

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

```
40 @dataclass
41 class RunContext(Generic[AgentDeps]):
42     """Information about the current call."""
43
44     deps: AgentDeps
45     """Dependencies for the agent."""
46     retry: int
47     """Number of retries so far."""
48     tool_name: str | None = None
49     """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.

```

128 @dataclass(init=False)
129 class Tool(Generic[AgentDeps]):
130     """A tool function for an agent."""
131
132     function: ToolFuncEither[AgentDeps, ...]
133     takes_ctx: bool
134     max_retries: int | None
135     name: str
136     description: str
137     prepare: ToolPrepareFunc[AgentDeps] | None
138     _is_async: bool = field(init=False)
139     _single_arg_name: str | None = field(init=False)
140     _positional_fields: list[str] = field(init=False)
141     _var_positional_field: str | None = field(init=False)
142     _validator: SchemaValidator = field(init=False, repr=False)
143     _parameters_json_schema: ObjectJsonSchema = field(init=False)
144     current_retry: int = field(default=0, init=False)
145
146     def __init__(
147         self,
148         function: ToolFuncEither[AgentDeps, ...],
149         *,
150         takes_ctx: bool | None = None,
151         max_retries: int | None = None,
152         name: str | None = None,
153         description: str | None = None,
154         prepare: ToolPrepareFunc[AgentDeps] | None = None,
155     ):
156         """Create a new tool instance.
157
158         Example usage:
159
160         ```py
161         from pydantic_ai import Agent, RunContext, Tool
162
163         async def my_tool(ctx: RunContext[int], x: int, y: int) -> str:
164             return f'{ctx.deps} {x} {y}'
165
166         agent = Agent('test', tools=[Tool(my_tool)])
167         ...
168
169         or with a custom prepare method:
170
171         ```py
172         from typing import Union
173
174         from pydantic_ai import Agent, RunContext, Tool
175         from pydantic_ai.tools import ToolDefinition
176
177         async def my_tool(ctx: RunContext[int], x: int, y: int) -> str:
178             return f'{ctx.deps} {x} {y}'
179
180         async def prep_my_tool(
181             ctx: RunContext[int], tool_def: ToolDefinition
182         ) -> Union[ToolDefinition, None]:
183             # only register the tool if 'deps' == 42
184             if ctx.deps == 42:
185                 return tool_def
186
187         agent = Agent('test', tools=[Tool(my_tool, prepare=prep_my_tool)])
188         ...
189
190         Args:
191             function: The Python function to call as the tool.
192             takes_ctx: Whether the function takes a ['RunContext'][pydantic_ai.tools.RunContext] first argument,
193                 this is inferred if unset.
194             max_retries: Maximum number of retries allowed for this tool, set to the agent default if 'None'.
195             name: Name of the tool, inferred from the function if 'None'.
196             description: Description of the tool, inferred from the function if 'None'.
197             prepare: custom method to prepare the tool definition for each step, return 'None' to omit this
198                 tool from a given step. This is useful if you want to customise a tool at call time,
199                 or omit it completely from a step. See ['ToolPrepareFunc'][pydantic_ai.tools.ToolPrepareFunc].
200         """
201         if takes_ctx is None:
202             takes_ctx = _pydantic.takes_ctx(function)
203
204         f = _pydantic.function_schema(function, takes_ctx)
205         self.function = function
206         self.takes_ctx = takes_ctx
207         self.max_retries = max_retries
208         self.name = name or function.__name__
209         self.description = description or f['description']
210         self.prepare = prepare
211         self._is_async = inspect.iscoroutinefunction(self.function)
212         self._single_arg_name = f['single_arg_name']
213         self._positional_fields = f['positional_fields']
214         self._var_positional_field = f['var_positional_field']
215         self._validator = f['validator']
216         self._parameters_json_schema = f['json_schema']
217
218     async def prepare_tool_def(self, ctx: RunContext[AgentDeps]) -> ToolDefinition | None:
219         """Get the tool definition.
220
221         By default, this method creates a tool definition, then either returns it, or calls 'self.prepare'
222         if it's set.
223
224         Returns:
225             return a 'ToolDefinition' or 'None' if the tools should not be registered for this run.
226         """
227         tool_def = ToolDefinition(
228             name=self.name,
229             description=self.description,
230             parameters_json_schema=self._parameters_json_schema,
231         )
232         if self.prepare is not None:
233             return await self.prepare(ctx, tool_def)
234         else:
235             return tool_def
236
237     async def run(self, deps: AgentDeps, message: messages.ToolCall) -> messages.Message:
238         """Run the tool function asynchronously."""
239         try:
240             if isinstance(message.args, messages.ArgsJson):
241                 args_dict = self._validator.validate_json(message.args.args_json)
242             else:
243                 args_dict = self._validator.validate_python(message.args.args_dict)
244         except ValidationError as e:
245             return self._on_error(e, message)
246
247         args, kwargs = self._call_args(deps, args_dict, message)
248         try:
249             if self._is_async:
250                 function = cast(Callable[[Any], Awaitable[str]], self.function)
251                 response_content = await function(*args, **kwargs)
252             else:
253                 function = cast(Callable[[Any], str], self.function)

```

```

255         response_content = await utils.run_in_executor(function, *args, **kwargs)
256     except ModelRetry as e:
257         return self._on_error(e, message)
258
259     self.current_retry = 0
260     return messages.ToolReturn(
261         tool_name=message.tool_name,
262         content=response_content,
263         tool_call_id=message.tool_call_id,
264     )
265
266     def _call_args(
267         self, deps: AgentDeps, args_dict: dict[str, Any], message: messages.ToolCall
268     ) -> tuple[list[Any], dict[str, Any]]:
269         if self._single_arg_name:
270             args_dict = {self._single_arg_name: args_dict}
271
272         args = [RunContext(deps, self.current_retry, message.tool_name)] if self.takes_ctx else []
273         for positional_field in self._positional_fields:
274             args.append(args_dict.pop(positional_field))
275         if self._var_positional_field:
276             args.extend(args_dict.pop(self._var_positional_field))
277
278         return args, args_dict
279
280     def _on_error(self, exc: ValidationError | ModelRetry, call_message: messages.ToolCall) -> messages.RetryPrompt:
281         self.current_retry += 1
282         if self.max_retries is None or self.current_retry > self.max_retries:
283             raise UnexpectedModelBehavior(f'Tool exceeded max retries count of {self.max_retries}') from exc
284         else:
285             if isinstance(exc, ValidationError):
286                 content = exc.errors(include_url=False)
287             else:
288                 content = exc.message
289             return messages.RetryPrompt(
290                 tool_name=call_message.tool_name,
291                 content=content,
292                 tool_call_id=call_message.tool_call_id,
293             )

```

__init__

```

__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:

```

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:

```

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	<code>ToolFuncEither[AgentDeps, ...]</code>	The Python function to call as the tool.	<i>required</i>
takes_ctx	<code>bool</code> <code>None</code>	Whether the function takes a <code>RunContext</code> first argument, this is inferred if unset.	<code>None</code>
max_retries	<code>int</code> <code>None</code>	Maximum number of retries allowed for this tool, set to the agent default if <code>None</code> .	<code>None</code>
name	<code>str</code> <code>None</code>	Name of the tool, inferred from the function if <code>None</code> .	<code>None</code>
description	<code>str</code> <code>None</code>	Description of the tool, inferred from the function if <code>None</code> .	<code>None</code>
prepare	<code>ToolPrepareFunc[AgentDeps]</code> <code>None</code>	custom method to prepare the tool definition for each step, return <code>None</code> 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 <code>ToolPrepareFunc</code> .	<code>None</code>

```
146 def __init__(
147     self,
148     function: ToolFuncEither[AgentDeps, ...],
149     *,
150     takes_ctx: bool | None = None,
151     max_retries: int | None = None,
152     name: str | None = None,
153     description: str | None = None,
154     prepare: ToolPrepareFunc[AgentDeps] | None = None,
155 ):
156     """Create a new tool instance.
157
158     Example usage:
159
160     ```py
161     from pydantic_ai import Agent, RunContext, Tool
162
163     async def my_tool(ctx: RunContext[int], x: int, y: int) -> str:
164         return f'{ctx.deps} {x} {y}'
165
166     agent = Agent('test', tools=[Tool(my_tool)])
167     ...
168
169     or with a custom prepare method:
170
171     ```py
172     from typing import Union
173
174     from pydantic_ai import Agent, RunContext, Tool
175     from pydantic_ai.tools import ToolDefinition
176
177     async def my_tool(ctx: RunContext[int], x: int, y: int) -> str:
178         return f'{ctx.deps} {x} {y}'
179
180     async def prep_my_tool(
181         ctx: RunContext[int], tool_def: ToolDefinition
182     ) -> Union[ToolDefinition, None]:
183         # only register the tool if 'deps' == 42
184         if ctx.deps == 42:
185             return tool_def
186
187     agent = Agent('test', tools=[Tool(my_tool, prepare=prep_my_tool)])
188     ...
189
190 Args:
191     function: The Python function to call as the tool.
192     takes_ctx: Whether the function takes a ['RunContext'] [pydantic_ai.tools.RunContext] first argument,
193         this is inferred if unset.
194     max_retries: Maximum number of retries allowed for this tool, set to the agent default if 'None'.
195     name: Name of the tool, inferred from the function if 'None'.
196     description: Description of the tool, inferred from the function if 'None'.
197     prepare: custom method to prepare the tool definition for each step, return 'None' to omit this
198         tool from a given step. This is useful if you want to customise a tool at call time,
199         or omit it completely from a step. See ['ToolPrepareFunc'] [pydantic_ai.tools.ToolPrepareFunc].
200 """
201
202 if takes_ctx is None:
203     takes_ctx = _pydantic.takes_ctx(function)
204
205 f = _pydantic.function_schema(function, takes_ctx)
206 self.function = function
207 self.takes_ctx = takes_ctx
208 self.max_retries = max_retries
209 self.name = name or function.__name__
210 self.description = description or f['description']
211 self.prepare = prepare
212 self.is_async = inspect.iscoroutinefunction(self.function)
213 self._single_arg_name = f['single_arg_name']
214 self._positional_fields = f['positional_fields']
215 self._var_positional_field = f['var_positional_field']
216 self._validator = f['validator']
217 self._parameters_json_schema = f['json_schema']
```

prepare_tool_def async

```
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
<code>ToolDefinition</code> <code>None</code>	return a <code>ToolDefinition</code> or <code>None</code> if the tools should not be registered for this run.

```
219 async def prepare_tool_def(self, ctx: RunContext[AgentDeps]) -> ToolDefinition | None:
220     """Get the tool definition.
221
222     By default, this method creates a tool definition, then either returns it, or calls 'self.prepare'
223     if it's set.
224
225     Returns:
226         return a 'ToolDefinition' or 'None' if the tools should not be registered for this run.
227     """
228     tool_def = ToolDefinition(
229         name=self.name,
230         description=self.description,
231         parameters_json_schema=self._parameters_json_schema,
232     )
233     if self.prepare is not None:
234         return await self.prepare(ctx, tool_def)
235     else:
236         return tool_def
```

run async

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

Run the tool function asynchronously.

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

238 async def run(self, deps: AgentDeps, message: messages.ToolCall) -> messages.Message:
239     """Run the tool function asynchronously."""
240     try:
241         if isinstance(message.args, messages.ArgsJson):
242             args_dict = self._validator.validate_json(message.args.args_json)
243         else:
244             args_dict = self._validator.validate_python(message.args.args_dict)
245     except ValidationError as e:
246         return self._on_error(e, message)
247
248     args, kwargs = self._call_args(deps, args_dict, message)
249     try:
250         if self._is_async:
251             function = cast(Callable[[Any], Awaitable[str]], self.function)
252             response_content = await function(*args, **kwargs)
253         else:
254             function = cast(Callable[[Any], str], self.function)
255             response_content = await _utils.run_in_executor(function, *args, **kwargs)
256     except ModelRetry as e:
257         return self._on_error(e, message)
258
259     self.current_retry = 0
260     return messages.ToolReturn(
261         tool_name=message.tool_name,
262         content=response_content,
263         tool_call_id=message.tool_call_id,
264     )
```

ObjectJsonSchema module-attribute

```
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.

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

305 @dataclass
306 class ToolDefinition:
307     """Definition of a tool passed to a model.
308
309     This is used for both function tools result tools.
310     """
311
312     name: str
313     """The name of the tool."""
314
315     description: str
316     """The description of the tool."""
317
318     parameters_json_schema: ObjectJsonSchema
319     """The JSON schema for the tool's parameters."""
320
321     outer_typed_dict_key: str | None = None
322     """The key in the outer [TypedDict] that wraps a result tool.
323
324     This will only be set for result tools which don't have an 'object' JSON schema.
325     """
```

name instance-attribute

```
name: str
```

The name of the tool.

description instance-attribute

```
description: str
```

The description of the tool.

parameters_json_schema instance-attribute

```
parameters_json_schema: ObjectJsonSchema
```

The JSON schema for the tool's parameters.

outer_typed_dict_key class-attribute instance-attribute

```
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.