```
import functools
import inspect
import json
from logging import getLogger
from typing import (
    Any,
    Callable,
    Dict,
    ForwardRef,
    List,
    Optional,
    Set,
    Tuple,
    Type,
    TypeVar,
    Union,
    get_args,
)

from pydantic import BaseModel, Field
from pydantic.version import VERSION as PYDANTIC_VERSION
from typing_extensions import Annotated, Literal, get_args, get_origin

T = TypeVar("T")

__all__ = (
```

```
    "JsonSchemaValue",
    "model_dump",
    "model_dump_json",
    "type2schema",
    "evaluate_forwardref",
)

PYDANTIC_V1 = PYDANTIC_VERSION.startswith("1.")

logger = getLogger(__name__)



if not PYDANTIC_V1:
    from pydantic import TypeAdapter
    from pydantic._internal._typing_extra import (
        eval_type_lenient as evaluate_forwardref,
    )
    from pydantic.json_schema import JsonSchemaValue

    def type2schema(t: Any) -> JsonSchemaValue:
        """Convert a type to a JSON schema

        Args:
            t (Type): The type to convert

        Returns:
```

```
        JsonSchemaValue: The JSON schema
    """
    return TypeAdapter(t).json_schema()


def model_dump(model: BaseModel) -> Dict[str, Any]:
    """Convert a pydantic model to a dict

    Args:
        model (BaseModel): The model to convert

    Returns:
        Dict[str, Any]: The dict representation of the model

    """
    return model.model_dump()


def model_dump_json(model: BaseModel) -> str:
    """Convert a pydantic model to a JSON string

    Args:
        model (BaseModel): The model to convert

    Returns:
        str: The JSON string representation of the model
    """
    return model.model_dump_json()
```

```
# Remove this once we drop support for pydantic 1.x
else:  # pragma: no cover
    from pydantic import schema_of
    from pydantic.typing import (
        evaluate_forwardref as evaluate_forwardref,  # type: ignore[no-redef]
    )

    JsonSchemaValue = Dict[str, Any]  # type: ignore[misc]

    def type2schema(t: Any) -> JsonSchemaValue:
        """Convert a type to a JSON schema

        Args:
            t (Type): The type to convert

        Returns:
            JsonSchemaValue: The JSON schema
        """
        if PYDANTIC_V1:
            if t is None:
                return {"type": "null"}
            elif get_origin(t) is Union:
                return {
                    "anyOf": [type2schema(tt) for tt in get_args(t)]
```

```
            }
        elif get_origin(t) in [Tuple, tuple]:
            prefixItems = [type2schema(tt) for tt in get_args(t)]
            return {
                "maxItems": len(prefixItems),
                "minItems": len(prefixItems),
                "prefixItems": prefixItems,
                "type": "array",
            }

    d = schema_of(t)
    if "title" in d:
        d.pop("title")
    if "description" in d:
        d.pop("description")

    return d

def model_dump(model: BaseModel) -> Dict[str, Any]:
    """Convert a pydantic model to a dict

    Args:
        model (BaseModel): The model to convert

    Returns:
        Dict[str, Any]: The dict representation of the model
```

```
        """
        return model.dict()

    def model_dump_json(model: BaseModel) -> str:
        """Convert a pydantic model to a JSON string

        Args:
            model (BaseModel): The model to convert

        Returns:
            str: The JSON string representation of the model
        """
        return model.json()


def get_typed_annotation(
    annotation: Any, globalns: Dict[str, Any]
) -> Any:
    """Get the type annotation of a parameter.

    Args:
        annotation: The annotation of the parameter
        globalns: The global namespace of the function

    Returns:
```

```
        The type annotation of the parameter
    """
    if isinstance(annotation, str):
        annotation = ForwardRef(annotation)
        annotation = evaluate_forwardref(
            annotation, globalns, globalns
        )
    return annotation



def get_typed_signature(
    call: Callable[..., Any]
) -> inspect.Signature:
    """Get the signature of a function with type annotations.

    Args:
        call: The function to get the signature for

    Returns:
        The signature of the function with type annotations
    """
    signature = inspect.signature(call)
    globalns = getattr(call, "__globals__", {})
    typed_params = [
        inspect.Parameter(
            name=param.name,
```

```
            kind=param.kind,
            default=param.default,
            annotation=get_typed_annotation(
                param.annotation, globalns
            ),
        )
        for param in signature.parameters.values()
    ]
    typed_signature = inspect.Signature(typed_params)
    return typed_signature


def get_typed_return_annotation(call: Callable[..., Any]) -> Any:
    """Get the return annotation of a function.

    Args:
        call: The function to get the return annotation for

    Returns:
        The return annotation of the function
    """
    signature = inspect.signature(call)
    annotation = signature.return_annotation

    if annotation is inspect.Signature.empty:
        return None
```

```
    globalns = getattr(call, "__globals__", {})
    return get_typed_annotation(annotation, globalns)


def get_param_annotations(
    typed_signature: inspect.Signature,
) -> Dict[str, Union[Annotated[Type[Any], str], Type[Any]]]:
    """Get the type annotations of the parameters of a function

    Args:
        typed_signature: The signature of the function with type annotations

    Returns:
        A dictionary of the type annotations of the parameters of the function
    """
    return {
        k: v.annotation
        for k, v in typed_signature.parameters.items()
        if v.annotation is not inspect.Signature.empty
    }


class Parameters(BaseModel):
    """Parameters of a function as defined by the OpenAI API"""
```

```
    type: Literal["object"] = "object"
    properties: Dict[str, JsonSchemaValue]
    required: List[str]


class Function(BaseModel):
    """A function as defined by the OpenAI API"""

    description: Annotated[
        str, Field(description="Description of the function")
    ]
    name: Annotated[str, Field(description="Name of the function")]
    parameters: Annotated[
        Parameters, Field(description="Parameters of the function")
    ]


class ToolFunction(BaseModel):
    """A function under tool as defined by the OpenAI API."""

    type: Literal["function"] = "function"
    function: Annotated[
        Function, Field(description="Function under tool")
    ]
```

```
def get_parameter_json_schema(
    k: str, v: Any, default_values: Dict[str, Any]
) -> JsonSchemaValue:
    """Get a JSON schema for a parameter as defined by the OpenAI API

    Args:
        k: The name of the parameter
        v: The type of the parameter
        default_values: The default values of the parameters of the function

    Returns:
        A Pydanitc model for the parameter
    """

    def type2description(
        k: str, v: Union[Annotated[Type[Any], str], Type[Any]]
    ) -> str:
        # handles Annotated
        if hasattr(v, "__metadata__"):
            retval = v.__metadata__[0]
            if isinstance(retval, str):
                return retval
            else:
                raise ValueError(
                    f"Invalid description {retval} for parameter {k}, should be a string."
                )
```

```
        else:
            return k

    schema = type2schema(v)
    if k in default_values:
        dv = default_values[k]
        schema["default"] = dv

    schema["description"] = type2description(k, v)

    return schema



def get_required_params(
    typed_signature: inspect.Signature,
) -> List[str]:
    """Get the required parameters of a function

    Args:
        signature: The signature of the function as returned by inspect.signature

    Returns:
        A list of the required parameters of the function
    """
    return [
        k
```

```
        for k, v in typed_signature.parameters.items()
        if v.default == inspect.Signature.empty
    ]


def get_default_values(
    typed_signature: inspect.Signature,
) -> Dict[str, Any]:
    """Get default values of parameters of a function

    Args:
        signature: The signature of the function as returned by inspect.signature

    Returns:
        A dictionary of the default values of the parameters of the function
    """
    return {
        k: v.default
        for k, v in typed_signature.parameters.items()
        if v.default != inspect.Signature.empty
    }


def get_parameters(
    required: List[str],
    param_annotations: Dict[
```

```
        str, Union[Annotated[Type[Any], str], Type[Any]]
    ],
    default_values: Dict[str, Any],
) -> Parameters:
    """Get the parameters of a function as defined by the OpenAI API

    Args:
        required: The required parameters of the function
        hints: The type hints of the function as returned by typing.get_type_hints

    Returns:
        A Pydantic model for the parameters of the function
    """
    return Parameters(
        properties={
            k: get_parameter_json_schema(k, v, default_values)
            for k, v in param_annotations.items()
            if v is not inspect.Signature.empty
        },
        required=required,
    )


def get_missing_annotations(
    typed_signature: inspect.Signature, required: List[str]
) -> Tuple[Set[str], Set[str]]:
```

```
    """Get the missing annotations of a function

     Ignores the parameters with default values as they are not required to be annotated, but logs a warning.
    Args:
        typed_signature: The signature of the function with type annotations
        required: The required parameters of the function

    Returns:
        A set of the missing annotations of the function
    """
    all_missing = {
        k
        for k, v in typed_signature.parameters.items()
        if v.annotation is inspect.Signature.empty
    }
    missing = all_missing.intersection(set(required))
    unannotated_with_default = all_missing.difference(missing)
    return missing, unannotated_with_default


def get_openai_function_schema_from_func(
    function: Callable[..., Any],
    *,
    name: Optional[str] = None,
    description: str = None,
```

) -> Dict[str, Any]:

"""Get a JSON schema for a function as defined by the OpenAI API

Args:

f: The function to get the JSON schema for

name: The name of the function

description: The description of the function

Returns:

A JSON schema for the function

Raises:

TypeError: If the function is not annotated

Examples:

```python
def f(a: Annotated[str, "Parameter a"], b: int = 2, c: Annotated[float, "Parameter c"] = 0.1) -> None:
    pass

get_function_schema(f, description="function f")

#   {'type': 'function',
#    'function': {'description': 'function f',
#        'name': 'f',
#        'parameters': {'type': 'object',
```

```
    #           'properties': {'a': {'type': 'str', 'description': 'Parameter a'},
    #               'b': {'type': 'int', 'description': 'b'},
    #               'c': {'type': 'float', 'description': 'Parameter c'}},
    #           'required': ['a']}}}
    ```


    """
    typed_signature = get_typed_signature(function)
    required = get_required_params(typed_signature)
    default_values = get_default_values(typed_signature)
    param_annotations = get_param_annotations(typed_signature)
    return_annotation = get_typed_return_annotation(function)
    missing, unannotated_with_default = get_missing_annotations(
        typed_signature, required
)

    if return_annotation is None:
        logger.warning(
            f"The return type of the function '{function.__name__}' is not annotated. Although annotating
it is "
            + "optional, the function should return either a string, a subclass of 'pydantic.BaseModel'."
)

    if unannotated_with_default != set():
        unannotated_with_default_s = [
            f"'{k}'" for k in sorted(unannotated_with_default)
```
```

```
        ]
        logger.warning(
            f"The following parameters of the function '{function.__name__}' with default values are not annotated: "
            + f"{', '.join(unannotated_with_default_s)}."
        )

    if missing != set():
        missing_s = [f"'{k}'" for k in sorted(missing)]
        raise TypeError(
            f"All parameters of the function '{function.__name__}' without default values must be annotated. "
            + f"The annotations are missing for the following parameters: {', '.join(missing_s)}"
        )

    fname = name if name else function.__name__

    parameters = get_parameters(
        required, param_annotations, default_values=default_values
    )

    function = ToolFunction(
        function=Function(
            description=description,
            name=fname,
            parameters=parameters,
```

```
        )
    )

    return model_dump(function)


#
def get_load_param_if_needed_function(
    t: Any,
) -> Optional[Callable[[Dict[str, Any], Type[BaseModel]], BaseModel]]:
    """Get a function to load a parameter if it is a Pydantic model

    Args:
        t: The type annotation of the parameter

    Returns:
        A function to load the parameter if it is a Pydantic model, otherwise None

    """
    if get_origin(t) is Annotated:
        return get_load_param_if_needed_function(get_args(t)[0])

    def load_base_model(
        v: Dict[str, Any], t: Type[BaseModel]
    ) -> BaseModel:
        return t(**v)
```

```
    return (
        load_base_model
        if isinstance(t, type) and issubclass(t, BaseModel)
        else None
    )


def load_basemodels_if_needed(
    func: Callable[..., Any]
) -> Callable[..., Any]:
    """A decorator to load the parameters of a function if they are Pydantic models

    Args:
        func: The function with annotated parameters

    Returns:
        A function that loads the parameters before calling the original function

    """
    # get the type annotations of the parameters
    typed_signature = get_typed_signature(func)
    param_annotations = get_param_annotations(typed_signature)

    # get functions for loading BaseModels when needed based on the type annotations
    kwargs_mapping_with_nones = {
```

```
    k: get_load_param_if_needed_function(t)
    for k, t in param_annotations.items()
}

# remove the None values
kwargs_mapping = {
    k: f
    for k, f in kwargs_mapping_with_nones.items()
    if f is not None
}

# a function that loads the parameters before calling the original function
@functools.wraps(func)
def _load_parameters_if_needed(*args: Any, **kwargs: Any) -> Any:
    # load the BaseModels if needed
    for k, f in kwargs_mapping.items():
        kwargs[k] = f(kwargs[k], param_annotations[k])

    # call the original function
    return func(*args, **kwargs)

@functools.wraps(func)
async def _a_load_parameters_if_needed(
    *args: Any, **kwargs: Any
) -> Any:
    # load the BaseModels if needed
```

```
        for k, f in kwargs_mapping.items():
            kwargs[k] = f(kwargs[k], param_annotations[k])

        # call the original function
        return await func(*args, **kwargs)

    if inspect.iscoroutinefunction(func):
        return _a_load_parameters_if_needed
    else:
        return _load_parameters_if_needed


def serialize_to_str(x: Any) -> str:
    if isinstance(x, str):
        return x
    elif isinstance(x, BaseModel):
        return model_dump_json(x)
    else:
        return json.dumps(x)
```