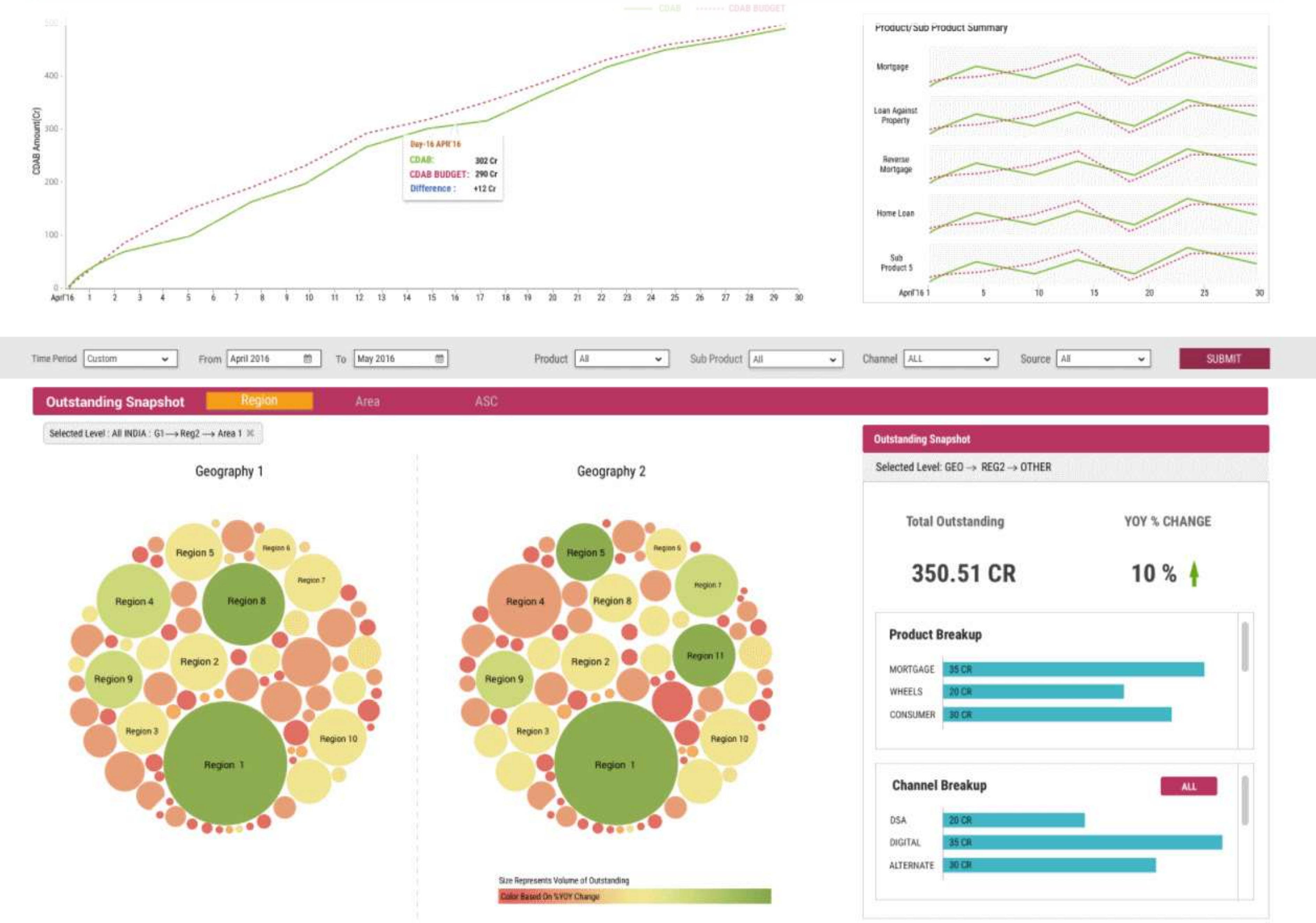
@SANAND0

Don't Repeat Yourself

Adventures in Re-use

Gramener
A Data Science Company

WE WERE BUILDING A BRANCH BALANCE DASHBOARD FOR A BANK



Day-16 APR'16
CDAB: 302 Cr
CDAB BUDGET: 290 Cr
Difference : +12 Cr

Time Period: Custom
From: April 2016
To: May 2016
Product: All
Sub Product: All
Channel: ALL
Source: All

Selected Level : All INDIA : G1 ⟶ Reg2 ⟶ Area 1

Outstanding Snapshot

Selected Level: GEO → REG2 → OTHER

Total Outstanding: 350.51 CR

YOY % CHANGE: 10 %

Product Breakup

MORTGAGE	35 CR
WHEELS	20 CR
CONSUMER	30 CR

Channel Breakup

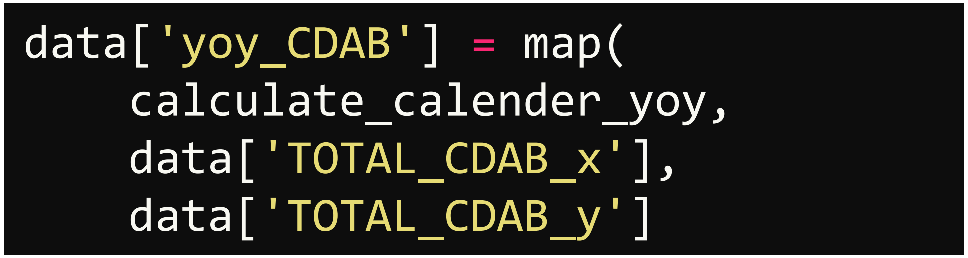
DSA	20 CR
DIGITAL	35 CR
ALTERNATE	30 CR

Size Represents Volume of Outstanding

Color Based On %YOY Change

This fragment of code was used to calculate the YoY growth

This is a piece of code we deployed at a large bank to calculate year-on-year growth of balance:

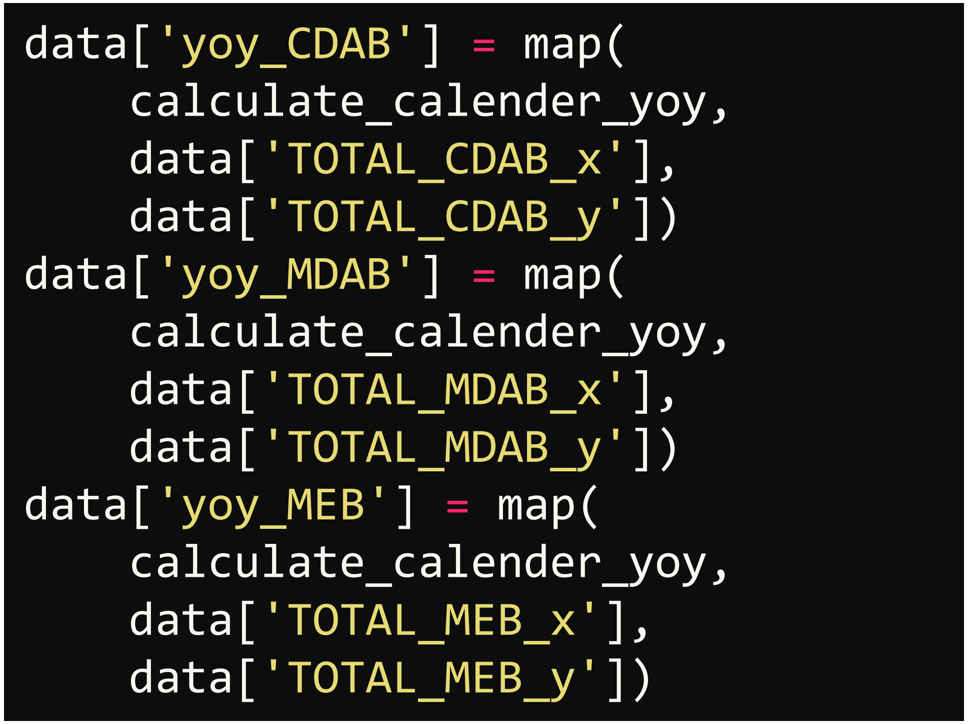


```
data['yoy_CDAB'] = map(
    calculate_calender_yoy,
    data['TOTAL_CDAB_x'],
    data['TOTAL_CDAB_y']
```

On 29 Aug, the bank added more metrics:

- **CDAB**: Cumulative Daily Average Balance (from start of year)
- **MDAB**: Monthly Daily Average Balance (from start of month)
- **MEB**: Month End Balance

This led to this piece of code



```
data['yoy_CDAB'] = map(
    calculate_calender_yoy,
    data['TOTAL_CDAB_x'],
    data['TOTAL_CDAB_y'])
data['yoy_MDAB'] = map(
    calculate_calender_yoy,
    data['TOTAL_MDAB_x'],
    data['TOTAL_MDAB_y'])
data['yoy_MEB'] = map(
    calculate_calender_yoy,
    data['TOTAL_MEB_x'],
    data['TOTAL_MEB_y'])
```

THE CLIENT ADDED MORE AREAS

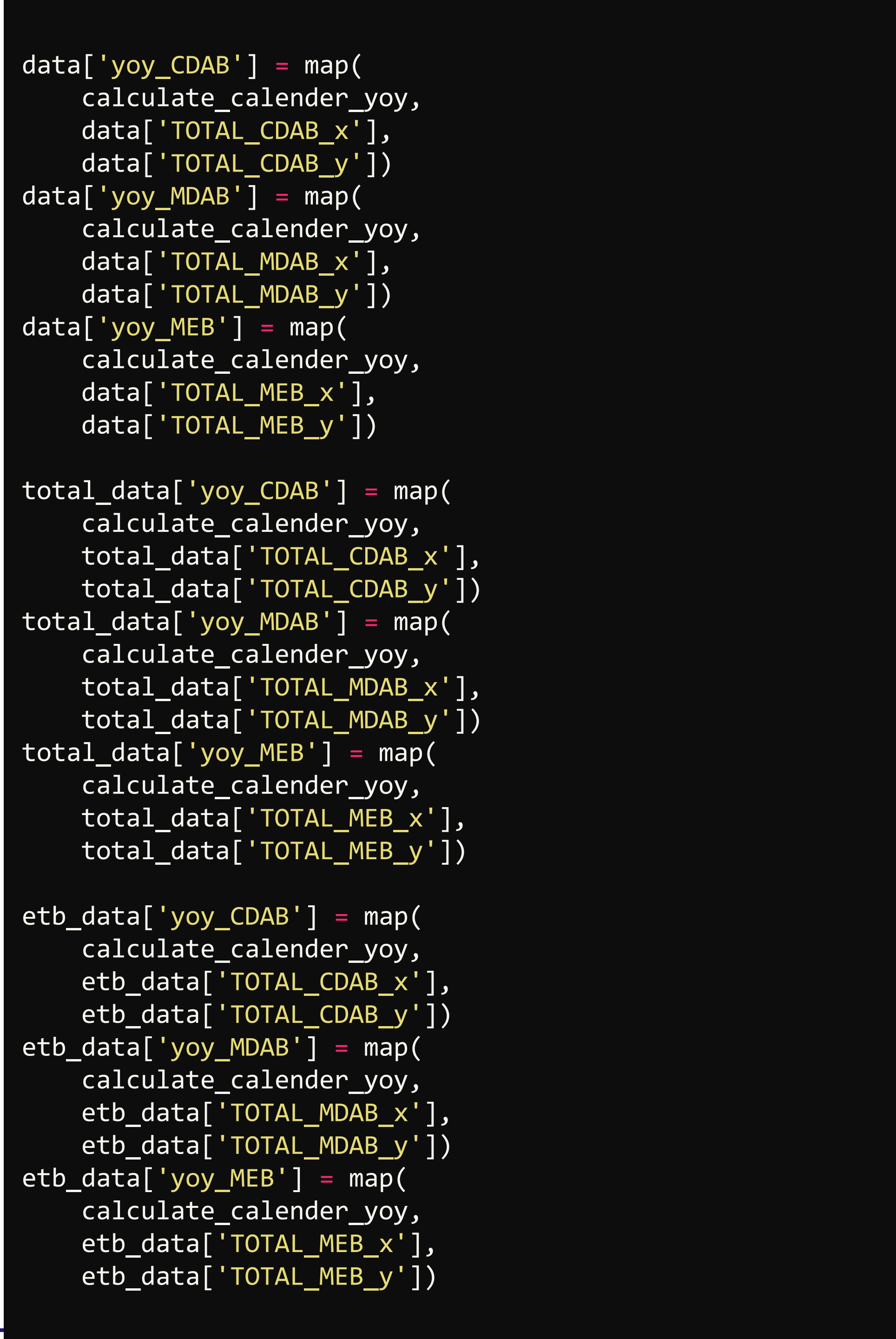
On 31 Aug, the bank wanted to see this across different areas:

- **NTB**: New to Bank accounts (clients added in the last 2 years)
- **ETB**: Existing to Bank accounts (clients older than 2 years)
- **Total**: All Bank accounts

This code is actually deployed in production.

Even today.

Really.



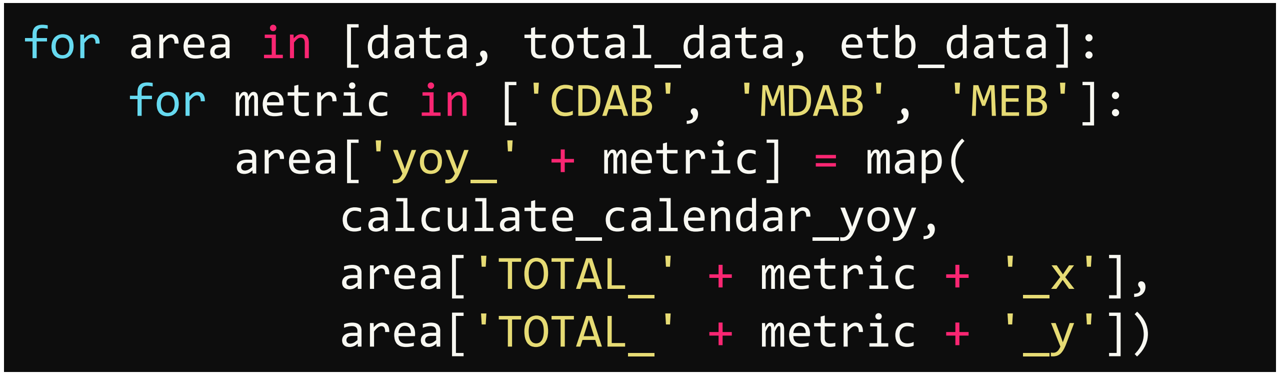
```
data['yoy_CDAB'] = map(
    calculate_calender_yoy,
    data['TOTAL_CDAB_x'],
    data['TOTAL_CDAB_y'])
data['yoy_MDAB'] = map(
    calculate_calender_yoy,
    data['TOTAL_MDAB_x'],
    data['TOTAL_MDAB_y'])
data['yoy_MEB'] = map(
    calculate_calender_yoy,
    data['TOTAL_MEB_x'],
    data['TOTAL_MEB_y'])

total_data['yoy_CDAB'] = map(
    calculate_calender_yoy,
    total_data['TOTAL_CDAB_x'],
    total_data['TOTAL_CDAB_y'])
total_data['yoy_MDAB'] = map(
    calculate_calender_yoy,
    total_data['TOTAL_MDAB_x'],
    total_data['TOTAL_MDAB_y'])
total_data['yoy_MEB'] = map(
    calculate_calender_yoy,
    total_data['TOTAL_MEB_x'],
    total_data['TOTAL_MEB_y'])

etb_data['yoy_CDAB'] = map(
    calculate_calender_yoy,
    etb_data['TOTAL_CDAB_x'],
    etb_data['TOTAL_CDAB_y'])
etb_data['yoy_MDAB'] = map(
    calculate_calender_yoy,
    etb_data['TOTAL_MDAB_x'],
    etb_data['TOTAL_MDAB_y'])
etb_data['yoy_MEB'] = map(
    calculate_calender_yoy,
    etb_data['TOTAL_MEB_x'],
    etb_data['TOTAL_MEB_y'])
```

USE LOOPS TO AVOID DUPLICATION

As you would have guessed, the same thing can be achieved much more compactly with loops.



```
for area in [data, total_data, etb_data]:
    for metric in ['CDAB', 'MDAB', 'MEB']:
        area['yoy_' + metric] = map(
            calculate_calendar_yoy,
            area['TOTAL_' + metric + '_x'],
            area['TOTAL_' + metric + '_y'])
```

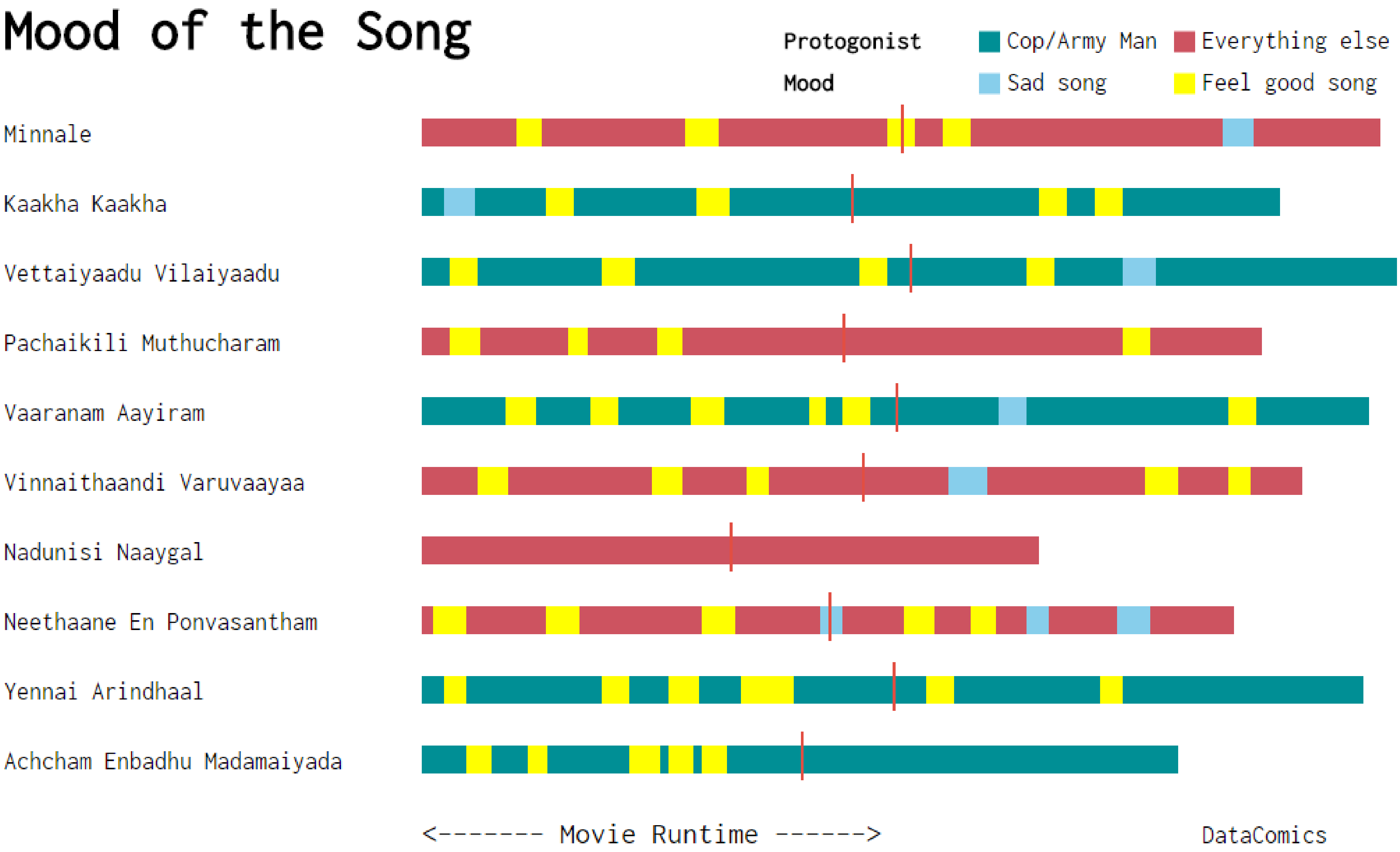
This is smaller – hence easier to **understand**
This uses data structures – hence easier to **extend**

WHY WOULD ANY SANE PERSON NOT USE LOOPS?

Don't blame the developer

He's actually brilliant. Here are some things he made

Data Comics: Songs in Gautham Menon Movies



FOOTBALLER'S CHERNOFF FACES

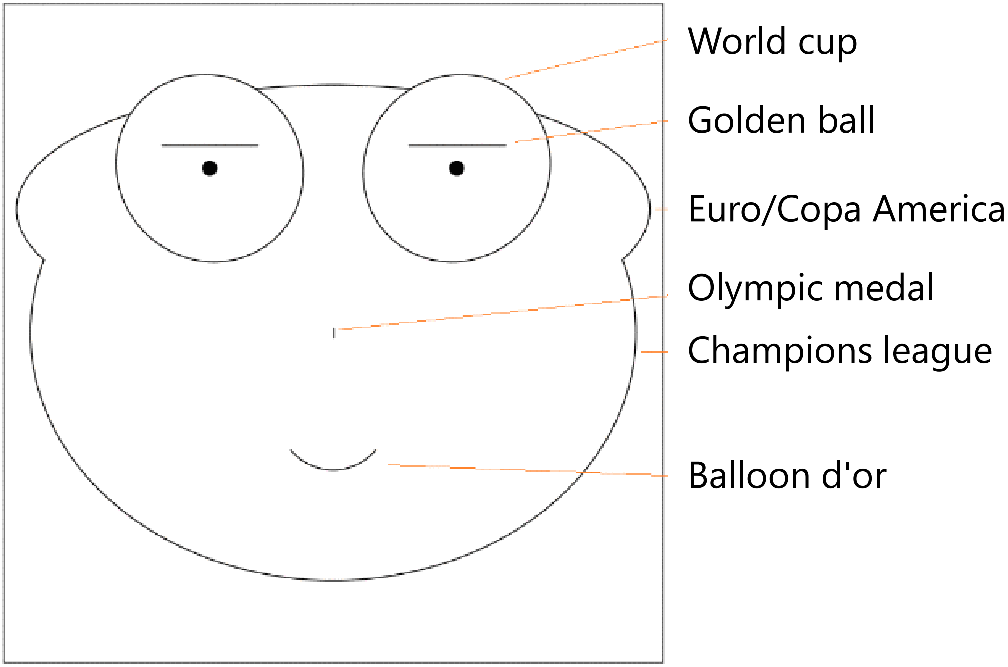
Chernoff Faces are a visualization that represent data using features in a human face like size of eyes, nose, their positioning etc..

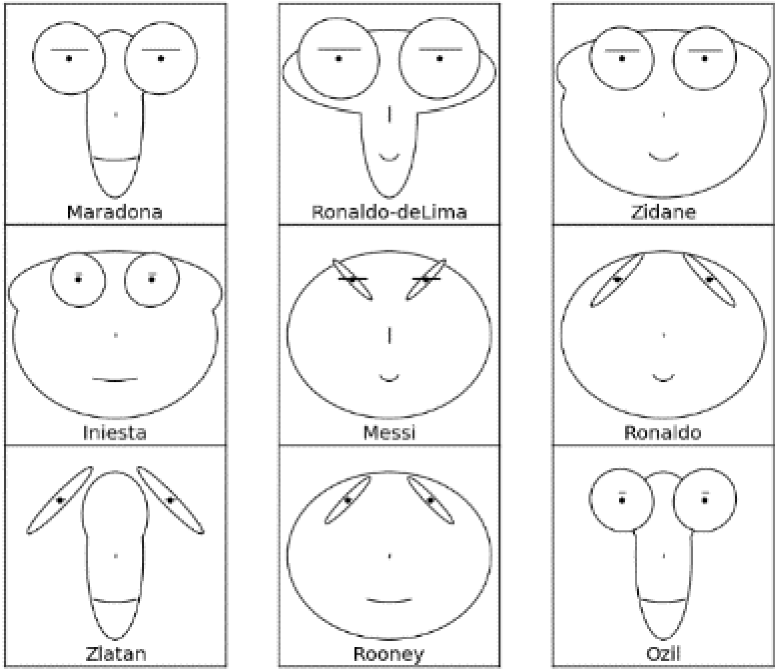
We applied this to a few well known faces of football with data representing their honors.

The size of the eyes is the direct representation of whether the player is a World Cup winner or not. Players with bigger eyes are World Cup winners.

The size of the eyebrows represent individual honors in the World Cup (Golden Ball). The width of the top half of the face represents whether the player is a Euro or Copa America winner and the bottom half represents whether the player is Champions League winner. . The curvature of smile represents Ballon d'or winners, higher the concavity higher the number of awards. The size of nose represents Olympic honors.

Below is what the faces of some of the famous footballers look like with this mapping





Re-use is not intuitive

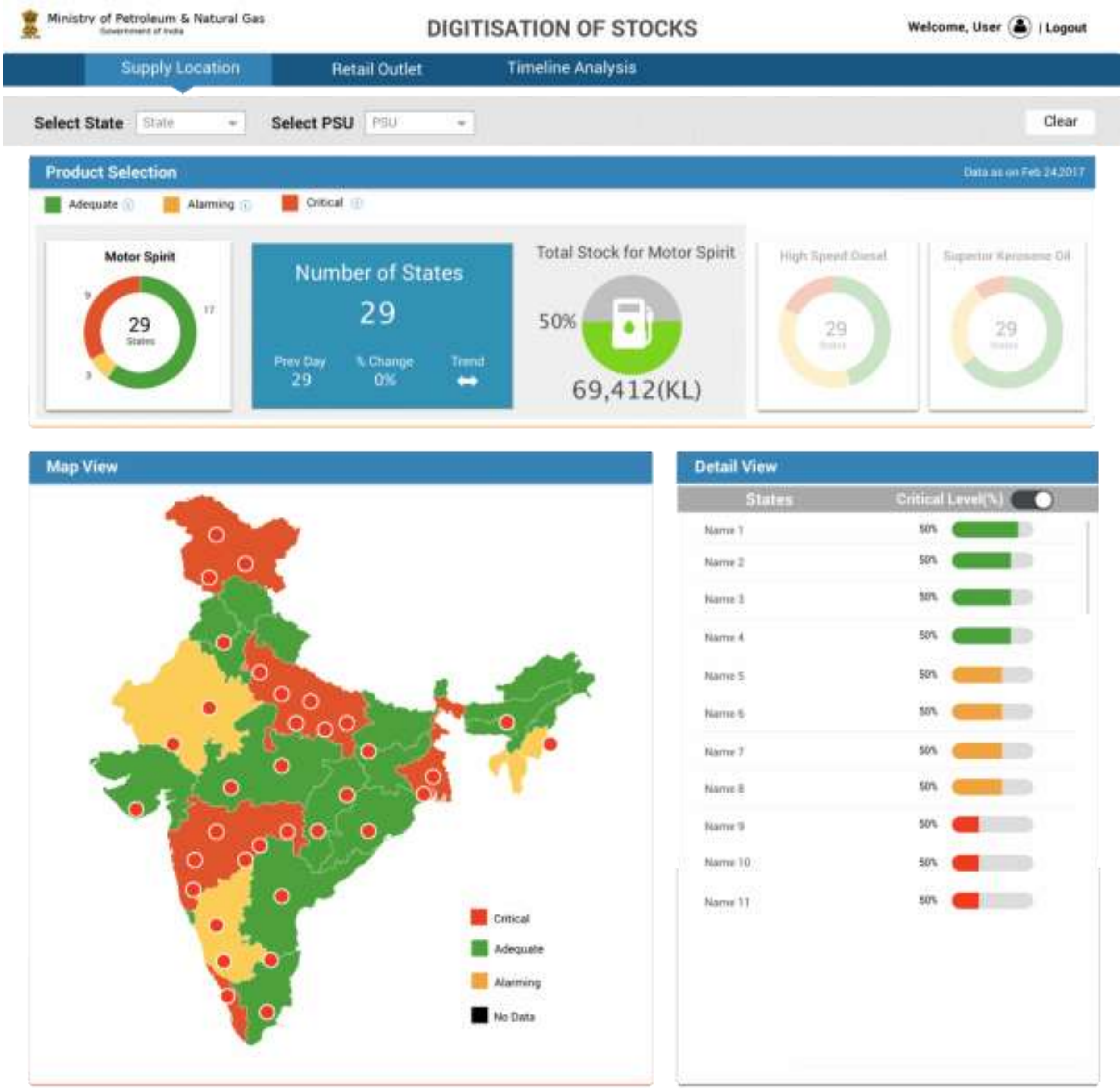
Copy-paste is very intuitive. That's what we're up against

Petroleum Stock

The Ministry of Petroleum and Natural Gas wanted to track stock levels of Motor Spirit and Diesel for all 3 OMC's across India. And also view Historical data for the same to take decisive business actions.

Gramener built a dashboard to view all the stock level data for all products and OMC's across India. The Dashboard was optimized to display daily data as well accumulate Historical data.

The dashboard manages Motor Spirit and Diesel stock worth ~Rs 4000 Cr. Acting on this can lead to ~Rs 42 Cr of annual savings on fuel wastage.



Ministry of Petroleum & Natural Gas
Government of India

DIGITISATION OF STOCKS

Welcome, User | Logout

Select State: State

Select PSU: PSU

Product Selection

Data as on Feb 24,2017

🟩 Adequate 🟧 Alarming 🟥 Critical

Number of States: 29

Prev Day	% Change	Trend
29	0%	⬌

Total Stock for Motor Spirit: 69,412(KL)

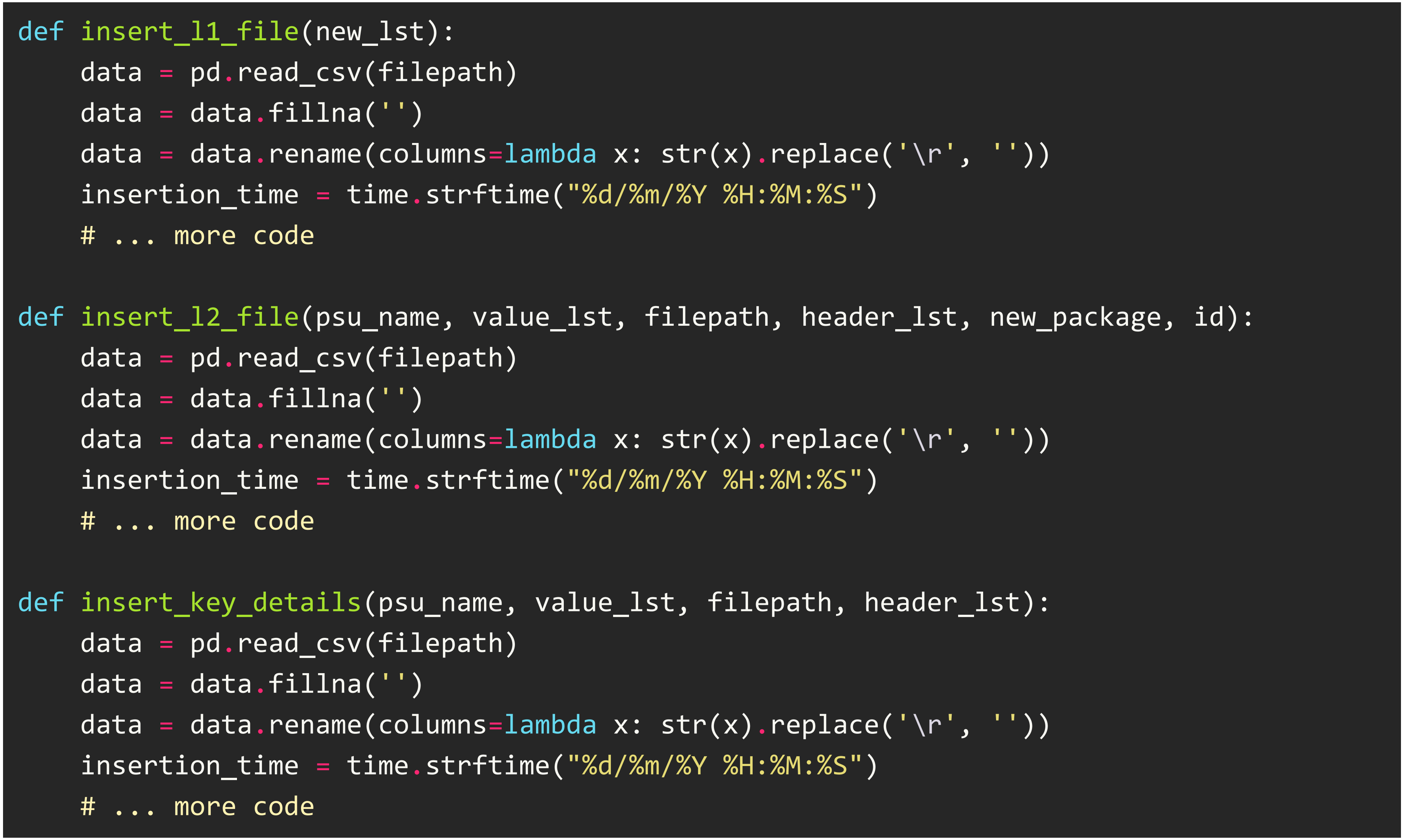
Map View

Detail View

States	Critical Level(%)
Name 1	50%
Name 2	50%
Name 3	50%
Name 4	50%
Name 5	50%
Name 6	50%
Name 7	50%
Name 8	50%
Name 9	50%
Name 10	50%
Name 11	50%

THIS FRAGMENT OF CODE WAS USED TO PROCESS DATA

When the same code is repeated across different functions like this:



```python
def insert_l1_file(new_lst):
    data = pd.read_csv(filepath)
    data = data.fillna('')
    data = data.rename(columns=lambda x: str(x).replace('\r', ''))
    insertion_time = time.strftime("%d/%m/%Y %H:%M:%S")
    # ... more code

def insert_l2_file(psu_name, value_lst, filepath, header_lst, new_package, id):
    data = pd.read_csv(filepath)
    data = data.fillna('')
    data = data.rename(columns=lambda x: str(x).replace('\r', ''))
    insertion_time = time.strftime("%d/%m/%Y %H:%M:%S")
    # ... more code

def insert_key_details(psu_name, value_lst, filepath, header_lst):
    data = pd.read_csv(filepath)
    data = data.fillna('')
    data = data.rename(columns=lambda x: str(x).replace('\r', ''))
    insertion_time = time.strftime("%d/%m/%Y %H:%M:%S")
    # ... more code
```

Group common code into functions

… create a common function and call it.

```
def load_data(filepath):
    data = pd.read_csv(filepath)
    data = data.fillna('')
    data = data.rename(columns=lambda x: str(x).replace('\r', ''))
    insertion_time = time.strftime("%d/%m/%Y %H:%M:%S")
    return data, insertion_time

def insert_l1_file(new_lst):
    data, insertion_time = load_data(filepath)
    # ... more code

def insert_l2_file(psu_name, value_lst, filepath, header_lst, new_package, id):
    data, insertion_time = load_data(filepath)
    # ... more code

def insert_key_details(psu_name, value_lst, filepath, header_lst):
    data, insertion_time = load_data(filepath)
    # ... more code
```

THIS FRAGMENT OF CODE WAS USED TO LOAD DATA

This code reads 3 datasets:

```
data_l1 = pd.read_csv('PSU_l1.csv')
data_l2 = pd.read_csv('PSU_l2.csv')
data_l3 = pd.read_csv('PSU_l3.csv')
```

Based on the user's input, the last row of the relevant dataset is picked:

```
if form_type == "l1":
    result = data_l1[:-1]
elif form_type == "l2":
    result = data_l2[:-1]
elif form_type == "l3":
    result = data_l3[:-1]
```

It's not trivial to replace this with a loop or a lookup.

Use loops to avoid duplication

Instead of loading into 4 datasets, use:

```
data = {
    level: pd.read_csv('PSU_' + level + '.csv')
    for level in ['l1', 'l2', 'l3']
}
result = data[form_type][:-1]
```

This cuts down the code, and it's easier to add new datasets.

BUT… (AND I HERE A LOT OF THESE “BUT”S)

BUT INPUTS ARE NOT CONSISTENT

The first 2 files are named **`PSU_l1.csv`** and **`PSU_l2.csv`**.

The third file alone is named **`PSU_Personnel.csv`** instead of **`PSU_l3.csv`**.

But we want to map it to **`data['l3']`**, because that's how the user will request it.

So use a mapping:

```
lookup = {
    'l1': 'PSU_l1.csv',
    'l2': 'PSU_l2.csv',
    'l3': 'PSU_Personnel.csv',  # different filename
}
data = {key: pd.read_csv(file) for key, file in lookup.items()}
result = data[form_type][:-1]
```

USE DATA STRUCTURES TO HANDLE VARIATIONS

BUT WE PERFORM DIFFERENT OPERATIONS ON DIFFERENT FILES

For **PSU_Personnel.csv**, we want to pick the first row, not the last row.

So add the row into the mapping as well:

```
lookup = {                             # Define row for each file
    'l1': dict(file='PSU_l1.csv',        row=-1),
    'l2': dict(file='PSU_l2.csv',        row=-1),
    'l3': dict(file='PSU_Personnel.csv', row=0),
}
data = {
    key: pd.read_csv(info['file'])
    for key, info in lookup.items()
}
result = data[form_type][:lookup[form_type]['row']]
```

USE DATA STRUCTURES TO HANDLE VARIATIONS

BUT WE PERFORM VERY DIFFERENT OPERATIONS ON DIFFERENT FILES

For **`PSU_l1.csv`**, we want to sort it.

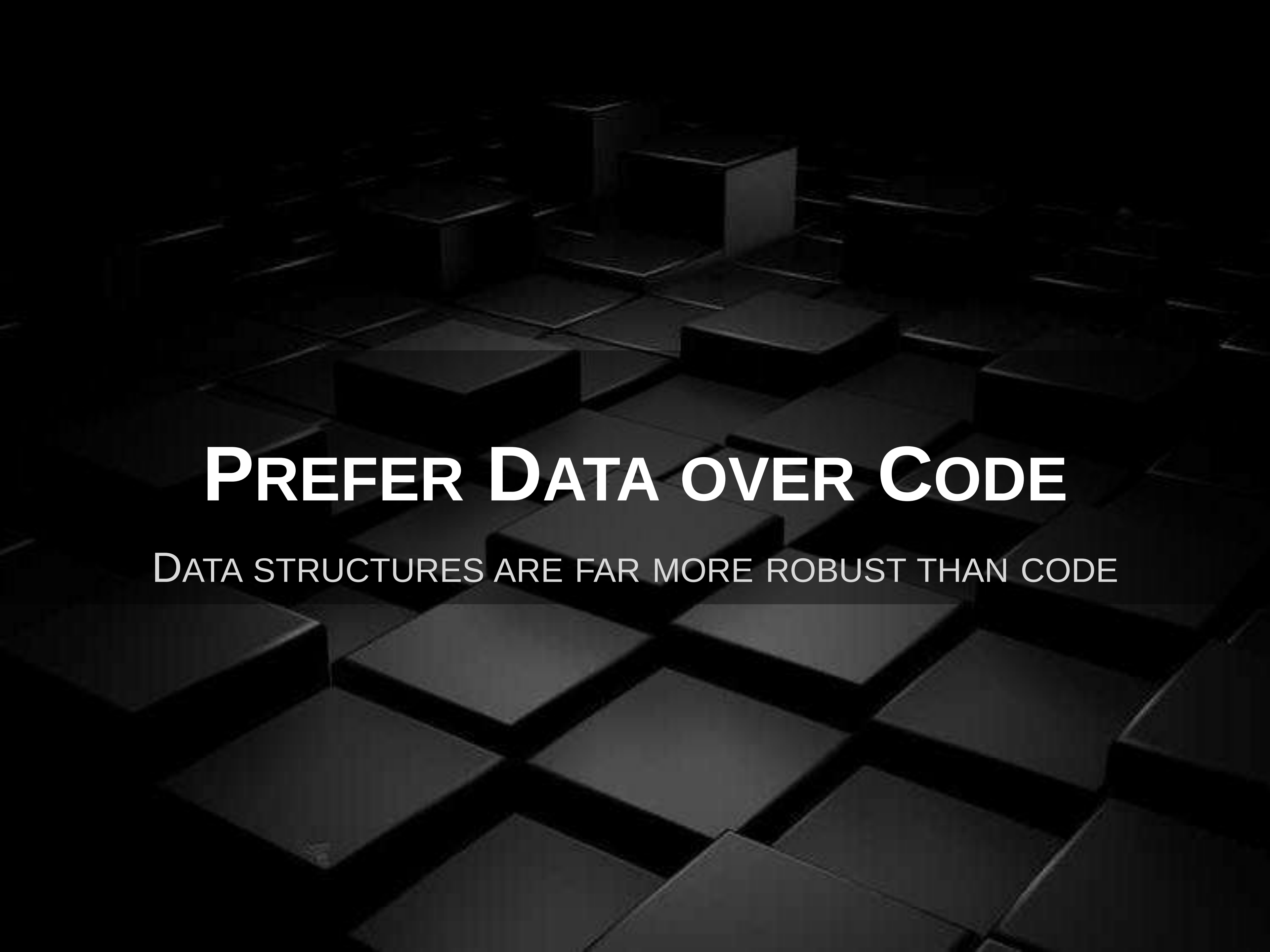
For **`PSU_l2.csv`**, we want to fill empty values.

Then use functions to define your operations.

```
lookup = {
    'l1': dict(file='PSU_l1.csv', op=lambda v: v.sort_values('X')),
    'l2': dict(file='PSU_l2.csv', op=lambda v: v.fillna('')),
    'l3': dict(file='PSU_Personnel.csv', op=lambda v: v),
}
data = {
    key: pd.read_csv(info['file'])
    for key, info in lookup.items()
}
result = lookup[form_type]['op'](data[form_type])
```

The functions need not be `lambdas`. They can be normal multi-line functions.

USE FUNCTIONS TO HANDLE VARIATIONS



Prefer Data over Code

Data structures are far more robust than code

Keep data in data files

Store data in data files, not Python files. This lets non-programmers (analysts, client IT teams, administrators) edit the data

You're a good programmer when you stop thinking *How to write code* and begin thinking *How will people use my code*.

```
lookup = {
    'l1': dict(file='PSU_l1.csv',        row=-1),
    'l2': dict(file='PSU_l2.csv',        row=-1),
    'l3': dict(file='PSU_Personnel.csv', row=0),
}
```

... is better stored as config.json:

```
{
    "l1": {"file": "PSU_l1.csv", "row": -1},
    "l2": {"file": "PSU_l2.csv", "row": -1},
    "l3": {"file": "PSU_Personnel.csv", "row": 0}
}
```

... and read via:

```
import json
lookup = json.load(open('config.json'))
```

Prefer YAML over JSON

YAML is be more intuitive less error-prone. There are no trailing commas or braces to get wrong.

It also supports data re-use.

```
l1:
    file: PSU_l1.csv
    row: -1
l2:
    file: PSU_l1.csv
    row: -1
l3:
    file: PSU_Personnel.csv
    row: 0
```

You can read this via:

```
import yaml
lookup = yaml.load(open('config.json'))
```

We used this in our cluster application

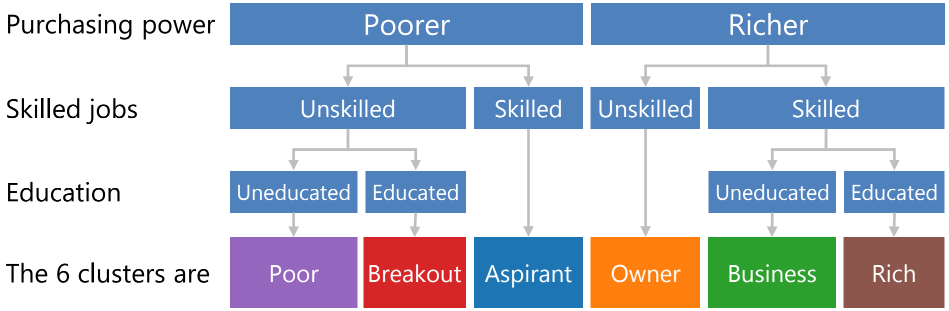
LINK

Previously, the client was treating contiguous regions as a homogenous entity, from a channel content perspective.

To deliver targeted content, we divided India into 6 clusters based on their demographic behavior. Specifically, three composite indices were created based on the economic development lifecycle:

- **Education** (literacy, higher education) that leads to…
- **Skilled jobs** (in mfg. or services) that leads to…
- **Purchasing power** (higher income, asset ownership)

Districts were divided (at the average cut-off) by:



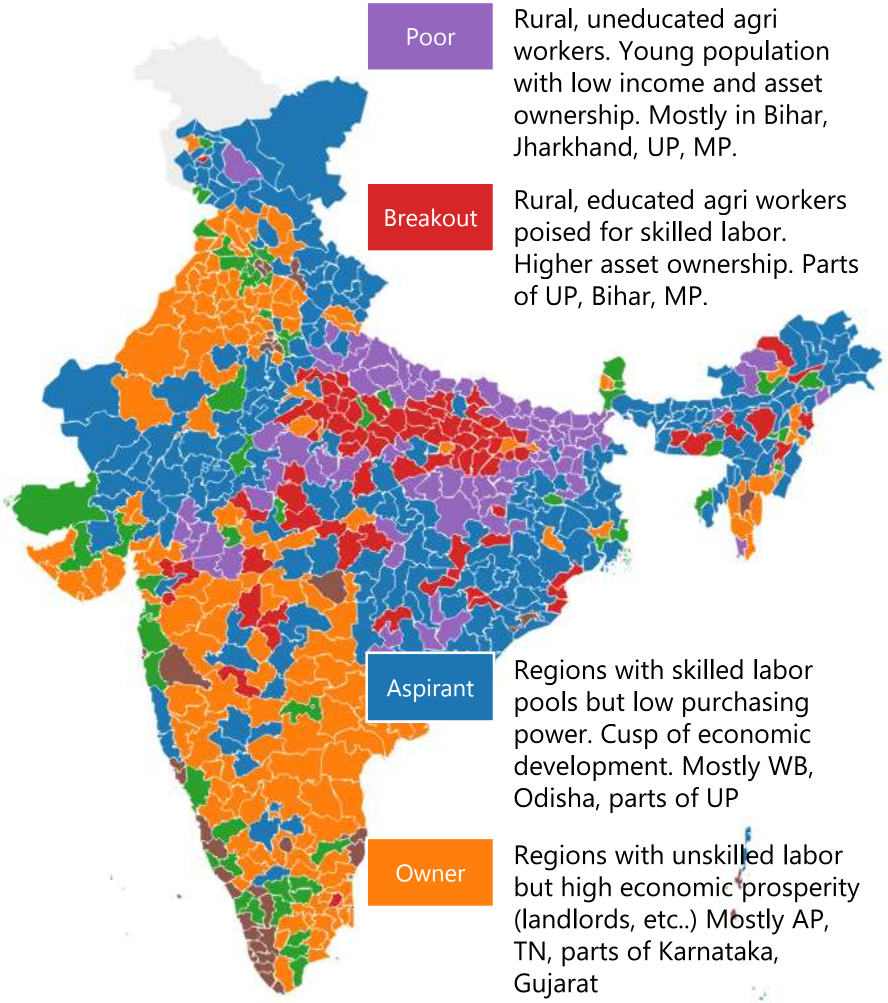
Purchasing power: Poorer | Richer

Skilled jobs: Poorer → Unskilled, Skilled; Richer → Unskilled, Skilled

Education: Poorer, Unskilled → Uneducated, Educated; Richer, Skilled → Uneducated, Educated

The 6 clusters are: Poor | Breakout | Aspirant | Owner | Business | Rich

Offering targeted content to these clusters will reach a more homogenous demographic population.



Poor: Rural, uneducated agri workers. Young population with low income and asset ownership. Mostly in Bihar, Jharkhand, UP, MP.

Breakout: Rural, educated agri workers poised for skilled labor. Higher asset ownership. Parts of UP, Bihar, MP.

Aspirant: Regions with skilled labor pools but low purchasing power. Cusp of economic development. Mostly WB, Odisha, parts of UP

Owner: Regions with unskilled labor but high economic prosperity (landlords, etc..) Mostly AP, TN, parts of Karnataka, Gujarat

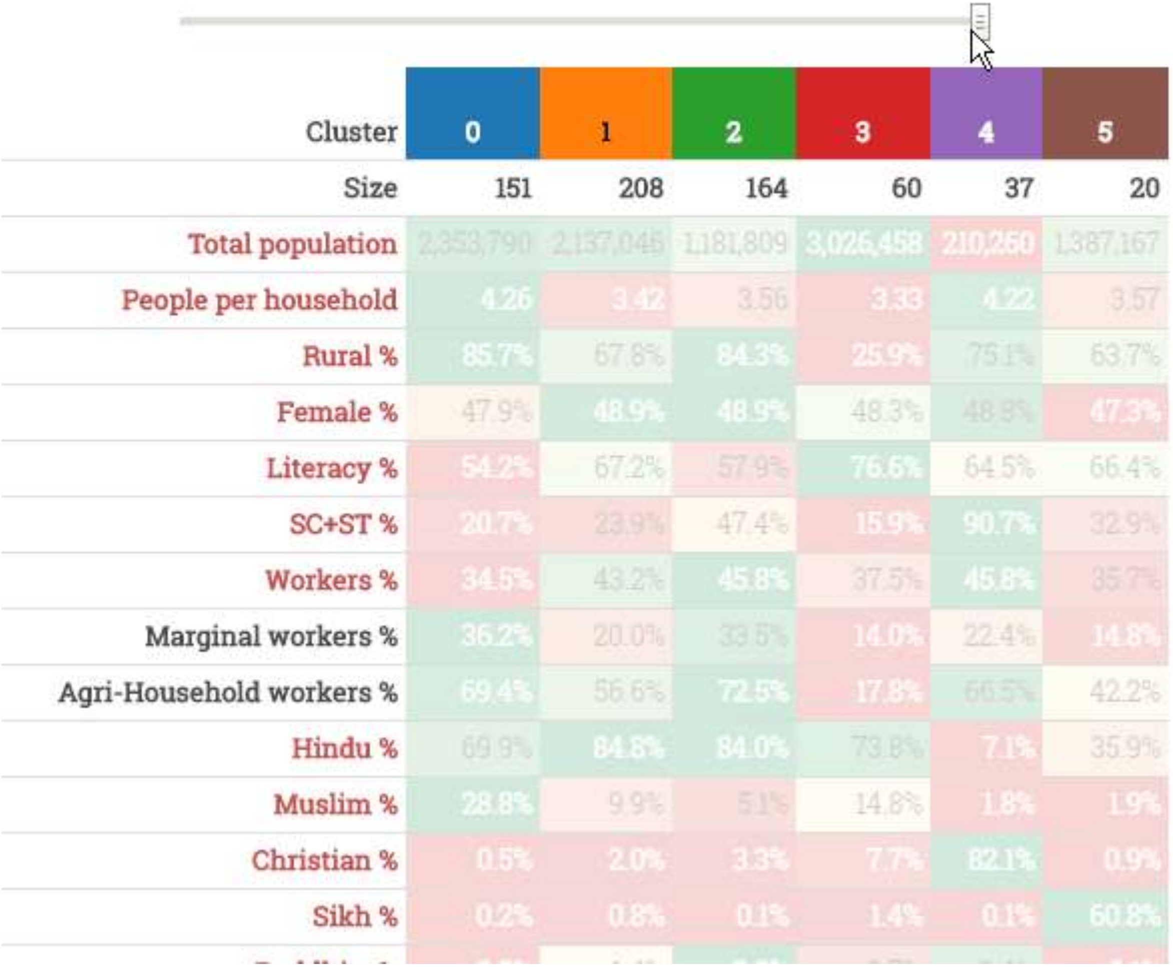
Rich: Urban educated population working in skilled jobs. All metros, large cities, parts of Kerala, TN

Business: Lower education but working in skilled jobs, and prosperous. Typical of business communities. Parts of Gujarat, TN, Urban UP, Punjab, etc.

THIS IS A FRAGMENT OF THE CONFIGURATION USED FOR THE OUTPUT

Our analytics team (who have never programmed in Python) were able to create the entire cluster setup in a few hours.

```
name: India Districts
csv: india-districts-census-2011.csv
columns:
  population:
    name: Total population
    value: Population
    scale: log
    description: Number of people
  household_size:
    name: People per household
    formula: Population / Households
rural_pc:
    name: Rural %
    formula: Rural_HH / Households
    description: % of rural households
clustering:
  kmeans:
    name: K-Means
    algo: KMeans
    description: Group closes points
    n_clusters: 6
  ...
```



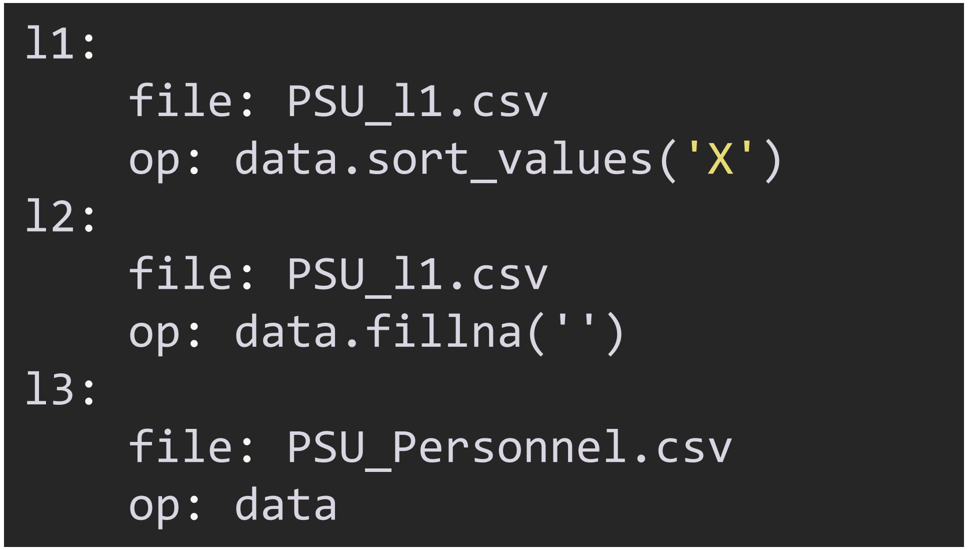
Cluster	0	1	2	3	4	5
Size	151	208	164	60	37	20
Total population	2,353,790	2,137,046	1,181,809	3,026,458	210,260	1,387,167
People per household	4.26	3.42	3.56	3.33	4.22	3.57
Rural %	85.7%	67.8%	84.3%	25.9%	75.1%	63.7%
Female %	47.9%	48.9%	48.9%	48.3%	48.8%	47.3%
Literacy %	54.2%	67.2%	57.9%	76.6%	64.5%	66.4%
SC+ST %	20.7%	23.9%	47.4%	15.9%	90.7%	32.9%
Workers %	34.5%	43.2%	45.8%	37.5%	45.8%	35.7%
Marginal workers %	36.2%	20.0%	33.5%	14.0%	22.4%	14.8%
Agri-Household workers %	69.4%	56.6%	72.5%	17.8%	66.5%	42.2%
Hindu %	69.9%	84.8%	84.0%	73.8%	7.1%	35.9%
Muslim %	28.8%	9.9%	5.1%	14.8%	1.8%	1.9%
Christian %	0.5%	2.0%	3.3%	7.7%	82.1%	0.9%
Sikh %	0.2%	0.8%	0.1%	1.4%	0.1%	60.8%

But, no functions in data

… or can there be?

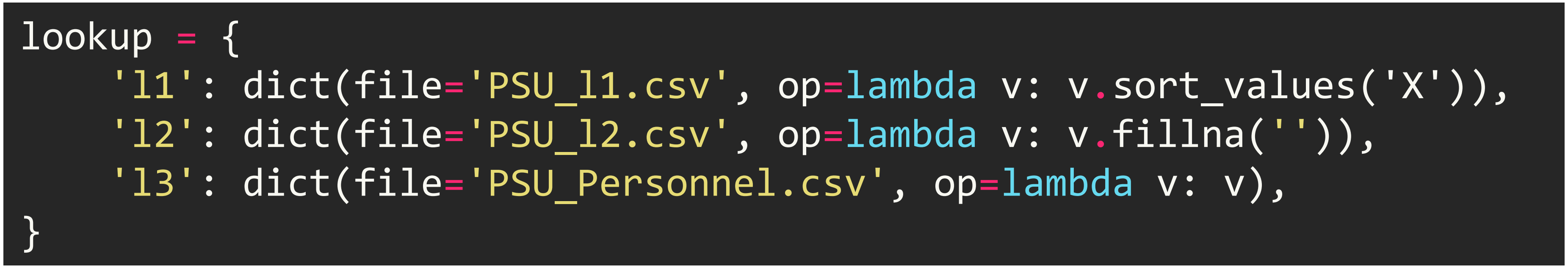
CAN WE JUST PUT THE FUNCTIONS IN THE YAML FILE?

How can we make this YAML file…



```
l1:
    file: PSU_l1.csv
    op: data.sort_values('X')
l2:
    file: PSU_l1.csv
    op: data.fillna('')
l3:
    file: PSU_Personnel.csv
    op: data
```

… compile into this data structure?



```
lookup = {
    'l1': dict(file='PSU_l1.csv', op=lambda v: v.sort_values('X')),
    'l2': dict(file='PSU_l2.csv', op=lambda v: v.fillna('')),
    'l3': dict(file='PSU_Personnel.csv', op=lambda v: v),
}
```

Yes. Python can compile Python code

This function compiles an expression into a function that takes a single argument: **data**

```python
def build_transform(expr):
    body = ['def transform(data):']
    body.append('    return %s' % expr)
    code = compile(''.join(body), filename='compiled', mode='exec')
    context = {}
    exec(code, context)
    return context['transform']
```

Here's an example of how it is used:

```python
>>> incr = build_transform('data + 1')
>>> incr(10)
11
```

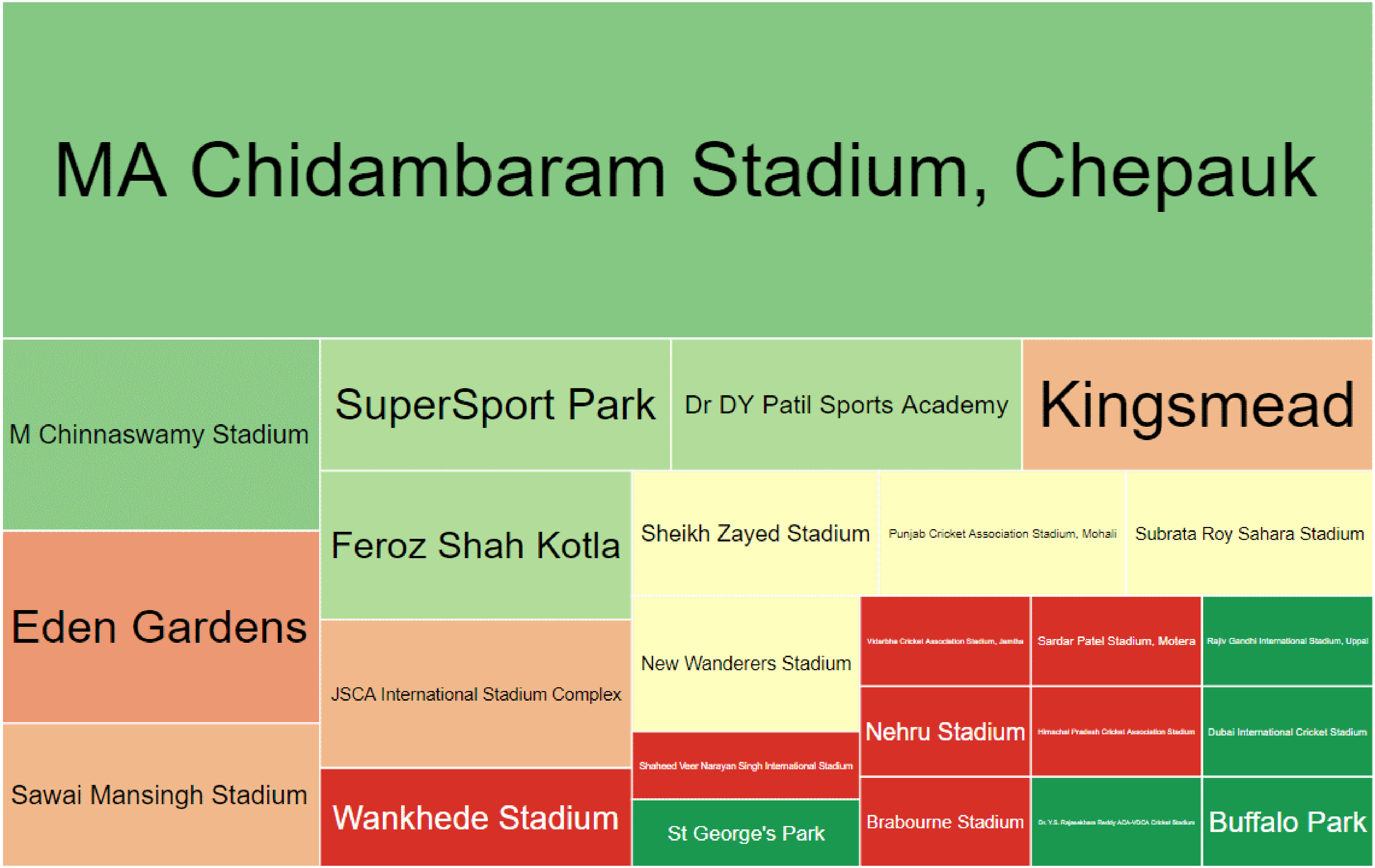
We'll need to handle imports, arbitrary input variables, caching, etc. But this is its core.

THIS IS, INCIDENTALLY, HOW TORNADO TEMPLATES WORK

We put this into a Data Explorer application

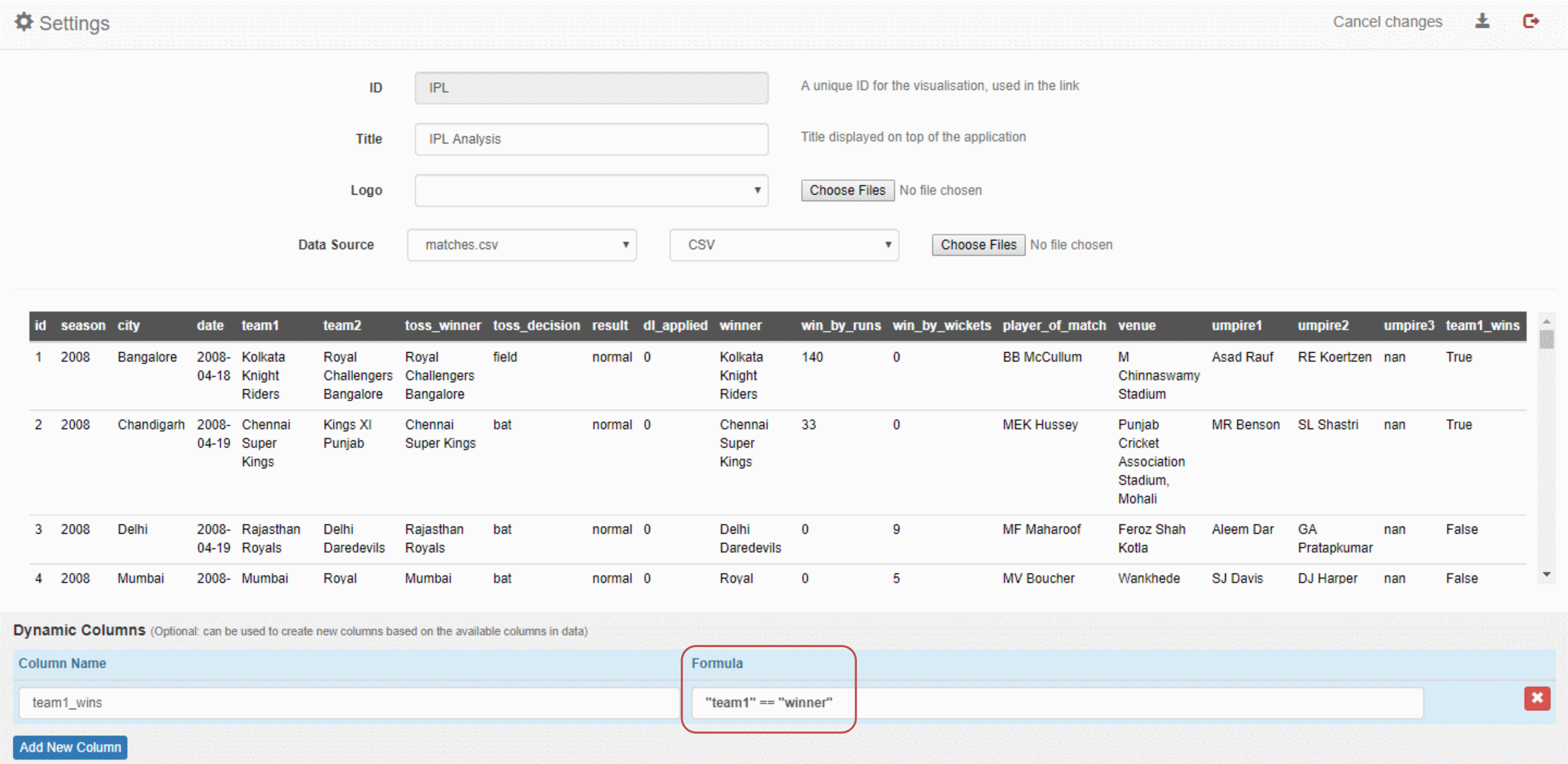
Link

Chennai Super Kings IPL win rate by stadium



It lets users create their own metrics

LINK



ID: IPL — A unique ID for the visualisation, used in the link

Title: IPL Analysis — Title displayed on top of the application

Logo: ▾ — Choose Files No file chosen

Data Source: matches.csv ▾ — CSV ▾ — Choose Files No file chosen

id	season	city	date	team1	team2	toss_winner	toss_decision	result	dl_applied	winner	win_by_runs	win_by_wickets	player_of_match	venue	umpire1	umpire2	umpire3	team1_wins
1	2008	Bangalore	2008-04-18	Kolkata Knight Riders	Royal Challengers Bangalore	Royal Challengers Bangalore	field	normal	0	Kolkata Knight Riders	140	0	BB McCullum	M Chinnaswamy Stadium	Asad Rauf	RE Koertzen	nan	True
2	2008	Chandigarh	2008-04-19	Chennai Super Kings	Kings XI Punjab	Chennai Super Kings	bat	normal	0	Chennai Super Kings	33	0	MEK Hussey	Punjab Cricket Association Stadium, Mohali	MR Benson	SL Shastri	nan	True
3	2008	Delhi	2008-04-19	Rajasthan Royals	Delhi Daredevils	Rajasthan Royals	bat	normal	0	Delhi Daredevils	0	9	MF Maharoof	Feroz Shah Kotla	Aleem Dar	GA Pratapkumar	nan	False
4	2008	Mumbai	2008-	Mumbai	Royal	Mumbai	bat	normal	0	Royal	0	5	MV Boucher	Wankhede	SJ Davis	DJ Harper	nan	False

Dynamic Columns (Optional: can be used to create new columns based on the available columns in data)

Column Name	Formula
team1_wins	"team1" == "winner"

Getting data from code

Can we actually inspect code to re-use its metadata?

How can we test our `build_transform`?

These two methods should be exactly the same.

```python
method = build_transform('data + 1')

def transform(data):
    return data + 1
```

How can we write a test case comparing 2 functions?

```python
from nose.tools import eq_

def eqfn(a, b):
    eq_(a.__code__.co_code, b.__code__.co_code)
    eq_(a.__code__.co_argcount, b.__code__.co_argcount)
```

WE'RE LEARNING MORE ABOUT THE CODE ITSELF

Here's a simple timer

```python
import timeit
_time = {'last': timeit.default_timer()}

def timer(msg):
    end = timeit.default_timer()
    print('%0.3fs %s' % (end - _time['last'], msg))
    _time['last'] = end
```

It prints the time taken since its last call:

```
>>> import time
>>> timer('start')
0.000s start
>>> time.sleep(0.5)
>>> timer('slept')
0.500s slept
```

CAN IT AUTOMATICALLY PRINT THE CALLER LINE NUMBER?

USE THE INSPECT MODULE TO INSPECT THE STACK

```python
import inspect

def caller():
    '''caller() returns caller's "file:function:line"'''
    parent = inspect.getouterframes(inspect.currentframe())[2]
    return '[%s:%s:%d]' % (parent[1], parent[3], parent[2])
```

```python
import time
import timeit
_time = {'last': timeit.default_timer()}

def timer(msg=None):
    end = timeit.default_timer()
    print('%0.3fs %s' % (end - _time['last'], msg or caller()))
    _time['last'] = end

timer()                 # Prints 0.000s [test.py:<module>:17]
time.sleep(0.4)
timer()                 # Prints 0.404s [test.py:<module>:19]
time.sleep(0.2)
```

OPEN FILE RELATIVE TO THE CALLER FUNCTION

Data files are stored in the same directory as the code, but the current directory is different

This code pattern is very common:

```
folder = os.path.dirname(os.path.abspath(__file__))
path = os.path.join(folder, 'data.csv')
data = pd.read_csv(path)
```

It is used across several modules in several files

We can convert this into a re-usable function.
But since **`__file__`** varies from module to module, it needs to be a parameter.

```
def open_csv(file, source):
    folder = os.path.dirname(os.path.abspath(source))
    path = os.path.join(folder, file)
    return pd.read_csv(path)

data = open_csv('data.csv', __file__)
```

INSPECT COMES TO OUR RESCUE AGAIN

We can completely avoid passing the source **`__file__`** because **`inspect`** can figure it out.

```python
def open_csv(file):
    stack = inspect.getouterframes(inspect.currentframe(), 2)
    folder = os.path.dirname(os.path.abspath(stack[1][1]))
    path = os.path.join(folder, path)
    return pd.read_csv(path)
```

Now, opening a data file relative to the current module is trivial:

```python
data = open_csv('data.csv')
```

I keep telling people this repeatedly:

Don't Repeat Yourself

I was repeating myself

AUTOMATING CODE REVIEWS

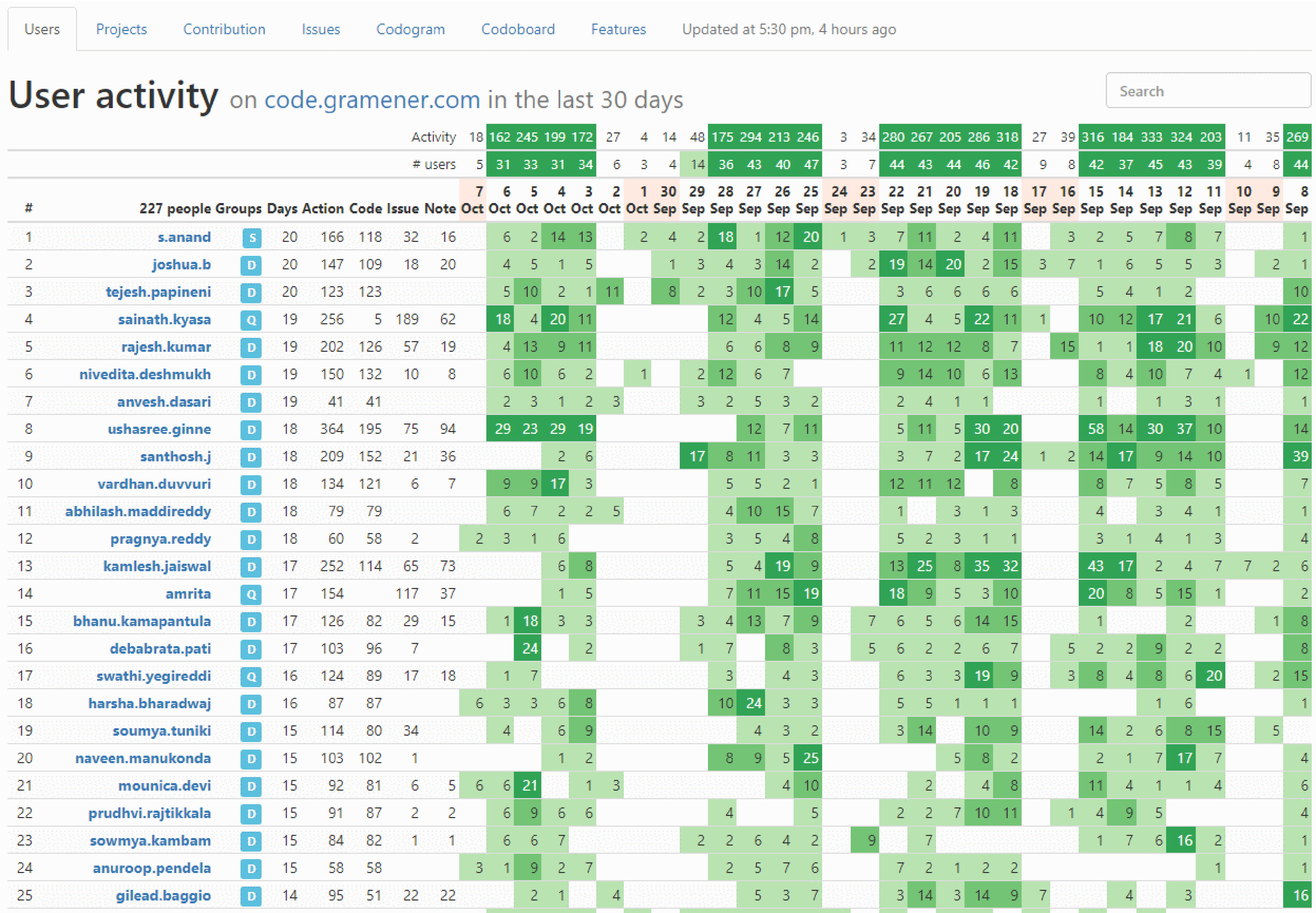
ADVENTURES IN AUTOMATED NIT-PICKING

Gramener
A Data Science Company

The first challenge is finding code

Not everyone was committing code into our Gitlab instance

WE GAMIFIED IT TO TRACK ACTIVITY, AND REWARDED REGULARITY



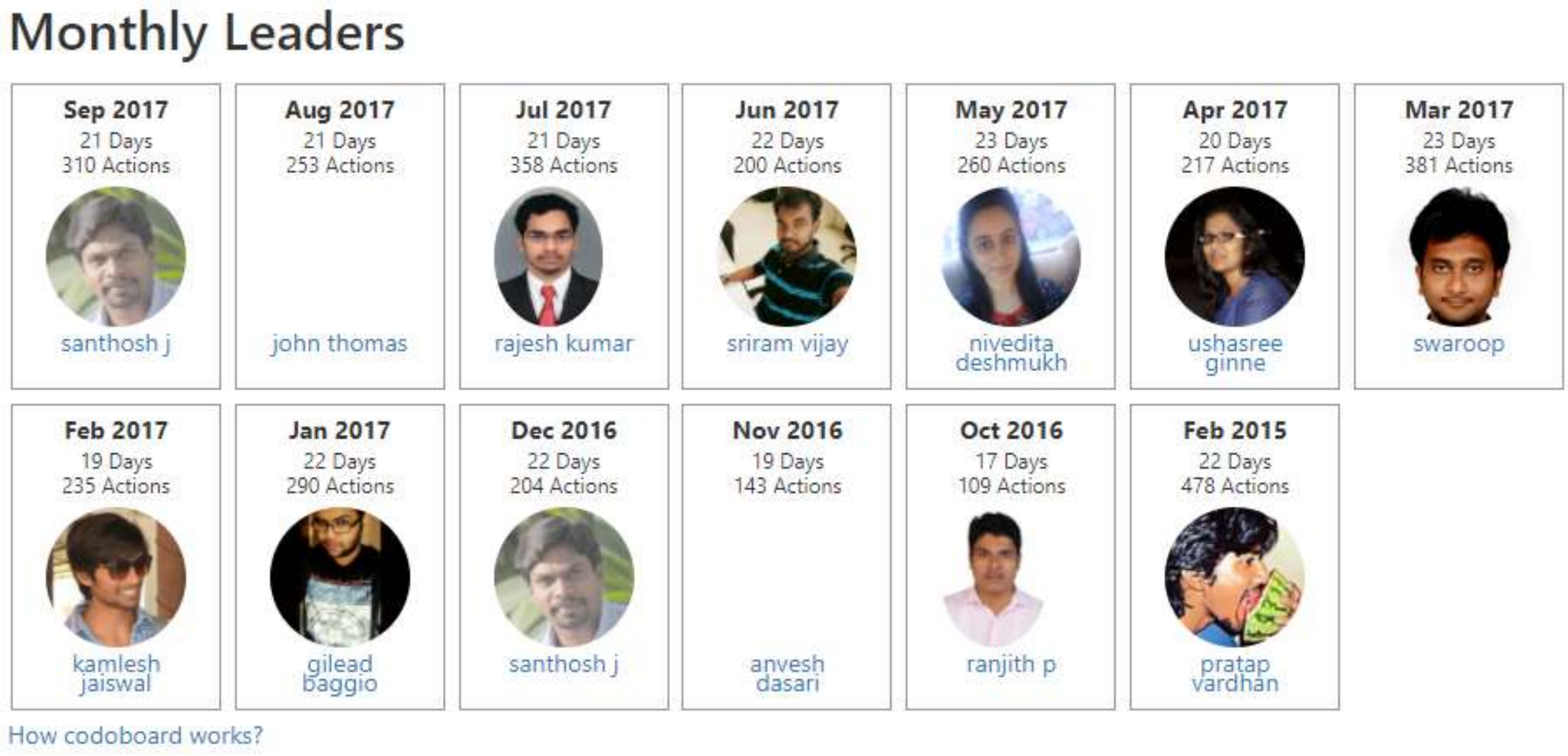
Users | Projects | Contribution | Issues | Codogram | Codoboard | Features | Updated at 5:30 pm, 4 hours ago

User activity on code.gramener.com in the last 30 days

Search

#	227 people	Groups	Days	Action	Code	Issue	Note	7 Oct	6 Oct	5 Oct	4 Oct	3 Oct	2 Oct	1 Oct	30 Sep	29 Sep	28 Sep	27 Sep	26 Sep	25 Sep	24 Sep	23 Sep	22 Sep	21 Sep	20 Sep	19 Sep	18 Sep	17 Sep	16 Sep	15 Sep	14 Sep	13 Sep	12 Sep	11 Sep	10 Sep	9 Sep	8 Sep
							Activity	18	162	245	199	172	27	4	14	48	175	294	213	246	3	34	280	267	205	286	318	27	39	316	184	333	324	203	11	35	269
							# users	5	31	33	31	34	6	3	4	14	36	43	40	47	3	7	44	43	44	46	42	9	8	42	37	45	43	39	4	8	44
1	s.anand	S	20	166	118	32	16		6	2	14	13		2	4	2	18	1	12	20	1	3	7	11	2	4	11		3	2	5	7	8	7			1
2	joshua.b	D	20	147	109	18	20		4	5	1	5			1	3	4	3	14	2		2	19	14	20	2	15	3	7	1	6	5	5	3		2	1
3	tejesh.papineni	D	20	123	123				5	10	2	1	11		8	2	3	10	17	5			3	6	6	6	6			5	4	1	2				10
4	sainath.kyasa	Q	19	256	5	189	62		18	4	20	11					12	4	5	14			27	4	5	22	11	1		10	12	17	21	6		10	22
5	rajesh.kumar	D	19	202	126	57	19		4	13	9	11					6	6	8	9			11	12	12	8	7		15	1	1	18	20	10		9	12
6	nivedita.deshmukh	D	19	150	132	10	8		6	10	6	2		1		2	12	6	7				9	14	10	6	13			8	4	10	7	4	1		12
7	anvesh.dasari	D	19	41	41				2	3	1	2	3			3	2	5	3	2			2	4	1	1				1		1	3	1			1
8	ushasree.ginne	D	18	364	195	75	94		29	23	29	19						12	7	11			5	11	5	30	20			58	14	30	37	10			14
9	santhosh.j	D	18	209	152	21	36				2	6				17	8	11	3	3			3	7	2	17	24	1	2	14	17	9	14	10			39
10	vardhan.duvvuri	D	18	134	121	6	7		9	9	17	3					5	5	2	1			12	11	12		8			8	7	5	8	5			7
11	abhilash.maddireddy	D	18	79	79				6	7	2	2	5				4	10	15	7			1		3	1	3			4		3	4	1			1
12	pragnya.reddy	D	18	60	58	2		2	3	1	6						3	5	4	8			5	2	3	1	1			3	1	4	1	3			4
13	kamlesh.jaiswal	D	17	252	114	65	73				6	8					5	4	19	9			13	25	8	35	32			43	17	2	4	7	7	2	6
14	amrita	Q	17	154		117	37				1	5					7	11	15	19			18	9	5	3	10			20	8	5	15	1			2
15	bhanu.kamapantula	D	17	126	82	29	15		1	18	3	3				3	4	13	7	9		7	6	5	6	14	15			1			2			1	8
16	debabrata.pati	D	17	103	96	7				24		2				1	7		8	3		5	6	2	2	6	7		5	2	2	9	2	2			8
17	swathi.yegireddi	Q	16	124	89	17	18		1	7							3		4	3			6	3	3	19	9		3	8	4	8	6	20		2	15
18	harsha.bharadwaj	D	16	87	87			6	3	3	6	8					10	24	3	3			5	5	1	1	1					1	6				1
19	soumya.tuniki	D	15	114	80	34			4		6	9						4	3	2			3	14		10	9			14	2	6	8	15		5	
20	naveen.manukonda	D	15	103	102	1					1	2					8	9	5	25					5	8	2			2	1	7	17	7			4
21	mounica.devi	D	15	92	81	6	5	6	6	21		1	3						4	10				2		4	8			11	4	1	1	4			6
22	prudhvi.rajtikkala	D	15	91	87	2	2		6	9	6	6					4			5			2	2	7	10	11		1	4	9	5					4
23	sowmya.kambam	D	15	84	82	1	1		6	6	7					2	2	6	4	2		9		7						1	7	6	16	2			1
24	anuroop.pendela	D	15	58	58			3	1	9	2	7					2	5	7	6			7	2	1	2	2							1			1
25	gilead.baggio	D	14	95	51	22	22			2	1		4					5	3	7			3	14	3	14	9	7			4		3				16

WE GAVE MONTHLY AWARDS TO THE TOPPERS



Monthly Leaders

Sep 2017
21 Days
310 Actions
santhosh j

Aug 2017
21 Days
253 Actions
john thomas

Jul 2017
21 Days
358 Actions
rajesh kumar

Jun 2017
22 Days
200 Actions
sriram vijay

May 2017
23 Days
260 Actions
nivedita deshmukh

Apr 2017
20 Days
217 Actions
ushasree ginne

Mar 2017
23 Days
381 Actions
swaroop

Feb 2017
19 Days
235 Actions
kamlesh jaiswal

Jan 2017
22 Days
290 Actions
gilead baggio

Dec 2016
22 Days
204 Actions
santhosh j

Nov 2016
19 Days
143 Actions
anvesh dasari

Oct 2016
17 Days
109 Actions
ranjith p

Feb 2015
22 Days
478 Actions
pratap vardhan

How codoboard works?

DIDN'T HELP. WE GOT MANAGERS TO ENFORCE COMMITS

Gamification works at the top

Processes & Rules work better at the bottom

BUT AT LAST, WE HAD ALL COMMITS IN ONE PLACE

THESE ARE OUR TOP ERRORS

1. Missing encoding when opening files
2. Printing unformatted numbers. e.g. `3.1415926535` instead of `3.14`
3. Magic constants.
   e.g. `x = v / 86400` instead of `x = v / seconds_per_day`
4. Non-vectorization
5. Local variable is assigned to but never used
6. Module imported but unused
7. Uninitialized variable used
8. Redefinition of unused variable
9. Blind `except:` statement
10. Dictionary key repeated with different values. e.g. `{'x': 1, 'x': 2}`

FLAKE8 DOES NOT CHECK FOR ALL. LET'S WRITE A PLUGIN

A FLAKE8 PLUGIN IS A CALLABLE WITH A SET OF ARGUMENTS

Flake8 inspects the plugin's signature to determine what parameters it expects. When processing a file, a plugin can ask for any of the following:

- `filename`
- `lines`
- `verbose`
- `tree`
- …

```
def parameters_for(plugin):
    func = plugin.plugin
    is_class = not inspect.isfunction(func)
    if is_class:
        func = plugin.plugin.__init__
    argspec = inspect.getargspec(func)
    start_of_optional_args = len(argspec[0]) - len(argspec[-1] or [])
    parameter_names = argspec[0]
    parameters = collections.OrderedDict([
        (name, position < start_of_optional_args)
        for position, name in enumerate(parameter_names)
    ])
    if is_class:
        parameters.pop('self', None)
    return parameters
```

IT ACCEPTS AN AST TREE THAT WE CAN PARSE

Let's take this file, `test.py`, as an example and parse it.

```
# test.py
import six
def to_str(val):
    return six.text_type(str(val))
```

- Parsing it returns a tree.
- The tree has a body attribute.
- The body is a list of nodes.
- The first node is an `Import` node.
- It has a list of names of imported modules
- The second is a `Function` node.
- It has a name and an argument spec
- It also has a body, which is a `Return` node, and has a value which is a `Call` node.

- In short, the Python program has been parsed into a **data structure**

```
>>> import ast
>>> tree = ast.parse(open('test.py').read())
>>> tree.body
<_ast.Import>, <_ast.FunctionDef>]
>>> ast.dump(tree.body[0])
"Import(names=[alias(name='six',
asname=None)])"
>>> type(tree.body[1])
_ast.FunctionDef
>>> tree.body[1].name
'to_str'
>>> ast.dump(tree.body[1].args)
'''arguments(
    args=[arg(arg='val', annotation=None)],
    vararg=None,
    kwonlyargs=[],
    kw_defaults=[],
    kwarg=None,
    defaults=[]
)'''
```

LET'S CHECK FOR LACK OF NUMBER FORMATTING

A classing issue is using `str` instead of formatting functions. We can check for all functions to see if it's an `str`

```
>>> for node in ast.walk(tree):
>>>     if isinstance(node, ast.Call):
>>>         print(ast.dump(node.func))
Attribute(value=Name(id='six', ctx=Load()), attr='text_type',
ctx=Load())
Name(id='str', ctx=Load())
```

This is, in fact, how many flake8 plugins work. See the [source]()

Today, each of 27 live projects is lint free

This happened just this week, after 3 months of effort!

Take-aways

- Use loops to avoid duplication
- Group common code into functions
- Prefer data over functions
  - Use data structures to handle variations in code
- Keep data in data files
- Prefer YAML over JSON
- Simple code can be embedded in data
- Code is a data structure. Inspect & modify it

THANK YOU

HAPPY TO TAKE QUESTIONS

Gramener
A Data Science Company