isolation_2D():
    atoms = ase.build.mx2(formula='MoS2', kind='2H', a=3.18, thickness=3.19)
    atoms.cell[2, 2] = 7
    atoms.set_pbc((1, 1, 1))
    atoms *= 2

    result = isolate_components(atoms)
    assert len(result) == 1
    key, components = list(result.items())[0]
    assert key == '2D'
    assert len(components) == 2
    for layer in components:
        empirical = atoms.get_chemical_formula(empirical=True)
        assert empirical == layer.get_chemical_formula(empirical=True)
        assert (layer.pbc == [True, True, False]).all()
```

Author: Peter Mahler

Another unit test

```
a = 4.1


@pytest.fixture
def atoms():
    atoms = bulk("Au", a=a)
    return atoms


def test_supercell_issue_938(atoms):
    assert atoms.cell.get_bravais_lattice().name == "FCC"

    # Since FCC and BCC are reciprocal, their product is cubic:
    P = BCC(2.0).tocell()

    # let P have negative determinant, make_supercell should not blow up
    P[0] *= -1
    assert np.allclose(np.linalg.det(P), -4)

    cubatoms = make_supercell(atoms, P)
    assert np.allclose(cubatoms.cell, a * np.diag((-1, 1, 1)))
    assert np.allclose(len(cubatoms), 4)
```

Test uses a “fixture”, initialization/cleanup code that can be shared among multiple tests. Author: Florian Knoop

Concluding remarks on testing

- If your code should survive, then write tests
- Learn pytest
- Many small tests are better than few big tests
- Tests should execute quickly

(This is easy to say. It's not trivial to test complex projects and it takes time to learn how to write good tests.)

Implement and test with minimum coupling

- You want to implement something new.
- Do not go to line 637 of somemodule.py and implement it there in the middle of everything. You'd be forced to call unwanted code in order to test.
- Instead, implement it in a standalone module. What is the input, and what is the output? Test it (and commit the test).
- Then integrate with the rest of the code.

Book: Clean code by Robert C. Martin

The Total Cost of Owning a Mess

If you have been a programmer for more than two or three years, you have probably been significantly slowed down by someone else's messy code. If you have been a programmer for longer than two or three years, you have probably been slowed down by messy code. The degree of the slowdown can be significant. Over the span of a year or two, teams that were moving very fast at the beginning of a project can find themselves moving at a snail's pace. Every change they make to the code breaks two or three other parts of the code. No change is trivial. Every addition or modification to the system requires that the tangles, twists, and knots be "understood" so that more tangles, twists, and knots can be added. Over time the mess becomes so big and so deep and so tall, they can not clean it up. There is no way at all.

Keep it simple

- What is a reasonable minimum input to compute a DOS?
  - Energies and weights
  - Not: A "gpw file"
- What is a reasonable minimum input for a band structure?
  - Energies and a band path
  - Not: A "gpw file"
- What is a reasonable minimum input for G_0W_0?
  - Well......
  - Not: 30 arguments including a "gpw file"

The "wrong" answers have one thing in common: They depend on unnecessary infrastructure. The feature is taken "hostage" by the necessity to provide the unnecessary infrastructure and hence ceases to be reusable.

Implement code in isolation and test it. *Then* add convenient wrappers to integrate with calculators, files, etc.

Take-home message: Don't allow your low-level processing to be taken hostage by non-essential infrastructure.

"How to write unmaintainable code"

By Roedy Green

General Principles

> *Quidquid latine dictum sit, altum sonatur.*
> - Whatever is said in Latin sounds profound.

To foil the maintenance programmer, you have to understand how he thinks. He has your giant program. He has no time to read it all, much less understand it. He wants to rapidly find the place to make his change, make it and get out and have no unexpected side effects from the change.

He views your code through a toilet paper tube. He can only see a tiny piece of your program at a time. You want to make sure he can never get at the big picture from doing that. You want to make it as hard as possible for him to find the code he is looking for. But even more important, you want to make it as awkward as possible for him to safely **ignore** anything.

On duplication

- "The cardinal rule of writing unmaintainable code is to specify each fact in as many places as possible and in as many ways as possible."
  — Roedy Green, *How to write unmaintainable code*
- In how many places do we hardcode "PBE", "gs.gpw", ...?
- In how many places do we redundantly check (or forget to check) that `all(pbc == [True, True, False])` because we don't have normalization layer?
- Don't duplicate code
- Don't duplicate functionality by effectively solving the same problem twice, either, even if code would be different
- Instead, look for an abstraction good enough to solve the problem well

Refactoring and code smells

When should you refactor? From *Refactoring: Improving the Design of Existing Code* by Martin Fowler *et al.*

I was mulling over this tricky issue when I visited Kent Beck in Zurich. Perhaps he was under the influence of the odors of his newborn daughter at the time, but he had come up with the notion describing the "when" of refactoring in terms of smells. "Smells," you say, "and that is supposed to be better than vague aesthetics?" Well, yes. We look at lots of code, written for projects that span the gamut from wildly successful to nearly dead. In doing so, we have learned to look for certain structures in the code that suggest (sometimes they scream for) the possibility of refactoring. (We are switching over to "we" in this chapter to reflect the fact that Kent and I wrote this chapter jointly. You can tell the difference because the funny jokes are mine and the others are his.)

Code smells

- Duplicated code
- Long function/method
- Long parameter list
- Shotgut surgery (changes necessary all over the place, not in one place)
- Feature envy (overuse features of other class)
- Switch statements (long/repeated if/else chains)
- Inappropriate intimacy (accessing implementation details of other class)

Invisible information passing

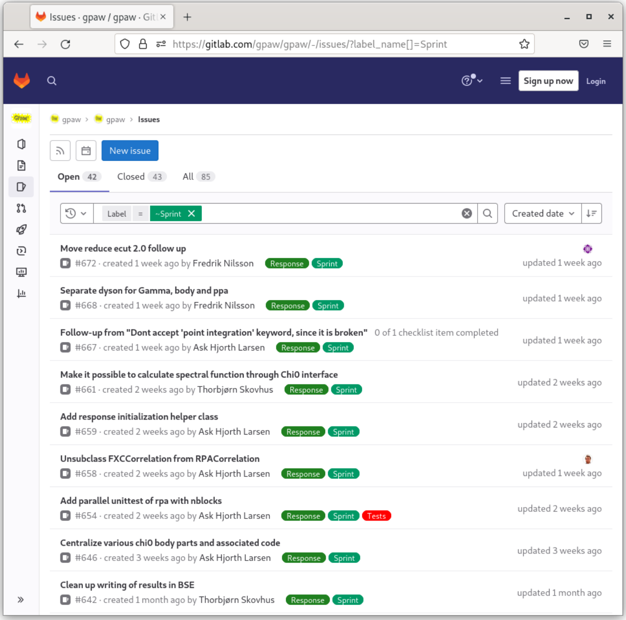
- Function 1 writes a file with a particular name
- Function 2 expects this file to exist after calling Function 1
- Problem: Both modules must redundantly implement the naming scheme for this to work. If either changes, code is broken.
- Solution 1: Function 1 takes target filename as an input, so Function 2 can choose what file is written
- Solution 2: Function 1 returns the path of the file it wrote, so Function 2 needs not reconstruct the name

Examples of this anti-pattern

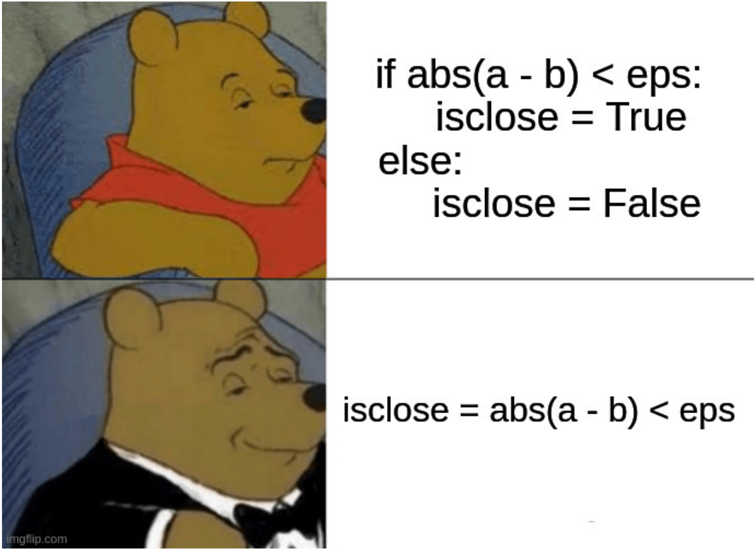
- ASE vibrations code (has since been factored out)
- GPAW response code (writing of “tags”)
- ASR old-master (all over the place! But it’s not as invisible because “dependencies” are declared)

The Response Code Focus Group

- Suggestion by Thorbjørn to do something about the quality of the response code
- Fredrik, Jens Jørgen, Julian, Mikael, myself, Tara, Thorbjørn
- Two-day code sprints every three weeks plus planning/follow-up meetings
- Tasks created and discussed as issues on Gitlab, then delegated to individuals or pairs during sprint
- Why does it work? The same people sit down, work together, learn, gradually improve things
- High level of satisfaction and productivity from working together rather than typical isolated postdoc work


Issues · gpaw / gpaw · Git
https://gitlab.com/gpaw/gpaw/-/issues/?label_name[]=Sprint
Sign up now
Login
gpaw › gpaw › Issues
New issue
Open 42
Closed 43
All 85
Label = ~Sprint
Created date
Move reduce ecut 2.0 follow up
#672 · created 1 week ago by Fredrik Nilsson
Response Sprint
updated 1 week ago
Separate dyson for Gamma, body and ppa
#668 · created 1 week ago by Fredrik Nilsson
Response Sprint
updated 1 week ago
Follow-up from "Dont accept 'point integration' keyword, since it is broken"
0 of 1 checklist item completed
#667 · created 1 week ago by Ask Hjorth Larsen
Response Sprint
updated 1 week ago
Make it possible to calculate spectral function through Chi0 interface
#661 · created 2 weeks ago by Thorbjørn Skovhus
Response Sprint
updated 2 weeks ago
Add response initialization helper class
#659 · created 2 weeks ago by Ask Hjorth Larsen
Response Sprint
updated 2 weeks ago
Unsubclass FXCCorrelation from RPACorrelation
#658 · created 2 weeks ago by Ask Hjorth Larsen
Response Sprint
updated 1 week ago
Add parallel unittest of rpa with nblocks
#654 · created 2 weeks ago by Ask Hjorth Larsen
Response Sprint Tests
updated 2 weeks ago
Centralize various chi0 body parts and associated code
#646 · created 3 weeks ago by Ask Hjorth Larsen
Response Sprint
updated 3 weeks ago
Clean up writing of results in BSE
#642 · created 1 month ago by Thorbjørn Skovhus
Response Sprint
updated 1 month ago

Refactoring level 1



Refactoring level 2


```python
W_GG[1:, 0] = 1. / nq * np.dot(
    np.sum(qdir_qv * u_q[:, np.newaxis], axis=0),
    S_vG0 * sqrtV_G[np.newaxis, 1:])

W_GG[0, 1:] = 1. / nq * np.dot(
    np.sum(qdir_qv * u_q[:, np.newaxis], axis=0),
    S_v0G * sqrtV_G[np.newaxis, 1:])
```

```python
def wing(S_vxx):
    return 1. / nq * np.dot(
        np.sum(qdir_qv * u_q[:, np.newaxis], axis=0)
        S_vxx * sqrtV_G[np.newaxis, 1:])

W_GG[1:, 0] = wing(S_vG0)
W_GG[0, 1:] = wing(S_v0G)
```

Refactoring level infinity

Refactoring the humongous response code constructors

```
class G0W0(PairDensity):
    def __init__(self, calc, filename='gw', restartfile=None,
                 kpts=None, bands=None, relbands=None, nbands=None, ppa=False,
                 xc='RPA', fxc_mode='GW', density_cut=1.e-6, do_GW_too=False,
                 av_scheme=None, Eg=None,
                 truncation=None, integrate_gamma=0,
                 ecut=150.0, eta=0.1, E0=1.0 * Ha,
                 domega0=0.025, omega2=10.0, q0_correction=False,
                 anisotropy_correction=None,
                 nblocks=1, savew=False, savepckl=True,
                 maxiter=1, method='G0W0', mixing=0.2,
                 world=mpi.world, ecut_extrapolation=False,
                 nblocksmax=False, gate_voltage=None,
                 paw_correction='brute-force'):
```

Inside G0W0

```
PairDensity.__init__(self, calc, ecut, world=world, nblocks=nblocks,
                     gate_voltage=gate_voltage, txt=txt,
                     paw_correction=paw_correction)

self.gate_voltage = gate_voltage
ecut /= Ha


...
 chi0 = Chi0(self.inputcalc,
             nbands=self.nbands,
             ecut=self.ecut * Ha,
             intraband=False,
             real_space_derivatives=False,
             txt=self.filename + '.w.txt',
             timer=self.timer,
             nblocks=self.blockcomm.size,
             gate_voltage=self.gate_voltage,
             paw_correction=self.paw_correction,
             **parameters)
```

Fun fact: G0W0 is a PairDensity, but it also creates Chi0 which itself creates a PairDensity. Thus, two completely redundant yet complex objects must (or maybe not?) be kept in sync.

```
class PairDensity:
    def __init__(self, gs, ecut=50, response='density',
                 ftol=1e-6, threshold=1,
                 real_space_derivatives=False,
                 world=mpi.world, txt='-', timer=None,
                 nblocks=1, gate_voltage=None,
                 paw_correction='brute-force', **unused):
```

...

```
class Chi0:
    """Class for calculating non-interacting response functions."""

    def __init__(self, calc, response='density',
                 frequencies=None, domega0=0.1, omega2=10.0, omegamax=None,
                 ecut=50, gammacentered=False, hilbert=True, nbands=None,
                 timeordered=False, eta=0.2, ftol=1e-6, threshold=1,
                 real_space_derivatives=False, intraband=True,
                 world=mpi.world, txt='-', timer=None,
                 nblocks=1, gate_voltage=None,
                 disable_point_group=False, disable_time_reversal=False,
                 disable_non_symmorphic=True,
                 integrationmode=None,
                 pbc=None, rate=0.0, eshift=0.0,
                 paw_correction='brute-force'):
```

Fun fact: PairDensity takes ecut as input, but it doesn't even use it.

So what's wrong then?

- Redundant creation of complex objects
- Endless passing of the same information
- Obviously code does not provide the correct abstractions
- Solution 1: Use `**kwargs` to pass everything instead. Tempting but fails to provide an adequate abstraction.
- When many pieces of data travel everywhere together, we call them "data clumps"
- Solution: Join data clumps into objects. Instead of passing all the information for g0w0 *and* chi0 *and* pair density to the G0W0 constructor, change G0W0 so it takes chi0 and pair density as inputs.

Constructors now

```
class G0W0Calculator:
    def __init__(self, filename='gw', *,
                 chi0calc,
                 wcalc,
                 kpts, bands, nbands=None,
                 fxc_modes,
                 eta,
                 ecut_e,
                 frequencies=None):
```

```
class Chi0Calculator:
    def __init__(self, wd, pair,
                 hilbert=True,
                 intraband=True,
                 nbands=None,
                 timeordered=False,
                 context=None,
                 ecut=None,
                 eta=0.2,
                 disable_point_group=False, disable_time_reversal=False,
                 disable_non_symmorphic=True,
                 integrationmode=None,
                 ftol=1e-6,
                 rate=0.0, eshift=0.0):
```

```
class PairDensityCalculator:
    def __init__(self, gs, context, *,
                 threshold=1, nblocks=1):
```

Sequential coupling in large objects

(slide reused from ASE 2019 workshop)

- Sequential coupling: Workflow becomes “magic incantation”
- Must call methods in right order:

```
obj = MyClass(...)
obj.initialize()
obj.calculate()
obj.read()
x = obj.useful_method()
```

- Complex state: Not clear what the object can do and when.

Source: Francisco Goya / Wikipedia

So what's an example of something good?

Numpy arrays

- Arrays can change, but are always qualitatively the same
- All methods work predictably at all times
- Limited scope: Arrays don't try to be anything more than an array. They will never have a "gate voltage" or a "logfile" inside them. That's somebody else's problem, SEP, a key principle in programming according to me.

How do we factor out sequential coupling?

- State changes many times: Object creation, initialization, halfway through calculation,
- Some methods work only when object is in a particular state
- Actually: An class named Chi0 should represent an already calculated Chi0. Not a way to calculate it.
- Conclusion: Split large classes into "data" objects (Chi0Data which holds arrays and information on parallelization) and calculators (e.g. Chi0Calculator) which know how to compute the data objects.

Work of response code focus group

- Split large sequentially-coupled classes to smaller “data” or “calculator” objects
- A data object wraps an array in order to provide abstractions to deal with (“hide” from the caller) details of its distribution and other properties
- A calculator can calculate something but does not (or should not have) mutable data inside it or otherwise change state
- Join more data clumps into helper classes (`ResponseContext`)
- Clean up completely tangled “integrators”

Concluding remarks

Literature

- *Clean code* by Robert C. Martin
- *The Pragmatic Programmer* by Andrew Hunt
- *Refactoring* by Martin Fowler *et al.*
- *How to write unmaintainable code* (humorous) by Roedy Green
- Finally, check out all the “anti-patterns” and “code smells” on wikipedia and elsewhere.