My Document

Author Name

July 23, 2024

1 Module `metagpt`

1.1 Sub-modules

- metagpt.assistants
- metagpt.distorters
- metagpt.evaluators
- metagpt.experiments
- metagpt.predictors
- metagpt.utils

2 Module `metagpt.assistants`

2.1 Sub-modules

- metagpt.assistants.assistant_AllAloneAugust
- metagpt.assistants.assistant_DiscriminatorDave
- metagpt.assistants.assistant_MappingMargarete
- metagpt.assistants.assistant_MelancholicMarvin
- metagpt.assistants.assistant_MissingMyrte
- metagpt.assistants.assistant_TargetTorben
- metagpt.assistants.assistant_TrashTimothy
- metagpt.assistants.assistant_ValidationVeronika
- metagpt.assistants.assistant_template

3 Module `metagpt.assistants.assistant_AllAloneAugust`

3.1 Classes

3.1.1 Class `AllAloneAugust`

```
class AllAloneAugust(
    ome_xsd_path,
    client
)
```

This assistant is the most basic assistant approach and attempts to translate the raw meta-data all alone :(.

Ancestors (in MRO)

- metagpt.assistants.assistant_template.AssistantTemplate

Class variables

Variable `AugustResponseModel` Usage docs: `https://docs.pydantic.dev/2.7/concepts/models/`
A base class for creating Pydantic models.
Attributes —–= **`__class_vars__`** : The names of classvars defined on the model.

`__private_attributes__` Metadata about the private attributes of the model.
`__signature__` The signature for instantiating the model.
`__pydantic_complete__` Whether model building is completed, or if there are still undefined fields.
`__pydantic_core_schema__` The pydantic-core schema used to build the SchemaValidator and SchemaSerializer.
`__pydantic_custom_init__` Whether the model has a custom ___init___ function.
`__pydantic_decorators__` Metadata containing the decorators defined on the model. This replaces Model.___validators___ and Model.___root_validators___ from Pydantic V1.
`__pydantic_generic_metadata__` Metadata for generic models; contains data used for a similar purpose to **args**, **origin**, **parameters** in typing-module generics. May eventually be replaced by these.
`__pydantic_parent_namespace__` Parent namespace of the model, used for automatic rebuilding of models.
`__pydantic_post_init__` The name of the post-init method for the model, if defined.
`__pydantic_root_model__` Whether the model is a RootModel.
`__pydantic_serializer__` The pydantic-core SchemaSerializer used to dump instances of the model.
`__pydantic_validator__` The pydantic-core SchemaValidator used to validate instances of the model.
`__pydantic_extra__` An instance attribute with the values of extra fields from validation when `model_config['extra'] == 'allow'`.
`__pydantic_fields_set__` An instance attribute with the names of fields explicitly set.
`__pydantic_private__` Instance attribute with the values of private attributes set on the model instance.

Methods

Method `run_assistant`

```
def run_assistant(
    self,
    msg,
    thread=None
)
```

Run the assistant :param thread: :param assistant: :param msg: :return:

4 Module `metagpt.assistants.assistant_DiscriminatorDave`

4.1 Classes

4.1.1 Class `DiscriminatorDave`

```
class DiscriminatorDave(
    ome_xsd_path,
    client
)
```

This assistants goal is to decide which part of the raw metadata is already contained in the start point OME XML.

Ancestors (in MRO)

- metagpt.assistants.assistant_template.AssistantTemplate

5 Module `metagpt.assistants.assistant_MappingMargarete`

5.1 Classes

5.1.1 Class `MappingMargarete`

```
class MappingMargarete(
    ome_xsd_path,
    client
)
```

This assistants goal is to take the metadata which can be natively be mapped to the OME XML and map it to the ome xml.

Ancestors (in MRO)

- metagpt.assistants.assistant_template.AssistantTemplate

6 Module `metagpt.assistants.assistant_MelancholicMarvin`

6.1 Classes

6.1.1 Class `MelancholicMarvin`

```
class MelancholicMarvin
```

Class variables

Variable `ResponseModel` This class defines the

Methods

Method `say`

```
def say(
    self,
    msg
)
```

Say something to the assistant. :return:

Method `validate`

```
def validate(
    self,
    ome_xml
) -> Exception
```

Validate the OME XML against the OME XSD :return:

7 Module `metagpt.assistants.assistant_MissingMyrte`

7.1 Classes

7.1.1 Class `MissingMyrte`

```
class MissingMyrte(
    ome_xsd_path,
    client
)
```

This class attempts to take the raw metadata file generated by DiscrminatorDave and which only contain missing metadata i.e. metadata that is not present in the OME XML. It then discriminates the data further into “missing map” and “missing target”. The “missing map” is the metadata that is present in the ome xsd adn therefore should be added to the OME XML according to the schema. The “missing target” is the metadata that is not present in the OME xsd and therefore should be added to ome xml according to a custom namespace/ ontology.

Ancestors (in MRO)

- metagpt.assistants.assistant_template.AssistantTemplate

8 Module `metagpt.assistants.assistant_TargetTorben`

8.1 Classes

8.1.1 Class `TargetTorben`

```
class TargetTorben(
    ome_xsd_path,
    client
)
```

This assistants goal is to take the metadata which can not natively be mapped to the OME XML and map it to an ontology.

Ancestors (in MRO)

- metagpt.assistants.assistant_template.AssistantTemplate

9 Module `metagpt.assistants.assistant_TrashTimothy`

9.1 Classes

9.1.1 Class `TrashTimothy`

```
class TrashTimothy(
    ome_xsd_path,
    client
)
```

This assistants goal is to take the in the end not mapped nor targeted metadata and add it as unstructured metadata.

Ancestors (in MRO)

- metagpt.assistants.assistant_template.AssistantTemplate

10 Module `metagpt.assistants.assistant_ValidationVeronika`

10.1 Classes

10.1.1 Class `ValidationVeronika`

```
class ValidationVeronika(
    ome_xsd_path,
    client
)
```

This assistants goal is to take the final ome xml and fix any errors in it until it is valid.

Ancestors (in MRO)

- metagpt.assistants.assistant_template.AssistantTemplate

Instance variables

Variable `instructions` self.tools.append({“type”: “function”, “function”: { “name”: “validate_ome_xml”, “description”: “Validates an OME XML document against the OME XSD schema.”, “parameters”: { “type”: “object”, “properties”: { “ome_xml”: {“type”: “string”, “description”: “The OME XML document to validate.”}, }, “required”: [“ome_xml”] } } })

11 Module `metagpt.assistants.assistant_template`

11.1 Classes

11.1.1 Class `AssistantTemplate`

```
class AssistantTemplate(
    ome_xsd_path: str = None,
    client: openai.OpenAI = None
)
```

This class is a template for all assistants. It contains the basic structure and methods that all assistants should have.

Descendants

- metagpt.assistants.assistant_AllAloneAugust.AllAloneAugust
- metagpt.assistants.assistant_DiscriminatorDave.DiscriminatorDave
- metagpt.assistants.assistant_MappingMargarete.MappingMargarete
- metagpt.assistants.assistant_MissingMyrte.MissingMyrte
- metagpt.assistants.assistant_TargetTorben.TargetTorben
- metagpt.assistants.assistant_TrashTimothy.TrashTimothy
- metagpt.assistants.assistant_ValidationVeronika.ValidationVeronika

Methods

Method `create_assistant`

```
def create_assistant(
    self,
    assistant_id_path: str = None
)
```

Define the assistant that will help the user with the task :return:

Method `new_assistant`

```
def new_assistant(
    self
)
```

Define the assistant that will help the user with the task :return:

Method `openai_schema`

```
def openai_schema(
    self,
    cls: pydantic.main.BaseModel = None
)
```

Return the schema in the format of OpenAI's schema as jsonschema

Note ——= Its important to add a docstring to describe how to best use this class, it will be included in the description attribute and be part of the prompt.

Returns ——= model_json_schema (dict): A dictionary in the format of OpenAI's schema as jsonschema

Method `save_assistant_id`

```
def save_assistant_id(
    self
)
```

Save the assistant id to a file. :return:

12 Namespace `metagpt.distorters`

12.1 Sub-modules

- metagpt.distorters.distorter_template

13 Module `metagpt.distorters.distorter_template`

13.1 Classes

13.1.1 Class `DistorterTemplate`

```
class DistorterTemplate
```

The distorter takes well formed ome xml as input and returns a "distorted" key value version of it. Distortion can include: - ome xml to key value - shuffling of the order of entried - keys get renamed to similair words

Methods

Method `distort`

```
def distort(
    self,
    ome_xml: str,
    out_path: str,
    should_pred: str = 'maybe'
) -> dict
```

Distort the ome xml

Method `extract_unique_keys`

```
def extract_unique_keys(
    self,
    metadata
)
```

Extract all unique key names from a dictionary, including nested structures, without full paths or indices.

Args: metadata (dict): The dictionary containing metadata.

Returns: list: A list of unique key names.

Method `gen_mapping`

```
def gen_mapping(
    self,
    dict_meta: dict
) -> dict
```

Rename the keys in the ome xml to similar words using a GPT model.

Method `isolate_keys`

```
def isolate_keys(
    self,
    dict_meta: dict
) -> dict
```

Isolate the keys in the ome xml

Method `load_fake_data`

```
def load_fake_data(
    self,
    path: str
) -> dict
```

Load the fake data from a file

Method `modify_metadata_structure`

```
def modify_metadata_structure(
    self,
    metadata,
    operations=None,
    probability=0.3
)
```

Modify the structure of a metadata dictionary systematically and randomly.

Args: metadata (dict): The original metadata dictionary. operations (list): List of operations to perform. If None, all operations are used. probability (float): Probability of applying an operation to each element (0.0 to 1.0).

Returns: dict: A new dictionary with modified structure.

Method `pred`

```
def pred(
    self,
    ome_xml: str,
    out_path: str
) -> dict
```

Predict the distorted data

Method `rename_metadata_keys`

```
def rename_metadata_keys(
    self,
    metadata,
    key_mapping
)
```

Rename keys in a metadata dictionary based on a provided mapping.

Args: metadata (dict): The original metadata dictionary. key_mapping (dict): A dictionary mapping original key names to new key names.

Returns: dict: A new dictionary with renamed keys.

Method `save_fake_data`

```
def save_fake_data(
    self,
    fake_data: dict,
    path: str
)
```

Save the fake data to a file

Method `shuffle_order`

```
def shuffle_order(
    self,
    dict_meta: dict
) -> dict
```

Shuffle the order of the keys in the ome xml

Method `xml_to_key_value`

```
def xml_to_key_value(
    self,
    ome_xml: str
) -> dict
```

Convert the ome xml to key value pairs

14 Namespace `metagpt.evaluators`

14.1 Sub-modules

- metagpt.evaluators.OME_evaluator

15 Module `metagpt.evaluators.OME_evaluator`

15.1 Classes

15.1.1 Class `OMEEvaluator`

```
class OMEEvaluator(
    schema: str = None,
    dataset: metagpt.utils.DataClasses.Dataset = None,
    out_path: str = None
)
```

This class evaluates the performance of a OME XML generation model by calculating the edit distance between the ground truth and the prediction. `https://github.com/timtadh/zhang-shasha`
:param path_to_raw_metadata: path to the raw metadata file

Methods

Method `align_paths`

```
def align_paths(
    self,
    paths_a,
    paths_b
)
```

Align the paths such that the sum of distances between the paths is minimized. paths_a: set of paths paths_b: set of paths :return: list of tuples of aligned paths

Method `align_sequences`

```
def align_sequences(
    self,
    s1,
    s2,
    cost=<function OMEEvaluator.<lambda>>
)
```

Method `align_sequences_score`

```
def align_sequences_score(
    self,
    s1,
    s2,
    cost=<function OMEEvaluator.<lambda>>
)
```

returns only the score for the alignment :param s1: :param s2: :param cost: :return:

Method `attempts_paths_plt`

```
def attempts_paths_plt(
    self,
    df_sample: pandas.core.frame.DataFrame = None
)
```

This plot shows the number of attempts per number of paths(of the original bioformats file). Each Method is its own line.

Method `format_counts_plt`

```
def format_counts_plt(
    self,
    df_sample: pandas.core.frame.DataFrame = None
)
```

This plot shows the formats on the x axis, and how many samples are in each format on the y axis. the different samples need to be identified via the name tag to not count the same file multiple times for each method.

Method `format_method_plt`

```
def format_method_plt(
    self,
    df_sample: pandas.core.frame.DataFrame = None
)
```

This plot compares the performance of the different methods based on the original image format. For each method several bars are plotted, one for each image format.

Method `format_n_paths_plt`

```
def format_n_paths_plt(
    self,
    df_sample: pandas.core.frame.DataFrame = None
)
```

This plots shows the format on the x axis and the number of paths of the y axis as a scatter plot for each data point. But only for the Bioformats method.

Method `get_graph`

```
def get_graph(
    self,
    xml_root: xml.etree.ElementTree.Element,
    root=None
) -> zss.simple_tree.Node
```

Helper function to get the graph representation of an ET XML tree as zss Node.

Method `json_to_pygram`

```
def json_to_pygram(
    self,
    json_data: dict
)
```

Convert a JSON structure to a pygram tree.

Method `method_attempts_plot`

```
def method_attempts_plot(
    self,
    df_sample: pandas.core.frame.DataFrame = None
)
```

This plots the number of attempts against the number of paths in the og image.

Method `method_cost_plot`

```
def method_cost_plot(
    self,
    df_sample: pandas.core.frame.DataFrame = None
)
```

This plot compares the performance of the different methods based on the cost. Wont work because OpenAI does not provide the cost of the methods.

Method `method_edit_distance_no_annot_plt`

```
def method_edit_distance_no_annot_plt(
    self,
    df_sample: pandas.core.frame.DataFrame = None
)
```

This function creates a plot which compares the inter sample standard deviation. The X-axis will be the used method, whereas the Y-axis will be the standard deviation.

Method `method_edit_distance_only_annot_plt`

```
def method_edit_distance_only_annot_plt(
    self,
    df_sample: pandas.core.frame.DataFrame = None
)
```

This function creates a plot which compares the inter sample standard deviation. The X-axis will be the used method, whereas the Y-axis will be the standard deviation.

Method `method_edit_distance_plt`

```
def method_edit_distance_plt(
    self,
    df_sample: pandas.core.frame.DataFrame = None
)
```

This function creates a plot which compares the inter sample standard deviation. The X-axis will be the used method, whereas the Y-axis will be the standard deviation.

Method `n_paths_method_plt`

```
def n_paths_method_plt(
    self,
    df_sample: pandas.core.frame.DataFrame = None
)
```

Plots the number of paths per method as a bar plot.
Parameters: - df_sample: pd.DataFrame, a DataFrame containing the data to plot.
Returns: - fig: The figure object. - ax: The axes object.

Method `path_df`

```
def path_df(
    self
) -> pandas.core.frame.DataFrame
```

This function creates a df with paths as Index and samples as Columns. The entries are True if the path is present in the sample and False if not.

Method `path_difference`

```
def path_difference(
    self,
    xml_a: xml.etree.ElementTree.Element,
    xml_b: xml.etree.ElementTree.Element
)
```

Calculates the length of the difference between the path sets in two xml trees.

Method `paths_annotation_stacked_plt`

```
def paths_annotation_stacked_plt(
    self,
    df_sample: pandas.core.frame.DataFrame = None
)
```

Plots the number of paths and annotations per sample as a stacked bar plot. Uses the seaborn library.

Method `paths_annotation_stacked_relative_plt`

```
def paths_annotation_stacked_relative_plt(
    self,
    df_sample: pandas.core.frame.DataFrame = None
)
```

Plots the relative to og_bioformats file number of paths and annotations per sample as a stacked bar plot. Uses the seaborn library.

Method `pygram_edit_distance`

```
def pygram_edit_distance(
    self,
    xml_a: ome_types._autogenerated.ome_2016_06.ome.OME,
    xml_b: ome_types._autogenerated.ome_2016_06.ome.OME
)
```

Calculate the edit distance between two xml trees on word level. Here an outline of the algorithm:

Method `report`

```
def report(
    self
)
```

Write evaluation report to file.

Method `sample_df`

```
def sample_df(
    self,
    df_paths: pandas.core.frame.DataFrame = None
)
```

This function creates a df with samples as Index and properties as Columns. TODO: Add docstring

Method `word_edit_distance`

```
def word_edit_distance(
    self,
    aligned_paths
) -> int
```

Calculate the word level edit distance between two sets of paths. aligned_paths: list of tuples of aligned paths

Method `zss_edit_distance`

```
def zss_edit_distance(
    self,
    xml_a: xml.etree.ElementTree.Element,
    xml_b: xml.etree.ElementTree.Element
)
```

TODO: add docstring

16 Namespace `metagpt.experiments`

16.1 Sub-modules

- metagpt.experiments.experiment_template

17 Module `metagpt.experiments.experiment_template`

17.1 Classes

17.1.1 Class `ExperimentTemplate`

```
class ExperimentTemplate
```

The experiment template class defines an experiment object that can be used to run experiments. The experiment defines the dataset, the predictors, the evaluator and more.

Methods

Method `run`

```
def run(
    self
)
```

Run the experiment

18 Module `metagpt.predictors`

18.1 Sub-modules

- metagpt.predictors.predictor_curation_swarm
- metagpt.predictors.predictor_discriminator
- metagpt.predictors.predictor_distorter
- metagpt.predictors.predictor_marvin
- metagpt.predictors.predictor_missing
- metagpt.predictors.predictor_network
- metagpt.predictors.predictor_network_annotator
- metagpt.predictors.predictor_seperator
- metagpt.predictors.predictor_simple
- metagpt.predictors.predictor_simple_annotator
- metagpt.predictors.predictor_state
- metagpt.predictors.predictor_state_tree
- metagpt.predictors.predictor_template
- metagpt.predictors.predictor_tree

19 Module `metagpt.predictors.predictor_curation_swarm`

19.1 Classes

19.1.1 Class `CurationSwarm`

```
class CurationSwarm(
    path_to_raw_metadata=None,
    path_to_ome_starting_point=None,
    ome_xsd_path=None,
    out_path=None
)
```

This class implements a swarm of AI assistants that work together to curate the metadata.

Ancestors (in MRO)

- metagpt.predictors.predictor_template.PredictorTemplate

Methods

Method `export_convo`

```
def export_convo(
    self
)
```

Method `hierachical_planning`

```
def hierachical_planning(
    self
)
```

Run the message

Method `predict`

```
def predict(
    self
)
```

Predict the OME XML :return:

Method `run_assistant`

```
def run_assistant(
    self,
    assistant,
    msg,
    thread=None
)
```

Run the assistant :param thread: :param assistant: :param msg: :return:

20 Module `metagpt.predictors.predictor_discriminator`

20.1 Classes

20.1.1 Class `DiscriminatorPredictor`

```
class DiscriminatorPredictor(
    path_to_raw_metadata=None,
    path_to_ome_starting_point=None,
    ome_xsd_path=None,
    out_path=None
)
```

This class implements only the discriminator assistant to predict the overhead metadata.

Ancestors (in MRO)

- metagpt.predictors.predictor_template.PredictorTemplate

Methods

Method `run_message`

```
def run_message(
    self
)
```

Predict the OME XML from the raw metadata

21 Module `metagpt.predictors.predictor_distorter`

21.1 Classes

21.1.1 Class `PredictorDistorter`

```
class PredictorDistorter(
    raw_meta: str
)
```

TODO: Add docstring

Ancestors (in MRO)

- metagpt.predictors.predictor_template.PredictorTemplate

Class variables

Variable `out_new_meta` Helper class to define the output of the assistant.

Methods

Method `init_assistant`

```
def init_assistant(
    self
)
```

Method `init_run`

```
def init_run(
    self
)
```

Method `predict`

```
def predict(
    self
) -> dict
```

TODO: Add docstring

22 Module `metagpt.predictors.predictor_marvin`

22.1 Classes

22.1.1 Class `PredictorMarvin`

```
class PredictorMarvin(
    raw_meta: str
)
```

TODO: Add docstring

Ancestors (in MRO)

- metagpt.predictors.predictor_template.PredictorTemplate

Methods

Method `init_state`

```
def init_state(
    self
)
```

Initialize the state

Method `predict`

```
def predict(
    self
)
```

Predict the image annotations using the Marvin model. :param image_path: The path to the image. :return: The predicted image annotations.

23 Module `metagpt.predictors.predictor_missing`

23.1 Classes

23.1.1 Class `MissingPredictor`

```
class MissingPredictor(
    path_to_raw_metadata=None,
    path_to_ome_starting_point=None,
    ome_xsd_path=None,
    out_path=None
)
```

Ancestors (in MRO)

- metagpt.predictors.predictor_template.PredictorTemplate

Methods

Method `run_message`

```
def run_message(
    self
)
```

Predict the OME XML from the raw metadata

24 Module `metagpt.predictors.predictor_network`

24.1 Classes

24.1.1 Class `PredictorNetwork`

```
class PredictorNetwork(
    raw_meta: str
)
```

This predictor approach uses two assistants, one for splitting the raw metadata into already contained and new metadata, and one for predicting the structured annotations from the new metadata.

Ancestors (in MRO)

- metagpt.predictors.predictor_template.PredictorTemplate

Methods

Method `predict`

```
def predict(
    self
) -> ome_types._autogenerated.ome_2016_06.structured_annotations.StructuredAnnotations
```

TODO: Add docstring

25 Module `metagpt.predictors.predictor_network_annotator`

25.1 Classes

25.1.1 Class `PredictorNetworkAnnotation`

```
class PredictorNetworkAnnotation(
    raw_meta: str
)
```

This predictor approach uses two assistants, one for splitting the raw metadata into already contained and new metadata, and one for predicting the structured annotations from the new metadata.

Ancestors (in MRO)

- metagpt.predictors.predictor_template.PredictorTemplate

Methods

Method `predict`

```
def predict(
    self
) -> tuple[str, float, float]
```

TODO: Add docstring

26 Module `metagpt.predictors.predictor_seperator`

26.1 Classes

26.1.1 Class `PredictorSeperator`

```
class PredictorSeperator(
    raw_meta: str
)
```

This predictor approach uses two assistants, one for splitting the raw metadata into already contained and new metadata, and one for predicting the structured annotations from the new metadata.

Ancestors (in MRO)

- metagpt.predictors.predictor_template.PredictorTemplate

Class variables

Variable `SepOutputTool` This tool automatically formats and structures the metadata in the appropriate way.

Methods

Method `init_assistant`

```
def init_assistant(
    self
)
```

Method `init_run`

```
def init_run(
    self
)
```

Method `init_vector_store`

```
def init_vector_store(
    self
)
```

Method `predict`

```
def predict(
    self
) -> tuple
```

TODO: Add docstring

27 Module `metagpt.predictors.predictor_simple`

27.1 Classes

27.1.1 Class `PredictorSimple`

```
class PredictorSimple(
    raw_meta: str
)
```

TODO: Add docstring

Ancestors (in MRO)

- metagpt.predictors.predictor_template.PredictorTemplate

Class variables

Variable `OMEXMLResponse` The response to the OME XML annotation

Methods

Method `init_assistant`

```
def init_assistant(
    self
)
```

Method `init_run`

```
def init_run(
    self
)
```

Method `init_vector_store`

```
def init_vector_store(
    self
)
```

Method `predict`

```
def predict(
    self
) -> dict
```

TODO: Add docstring

28 Module `metagpt.predictors.predictor_simple_annotator`

28.1 Classes

28.1.1 Class `PredictorSimpleAnnotation`

```
class PredictorSimpleAnnotation(
    raw_meta: str
)
```

TODO: Add docstring

Ancestors (in MRO)

- metagpt.predictors.predictor__template.PredictorTemplate

Class variables

Variable `XMLAnnotationFunction` The function call to hand in the structured annotations to the OME XML.

Methods

Method `init_assistant`

```
def init_assistant(
    self
)
```

Method `init_run`

```
def init_run(
    self
)
```

Method `predict`

```
def predict(
    self
) -> dict
```

TODO: Add docstring

29 Module `metagpt.predictors.predictor_state`

29.1 Classes

29.1.1 Class `AddReplaceTestOperation`

```
class AddReplaceTestOperation(
    **data: Any
)
```

Usage docs: `https://docs.pydantic.dev/2.7/concepts/models/`

A base class for creating Pydantic models.

Attributes ——= **`__class_vars__`** : The names of classvars defined on the model.

`__private_attributes__` Metadata about the private attributes of the model.
`__signature__` The signature for instantiating the model.
`__pydantic_complete__` Whether model building is completed, or if there are still undefined fields.
`__pydantic_core_schema__` The pydantic-core schema used to build the SchemaValidator and SchemaSerializer.
`__pydantic_custom_init__` Whether the model has a custom ___init___ function.
`__pydantic_decorators__` Metadata containing the decorators defined on the model. This replaces Model.___validators___ and Model.___root_validators___ from Pydantic V1.
`__pydantic_generic_metadata__` Metadata for generic models; contains data used for a similar purpose to **args**, **origin**, **parameters** in typing-module generics. May eventually be replaced by these.
`__pydantic_parent_namespace__` Parent namespace of the model, used for automatic rebuilding of models.
`__pydantic_post_init__` The name of the post-init method for the model, if defined.
`__pydantic_root_model__` Whether the model is a RootModel.
`__pydantic_serializer__` The pydantic-core SchemaSerializer used to dump instances of the model.
`__pydantic_validator__` The pydantic-core SchemaValidator used to validate instances of the model.
`__pydantic_extra__` An instance attribute with the values of extra fields from validation when `model_config['extra'] == 'allow'`.
`__pydantic_fields_set__` An instance attribute with the names of fields explicitly set.
`__pydantic_private__` Instance attribute with the values of private attributes set on the model instance.

Create a new model by parsing and validating input data from keyword arguments.

Raises [`ValidationError`][pydantic_core.ValidationError] if the input data cannot be validated to form a valid model.

self is explicitly positional-only to allow self as a field name.

Ancestors (in MRO)

- pydantic.main.BaseModel

Class variables

Variable `model_computed_fields`

Variable `model_config`

Variable `model_fields`

Variable `op` Type: `Literal['add', 'replace', 'test']`

Variable `path` Type: `str`

Variable `value` Type: `Any`

29.1.2 Class `JsonPatch`

```
class JsonPatch(
    **data: Any
)
```

Usage docs: `https://docs.pydantic.dev/2.7/concepts/models/`

A base class for creating Pydantic models.

Attributes ——= **`__class_vars__`** : The names of classvars defined on the model.

`__private_attributes__` Metadata about the private attributes of the model.
`__signature__` The signature for instantiating the model.
`__pydantic_complete__` Whether model building is completed, or if there are still undefined fields.
`__pydantic_core_schema__` The pydantic-core schema used to build the SchemaValidator and SchemaSerializer.
`__pydantic_custom_init__` Whether the model has a custom ___init___ function.
`__pydantic_decorators__` Metadata containing the decorators defined on the model. This replaces Model.___validators___ and Model.___root_validators___ from Pydantic V1.
`__pydantic_generic_metadata__` Metadata for generic models; contains data used for a similar purpose to **args**, **origin**, **parameters** in typing-module generics. May eventually be replaced by these.
`__pydantic_parent_namespace__` Parent namespace of the model, used for automatic rebuilding of models.
`__pydantic_post_init__` The name of the post-init method for the model, if defined.
`__pydantic_root_model__` Whether the model is a RootModel.
`__pydantic_serializer__` The pydantic-core SchemaSerializer used to dump instances of the model.
`__pydantic_validator__` The pydantic-core SchemaValidator used to validate instances of the model.
`__pydantic_extra__` An instance attribute with the values of extra fields from validation when `model_config['extra'] == 'allow'`.
`__pydantic_fields_set__` An instance attribute with the names of fields explicitly set.
`__pydantic_private__` Instance attribute with the values of private attributes set on the model instance.

Create a new model by parsing and validating input data from keyword arguments.

Raises [`ValidationError`][pydantic_core.ValidationError] if the input data cannot be validated to form a valid model.

self is explicitly positional-only to allow self as a field name.

Ancestors (in MRO)

- pydantic.main.BaseModel

Class variables

Variable `Config`

Variable `model_computed_fields`

Variable `model_config`

Variable `model_fields`

Variable `root` Type: `List[Union[metagpt.predictors.predictor_state.AddReplaceTestOperation, metagp`

29.1.3 Class `MoveCopyOperation`

```
class MoveCopyOperation(
    **data: Any
)
```

Usage docs: `https://docs.pydantic.dev/2.7/concepts/models/`

A base class for creating Pydantic models.

Attributes ——= **`__class_vars__`** : The names of classvars defined on the model.

`__private_attributes__` Metadata about the private attributes of the model.
`__signature__` The signature for instantiating the model.
`__pydantic_complete__` Whether model building is completed, or if there are still undefined fields.
`__pydantic_core_schema__` The pydantic-core schema used to build the SchemaValidator and SchemaSerializer.
`__pydantic_custom_init__` Whether the model has a custom ___init___ function.
`__pydantic_decorators__` Metadata containing the decorators defined on the model. This replaces Model.___validators___ and Model.___root_validators___ from Pydantic V1.
`__pydantic_generic_metadata__` Metadata for generic models; contains data used for a similar purpose to **args**, **origin**, **parameters** in typing-module generics. May eventually be replaced by these.
`__pydantic_parent_namespace__` Parent namespace of the model, used for automatic rebuilding of models.
`__pydantic_post_init__` The name of the post-init method for the model, if defined.
`__pydantic_root_model__` Whether the model is a RootModel.
`__pydantic_serializer__` The pydantic-core SchemaSerializer used to dump instances of the model.
`__pydantic_validator__` The pydantic-core SchemaValidator used to validate instances of the model.
`__pydantic_extra__` An instance attribute with the values of extra fields from validation when `model_config['extra'] == 'allow'`.
`__pydantic_fields_set__` An instance attribute with the names of fields explicitly set.
`__pydantic_private__` Instance attribute with the values of private attributes set on the model instance.

Create a new model by parsing and validating input data from keyword arguments.

Raises [`ValidationError`][pydantic_core.ValidationError] if the input data cannot be validated to form a valid model.

self is explicitly positional-only to allow self as a field name.

Ancestors (in MRO)

- pydantic.main.BaseModel

Class variables

Variable `from_` Type: `str`

Variable `model_computed_fields`

Variable `model_config`

Variable `model_fields`

Variable `op` Type: `Literal['move', 'copy']`

Variable `path` Type: `str`

29.1.4 Class `PredictorState`

```
class PredictorState(
    raw_meta: str,
    state: pydantic.main.BaseModel = None
)
```

TODO: Add docstring

Ancestors (in MRO)

- metagpt.predictors.predictor_template.PredictorTemplate

Methods

Method `init_assistant`

```
def init_assistant(
    self
)
```

Method `init_run`

```
def init_run(
    self
)
```

Method `init_vector_store`

```
def init_vector_store(
    self
)
```

Method `predict`

```
def predict(
    self,
    indent: Optional[int] = 0
) -> str
```

TODO: Add docstring

29.1.5 Class `RemoveOperation`

```
class RemoveOperation(
    **data: Any
)
```

Usage docs: `https://docs.pydantic.dev/2.7/concepts/models/`
A base class for creating Pydantic models.
Attributes ——= **`__class_vars__`** : The names of classvars defined on the model.

`__private_attributes__` Metadata about the private attributes of the model.
`__signature__` The signature for instantiating the model.
`__pydantic_complete__` Whether model building is completed, or if there are still undefined fields.
`__pydantic_core_schema__` The pydantic-core schema used to build the SchemaValidator and SchemaSerializer.
`__pydantic_custom_init__` Whether the model has a custom ___init___ function.
`__pydantic_decorators__` Metadata containing the decorators defined on the model. This replaces Model.___validators___ and Model.___root_validators___ from Pydantic V1.

`__pydantic_generic_metadata__` Metadata for generic models; contains data used for a similar purpose to **args**, **origin**, **parameters** in typing-module generics. May eventually be replaced by these.
`__pydantic_parent_namespace__` Parent namespace of the model, used for automatic rebuilding of models.
`__pydantic_post_init__` The name of the post-init method for the model, if defined.
`__pydantic_root_model__` Whether the model is a RootModel.
`__pydantic_serializer__` The pydantic-core SchemaSerializer used to dump instances of the model.
`__pydantic_validator__` The pydantic-core SchemaValidator used to validate instances of the model.
`__pydantic_extra__` An instance attribute with the values of extra fields from validation when `model_config['extra'] == 'allow'`.
`__pydantic_fields_set__` An instance attribute with the names of fields explicitly set.
`__pydantic_private__` Instance attribute with the values of private attributes set on the model instance.

Create a new model by parsing and validating input data from keyword arguments.
Raises [`ValidationError`][pydantic_core.ValidationError] if the input data cannot be validated to form a valid model.
self is explicitly positional-only to allow self as a field name.

Ancestors (in MRO)

- pydantic.main.BaseModel

Class variables

Variable `model_computed_fields`

Variable `model_config`

Variable `model_fields`

Variable `op` Type: `Literal['remove']`

Variable `path` Type: `str`

29.1.6 Class `update_json_state`

```
class update_json_state(
    **data: Any
)
```

Update the state of the predictor from a list of json patches.
Create a new model by parsing and validating input data from keyword arguments.
Raises [`ValidationError`][pydantic_core.ValidationError] if the input data cannot be validated to form a valid model.
self is explicitly positional-only to allow self as a field name.

Ancestors (in MRO)

- pydantic.main.BaseModel

Class variables

Variable `json_patches` Type: `Optional[list[metagpt.predictors.predictor_state.JsonPatch]]`

Variable `model_computed_fields`

Variable `model_config`

Variable `model_fields`

30 Module `metagpt.predictors.predictor_state_tree`

30.1 Functions

30.1.1 Function `create_instance`

```
def create_instance(
    instance,
    obj_dict: dict
)
```

30.2 Classes

30.2.1 Class `PredictorStateTree`

```
class PredictorStateTree(
    raw_meta: str,
    model: pydantic.main.BaseModel = None
)
```

A template for creating a new predictor. A predictor utilizes one or several assistants to predict the OME XML from the raw metadata.

Ancestors (in MRO)

- metagpt.predictors.predictor_template.PredictorTemplate

Methods

Method `build_tree`

```
def build_tree(
    self,
    root_model: Type[pydantic.main.BaseModel]
) -> metagpt.predictors.predictor_state_tree.TreeNode
```

Method `collect_dependencies`

```
def collect_dependencies(
    self,
    model: Type[pydantic.main.BaseModel],
    known_models: Dict[str, Type[pydantic.main.BaseModel]],
    collected: Dict[str, Type[pydantic.main.BaseModel]]
)
```

Method `create_dependency_tree`

```
def create_dependency_tree(
    self,
    model: Type[pydantic.main.BaseModel],
    known_models: Dict[str, Type[pydantic.main.BaseModel]],
    visited: Set[str]
) -> metagpt.predictors.predictor_state_tree.TreeNode
```

Method `print_tree`

```
def print_tree(
    self,
    node: metagpt.predictors.predictor_state_tree.TreeNode = None,
    indent: str = ''
)
```

30.2.2 Class `TreeNode`

```
class TreeNode(
    model: Type[pydantic.main.BaseModel]
)
```

Methods

Method `add_child`

```
def add_child(
    self,
    child: TreeNode
)
```

Method `instantiate_model`

```
def instantiate_model(
    self,
    child_objects
) -> pydantic.main.BaseModel
```

Method `predict_meta`

```
def predict_meta(
    self,
    raw_meta: str,
    indent: int = 0
) -> pydantic.main.BaseModel
```

Method `required_fields`

```
def required_fields(
    self,
    model: type[pydantic.main.BaseModel],
    recursive: bool = False
) -> collections.abc.Iterator[str]
```

`https://stackoverflow.com/questions/75146792/get-all-required-fields-of-a-nested-pydantic-model`

31 Module `metagpt.predictors.predictor_template`

31.1 Classes

31.1.1 Class `PredictorTemplate`

```
class PredictorTemplate
```

A template for creating a new predictor. A predictor utilizes one or several assistants to predict the OME XML from the raw metadata.

Descendants

- metagpt.predictors.predictor_curation_swarm.CurationSwarm
- metagpt.predictors.predictor_discriminator.DiscriminatorPredictor
- metagpt.predictors.predictor_distorter.PredictorDistorter
- metagpt.predictors.predictor_marvin.PredictorMarvin
- metagpt.predictors.predictor_missing.MissingPredictor
- metagpt.predictors.predictor_network.PredictorNetwork
- metagpt.predictors.predictor_network_annotator.PredictorNetworkAnnotation
- metagpt.predictors.predictor_seperator.PredictorSeperator
- metagpt.predictors.predictor_simple.PredictorSimple
- metagpt.predictors.predictor_simple_annotator.PredictorSimpleAnnotation
- metagpt.predictors.predictor_state.PredictorState
- metagpt.predictors.predictor_state_tree.PredictorStateTree

Methods

Method `add_attempts`

```
def add_attempts(
    self,
    i: float = 1
)
```

Add an attempt to the attempt counter. Normalized by the number of assistants.

Method `clean_assistants`

```
def clean_assistants(
    self
)
```

Clean up the assistants

Method `export_ome_xml`

```
def export_ome_xml(
    self
)
```

Export the OME XML to a file

Method `generate_message`

```
def generate_message(
    self,
    msg=None
)
```

Generate the prompt from the raw metadata

Method `get_cost`

```
def get_cost(
    self,
    run
)
```

Get the cost of the prediction

Method `get_response`

```
def get_response(
    self
)
```

Predict the OME XML from the raw metadata

Method `init_thread`

```
def init_thread(
    self
)
```

Initialize the thread

Method `predict`

```
def predict(
    self
) -> dict
```

Predict the OME XML from the raw metadata

Method `read_ome_as_string`

```
def read_ome_as_string(
    self,
    path
)
```

Read the OME XML as a string

Method `read_ome_as_xml`

```
def read_ome_as_xml(
    self,
    path
)
```

This method reads the ome xml file and returns the root element.

Method `read_raw_metadata`

```
def read_raw_metadata(
    self
)
```

Read the raw metadata from the file

Method `subdivide_raw_metadata`

```
def subdivide_raw_metadata(
    self
)
```

Subdivide the raw metadata into appropriate chunks

Method `validate`

```
def validate(
    self,
    ome_xml
) -> Exception
```

Validate the OME XML against the OME XSD :return:

32 Module `metagpt.predictors.predictor_tree`

32.1 Functions

32.1.1 Function `build_tree`

```
def build_tree(
    root_model: Type[pydantic.main.BaseModel]
) -> metagpt.predictors.predictor_tree.TreeNode
```

32.1.2 Function `collect_dependencies`

```
def collect_dependencies(
    model: Type[pydantic.main.BaseModel],
    known_models: Dict[str, Type[pydantic.main.BaseModel]],
    collected: Dict[str, Type[pydantic.main.BaseModel]]
)
```

32.1.3 Function `create_dependency_tree`

```
def create_dependency_tree(
    model: Type[pydantic.main.BaseModel],
    known_models: Dict[str, Type[pydantic.main.BaseModel]],
    visited: Set[str]
) -> metagpt.predictors.predictor_tree.TreeNode
```

32.1.4 Function `create_instance`

```
def create_instance(
    instance,
    obj_dict: dict
)
```

32.1.5 Function `print_tree`

```
def print_tree(
    node: metagpt.predictors.predictor_tree.TreeNode,
    indent: str = ''
)
```

32.2 Classes

32.2.1 Class `TreeNode`

```
class TreeNode(
    model: Type[pydantic.main.BaseModel]
)
```

Class variables

Variable `thread`

Variable `thread_id`

Methods

Method `add_child`

```
def add_child(
    self,
    child: TreeNode
)
```

Method `instantiate_model`

```
def instantiate_model(
    self,
    child_objects
) -> pydantic.main.BaseModel
```

Method `predict_meta`

```
def predict_meta(
    self,
    raw_meta
) -> pydantic.main.BaseModel
```

Method `required_fields`

```
def required_fields(
    self,
    model: type[pydantic.main.BaseModel],
    recursive: bool = False
) -> collections.abc.Iterator[str]
```

`https://stackoverflow.com/questions/75146792/get-all-required-fields-of-a-nested-pydantic-model`

33 Module `metagpt.utils`

33.1 Sub-modules

- metagpt.utils.BioformatsReader
- metagpt.utils.DataClasses
- metagpt.utils.utils

34 Module `metagpt.utils.BioformatsReader`

This file implements functions to read the proprietary images and returns their metadata in OME-XML format and the raw metadata key-value pairs.

34.1 Functions

34.1.1 Function `get_omexml_metadata`

```
def get_omexml_metadata(
    path=None,
    url=None
)
```

Read the OME metadata from a file using Bio-formats
:param path: path to the file
:param groupfiles: utilize the groupfiles option to take the directory structure into account.
:returns: the metdata as XML.

34.1.2 Function `get_raw_metadata`

```
def get_raw_metadata(
    path: str = None
) -> dict[str, str]
```

Read the raw metadata from a file using Bio-formats
:param path: path to the file :return: the metadata as a dictionary

34.1.3 Function `raw_to_tree`

```
def raw_to_tree(
    raw_metadata: dict[str, str]
)
```

Convert the raw metadata to a tree structure, by seperating the key on the "|" character.

35 Module `metagpt.utils.DataClasses`

Data classes for the metagpt package.

35.1 Classes

35.1.1 Class `Dataset`

```
class Dataset(
    name: str = None,
    samples: dict[slice(<class 'str'>, <class 'metagpt.utils.DataClasses.Sample'>, None)] = Fiel
    cost: Optional[float] = None,
    time: Optional[float] = None
)
```

Dataset(name: str = None, samples: dict[slice(<class 'str'>, <class 'metagpt.utils.DataClasses.Sample'>, None)] = FieldInfo(annotation=NoneType, required=False, default_factory=dict), cost: Optional[float] = None, time: Optional[float] = None)

Class variables

Variable `cost` Type: `Optional[float]`

Variable `name` Type: `str`

Variable `samples` Type: `dict[slice(<class 'str'>, <class 'metagpt.utils.DataClasses.Sample'>, None`

Variable `time` Type: `Optional[float]`

Methods

Method `add_sample`

```
def add_sample(
    self,
    sample: metagpt.utils.DataClasses.Sample
)
```

35.1.2 Class `Sample`

```
class Sample(
    name: str = None,
    metadata_str: str = None,
    method: str = None,
    metadata_xml: ome_types._autogenerated.ome_2016_06.ome.OME = FieldInfo(annotation=NoneType,
    cost: Optional[float] = None,
    paths: Optional[list[str]] = None,
    time: Optional[float] = None,
    format: Optional[str] = None,
    attempts: Optional[float] = None,
    iter: Optional[int] = None,
    gpt_model: Optional[str] = None
)
```

Sample(name: str = None, metadata_str: str = None, method: str = None, metadata_xml: ome_types._autogenerated.ome_2016_06.ome.OME = FieldInfo(annotation=NoneType, required=False, default_factory=OME, description='The metadata as an OME object'), cost: Optional[float] = None, paths: Optional[list[str]] = None, time: Optional[float] = None, format: Optional[str] = None, attempts: Optional[float] = None, iter: Optional[int] = None, gpt_model: Optional[str] = None)

Class variables

Variable `attempts` Type: `Optional[float]`

Variable `cost` Type: `Optional[float]`

Variable `format` Type: `Optional[str]`

Variable `gpt_model` Type: `Optional[str]`

Variable `iter` Type: `Optional[int]`

Variable `metadata_str` Type: `str`

Variable `metadata_xml` Type: `ome_types._autogenerated.ome_2016_06.ome.OME`

Variable `method` Type: `str`

Variable `name` Type: `str`

Variable `paths` Type: `Optional[list[str]]`

Variable `time` Type: `Optional[float]`

36 Module `metagpt.utils.utils`

36.1 Functions

36.1.1 Function `browse_schema`

```
def browse_schema(
    cls: pydantic.main.BaseModel,
    additional_ignored_keywords: List[str] = [],
    max_depth: int = inf
) -> Dict[str, Any]
```

Browse a schema as jsonschema, with depth control.
Args ——= **cls** : BaseModel : The Pydantic model to convert to a schema.

additional_ignored_keywords : List[str], optional Additional keywords to ignore in the schema. Defaults to [].
max_depth : int, optional Maximum depth of nesting to include in the schema. Defaults to infinity.

Returns ——= dict : A dictionary in the format of OpenAI's schema as jsonschema

36.1.2 Function `collect_dependencies`

```
def collect_dependencies(
    model: Type[pydantic.main.BaseModel],
    known_models: Dict[str, Type[pydantic.main.BaseModel]],
    collected: Dict[str, Type[pydantic.main.BaseModel]]
)
```

Function to identify dependencies and collect all models

36.1.3 Function `custom_apply`

```
def custom_apply(
    patch: jsonpatch.JsonPatch,
    data: Dict[str, Any]
) -> Dict[str, Any]
```

Apply the JSON Patch, automatically creating missing nodes.

36.1.4 Function `dict_to_xml_annotation`

```
def dict_to_xml_annotation(
    value: dict
) -> ome_types._autogenerated.ome_2016_06.xml_annotation.XMLAnnotation
```

Convert a dictionary to an XMLAnnotation object, handling nested dictionaries.
value: dict - The dictionary to be converted to an XMLAnnotation object. It requires the key 'annotations' which is a dictionary of key-value pairs.

36.1.5 Function `ensure_path_exists`

```
def ensure_path_exists(
    data: Dict[str, Any],
    path: str
) -> None
```

Ensure that the path exists in the data structure, creating empty lists or dicts as needed.

36.1.6 Function `flatten`

```
def flatten(
    container
)
```

36.1.7 Function `from_dict`

```
def from_dict(
    ome_dict,
    state: pydantic.main.BaseModel = None
) -> ome_types._autogenerated.ome_2016_06.ome.OME
```

Convert a dictionary to an OME object.

36.1.8 Function `generate_paths`

```
def generate_paths(
    json_data: Union[dict, list],
    current_path: str = '',
    paths: list = None
) -> list
```

Generate all possible paths from a nested JSON structure.

This function traverses a nested JSON structure (which may contain dictionaries and lists) and generates a list of all possible paths within it. For dictionaries, it uses the keys to build the path. For lists, it uses indices, except when a list item is a dictionary with an 'id' key, in which case it uses the id value in the path.

Args ——= **`json_data`** : dict or list : The nested JSON structure to traverse.

`current_path` : str, optional The current path being built. Used in recursive calls. Defaults to an empty string.

`paths` : list, optional The list to store all generated paths. Used in recursive calls. Defaults to None, which initializes an empty list.

Returns ——= list : A list of strings, where each string represents a path in the format "path/to/element = value".

Examples ——=

```
>>> json_data = {
...     "test": 5,
...     "images": [
...         {"image": {"id": "image:0"}},
...         {"image": {"id": "image:1"}}
...     ],
...     "nested": {
...         "key": "value",
...         "list": [1, 2, 3]
...     }
... }
>>> result = generate_paths(json_data)
>>> for path in result:
...     print(path)
test = 5
images/0/image/id = image:0
images/1/image/id = image:1
nested/key = value
nested/list/0 = 1
nested/list/1 = 2
nested/list/2 = 3
```

36.1.9 Function `get_dependencies`

```
def get_dependencies(
    model: Type[pydantic.main.BaseModel],
    known_models: Dict[str, Type[pydantic.main.BaseModel]]
) -> Set[str]
```

Function to identify dependencies for sorting

36.1.10 Function `get_json`

```
def get_json(
    xml_root: xml.etree.ElementTree.Element,
    paths={}
) -> dict
```

Helper function to get all paths in an XML tree. :return: set of paths

36.1.11 Function `load_output`

```
def load_output(
    path: str
) -> tuple[typing.Optional[str], typing.Optional[float]]
```

Load output from a file.
Args —–= **`path`** : str : The file path to load from.
Returns —–= Optional[str] : The loaded output, or None if an error occurred.

36.1.12 Function `make_prediction`

```
def make_prediction(
    predictor: metagpt.predictors.predictor_template.PredictorTemplate,
    in_data,
    dataset,
    name,
    should_predict='maybe',
    start_point=None,
    data_format=None,
    iter: int = None,
    model: str = None
)
```

TODO: add docstring

36.1.13 Function `merge_xml_annotation`

```
def merge_xml_annotation(
    annot: Dict[str, Any],
    ome: str = None
) -> Optional[str]
```

Merge the annotation section with the OME XML.
Args —–= **`ome`** : Optional[str] : The OME XML string.

`annot` : Optional[Dict[str, Any]] The annotation dictionary.

Returns —–= Optional[str] : The merged XML string, or None if inputs are invalid.

36.1.14 Function `openai_schema`

```
def openai_schema(
    cls: pydantic.main.BaseModel,
    additional_ignored_keywords: list[str] = []
) -> Dict[str, Any]
```

Return the schema in the format of OpenAI's schema as jsonschema
Note —–= It's important to add a docstring to describe how to best use this class, it will be included in the description attribute and be part of the prompt.
Returns —–= dict : A dictionary in the format of OpenAI's schema as jsonschema

36.1.15 Function `read_ome_xml`

```
def read_ome_xml(
    path: str
) -> xml.etree.ElementTree.Element
```

This method reads the ome xml file and returns the ET root element.

36.1.16 Function `render_cell_output`

```
def render_cell_output(
    output_path
)
```

Load the captured output from a file and render it.
Parameters: output_path (str): Path to the output file where the cell output is saved.

36.1.17 Function `safe_float`

```
def safe_float(
    value
)
```

Safely convert a value to float, returning None if conversion is not possible.
Args —–= **`value`** : The value to convert to float.
Returns —–= float or None : The float value if conversion is successful, None otherwise.

36.1.18 Function `save_and_stream_output`

```
def save_and_stream_output(
    output_path='out/jupyter_cell_outputs/cell_output_2024-07-23T16:27:21.931842_.json'
)
```

Context manager to capture the output of a code block, save it to a file, and print it to the console in real-time.
Parameters: output_path (str): Path to the output file where the cell output will be saved.

36.1.19 Function `save_output`

```
def save_output(
    output: str,
    cost: float,
    attempts: float,
    path: str
) -> bool
```

Save output to a file.
Args —–= **`output`** : str : The output to save.

`path : str` The file path to save to.

Returns —–= bool : True if save was successful, False otherwise.

36.1.20 Function `sort_models_by_dependencies`

```
def sort_models_by_dependencies(
    root_model: Type[pydantic.main.BaseModel]
) -> List[Type[pydantic.main.BaseModel]]
```

Function to sort models by dependencies

36.1.21 Function `update_state`

```
def update_state(
    current_state: ome_types._autogenerated.ome_2016_06.ome.OME,
    proposed_change: list
) -> ome_types._autogenerated.ome_2016_06.ome.OME
```

Update the OME state based on proposed changes using JSONPatch, automatically creating missing nodes.
Args —–= **`current_state`** : OME : The current OME state.

`proposed_change` : list The change proposed as a JSON Patch document.

Returns ——= OME : The updated OME state.
Raises ——= jsonpatch.JsonPatchException : If the patch is invalid or cannot be applied.

ValueError If the resulting document is not a valid OME model.

36.2 Classes

36.2.1 Class `Tee`

```
class Tee(
    *streams
)
```

Methods

Method `flush`

```
def flush(
    self
)
```

Method `write`

```
def write(
    self,
    data
)
```

Generated by *pdoc* 0.11.1 (`https://pdoc3.github.io`).