Byte-Sized

Computer Science for Data People

Part 2: Maintainability

Principles of Successful Tech. Products

1

Scalability

Can the product handle growing data volumes?

2

Maintainability

Can other people work on the system productively?

3

Reliability

Can the product tolerate hardware, software, and human faults?

Framework adapted from Designing Data-Intensive Applications by Martin Kleppmann

Aspects of Maintainability

Modularity

Splitting your product and teams into modules that are loosely coupled, tightly aligned

Clean Code

Writing developer-friendly code that encourages others to build on top of and reuse your work

Version Control

Using version control tools like Git to manage the collaboration process so you don’t have to

Deep-Dive: Modularity

Big Ideas

- **Every module / function should do only one thing**
- **Modules should interact using tightly-defined ‘interfaces’**
- **Teams should be similarly organized, collaborating at their intersections**

Trying to find a bug in one long script that does 15+ things

Writing fifteen smaller functions and unit tests so that you don’t have to

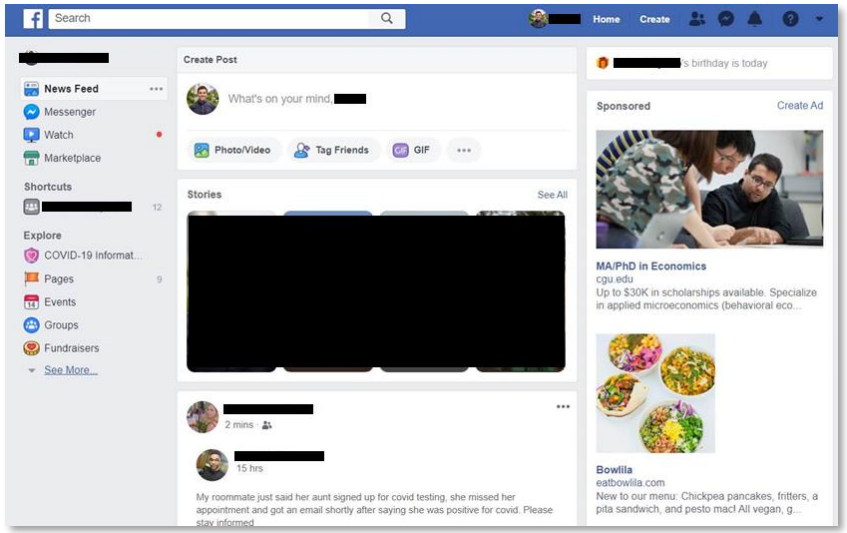
Benefits

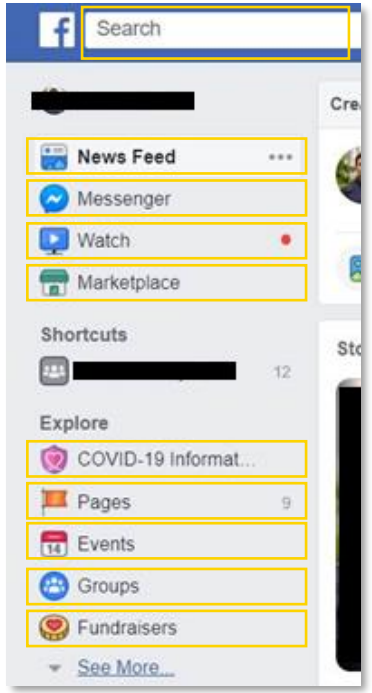
- Tightly-defined modules can build on one another and be reused
- Modules can be swapped-out when requirements change
- Errors can be caught at the source using automated testing

Related CS Concepts

- Single-Responsibility Principle
- Design by Contract
- Law of Demeter (‘shy code’)

Real-World Example





What users see

One cohesive product

What users don't see

Dozens of loosely-coupled, tightly-aligned teams owning individual features that share data through a shared hidden state

Deep-Dive: Clean Code

Big Ideas

- **Don’t repeat yourself: write short*, simple functions**
- **Tell a story using descriptive names, not comments**
- **Pass objects, collections, and functions as parameters**

Hastily writing code that’s easier for a future developer to abandon than to fix

Writing clean code once and having future users thank you forever

Benefits

- Errors are easier to spot and only have to be fixed in one place
- Reusing patterns means less code to write and read
- New team members are easier to onboard and get up to speed

Related CS Concepts

- Evils of Duplication
- Software Entropy (‘broken windows’)
- Functional Programming

**Clean Code* recommends keeping functions to <4 lines, <3 parameters*

Real-World Examples

```
#Downcast in order to save memory
def downcast(df):
    cols = df.dtypes.index.tolist()
    types = df.dtypes.values.tolist()
    for i,t in enumerate(types):
        if 'int' in str(t):
            if df[cols[i]].min() > np.iinfo(np.int8).min and df[cols[i]].max() < np.iinfo(np.in
t8).max:
                df[cols[i]] = df[cols[i]].astype(np.int8)
            elif df[cols[i]].min() > np.iinfo(np.int16).min and df[cols[i]].max() < np.iinfo(n
p.int16).max:
                df[cols[i]] = df[cols[i]].astype(np.int16)
            elif df[cols[i]].min() > np.iinfo(np.int32).min and df[cols[i]].max() < np.iinfo(n
p.int32).max:
                df[cols[i]] = df[cols[i]].astype(np.int32)
            else:
                df[cols[i]] = df[cols[i]].astype(np.int64)
        elif 'float' in str(t):
            if df[cols[i]].min() > np.finfo(np.float16).min and df[cols[i]].max() < np.finfo(n
p.float16).max:
                df[cols[i]] = df[cols[i]].astype(np.float16)
            elif df[cols[i]].min() > np.finfo(np.float32).min and df[cols[i]].max() < np.finfo
(np.float32).max:
                df[cols[i]] = df[cols[i]].astype(np.float32)
            else:
                df[cols[i]] = df[cols[i]].astype(np.float64)
        elif t == np.object:
            if cols[i] == 'date':
                df[cols[i]] = pd.to_datetime(df[cols[i]], format='%Y-%m-%d')
            else:
                df[cols[i]] = df[cols[i]].astype('category')
    return df
```

Real-World Examples

```python
#Downcast in order to save memory
def downcast(df):
    cols = df.dtypes.index.tolist()
    types = df.dtypes.values.tolist()
    for i,t in enumerate(types):
        if 'int' in str(t):
            df[cols[i]] = pd.to_numeric(df[cols[i]], downcast='integer')
        elif 'float' in str(t):
            df[cols[i]] = pd.to_numeric(df[cols[i]], downcast='float')
        elif t == np.object:
            if cols[i] == 'date':
                df[cols[i]] = pd.to_datetime(df[cols[i]], format='%Y-%m-%d')
            else:
                df[cols[i]] = df[cols[i]].astype('category')
    return df
```

Real-World Examples

```python
def compress_dataframe(df):
    """
    Downcast dataframe and convert objects to categories to save memory
    """
    def handle_numeric_downcast(array, type_):
        return pd.to_numeric(array, downcast=type_)

    for type_ in ['integer', 'float', 'object']:
        column_list = df.select_dtypes(include=type_)

        if type_ == 'object':
            df[column_list] = df[column_list].astype('category')
        else:
            df[column_list] = handle_numeric_downcast(df[column_list], type_)
```

Deep-Dive: Version Control

Big Ideas

- **<u>Always</u> use a version control system**
- **Don’t keep multiple versions of the same code; use pull requests instead**
- **Keep your repo fresh with descriptive commits and good branch hygiene**

Writing new code in a copied script, then dealing with versioning issues when you want to incorporate it into the master

Handling versioning issues automatically using Git

Benefits

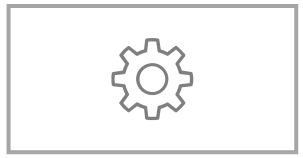
- Collaborate on the same codebase without fear of versioning issues
- Quickly and easily reverse bad commits
- Provides transparency to minimize duplicative work

Related CS Concepts

- Reversibility
- Metadata

Suggested Workflow

Import and call functions in main



main.py
(where the magic happens)

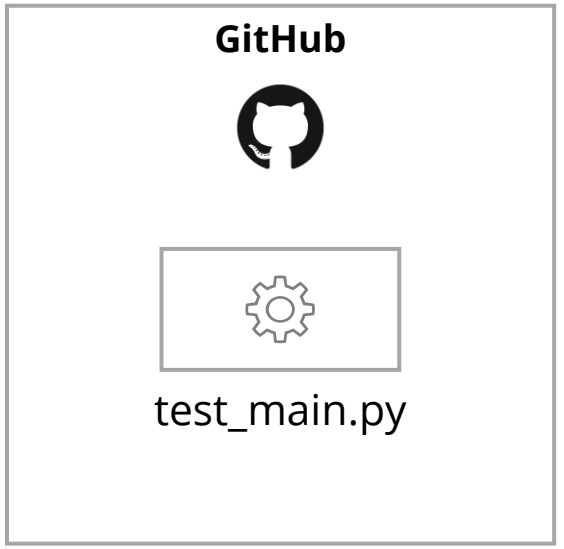
Write functions in .py scripts within /src

utilities.py

feature_engineering.py

modeling.py

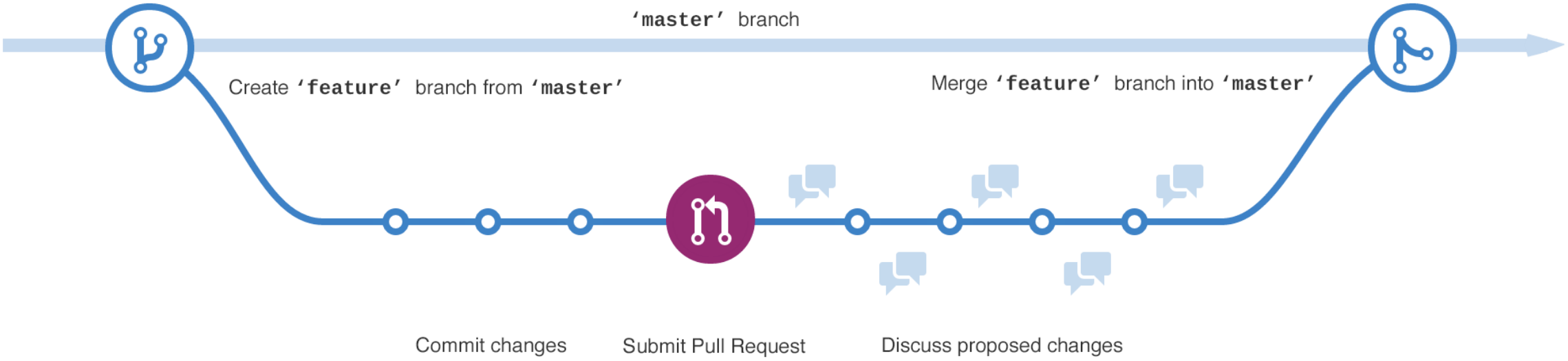
Trigger automated tests with every push



Application (calls functions)

Module (defines functions)

Collaborating on GitHub



Source: GitHub Guides