Introducing MESCAL

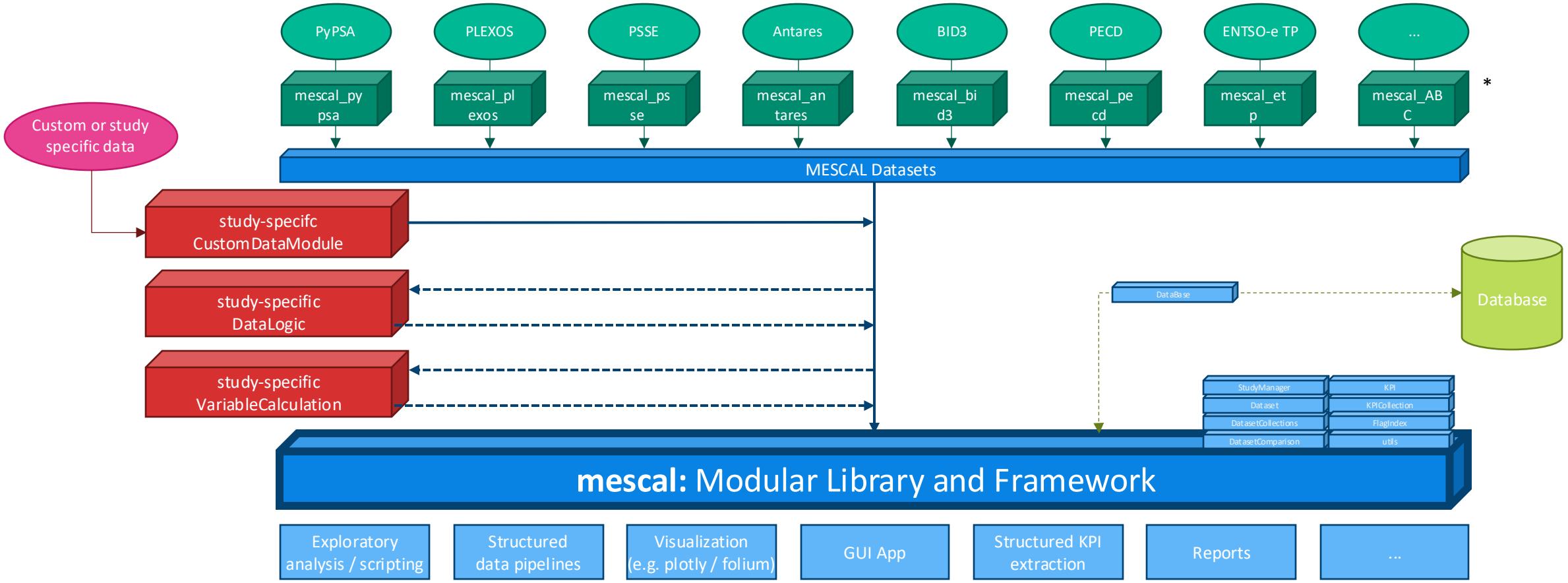
Modular Energy Scenario Comparison Analysis Library

Helge Esch

Agenda

1) **MESCAL:** A modular library and framework
2) **StudyManager:** A structured approach to dealing with multi-scenario studies
3) **Dataset class:** The ultimate modular class to combine, link, merge, concatenate, compare various Datasets
4) **A unique relationship:** Integrating Model DFs and Variable Time-Series DFs
5) **Shared Repo Structure:** Template Structure for a Shared Studies-Repo
6) And there's so much more...

One Modular Library and Framework to Process and Analyze All Energy System Studies

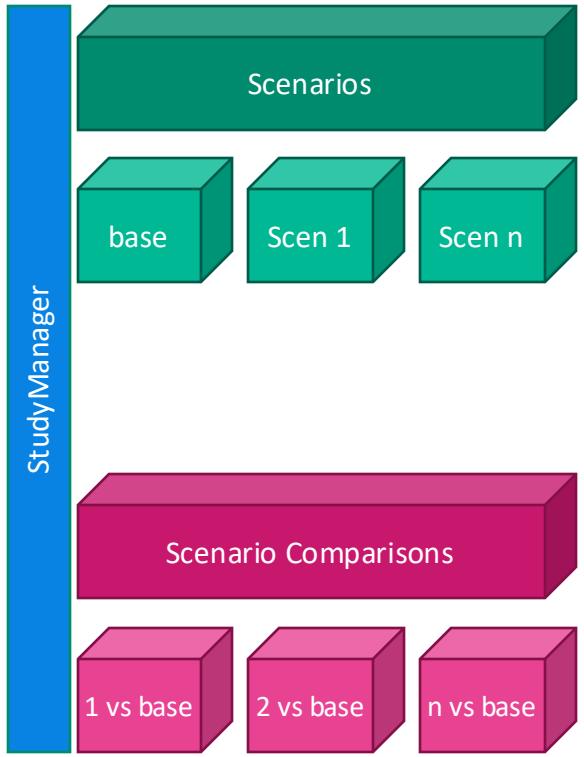


*Each platform-interface is a separate package that implements the abstract classes from the generic mescal package.

Multi-Scenario Data Fetching with the StudyManager

1) StudyManager setup

study = StudyManager(...)



→ All Datasets provide the same functionality.

2) Fetching DFs from Datasets for flag = 'ST.BiddingZone.MarketPrice'

2.1 Fetch DF for all scenarios (MultiIndex)

study.scen.fetch(flag)

	base		...	Scen n
	BZ A	BZ B	...	BZ X
00:00	2.2	5.0	...	1.2
01:00	2.5	5.1	...	1.3
02:00	2.4	5.0	...	1.4
...	...	...	...	...

2.2 Fetch DF for single scenario

study.scen.get_dataset('base').fetch(flag)

	BZ A	BZ B	...	BZ X
00:00	2.2	5.0	...	1.0
01:00	2.5	5.1	...	1.0
02:00	2.4	5.0	...	1.0
...	...	...	...	...

2.3 Fetch DF for all comparisons (MultiIndex)

study.comp.fetch(flag)

	1 vs base		...	n vs base
	BZ A	BZ B	...	BZ X
00:00	+0.2	-0.3	...	+2.1
01:00	+0.2	-0.5	...	+2.1
02:00	+0.1	-0.2	...	+2.2
...	...	...	...	...

2.4 Fetch DF for single comparison

study.comp.get_dataset('1 vs base').fetch(flag)

	BZ A	BZ B	...	BZ X
00:00	+0.2	-0.3	...	+0.0
01:00	+0.2	-0.5	...	+0.1
02:00	+0.1	-0.2	...	+0.0
...	...	...	...	...

Everything is a Dataset!

MESCAL's ultimate modular class to handle data

1) Dataset class and its primary methods

```python
class Dataset(ABC):
    def __init__(
        name: str = None,
        attributes: Dict = None,
        parent_dataset: Dataset = None,
        flag_index: FlagIndex = None,
        database: Database = None,
    )

    @abstractmethod
    def accepted_flags() -> set[Flagtype]

    @flag_must_be_accepted
    def fetch(flag: Flagtype, **kwargs) -> pd.DataFrame | pd.Series

    def add_kpi(kpi: Union[KPI, KPIFactory])

    def get_kpi_collection() -> KPICollection

    ...
```

2) Collections of Datasets and key underlying logic

```python
class DatasetCollection(Dataset, ABC):
    """Abstract class to collect multiple Dataset instances
       and handle them according to a specific logic.
       Inherits all methods / functionalities from Dataset."""

class DatasetLinkCollection(DatasetCollection):
    """Links multiple Dataset instances so that:
       - the parent-Dataset accepts flags of all child-Datasets and automatically
         the data from the child-Dataset one that accepts the flag.
       - the child-Dataset instances have access to the parent-Dataset so that they can
         fetch from other, e.g. child_ds.parent_dataset.fetch(...)."""

class DatasetMergeCollection(DatasetCollection):
    """"Fetch method will merge fragmented Datasets for same flag, e.g.:
       - fragmented simulation runs, e.g. CW1, CW2, CW3, CWn.
       - fragmented data sources, e.g. mapping from Excel file
         with model from simulation platform."""

class DatasetConcatCollection(DatasetCollection):
    """"Fetch method will return a concatenation of all child-Datasets
       with an additional Index-level."""

class DatasetComparison(DatasetCollection):
    """"Takes two Datasets (variation and reference) and
       fetch method will return the delta between the two (var-ref)."""
```

*For readability purposes, all self attributes are removed from dummy code.

→ **Fully modular:** All collection-types can be combined / nested with each other.

Two DataFrame Formats with a Unique Purpuse: Integrating Model DFs and Variable Time-Series DFs

Model df: Static attributes, properties and memberships per object: *e.g. 'Generator.Model'*

	tech	max_cap	is_intermittent_res	lat_lon	BZ	...
Gen 1	Wind	100	True	(45, 15)	DE_LU	...
Gen 2	RoR	250	False	(35, 10)	AT	...
...	...	...	...	...	...	...
Gen n	PV	300	True	(25, -5)	ES	...

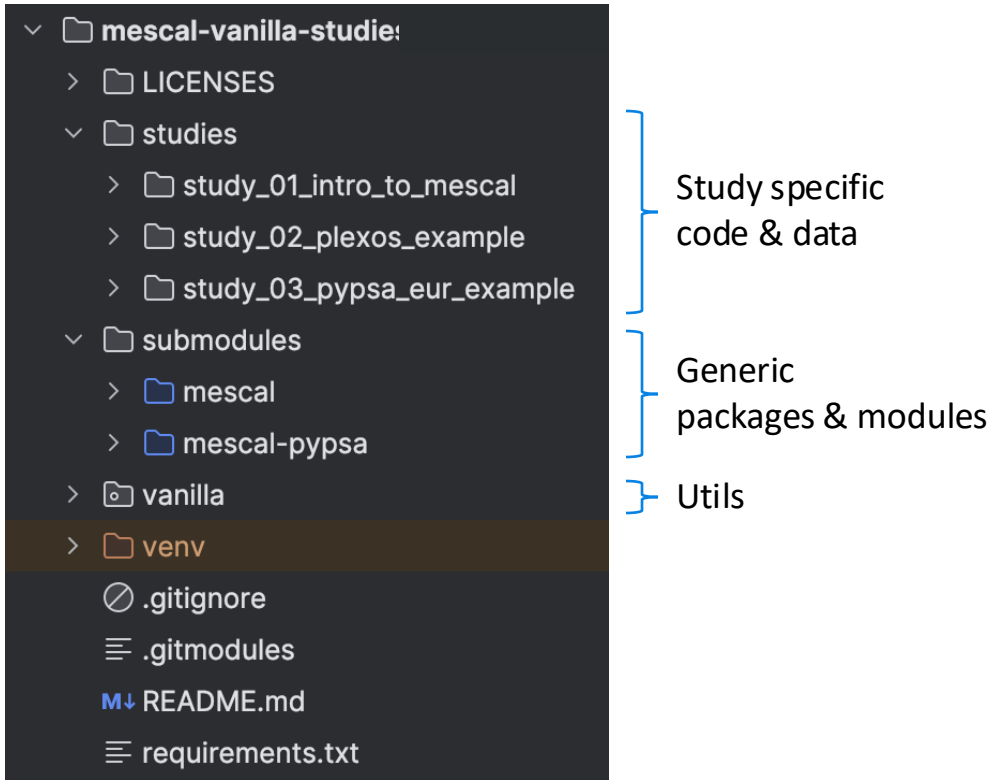
Variable time-series df: time-series per object *e.g. 'ST.Generator.Generation'*

	Gen 1	Gen 2	...	Gen n
00:00	2.2	5.0	...	0.0
01:00	2.5	5.1	...	0.0
02:00	2.4	5.0	...	0.0
...	...	...	...	...

- **A model df describes all objects of a specific object class:** *e.g. generators, bidding zones, lines, ...*
- **Quick builtin filter and groupby_agg operations** thanks to implicit integration between variable time-series and model df: *e.g. filter DE; groupby tech & control_area; sum.*
- **Properties can come from various sources or custom internal logic:** *e.g. Plexos model + Excel Mapping + geojson; even study specific.*
- **Model df should always be exhaustive** and contain all properties.
- **Automatic preservation / mapping of nested properties:** e.g. If line has node_from and node_to, and nodes have a BZ, then the Line.Model will also have BZ_from, and so on...)
- Variable time-series can be simulation inputs, simulation outputs or custom variable calculations. They can come from various sources.
- Variable time-series can be in / switch between DatetimeIndex, session_id+period, MTU enumeration, ..., allowing you to compare different time periods against each other (e.g. Climate years; months, ...).

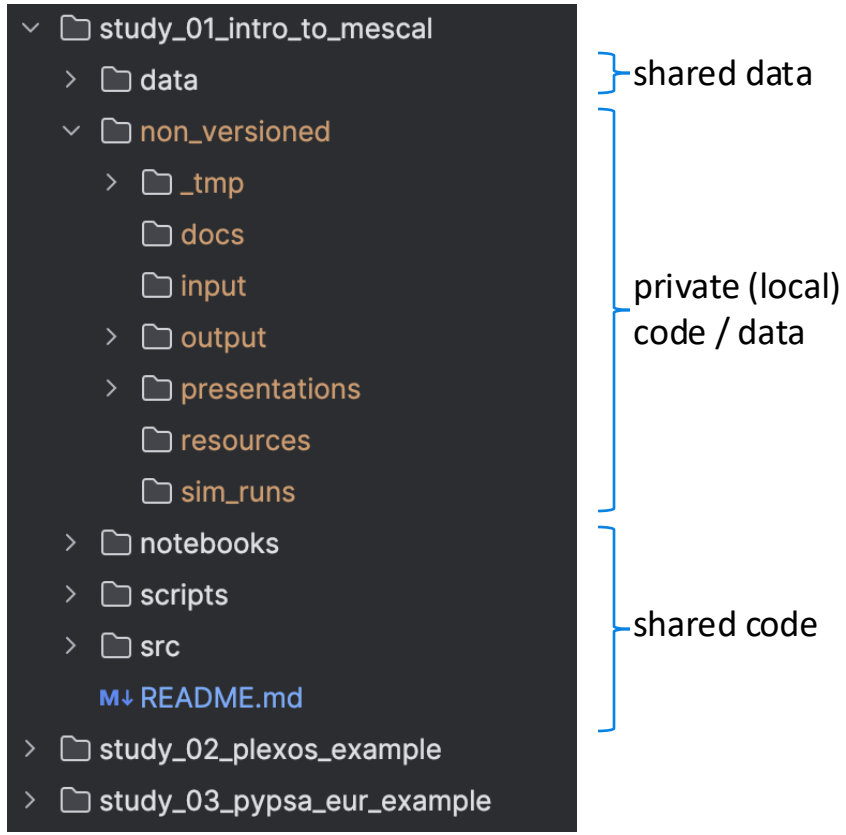
Template Structure for a Shared Studies-Repo

Generic vs study-specific code / data
in a shared repository

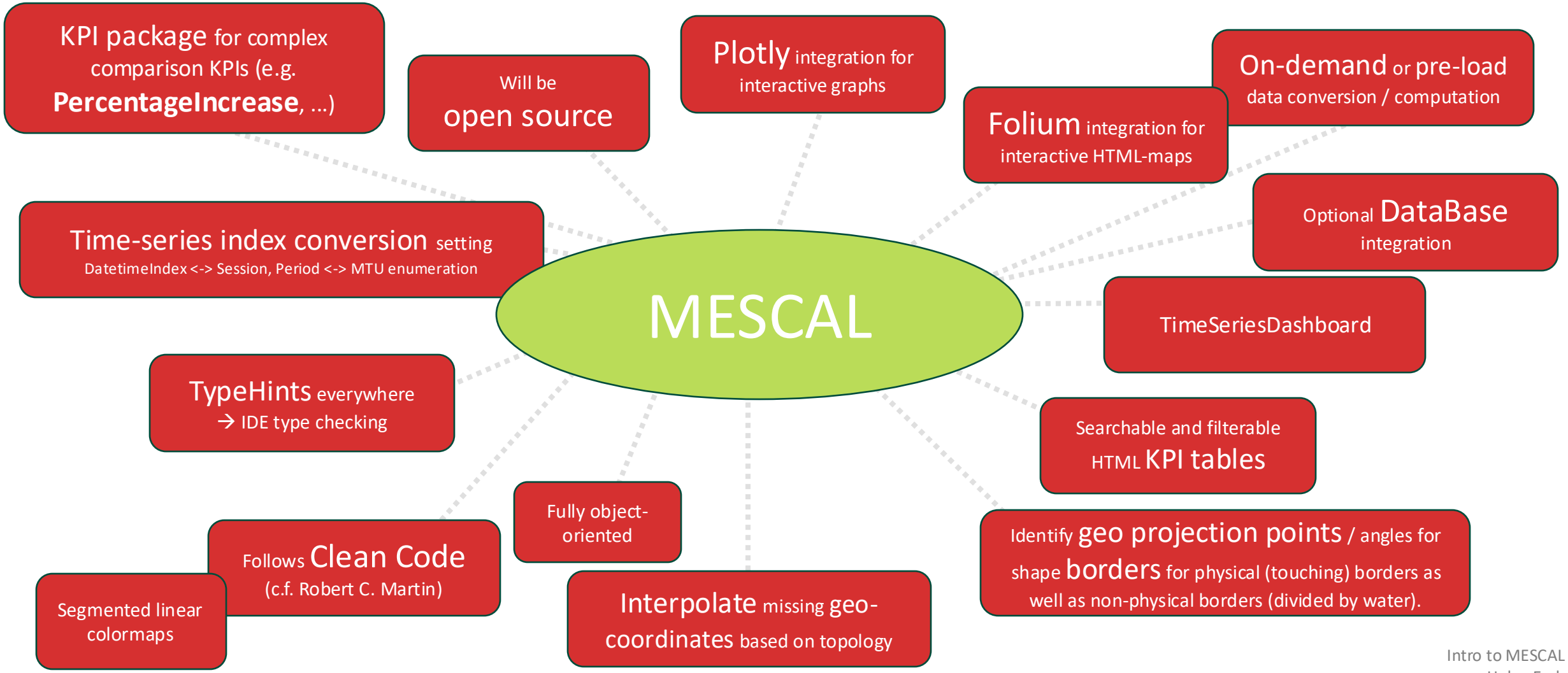


*Example repository structure. Repo does not exist (yet).

Shared vs private code / data
in a shared repository



And there's so much more



APPENDIX

High level overview of the KPI package

1) KPI class

```
class KPI(ABC):
  def __init__(dataset: DatasetType)

  def name() -> str

  def value() -> int | float | bool

  def compute()

  def unit() -> Units.Unit

  def get_pretty_text_value(
        decimals: int = None,
        order_of_magnitude: int = None,
        include_unit: bool = None
  ) -> str:


class FlagAggKPI(KPI):
  def __init__(
        dataset: DatasetType,
        aggregation: Aggregation,
        flag: Flagtype = None,
        subset: Hashable | list[Hashable]] = None,
        model_query: str = None,
  )


class ComparisonKPI(KPI):
  def __init__(
        variation_kpi: KPI,
        reference_kpi: KPI,
        Comparison: ValueComparison,
  )
```

2) Aggregations & Comparisons

```
class Aggregations:
  Total
  Sum
  Max
  Mean
  Min
  AbsSum
  AbsMax
  AbsMean
  AbsMin
  SumGeqZero
  SumLeqZero
  MeanGeqZero
  MeanLeqZero
  MTUsWithNaN
  MTUsNonZero
  MTUsEqZero
  MTUsAboveZero
  MTUsBelowZero
  ...
  CustomAggregation


class ValueComparisons:
  Increase
  Decrease
  PercentageIncrease
  PercentageDecrease
  Share
  Delta
  Diff
```

3) KPICollection class

```
class KPICollection
    """Class to hold a set of KPIs, e.g. for a Dataset.
      Contains methods to add or retrieve KPIs in various formats,
      e.g. as pd.Series."""


class KPIGroup(KPICollection)
    """Class to hold a group of KPIs. A "group" of KPIs has something in
      commong, e.g. they all belong to the same object,
      or they all belong to the same object class, etc."""

class KPIGrouper(KPICollection)
    """Class to group KPIs by specific commonalities,
      e.g. KPIs that all belong to the same object,
      or they all belong KPIs to the same object class, etc."""
```

Key Principles of Clean Code (Robert C. Martin)

1. **Meaningful Names**: Descriptive, intention-revealing names. Avoid abbreviations or misleading terms.
2. **Small Functions**: Functions should do one thing and be small. Limit arguments (0-2).
3. **DRY (Don't Repeat Yourself)**: Eliminate duplication. Abstract common behaviors.
4. **Single Responsibility Principle (SRP)**: Each class/function should have one responsibility, one reason to change.
5. **Open/Closed Principle (OCP)**: Code should be open for extension, closed for modification.
6. **Avoid Side Effects**: Functions should not have unexpected behavior or change the program's state unpredictably.
7. **Exceptions Over Error Codes**: Use exceptions for error handling, not return codes.
8. **Command-Query Separation**: Functions should either perform an action (command) or answer a question (query).
9. **Encapsulation**: Hide internal details, exposing only what's necessary.
10. **Readable Code**: Favor readability. Consistent formatting and self-documenting code minimize comments.
11. **Minimize Comments**: Use comments only to explain complex logic or decisions, not how the code works.
12. **Unit Testing**: Code should be easy to test in isolation. Automate tests to ensure functionality.
13. **Separation of Concerns**: Divide code into modules, each handling a distinct concern.
14. **Avoid Premature Optimization**: Focus on clarity first, optimize only when necessary.
15. **Consistent Naming**: Stick to agreed naming conventions (camelCase, PascalCase, snake_case).