Results

Results are the final values returned from running an agent. The result values are wrapped in `RunResult` and `StreamedRunResult` so you can access other data like cost of the run and message history

Both `RunResult` and `StreamedRunResult` are generic in the data they wrap, so typing information about the data returned by the agent is preserved.

olympics.py

```python
from pydantic import BaseModel

from pydantic_ai import Agent


class CityLocation(BaseModel):
    city: str
    country: str


agent = Agent('gemini-1.5-flash', result_type=CityLocation)
result = agent.run_sync('Where the olympics held in 2012?')
print(result.data)
#> city='London' country='United Kingdom'
print(result.cost())
#> Cost(request_tokens=56, response_tokens=8, total_tokens=64, details=None)
```

(This example is complete, it can be run "as is")

Runs end when either a plain text response is received or the model calls a tool associated with one of the structured result types. We will add limits to make sure a run doesn't go on indefinitely, see #70.

Result data

When the result type is `str`, or a union including `str`, plain text responses are enabled on the model, and the raw text response from the model is used as the response data.

If the result type is a union with multiple members (after remove `str` from the members), each member is registered as a separate tool with the model in order to reduce the complexity of the tool schemas and maximise the changes a model will respond correctly.

If the result type schema is not of type `"object"`, the result type is wrapped in a single element object, so the schema of all tools registered with the model are object schemas.

Structured results (like tools) use Pydantic to build the JSON schema used for the tool, and to validate the data returned by the model.

Bring on PEP-747

Until PEP-747 "Annotating Type Forms" lands, unions are not valid as `type` s in Python.

When creating the agent we need to `# type: ignore` the `result_type` argument, and add a type hint to tell type checkers about the type of the agent.

Here's an example of returning either text or a structured value

box_or_error.py

```python
from typing import Union

from pydantic import BaseModel

from pydantic_ai import Agent


class Box(BaseModel):
    width: int
    height: int
    depth: int
    units: str


agent: Agent[None, Union[Box, str]] = Agent(
    'openai:gpt-4o-mini',
    result_type=Union[Box, str],  # type: ignore
    system_prompt=(
        "Extract me the dimensions of a box, "
        "if you can't extract all data, ask the user to try again."
    ),
)

result = agent.run_sync('The box is 10x20x30')
print(result.data)
#> Please provide the units for the dimensions (e.g., cm, in, m).

result = agent.run_sync('The box is 10x20x30 cm')
print(result.data)
#> width=10 height=20 depth=30 units='cm'
```

(This example is complete, it can be run "as is")

Here's an example of using a union return type which registered multiple tools, and wraps non-object schemas in an object:

colors_or_sizes.py

```python
from typing import Union

from pydantic_ai import Agent

agent: Agent[None, Union[list[str], list[int]]] = Agent(
    'openai:gpt-4o-mini',
    result_type=Union[list[str], list[int]],  # type: ignore
    system_prompt='Extract either colors or sizes from the shapes provided.',
)

result = agent.run_sync('red square, blue circle, green triangle')
print(result.data)
#> ['red', 'blue', 'green']

result = agent.run_sync('square size 10, circle size 20, triangle size 30')
```

```
print(result.data)
#> [10, 20, 30]
```

(This example is complete, it can be run "as is")

Result validators functions

Some validation is inconvenient or impossible to do in Pydantic validators, in particular when the validation requires IO and is asynchronous. PydanticAI provides a way to add validation functions via the `agent.result_validator` decorator.

Here's a simplified variant of the SQL Generation example:

sql_gen.py

```
from typing import Union

from fake_database import DatabaseConn, QueryError
from pydantic import BaseModel

from pydantic_ai import Agent, RunContext, ModelRetry


class Success(BaseModel):
    sql_query: str


class InvalidRequest(BaseModel):
    error_message: str


Response = Union[Success, InvalidRequest]
agent: Agent[DatabaseConn, Response] = Agent(
    'gemini-1.5-flash',
    result_type=Response,  # type: ignore
    deps_type=DatabaseConn,
    system_prompt='Generate PostgreSQL flavored SQL queries based on user input.',
)


@agent.result_validator
async def validate_result(ctx: RunContext[DatabaseConn], result: Response) -> Response:
    if isinstance(result, InvalidRequest):
        return result
    try:
        await ctx.deps.execute(f'EXPLAIN {result.sql_query}')
    except QueryError as e:
        raise ModelRetry(f'Invalid query: {e}') from e
    else:
        return result


result = agent.run_sync(
    'get me uses who were last active yesterday.', deps=DatabaseConn()
)
print(result.data)
#> sql_query='SELECT * FROM users WHERE last_active::date = today() - interval 1 day'
```

(This example is complete, it can be run "as is")

Streamed Results

There two main challenges with streamed results:

1. Validating structured responses before they're complete, this is achieved by "partial validation" which was recently added to Pydantic in pydantic/pydantic#10748.
2. When receiving a response, we don't know if it's the final response without starting to stream it and peeking at the content. PydanticAI streams just enough of the response to sniff out if it's a tool call or a result, then streams the whole thing and calls tools, or returns the stream as a `StreamedRunResult`.

Streaming Text

Example of streamed text result:

streamed_hello_world.py

```
from pydantic_ai import Agent

agent = Agent('gemini-1.5-flash')  # 1


async def main():
    async with agent.run_stream('Where does "hello world" come from?') as result:  # 2
        async for message in result.stream():  # 3
            print(message)
            #> The first known
            #> The first known use of "hello,
            #> The first known use of "hello, world" was in
            #> The first known use of "hello, world" was in a 1974 textbook
            #> The first known use of "hello, world" was in a 1974 textbook about the C
            #> The first known use of "hello, world" was in a 1974 textbook about the C programming language.
```

1. Streaming works with the standard `Agent` class, and doesn't require any special setup, just a model that supports streaming (currently all models support streaming).
2. The `Agent.run_stream()` method is used to start a streamed run, this method returns a context manager so the connection can be closed when the stream completes.
3. Each item yield by `StreamedRunResult.stream()` is the complete text response, extended as new data is received.

(This example is complete, it can be run "as is")

We can also stream text as deltas rather than the entire text in each item:

streamed_delta_hello_world.py

```
from pydantic_ai import Agent

agent = Agent('gemini-1.5-flash')


async def main():
    async with agent.run_stream('Where does "hello world" come from?') as result:
        async for message in result.stream_text(delta=True):  # 1
            print(message)
            #> The first known
            #> use of "hello,
```

```
            #> world" was in
            #> a 1974 textbook
            #> about the C
            #> programming language.
```

1. `stream_text` will error if the response is not text

(This example is complete, it can be run "as is")

> **Result message not included in `messages`**
>
> The final result message will **NOT** be added to result messages if you use `.stream_text(delta=True)`, see Messages and chat history for more information.

Streaming Structured Responses

Not all types are supported with partial validation in Pydantic, see pydantic/pydantic#10748, generally for model-like structures it's currently best to use `TypeDict`.

Here's an example of streaming a use profile as it's built:

streamed_user_profile.py

```python
from datetime import date

from typing_extensions import TypedDict

from pydantic_ai import Agent


class UserProfile(TypedDict, total=False):
    name: str
    dob: date
    bio: str


agent = Agent(
    'openai:gpt-4o',
    result_type=UserProfile,
    system_prompt='Extract a user profile from the input',
)


async def main():
    user_input = 'My name is Ben, I was born on January 28th 1990, I like the chain the dog and the pyramid.'
    async with agent.run_stream(user_input) as result:
        async for profile in result.stream():
            print(profile)
            #> {'name': 'Ben'}
            #> {'name': 'Ben'}
            #> {'name': 'Ben', 'dob': date(1990, 1, 28), 'bio': 'Likes'}
            #> {'name': 'Ben', 'dob': date(1990, 1, 28), 'bio': 'Likes the chain the '}
            #> {'name': 'Ben', 'dob': date(1990, 1, 28), 'bio': 'Likes the chain the dog and the pyr'}
            #> {'name': 'Ben', 'dob': date(1990, 1, 28), 'bio': 'Likes the chain the dog and the pyramid'}
            #> {'name': 'Ben', 'dob': date(1990, 1, 28), 'bio': 'Likes the chain the dog and the pyramid'}
```

(This example is complete, it can be run "as is")

If you want fine-grained control of validation, particularly catching validation errors, you can use the following pattern:

streamed_user_profile.py

```python
from datetime import date

from pydantic import ValidationError
from typing_extensions import TypedDict

from pydantic_ai import Agent


class UserProfile(TypedDict, total=False):
    name: str
    dob: date
    bio: str


agent = Agent('openai:gpt-4o', result_type=UserProfile)


async def main():
    user_input = 'My name is Ben, I was born on January 28th 1990, I like the chain the dog and the pyramid.'
    async with agent.run_stream(user_input) as result:
        async for message, last in result.stream_structured(debounce_by=0.01):  # (1)
            try:
                profile = await result.validate_structured_result(  # (2)
                    message,
                    allow_partial=not last,
                )
            except ValidationError:
                continue
            print(profile)
            #> {'name': 'Ben'}
            #> {'name': 'Ben'}
            #> {'name': 'Ben', 'dob': date(1990, 1, 28), 'bio': 'Likes'}
            #> {'name': 'Ben', 'dob': date(1990, 1, 28), 'bio': 'Likes the chain the '}
            #> {'name': 'Ben', 'dob': date(1990, 1, 28), 'bio': 'Likes the chain the dog and the pyr'}
            #> {'name': 'Ben', 'dob': date(1990, 1, 28), 'bio': 'Likes the chain the dog and the pyramid'}
            #> {'name': 'Ben', 'dob': date(1990, 1, 28), 'bio': 'Likes the chain the dog and the pyramid'}
```

1. `stream_structured` streams the data as `ModelStructuredResponse` objects, thus iteration can't fail with a `ValidationError`.
2. `validate_structured_result` validates the data, `allow_partial=True` enables pydantic's `experimental_allow_partial` flag on `TypeAdapter`.

(This example is complete, it can be run "as is")

Examples

The following examples demonstrate how to use streamed responses in PydanticAI:

- Stream markdown
- Stream Whales