Byte-Sized

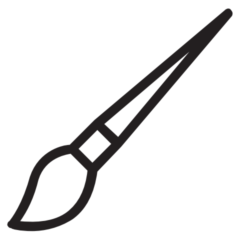
Computer Science for Data People

Part 1: Clean Code

Topics We'll Cover

1

Clean Code



Writing performant code that others will be excited to reuse

2

System Design

Building systems and products that scale

3

Collaboration

Working productively with other people

Principles of Clean Code

Style

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

Speed

Ability to handle larger volumes of data without slowing down

Space

Ability to handle larger volumes of data without breaking

Deep-Dive: Style

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

<u>Clean Code</u> recommends keeping functions to <4 lines, <3 parameters

Deep-Dive: Speed

Big Ideas

- **Pick the right data structure (don't distribute the small stuff)**
- **Use arrays, not for-loops**
- **Double-check if a built-in exists before writing your own function**

Running nested for-loops on a Spark DataFrame

Using a built-in function on a filtered pandas DataFrame instead

Thinking Exercise

Your modeling pipeline was very fast when running on a small sample of data, but started to hang when you tried the same code on a larger dataframe. What should you do first?

Related CS Concepts

- Serialization
- Vectorization & Linear Algebra
- Big O Notation

Deep-Dive: Space

Big Ideas

- **Filter first, especially before joining**
- **Drop columns, downcast, and use in-place operations and sparse data structures to compress your data**
- **Index, chunk, and distribute the big stuff**

Wasting money on expensive storage and compute clusters

Profiling your code and compressing your data so you don't have to

Thinking Exercise

Your modeling pipeline was working fine when you were filtered on one region, but suddenly throws an out-of-memory error when include all regions in your dataframe. What steps would you take to solve this problem?

Related CS Concepts

- Lossless / Lossy Compression
- Parallelism & Concurrency
- Distributed Computing

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.int8).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(np.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(np.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(np.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

```
#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
```

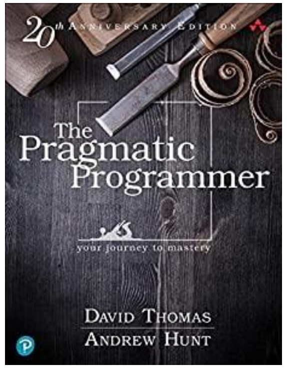
Real-World Examples

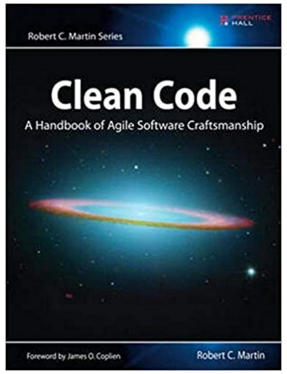
```python
def compress_dataframe(df):
    """
    Downcast dataframe and convert objects to categories to save memory
    """
    def handle_numeric_downcast(array, type_):
        return pd.to_numeric(array, downcast=type_)

    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_)
```

Book Recommendations

1

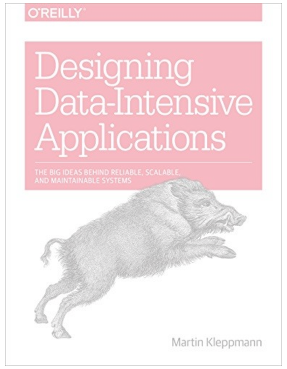
Clean Code





2

System Design



3

Collaboration