Designing for Interaction

Scott Sanderson

- GitHub: @ssanderson
- Twitter: @scottbsanderson
- Email: ssanderson@quantopian.com

*Values are what's important to you. And we all believe that we have the same things that are important to us, because they're important to us, so **of course they're important to you**. But that's not quite the case.*

*We always talk about "the right tool for the job". When it comes to choosing a software platform, it's not really the right **tool** for the job, it's the right **values** for the job.*

- ***Bryan Cantrill**: Platform as a reflection of values. Node Summit 2017*

Thesis: Python library design has unique challenges because Python code is used by communities with **divergent technical values**.

Sub-Thesis: Value divergence is a natural product of different **modes of interaction** users have with code.

Sub-Sub-Thesis: Data science and traditional engineering teams often struggle to collaborate because of this divergence.

Goals

- Explain how interactive and non-interactive development cultivate different software values.
- Identify design decisions that make tradeoffs between values.
- Give advice for structuring code in a way that affords both interactive and non-interactive use.

About Me

- Principal Engineer at Quantopian
- Lead team responsible for Backtesting API and (Jupyter-based) Research API.
- My day-to-day is mostly API design and "data infrastructure".
  - Code I commit is mostly "traditional" application code...
  - ...but I do a lot of exploratory analysis for validating our data.

What makes a piece of software "well-designed"?

... it depends.

Okay...on what?

..."context".

Context - Language Features

- Type System
- Object Model
- Runtime (GC, Concurrency, Compilation)

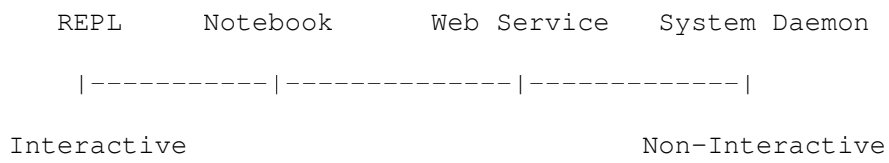
Context - Ecosystem

- Standard Library
- Third-Party Packages
- Tools (Compilers, Linters, Test Frameworks, Packaging Tools, etc.)
- Documentation Standards

Context - Modes of Use

- Servers
- Command Line Applications
- Notebooks
- REPLs

Spectrum of Interaction



Interactive Development

- Most development independent of previously-written code.
- Author, end-user, and operator often the same.
- Results are "directly visible" to the user.
- Few authors.
- Often don't know ahead of time what you're building.

Application Development

- Most development modifies existing work.
- Author, end-user, and operator usually not the same.
- Results are not directly visible.
- Many authors.
- Generally know ahead of time what you're building.

What I Care About When I'm Building an Application

Maintainability

- How easily can I change the application **and be confident that the change is correct**?
- How confident am I that other developers can change my code without breaking it?

Testability

- Can I easily test individual components of the application in isolation?
- Can I run the application without access to production systems? Without access to the internet?
- How long does it take to comprehensively test the application?

Modularity

- How easily can I replace components of the application?
- How easily can I re-use components of the application in new contexts?

Robustness

- How well does the application handle unexpected or erroneous inputs?
- How well does the application **detect** unexpected or erroneous inputs?

Operability

- How easy is it to deploy or upgrade the application?
- How easy is it for a non-expert to detect and recover from failures?

Performance

- How fast does the application solve the problems that it aims to solve?
- Does the application effectively use the resources available to it?
- How quickly does the application start?

What I Care About When I'm Working Interactively

Ergonomics

- How easy is it for me to physically type the code necessary to solve my problem?
- Does the library work well with tools like auto-completers to help me express ideas quickly?

Discoverability

- How easily can I find the things I need?
- How easily can I learn how to use the things I need?
- Can I do the above without disrupting my train of thought?

Flexibility

- How easy is it to use the tool in different ways as new problems arise?
- How easily can I use parts of the tool without requiring access to all of it?

Intuitiveness

- Does the library's model of its domain match my mental model?
- Does the library respond to my inputs in the way that I expect?

In Short

When I'm working an application, I'm concerned with minimizing the **long-term** cost of **maintaining** and **operating** the application.

When I'm working interactively, I'm concerned with **getting ideas out of my brain** as quickly and with as little friction as posssible.

Conflicting Values

Many design decisions force us to choose between conflicting values.

Often we make these decisions without thinking carefully about the values we're implicitly prioritizing.

This makes it harder to understand why others disagree with us.

Example: Optional Arguments

Allow you to ignore rarely-needed parameters.

- Win for **ergonomics** (less typing).
- Win for **flexibility** (more options).

- Loss for **robustness** (can mask errors).
- Loss for **testability** (harder to test all configurations).
- Loss for **maintainability** (hard to keep defaults in sync across many functions)

Example: Globally-Visible Configuration

Allow you to avoid passing parameters that you're always going to pass.

- Win for **ergonomics** (less typing).

- Loss for **maintainability** (hard to know who depends on what).
- Loss for **testability** (requires monkey-patching to test in isolation).
- Loss for **modularity** (requires all consumers to have same global context).

Code Example: `get_algo`

In [2]:
```
cat get_algo_notebook.py
```

```
# Original version of a function written for notebook use.
def get_algo(harness_id,
             metadata_db=None,
             results_db=None,
             session=None,
             dp=None):

    if metadata_db is None:
        import config
        metadata_db = create_engine(config.META_CONF)


    if results_db is None:
        import config
        results_db = create_engine(config.RESULT_CONF)

    if session is None:
        import config
        session = cassandra_session(config.CASSANDRA_CONFIG)

    if dp is None:
        dp = make_dataportal()

    return business_logic(harness_id, metadata_db, results_db,
                          session, dp)
```

In [3]:
```
cat get_algo_application.py
```

```
# Same function, refactored for use in an application.
def get_algo(harness_id,
             data_portal,
             metadata_reader,
             backtest_results_reader):


    algo_metadata = metadata_reader.get_metadata(harness_id)
    backtest_result = backtest_results_reader.get(
        algo_metadata['remote_id']
    )
    return business_logic(algo_metadata, backtest_result)
```

Example: Ambiguous Inputs

> *In the face of ambiguity, refuse the temptation to guess.*
>
> - *The Zen of Python*

The temptation to guess becomes **very strong** for code that's used interactively.

From an interactive user's perspective, it feels pedantic to throw an error on an unexpected input with a "obvious" interpretation.

*"C'mon, you **know** what I mean..."*	• *Every interactive user*

Example: Ambiguous Inputs

In [4]:

```
df = pd.DataFrame(index=pd.date_range('2014', '2015'), data={'a': 'a', 'b': 'b'})
#df.loc[pd.Timestamp('2014-01-02')]
df.loc['2014']
```

Out[4]:

	a	b
2014-01-01	a	b
2014-01-02	a	b
2014-01-03	a	b
2014-01-04	a	b
2014-01-05	a	b
2014-01-06	a	b
2014-01-07	a	b
2014-01-08	a	b
2014-01-09	a	b
2014-01-10	a	b
2014-01-11	a	b
2014-01-12	a	b
...	...	...
2014-12-20	a	b
2014-12-21	a	b

In [5]:

```
df = pd.DataFrame({'x': np.arange(10), 'x ** 2': np.arange(10) ** 2})
df.loc[4, 'x ** 2'] = 5.0
print(df.dtypes)
df
```

```
x           int64
x ** 2    float64
dtype: object
```

Out[5]:

	x	x ** 2
0	0	0.0
1	1	1.0
2	2	4.0
3	3	9.0
4	4	5.0
5	5	25.0
6	6	36.0
7	7	49.0
8	8	64.0
9	9	81.0

Example: Ambiguous Inputs

- Win for **ergonomics**. (Often can accept "shorthands".)
- Win for **intuitiveness**. (Library "figures out what I meant".)

- Loss for **robustness**. (Can hide bugs)
- Loss for **performance**. (Can be expensive to "figure out what you mean")
- Loss for **operability**. (Makes it harder to provide a clear error)

Other Examples:

- Flat vs. nested API surface (flat tends to be more discoverable, but harder to maintain, and can hurt startup time).
- Automatic/unbounded caching (convenient for interactive work, dangerous for applications).

I'm **not** arguing that one set of values is intrinsically better than another. Differences in values are driven by differences in modes of work.

Ergonomics matters more when you're starting from scratch much more often.

Robustness and **Operability** matter less when your results are immediately visible to you.

Testability matters less when you don't know what you're building yet.

Most interactively-developed software either stops being used or gets deployed as part of an application.

"Interactive values" are mostly "velocity-oriented". They aim to minimize the cost of exploration and new development.

"Non-interactive values" are mostly "production-oriented". They aim to minimize the total cost of development over the long run.

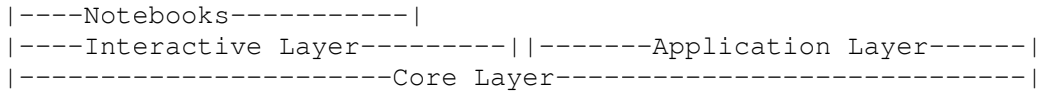
For software that stops being used, it makes sense to optimize for velocity-oriented values.

For software that's used for a long time, it makes sense to optimize for production-oriented values.

Problem: It's hard to tell which is which ahead of time.

(Partial) Solution:

- Build a "core" layer optimized for production-oriented values.
- Build a thin interactive-friendly layer in terms of the core layer.
- When it comes time to convert from notebook to application, only the interactive layer needs to change.



This pattern has applications at different scales of code organization.

Libraries

- `h5py` has a "high-level" API built on top of an explicit low-level API.
- `trio` and `cryptography` have "hazmat" modules that export low-level primitives.
- `zarr` has a `zarr.convenience` module.

Classes

```
class ComplexObject:

    def __init__(self, lots, of, complicated, parameters):
        # Initialize class.

    @classmethod
    def from_config_file(cls, path):
        with open(path) as f:
            # Build configuration from file.
            return cls(...)
```

Functions

```
from core_layer import get_algo
from interactive_layer import global_interactive_context

def get_algo_interactive(harness_id):
    return get_algo(
        harness_id,
        global_interactive_context.data_portal,
        global_interactive_context.metadata_reader,
        global_interactive_context.backtest_results_reader,
    )
```

No Silver Bullet

- More API surface to maintain.
- More work in the short term.
  - Might be work you were going to do anyway.
- Still hard to test and refactor interactive-friendly layers.
  - Probably easier than it was before though...

Conclusions and Takeaways

If you disagree with the design of a piece of code, try to understand the ways that their **software values** may differ from yours.

When designing code that will be (eventually) used in an application, consider the implications of your decisions for **maintainability**, **testability**, **modularity**, and **robustness**.

When designing code that might be used interactively, consider the implications of your decisions for **ergonomics**, **discoverability**, and **flexibility**.

Conclusions and Takeaways

When faced with tradeoffs between divergent values, considering separating your code into **layers** that can cater to different **modes of usage**.

- Organize modules into "core" and "convenience" layers.
- Consider adding convenient "smart constructors" as classmethods of objects.
- Write helper functions that make common patterns more ergonomic.

Questions?

- Slides: https://github.com/ssanderson/jupytercon-2018
- GitHub: @ssanderson
- Twitter: @scottbsanderson
- Email: ssanderson@quantopian.com