`pydantic_ai.models.function`

A model controlled by a local function.

`FunctionModel` is similar to `TestModel`, but allows greater control over the model's behavior.

Its primary use case is for more advanced unit testing than is possible with `TestModel`.

FunctionModel `dataclass`

Bases: `Model`

A model controlled by a local function.

Apart from `__init__`, all methods are private or match those of the base class.

Source code in `pydantic_ai_slim/pydantic_ai/models/function.py`

```python
@dataclass(init=False)
class FunctionModel(Model):
    """A model controlled by a local function.

    Apart from `__init__`, all methods are private or match those of the base class.
    """

    function: FunctionDef | None = None
    stream_function: StreamFunctionDef | None = None

    @overload
    def __init__(self, function: FunctionDef) -> None: ...

    @overload
    def __init__(self, *, stream_function: StreamFunctionDef) -> None: ...

    @overload
    def __init__(self, function: FunctionDef, *, stream_function: StreamFunctionDef) -> None: ...

    def __init__(self, function: FunctionDef | None = None, *, stream_function: StreamFunctionDef | None = None):
        """Initialize a `FunctionModel`.

        Either `function` or `stream_function` must be provided, providing both is allowed.

        Args:
            function: The function to call for non-streamed requests.
            stream_function: The function to call for streamed requests.
        """
        if function is None and stream_function is None:
            raise TypeError('Either `function` or `stream_function` must be provided')
        self.function = function
        self.stream_function = stream_function

    async def agent_model(
        self,
        *,
        function_tools: list[ToolDefinition],
        allow_text_result: bool,
        result_tools: list[ToolDefinition],
    ) -> AgentModel:
        return FunctionAgentModel(
            self.function, self.stream_function, AgentInfo(function_tools, allow_text_result, result_tools)
        )

    def name(self) -> str:
        labels: list[str] = []
        if self.function is not None:
            labels.append(self.function.__name__)
        if self.stream_function is not None:
            labels.append(f'stream-{self.stream_function.__name__}')
        return f'function:{",".join(labels)}'
```

__init__

```python
__init__(function: FunctionDef) -> None
```

```python
__init__(*, stream_function: StreamFunctionDef) -> None
```

```python
__init__(
    function: FunctionDef,
    *,
    stream_function: StreamFunctionDef
) -> None
```

```python
__init__(
    function: FunctionDef | None = None,
    *,
    stream_function: StreamFunctionDef | None = None
)
```

Initialize a `FunctionModel`.

Either `function` or `stream_function` must be provided, providing both is allowed.

Parameters:

Name	Type	Description	Default
`function`	`FunctionDef | None`	The function to call for non-streamed requests.	`None`
`stream_function`	`StreamFunctionDef | None`	The function to call for streamed requests.	`None`

Source code in `pydantic_ai_slim/pydantic_ai/models/function.py`

```python
def __init__(self, function: FunctionDef | None = None, *, stream_function: StreamFunctionDef | None = None):
    """Initialize a `FunctionModel`.

    Either `function` or `stream_function` must be provided, providing both is allowed.

    Args:
        function: The function to call for non-streamed requests.
        stream_function: The function to call for streamed requests.
    """
    if function is None and stream_function is None:
        raise TypeError('Either `function` or `stream_function` must be provided')
    self.function = function
    self.stream_function = stream_function
```

AgentInfo `dataclass`

Information about an agent.

This is passed as the second to functions used within `FunctionModel`.

Source code in `pydantic_ai_slim/pydantic_ai/models/function.py`

```python
@dataclass(frozen=True)
class AgentInfo:
    """Information about an agent.

    This is passed as the second to functions used within [`FunctionModel`][pydantic_ai.models.function.FunctionModel].
    """

    function_tools: list[ToolDefinition]
    """The function tools available on this agent.

    These are the tools registered via the [`tool`][pydantic_ai.Agent.tool] and
    [`tool_plain`][pydantic_ai.Agent.tool_plain] decorators.
    """
    allow_text_result: bool
    """Whether a plain text result is allowed."""
    result_tools: list[ToolDefinition]
    """The tools that can called as the final result of the run."""
```

function_tools `instance-attribute`

```python
function_tools: list[ToolDefinition]
```

The function tools available on this agent.

These are the tools registered via the `tool` and `tool_plain` decorators.

allow_text_result `instance-attribute`

```python
allow_text_result: bool
```

Whether a plain text result is allowed.

result_tools `instance-attribute`

```python
result_tools: list[ToolDefinition]
```

The tools that can called as the final result of the run.

DeltaToolCall `dataclass`

Incremental change to a tool call.

Used to describe a chunk when streaming structured responses.

Source code in `pydantic_ai_slim/pydantic_ai/models/function.py`

```python
@dataclass
class DeltaToolCall:
    """Incremental change to a tool call.

    Used to describe a chunk when streaming structured responses.
    """

    name: str | None = None
    """Incremental change to the name of the tool."""
    json_args: str | None = None
    """Incremental change to the arguments as JSON"""
```

name `class-attribute` `instance-attribute`

```python
name: str | None = None
```

Incremental change to the name of the tool.

json_args `class-attribute` `instance-attribute`

```python
json_args: str | None = None
```

Incremental change to the arguments as JSON

DeltaToolCalls `module-attribute`

```python
DeltaToolCalls: TypeAlias = dict[int, DeltaToolCall]
```

A mapping of tool call IDs to incremental changes.

FunctionDef `module-attribute`

```
FunctionDef: TypeAlias = Callable[
    [list[Message], AgentInfo],
    Union[ModelAnyResponse, Awaitable[ModelAnyResponse]],
]
```

A function used to generate a non-streamed response.

StreamFunctionDef `module-attribute`

```
StreamFunctionDef: TypeAlias = Callable[
    [list[Message], AgentInfo],
    AsyncIterator[Union[str, DeltaToolCalls]],
]
```

A function used to generate a streamed response.

While this is defined as having return type of `AsyncIterator[Union[str, DeltaToolCalls]]`, it should really be considered as `Union[AsyncIterator[str], AsyncIterator[DeltaToolCalls]`,

E.g. you need to yield all text or all `DeltaToolCalls`, not mix them.

FunctionAgentModel `dataclass`

Bases: `AgentModel`

Implementation of `AgentModel` for FunctionModel.

Source code in `pydantic_ai_slim/pydantic_ai/models/function.py`

```python
@dataclass
class FunctionAgentModel(AgentModel):
    """Implementation of `AgentModel` for [FunctionModel][pydantic_ai.models.function.FunctionModel]."""

    function: FunctionDef | None
    stream_function: StreamFunctionDef | None
    agent_info: AgentInfo

    async def request(self, messages: list[Message]) -> tuple[ModelAnyResponse, result.Cost]:
        assert self.function is not None, 'FunctionModel must receive a `function` to support non-streamed requests'
        if inspect.iscoroutinefunction(self.function):
            response = await self.function(messages, self.agent_info)
        else:
            response_ = await _utils.run_in_executor(self.function, messages, self.agent_info)
            response = cast(ModelAnyResponse, response_)
        # TODO is `messages` right here? Should it just be new messages?
        return response, _estimate_cost(chain(messages, [response]))

    @asynccontextmanager
    async def request_stream(self, messages: list[Message]) -> AsyncIterator[EitherStreamedResponse]:
        assert (
            self.stream_function is not None
        ), 'FunctionModel must receive a `stream_function` to support streamed requests'
        response_stream = self.stream_function(messages, self.agent_info)
        try:
            first = await response_stream.__anext__()
        except StopAsyncIteration as e:
            raise ValueError('Stream function must return at least one item') from e

        if isinstance(first, str):
            text_stream = cast(AsyncIterator[str], response_stream)
            yield FunctionStreamTextResponse(first, text_stream)
        else:
            structured_stream = cast(AsyncIterator[DeltaToolCalls], response_stream)
            yield FunctionStreamStructuredResponse(first, structured_stream)
```

FunctionStreamTextResponse `dataclass`

Bases: `StreamTextResponse`

Implementation of `StreamTextResponse` for FunctionModel.

Source code in `pydantic_ai_slim/pydantic_ai/models/function.py`

```python
@dataclass
class FunctionStreamTextResponse(StreamTextResponse):
    """Implementation of `StreamTextResponse` for [FunctionModel][pydantic_ai.models.function.FunctionModel]."""

    _next: str | None
    _iter: AsyncIterator[str]
    _timestamp: datetime = field(default_factory=_utils.now_utc, init=False)
    _buffer: list[str] = field(default_factory=list, init=False)

    async def __anext__(self) -> None:
        if self._next is not None:
            self._buffer.append(self._next)
            self._next = None
        else:
            self._buffer.append(await self._iter.__anext__())

    def get(self, *, final: bool = False) -> Iterable[str]:
        yield from self._buffer
        self._buffer.clear()

    def cost(self) -> result.Cost:
        return result.Cost()

    def timestamp(self) -> datetime:
        return self._timestamp
```

FunctionStreamStructuredResponse `dataclass`

Bases: `StreamStructuredResponse`

Implementation of `StreamStructuredResponse` for FunctionModel.

Source code in `pydantic_ai_slim/pydantic_ai/models/function.py`

```python
@dataclass
class FunctionStreamStructuredResponse(StreamStructuredResponse):
    """Implementation of `StreamStructuredResponse` for [FunctionModel][pydantic_ai.models.function.FunctionModel]."""

    _next: DeltaToolCalls | None
    _iter: AsyncIterator[DeltaToolCalls]
    _delta_tool_calls: dict[int, DeltaToolCall] = field(default_factory=dict)
    _timestamp: datetime = field(default_factory=_utils.now_utc)

    async def __anext__(self) -> None:
        if self._next is not None:
            tool_call = self._next
            self._next = None
        else:
            tool_call = await self._iter.__anext__()

        for key, new in tool_call.items():
            if current := self._delta_tool_calls.get(key):
                current.name = _utils.add_optional(current.name, new.name)
                current.json_args = _utils.add_optional(current.json_args, new.json_args)
            else:
                self._delta_tool_calls[key] = new

    def get(self, *, final: bool = False) -> ModelStructuredResponse:
        calls: list[ToolCall] = []
        for c in self._delta_tool_calls.values():
            if c.name is not None and c.json_args is not None:
                calls.append(ToolCall.from_json(c.name, c.json_args))

        return ModelStructuredResponse(calls, timestamp=self._timestamp)

    def cost(self) -> result.Cost:
        return result.Cost()

    def timestamp(self) -> datetime:
        return self._timestamp
```