

# AOP & MOP in Python and JavaScript

Authors:

# Python MOP

Every dataset should come with a domain for each attribute.

The validation process makes sure that the data falls between these ranges to be used smoothly further in the application.

#	Attribute	Domain
0.	Sample code number	id number
1.	Class:	(2 for benign, 4 for malignant)
2.	Clump Thickness	1 - 10
3.	Uniformity of Cell Size	1 - 10
4.	Uniformity of Cell Shape	1 - 10
5.	Marginal Adhesion	1 - 10
6.	Single Epithelial Cell Size	1 - 10
7.	Bare Nuclei	1 - 10
8.	Bland Chromatin	1 - 10
9.	Normal Nucleoli	1 - 10
10.	Mitoses	1 - 10

# Python MOP

Cleaning the data before validating a certain dataset has the role to delete only the invalid rows that contain errors.

With this approach we avoid losing a majority of good rows because of just a few invalid ones.

id	class	clump thickness	uniformity of cell size	uniformity of cell shape	marginal adhesion	single epithelial cell size	bare nuclei	bland chromatin	normal nucleoli	mitoses
1002945	2	1	4	4	5	7	10	3	2	1
1015425	2	3	1	75	1	2	2	3	1	1
1016277	2	6	1234	8	1	3	175	3	7	1
1017023	2	4	1	1	83	2	1	3	1	1
1017122	4	8	10	10	8	7	10	9	7	1
1018099	2	1	1	1	1	2	10	3	1	1
1018561	2	2	1	2	1	2	1	3	1	1
1033078	2	2	12222	1	1	2	1	1	1	5
1033078	2	4	2	1	1	2	1	2	1	1
1035283	2	1	1	1	1	1	1	3	1	1
1036172	2	2	1	1	1	2	1	2	1	1
1041801	4	5	3	3	3	2	3	4	4	1
1043999	2	1	1	1	1	2	3	3	1	1
1044572	4	8	7	5	10	7	9	5	5	4
1047630	4	7	4	6	4	6	1	4	3	1

# Python MOP

The cleaning function drops the rows that contain different values than expected and the validation function determines if the dataset can be used or not.

```
def clean_wrong_rows(dir, df, project_name):
    print("Executing CLEAN")
    possible_columns = get_possible_columns_dict(project_name, df.columns)

    for col in df.columns:
        if possible_columns[col]:
            wrong_values = set(df[col]).difference(possible_columns[col])
            for value in wrong_values:
                df.drop(df.loc[df[col]==value].index, inplace=True)

    return df

def validate_csv(dir, df, project_name):
    print("Executing VALIDATE")

    official_columns = ["id", "class", "clump thickness", "uniformity of cell size",
                        "uniformity of cell shape", "marginal adhesion", "single epithelial cell size",
                        "bare nuclei", "bland chromatin", "normal nucleoli", "mitoses"]

    columns_from_csv = df.columns
    if set(columns_from_csv) != set(official_columns):
        return False

    possible_columns = get_possible_columns_dict(project_name, columns_from_csv)

    for col in columns_from_csv:
        if possible_columns[col]:
            if set(df[col]) - possible_columns[col] != set():
                return False

    return True
```

# Python MOP

The “spec\_validate\_then\_send” function verifies not to forward a CSV if it was not validated before. Also, if there was an upload of a new CSV, it also has to be verified before forwarding it.

```
49 # Forward CSV without validating in between /
50 @rv.spec(history_size=5)
51 @rv.monitor(validate=trusted_middleware.valid_csv, send=trusted_middleware.forward_csv, upload=trusted_middleware.upload)
52 def spec_validate_then_send(event):
53     if event.fn.send.called:
54         count=0
55         count_validate=0
56         count_upload=0
57         for old_event in event.history:
58             count+=1
59             if old_event.called_function==old_event.fn.validate:
60                 count_validate=count
61             if old_event.called_function==old_event.fn.upload:
62                 count_upload=count
63         assert count_upload>count_validate , "Forward CSV without validating it between"
64 # @rv.monitor(send=trusted_middleware.upload)
```

# Python MOP

The “simple\_specification” function verifies if function “valid\_csv” returns bool type.

```
9
10 @rv.monitor(up=trusted_middleware.valid_csv)
11 @rv.spec(when=rv.POST)
12 def simple_specification(event):
13     assert type(event.fn.up.result) is bool
14     print("result of validation is bool: OK")
15
```

The “more\_specifications2” function verifies if after each call of the function “simulate\_interraction”, it is called the “decrypt” function.

```
27
28 @rv.monitor(si=trusted_middleware.simulate_interraction, d=trusted_middleware.decrypt)
29 @rv.spec(when=rv.POST, history_size=20)
30 def more_specifications2(event):
31
32     if event.called_function == event.fn.si:
33         event.next_called_should_be(event.fn.d)
34
35
```

# Python MOP

Monitoring makes sure to start the cleaning function each time the validation takes place in case it was omitted. This way we make sure to keep good data that without being cleaned would be discarded in the validation process.

```
from pythonrv import rv
import mop_valid

@rv.monitor(clean=mop_valid.clean_wrong_rows, validate=mop_valid.validate_csv)
def specifications(event):
    if event.called_function == event.fn.validate:
        if len([old_event for old_event in event.history if old_event.called_function == old_event.fn.clean]) == 0:
            print("Starting CLEAN from monitoring function")
            df = mop_valid.read_df_from_csv("./Breast_Cancer_Data", "invalid_data.csv")
            df = mop_valid.clean_wrong_rows("./Breast_Cancer_Data", df, "Breast_Cancer_Data")
            assert len([old_event for old_event in event.history if old_event.called_function == old_event.fn.clean]) > 0

df = mop_valid.read_df_from_csv("./Breast_Cancer_Data", "invalid_data.csv")
# df = mop_valid.clean_wrong_rows("./Breast_Cancer_Data", df, "Breast_Cancer_Data")
response = mop_valid.validate_csv("./Breast_Cancer_Data", df, "Breast_Cancer_Data")
```

# JavaScript AOP

- Using the help of *kaop-ts* library, we can import AOP support such as *afterMethod* and *beforeMethod* and create custom annotation to use in our production code.
- In this example, our target is to create two annotations that are used to log when the execution of the program enters and exits from a functions code block. The enter arguments and the exit result are also provided in our logs.

```
import { beforeMethod, afterMethod } from "kaop-ts";

const EnteringMethodAspect = function(meta){
  const methodName = `${meta.target.constructor.name}::${meta.method.name}`;
  const args = JSON.stringify(meta.args);
  console.info(`Entering ${methodName} with parameters ${args}!`);
};

const ExitingMethodAspect = function(meta){
  const methodName = `${meta.target.constructor.name}::${meta.method.name}`;
  const result = JSON.stringify(meta.result);
  console.info(`Exiting ${methodName} with result ${result}!`);
};

export const EnteringLog = beforeMethod(EnteringMethodAspect);
export const ExitingLog = afterMethod(ExitingMethodAspect);
```

```
@EnteringLog
@ExitingLog
build_id3_node(attributes, label, possible_values, decisions) {...}
```

# Javascript AOP

Basic annotations are useful, but their power comes from customization (parametrization for decorators).

In *kaop-ts*, this can be achieved by using *currying*, in our specific case, logging, by wrapping our initial decorator inside another generator method.

On the left, an example of parameterization is provided. The method we wish to log is decorated as follows:

```
@LogEnter('/tmp/ID3.log')
@LogExit('/tmp/ID3.log')
build_id3_node(attributes, label, possible_values,
decisions) {
  ...
}
```

```
export function LogEnter(filename) {
  return beforeMethod(meta => {
    const message = createEnterMethodMessage(meta);
    logMessage(filename, wrapInDate(message),
onError(filename, meta));
  });
}

export function LogExit(filename) {
  return afterMethod(meta => {
    const message = createExitMethodMessage(meta);
    logMessage(filename, wrapInDate(message),
onError(filename, meta));
  });
}
```

# JavaScript AOP

- Still, annotations functionality in JavaScript are in testing so developers must pay attention if they really need them and must make some configuration to enable annotations
- The configuration file is named **.babelrc** which should be placed in the root of the project and could look like in the image from the right side of the slide
- Babel requirements should be installed as well

```
{
  "presets": [
    "@babel/preset-env"
  ],
  "plugins": [
    [
      "@babel/plugin-proposal-decorators",
      {
        "legacy": true
      }
    ]
  ]
}
```

# JavaScript/React MOP

- In JavaScript Framework ReactJS we can simulate MOP by using the built in lifecycle events
- For example, in a React component that handles file upload (training dataset in our case) at some specific point our upload progress could go over 100%
- The easy fix for this is to implement the component lifecycle function ***componentDidUpdate*** to check the previous state of the component (upload progress) and correct it for the user to see a logic value

```
constructor(props) {
  super(props);
  this.state = {
    dataSetPermission: DATA_SET_PERMISSIONS.PUBLIC,
    uploadProgress: 0
  };
}

handlePermissionChange = event => {
  this.setState({ dataSetPermission: event.target.value });
}

componentDidUpdate(previousProps, previousState) {
  if (previousState.uploadProgress > 100) {
    this.setState({ uploadProgress: 100 })
  }
}

shouldComponentUpdate(nextProps, nextState) {
  return nextState.uploadProgress <= 100;
}
```

# Util Links

1. [Python MOP] <https://pypi.org/project/pythonrv/>
2. [Python AOP] <https://pypi.org/project/aspectlib/>
3. [JavaScript AOP] <https://www.npmjs.com/package/kaop-ts>
4. [React Lifecycle] <https://reactjs.org/docs/react-component.html>