pydantic_ai.tools

AgentDeps module-attribute

```
AgentDeps = TypeVar('AgentDeps')
```

Type variable for agent dependencies.

RunContext dataclass

Bases: `Generic[AgentDeps]`

Information about the current call.

Source code in `pydantic_ai_slim/pydantic_ai/tools.py`

```
@dataclass
class RunContext(Generic[AgentDeps]):
    """Information about the current call."""

    deps: AgentDeps
    """Dependencies for the agent."""
    retry: int
    """Number of retries so far."""
    tool_name: str | None = None
    """Name of the tool being called."""
```

deps instance-attribute

```
deps: AgentDeps
```

Dependencies for the agent.

retry instance-attribute

```
retry: int
```

Number of retries so far.

tool_name class-attribute instance-attribute

```
tool_name: str | None = None
```

Name of the tool being called.

ToolParams module-attribute

```
ToolParams = ParamSpec('ToolParams')
```

Retrieval function param spec.

SystemPromptFunc module-attribute

```
SystemPromptFunc = Union[
    Callable[[RunContext[AgentDeps]], str],
    Callable[[RunContext[AgentDeps]], Awaitable[str]],
    Callable[[], str],
    Callable[[], Awaitable[str]],
]
```

A function that may or maybe not take `RunContext` as an argument, and may or may not be async.

Usage `SystemPromptFunc[AgentDeps]`.

ResultValidatorFunc module-attribute

```
ResultValidatorFunc = Union[
    Callable[
        [RunContext[AgentDeps], ResultData], ResultData
    ],
    Callable[
        [RunContext[AgentDeps], ResultData],
        Awaitable[ResultData],
    ],
    Callable[[ResultData], ResultData],
    Callable[[ResultData], Awaitable[ResultData]],
]
```

A function that always takes `ResultData` and returns `ResultData`, but may or maybe not take `CallInfo` as a first argument, and may or may not be async.

Usage `ResultValidator[AgentDeps, ResultData]`.

ToolFuncContext module-attribute

```
ToolFuncContext = Callable[
    Concatenate[RunContext[AgentDeps], ToolParams], Any
]
```

A tool function that takes `RunContext` as the first argument.

Usage `ToolContextFunc[AgentDeps, ToolParams]`.

ToolFuncPlain module-attribute

```
ToolFuncPlain = Callable[ToolParams, Any]
```

A tool function that does not take `RunContext` as the first argument.

Usage `ToolPlainFunc[ToolParams]`.

ToolFuncEither `module-attribute`

```
ToolFuncEither = Union[
    ToolFuncContext[AgentDeps, ToolParams],
    ToolFuncPlain[ToolParams],
]
```

Either kind of tool function.

This is just a union of `ToolFuncContext` and `ToolFuncPlain`.

Usage `ToolFuncEither[AgentDeps, ToolParams]`.

ToolPrepareFunc `module-attribute`

```
ToolPrepareFunc: TypeAlias = (
    "Callable[[RunContext[AgentDeps], ToolDefinition], Awaitable[ToolDefinition | None]]"
)
```

Definition of a function that can prepare a tool definition at call time.

See tool docs for more information.

Example — here `only_if_42` is valid as a `ToolPrepareFunc`:

```
from typing import Union

from pydantic_ai import RunContext, Tool
from pydantic_ai.tools import ToolDefinition

async def only_if_42(
    ctx: RunContext[int], tool_def: ToolDefinition
) -> Union[ToolDefinition, None]:
    if ctx.deps == 42:
        return tool_def

def hitchhiker(ctx: RunContext[int], answer: str) -> str:
    return f'{ctx.deps} {answer}'

hitchhiker = Tool(hitchhiker, prepare=only_if_42)
```

Usage `ToolPrepareFunc[AgentDeps]`.

Tool `dataclass`

Bases: `Generic[AgentDeps]`

A tool function for an agent.

Source code in `pydantic_ai_slim/pydantic_ai/tools.py`

~~~python
@dataclass(init=False)
class Tool(Generic[AgentDeps]):
    """A tool function for an agent."""

    function: ToolFuncEither[AgentDeps, ...]
    takes_ctx: bool
    max_retries: int | None
    name: str
    description: str
    prepare: ToolPrepareFunc[AgentDeps] | None
    _is_async: bool = field(init=False)
    _single_arg_name: str | None = field(init=False)
    _positional_fields: list[str] = field(init=False)
    _var_positional_field: str | None = field(init=False)
    _validator: SchemaValidator = field(init=False, repr=False)
    _parameters_json_schema: ObjectJsonSchema = field(init=False)
    current_retry: int = field(default=0, init=False)

    def __init__(
        self,
        function: ToolFuncEither[AgentDeps, ...],
        *,
        takes_ctx: bool | None = None,
        max_retries: int | None = None,
        name: str | None = None,
        description: str | None = None,
        prepare: ToolPrepareFunc[AgentDeps] | None = None,
    ):
        """Create a new tool instance.

        Example usage:

        ```py
        from pydantic_ai import Agent, RunContext, Tool

        async def my_tool(ctx: RunContext[int], x: int, y: int) -> str:
            return f'{ctx.deps} {x} {y}'

        agent = Agent('test', tools=[Tool(my_tool)])
        ~~~

        or with a custom prepare method:

        ```py
        from typing import Union

        from pydantic_ai import Agent, RunContext, Tool
        from pydantic_ai.tools import ToolDefinition

        async def my_tool(ctx: RunContext[int], x: int, y: int) -> str:
            return f'{ctx.deps} {x} {y}'

        async def prep_my_tool(
            ctx: RunContext[int], tool_def: ToolDefinition
        ) -> Union[ToolDefinition, None]:
            # only register the tool if `deps == 42`
            if ctx.deps == 42:
                return tool_def

        agent = Agent('test', tools=[Tool(my_tool, prepare=prep_my_tool)])
        ```

        Args:
            function: The Python function to call as the tool.
            takes_ctx: Whether the function takes a [`RunContext`][pydantic_ai.tools.RunContext] first argument,
                this is inferred if unset.
            max_retries: Maximum number of retries allowed for this tool, set to the agent default if `None`.
            name: Name of the tool, inferred from the function if `None`.
            description: Description of the tool, inferred from the function if `None`.
            prepare: custom method to prepare the tool definition for each step, return `None` to omit this
                tool from a given step. This is useful if you want to customise a tool at call time,
                or omit it completely from a step. See [`ToolPrepareFunc`][pydantic_ai.tools.ToolPrepareFunc].
        """
        if takes_ctx is None:
            takes_ctx = _pydantic.takes_ctx(function)

        f = _pydantic.function_schema(function, takes_ctx)
        self.function = function
        self.takes_ctx = takes_ctx
        self.max_retries = max_retries
        self.name = name or function.__name__
        self.description = description or f['description']
        self.prepare = prepare
        self._is_async = inspect.iscoroutinefunction(self.function)
        self._single_arg_name = f['single_arg_name']
        self._positional_fields = f['positional_fields']
        self._var_positional_field = f['var_positional_field']
        self._validator = f['validator']
        self._parameters_json_schema = f['json_schema']

    async def prepare_tool_def(self, ctx: RunContext[AgentDeps]) -> ToolDefinition | None:
        """Get the tool definition.

        By default, this method creates a tool definition, then either returns it, or calls `self.prepare`
        if it's set.

        Returns:
            return a `ToolDefinition` or `None` if the tools should not be registered for this run.
        """
        tool_def = ToolDefinition(
            name=self.name,
            description=self.description,
            parameters_json_schema=self._parameters_json_schema,
)
        if self.prepare is not None:
            return await self.prepare(ctx, tool_def)
        else:
            return tool_def

    async def run(self, deps: AgentDeps, message: messages.ToolCall) -> messages.Message:
        """Run the tool function asynchronously."""
        try:
            if isinstance(message.args, messages.ArgsJson):
                args_dict = self._validator.validate_json(message.args.args_json)
            else:
                args_dict = self._validator.validate_python(message.args.args_dict)
        except ValidationError as e:
            return self._on_error(e, message)

        args, kwargs = self._call_args(deps, args_dict, message)
        try:
            if self._is_async:
                function = cast(Callable[[Any], Awaitable[str]], self.function)
                response_content = await function(*args, **kwargs)
            else:
                function = cast(Callable[[Any], str], self.function)
```
```

```python
                response_content = await _utils.run_in_executor(function, *args, **kwargs)
        except ModelRetry as e:
            return self._on_error(e, message)

        self.current_retry = 0
        return messages.ToolReturn(
            tool_name=message.tool_name,
            content=response_content,
            tool_call_id=message.tool_call_id,
        )

    def _call_args(
        self, deps: AgentDeps, args_dict: dict[str, Any], message: messages.ToolCall
    ) -> tuple[list[Any], dict[str, Any]]:
        if self._single_arg_name:
            args_dict = {self._single_arg_name: args_dict}

        args = [RunContext(deps, self.current_retry, message.tool_name)] if self.takes_ctx else []
        for positional_field in self._positional_fields:
            args.append(args_dict.pop(positional_field))
        if self._var_positional_field:
            args.extend(args_dict.pop(self._var_positional_field))

        return args, args_dict

    def _on_error(self, exc: ValidationError | ModelRetry, call_message: messages.ToolCall) -> messages.RetryPrompt:
        self.current_retry += 1
        if self.max_retries is None or self.current_retry > self.max_retries:
            raise UnexpectedModelBehavior(f'Tool exceeded max retries count of {self.max_retries}') from exc
        else:
            if isinstance(exc, ValidationError):
                content = exc.errors(include_url=False)
            else:
                content = exc.message
            return messages.RetryPrompt(
                tool_name=call_message.tool_name,
                content=content,
                tool_call_id=call_message.tool_call_id,
            )
```

__init__

```python
__init__(
    function: ToolFuncEither[AgentDeps, ...],
    *,
    takes_ctx: bool | None = None,
    max_retries: int | None = None,
    name: str | None = None,
    description: str | None = None,
    prepare: ToolPrepareFunc[AgentDeps] | None = None
)
```

Create a new tool instance.

Example usage:

```python
from pydantic_ai import Agent, RunContext, Tool

async def my_tool(ctx: RunContext[int], x: int, y: int) -> str:
    return f'{ctx.deps} {x} {y}'

agent = Agent('test', tools=[Tool(my_tool)])
```

or with a custom prepare method:

```python
from typing import Union

from pydantic_ai import Agent, RunContext, Tool
from pydantic_ai.tools import ToolDefinition

async def my_tool(ctx: RunContext[int], x: int, y: int) -> str:
    return f'{ctx.deps} {x} {y}'

async def prep_my_tool(
    ctx: RunContext[int], tool_def: ToolDefinition
) -> Union[ToolDefinition, None]:
    # only register the tool if `deps == 42`
    if ctx.deps == 42:
        return tool_def

agent = Agent('test', tools=[Tool(my_tool, prepare=prep_my_tool)])
```

Parameters:

Name	Type	Description	Default
`function`	`ToolFuncEither[AgentDeps, ...]`	The Python function to call as the tool.	*required*
`takes_ctx`	`bool | None`	Whether the function takes a `RunContext` first argument, this is inferred if unset.	`None`
`max_retries`	`int | None`	Maximum number of retries allowed for this tool, set to the agent default if `None`.	`None`
`name`	`str | None`	Name of the tool, inferred from the function if `None`.	`None`
`description`	`str | None`	Description of the tool, inferred from the function if `None`.	`None`
`prepare`	`ToolPrepareFunc[AgentDeps] | None`	custom method to prepare the tool definition for each step, return `None` to omit this tool from a given step. This is useful if you want to customise a tool at call time, or omit it completely from a step. See `ToolPrepareFunc`.	`None`

Source code in `pydantic_ai_slim/pydantic_ai/tools.py`

~~~python
def __init__(
    self,
    function: ToolFuncEither[AgentDeps, ...],
    *,
    takes_ctx: bool | None = None,
    max_retries: int | None = None,
    name: str | None = None,
    description: str | None = None,
    prepare: ToolPrepareFunc[AgentDeps] | None = None,
):
    """Create a new tool instance.

    Example usage:

    ```py
    from pydantic_ai import Agent, RunContext, Tool

    async def my_tool(ctx: RunContext[int], x: int, y: int) -> str:
        return f'{ctx.deps} {x} {y}'

    agent = Agent('test', tools=[Tool(my_tool)])
    ~~~

    or with a custom prepare method:

    ```py
    from typing import Union

    from pydantic_ai import Agent, RunContext, Tool
    from pydantic_ai.tools import ToolDefinition

    async def my_tool(ctx: RunContext[int], x: int, y: int) -> str:
        return f'{ctx.deps} {x} {y}'

    async def prep_my_tool(
        ctx: RunContext[int], tool_def: ToolDefinition
    ) -> Union[ToolDefinition, None]:
        # only register the tool if `deps == 42`
        if ctx.deps == 42:
            return tool_def

    agent = Agent('test', tools=[Tool(my_tool, prepare=prep_my_tool)])
    ```

    Args:
        function: The Python function to call as the tool.
        takes_ctx: Whether the function takes a [`RunContext`][pydantic_ai.tools.RunContext] first argument,
            this is inferred if unset.
        max_retries: Maximum number of retries allowed for this tool, set to the agent default if `None`.
        name: Name of the tool, inferred from the function if `None`.
        description: Description of the tool, inferred from the function if `None`.
        prepare: custom method to prepare the tool definition for each step, return `None` to omit this
            tool from a given step. This is useful if you want to customise a tool at call time,
            or omit it completely from a step. See [`ToolPrepareFunc`][pydantic_ai.tools.ToolPrepareFunc].
    """
    if takes_ctx is None:
        takes_ctx = _pydantic.takes_ctx(function)

    f = _pydantic.function_schema(function, takes_ctx)
    self.function = function
    self.takes_ctx = takes_ctx
    self.max_retries = max_retries
    self.name = name or function.__name__
    self.description = description or f['description']
    self.prepare = prepare
    self._is_async = inspect.iscoroutinefunction(self.function)
    self._single_arg_name = f['single_arg_name']
    self._positional_fields = f['positional_fields']
    self._var_positional_field = f['var_positional_field']
    self._validator = f['validator']
    self._parameters_json_schema = f['json_schema']

prepare_tool_def `async`

```python
prepare_tool_def(
    ctx: RunContext[AgentDeps],
) -> ToolDefinition | None
```

Get the tool definition.

By default, this method creates a tool definition, then either returns it, or calls `self.prepare` if it's set.

Returns:

Type	Description
`ToolDefinition | None`	return a `ToolDefinition` or `None` if the tools should not be registered for this run.

Source code in `pydantic_ai_slim/pydantic_ai/tools.py`

```python
async def prepare_tool_def(self, ctx: RunContext[AgentDeps]) -> ToolDefinition | None:
    """Get the tool definition.

    By default, this method creates a tool definition, then either returns it, or calls `self.prepare`
    if it's set.

    Returns:
        return a `ToolDefinition` or `None` if the tools should not be registered for this run.
    """
    tool_def = ToolDefinition(
        name=self.name,
        description=self.description,
        parameters_json_schema=self._parameters_json_schema,
    )
    if self.prepare is not None:
        return await self.prepare(ctx, tool_def)
    else:
        return tool_def
```

run `async`

```python
run(deps: AgentDeps, message: ToolCall) -> Message
```

Run the tool function asynchronously.

Source code in `pydantic_ai_slim/pydantic_ai/tools.py`

```python
async def run(self, deps: AgentDeps, message: messages.ToolCall) -> messages.Message:
    """Run the tool function asynchronously."""
    try:
        if isinstance(message.args, messages.ArgsJson):
            args_dict = self._validator.validate_json(message.args.args_json)
        else:
            args_dict = self._validator.validate_python(message.args.args_dict)
    except ValidationError as e:
        return self._on_error(e, message)

    args, kwargs = self._call_args(deps, args_dict, message)
    try:
        if self._is_async:
            function = cast(Callable[[Any], Awaitable[str]], self.function)
            response_content = await function(*args, **kwargs)
        else:
            function = cast(Callable[[Any], str], self.function)
            response_content = await _utils.run_in_executor(function, *args, **kwargs)
    except ModelRetry as e:
        return self._on_error(e, message)

    self.current_retry = 0
    return messages.ToolReturn(
        tool_name=message.tool_name,
        content=response_content,
        tool_call_id=message.tool_call_id,
    )
```

ObjectJsonSchema `module-attribute`

```python
ObjectJsonSchema: TypeAlias = dict[str, Any]
```

Type representing JSON schema of an object, e.g. where `"type": "object"`.

This type is used to define tools parameters (aka arguments) in ToolDefinition.

With PEP-728 this should be a TypedDict with `type: Literal['object']`, and `extra_items=Any`

ToolDefinition `dataclass`

Definition of a tool passed to a model.

This is used for both function tools result tools.

Source code in `pydantic_ai_slim/pydantic_ai/tools.py`

```python
@dataclass
class ToolDefinition:
    """Definition of a tool passed to a model.

    This is used for both function tools result tools.
    """

    name: str
    """The name of the tool."""

    description: str
    """The description of the tool."""

    parameters_json_schema: ObjectJsonSchema
    """The JSON schema for the tool's parameters."""

    outer_typed_dict_key: str | None = None
    """The key in the outer [TypedDict] that wraps a result tool.

    This will only be set for result tools which don't have an `object` JSON schema.
    """
```

name `instance-attribute`

```python
name: str
```

The name of the tool.

description `instance-attribute`

```python
description: str
```

The description of the tool.

parameters_json_schema `instance-attribute`

```python
parameters_json_schema: ObjectJsonSchema
```

The JSON schema for the tool's parameters.

outer_typed_dict_key `class-attribute` `instance-attribute`

```python
outer_typed_dict_key: str | None = None
```

The key in the outer [TypedDict] that wraps a result tool.

This will only be set for result tools which don't have an `object` JSON schema.