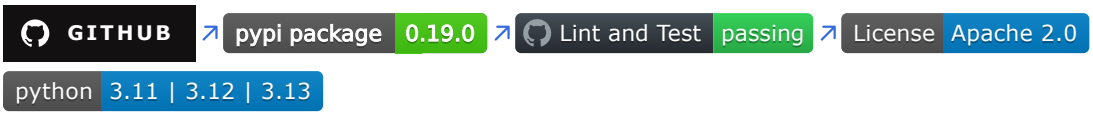
Aviary

aviary



Gymnasium framework for training language model agents on constructive tasks.

Installation

To install aviary (note `fh` stands for FutureHouse):

```
pip install fhaviary
```

To install aviary with the bundled environments,
please see the Environments section below.

Google Colab

As of 10/25/2024, unfortunately Google Colab does not yet support Python 3.11 or 3.12 (issue ↗).

Thus, as a workaround, you will need to install Python 3.11 into your notebook.
Here is a sample notebook showing how to do this:
https://colab.research.google.com/drive/1mejZ5cxgKZrMpYEe0iRoanaGGQ0Cr6WI?usp=sharing

Also, note that `async` code works in Google Colab.

Developer Installation

For local development, please see the CONTRIBUTING.md.

Messages

Communication between the agent and environment is done through messages.
Messages have two attributes:

```
msg = Message(content="Hello, world!", role="assistant")
```

The `content` is a string with a text, a JSON serializable list of `dict`s, or a null value.
A list of dicts is used to encode multi-modal content. The method `create_message` can be used to create a message with images:

```
from PIL import Image
import numpy as np

img = Image.open("your_image.jpg")
img_array = np.array(img)

msg = Message.create_message(role="user", text="Hello, world!", images=
[img_array])
```

`create_message` supports images as numpy array or base64 encoded images. In this case, `content` will be a list of dictionaries with the keys `text` and `image_url`.

```
{
    {"type": "text", "text": "Hello World!"},
    {"text": "image_url", "image_url": "
{base64_image}"},
}
```

We follow the structure adopted by OpenAI ↗.

For the meaning of role, see the table below.
You can change around roles as desired,
except for `tool` which has a special meaning in aviary.

Role	Host	Example(s)
assistant	Agent	A tool selector agent's tool selection message
system	Agent system prompt	"You are an agent."
user	Environment system prompt or emitted observation	HotPotQA problem to solve, or details of an internal env failure
tool	Result of tool run in the environment	Some number crunching program's output

`Message` is extended in `ToolRequestMessage` and `ToolResponseMessage` to include the relevant tool name and arguments.

Environment

An environment should have two functions:

```
obs_msgs, tools = await env.reset()
new_obs_msgs, reward, done, truncated = await env.step(action_msg)
```

where messages are how communication is passed. The `action_msg` should be `ToolRequestMessage` which is 1 or more calls to tools provided by the `reset`. The `obs_msgs` returned from the environment are `ToolResponseMessage` or other general messages that are observations. The `reward` is a scalar value. The `done` is a boolean value. The `truncated` is a boolean value.

Functional Environments

The easiest way to create an environment is using the functional interface, which just uses functions and decorators to define environments. First, let's define what the environment looks like by defining its `start` function:

```
from aviary.core import fenv


@fenv.start()
def my_env(topic):
    # return first observation, and the starting environment state
    # (empty in this case)
    return f"Write a story about {topic}", {}
```

Note that the decorator is a call (`start()`). The `start` decorator starts the definition of an environment. The function, `my_env`, can take whatever you would like and should return a tuple containing the first observation and anything you would like to store about the state of the environment (used to persist/share things between tools). The state will

always automatically have an optional `reward` and a boolean `done` that indicates if the environment is complete.

Now we can define some tools:

```
@my_env.tool()
def multiply(x: float, y: float) -> float:
    """Multiply two numbers."""
    return x * y


@my_env.tool()
def print_story(story: str | bytes, state) -> None:
    """Print a story to user and complete task."""
    print(story)
    state.reward = 1
    state.done = True
```

The tools will be converted into things visible for LLMs using the type hints and the variable descriptions. Thus, the type hinting can be valuable for the agent using it correctly. The docstrings are also passed to the LLM, and is the primary way (along with function name) for communicating about intended tool usage.

You can access the `state` variable in tools, which will have any fields you passed in the return tuple of `start()`. For example, if you returned `{'foo': 'bar'}`, then you could access `state.foo` in the tools.

Stop an environment or set a reward via the `state` variable as shown the second tool. If the reward is not set, it is treated as zero.

Now we can use our environment:

```
env = my_env(topic="foo")
obs, tools = await env.reset()
```

Subclass Environments

If you need more control over Environments and tools, you'll want to subclass the `Environment`

First we define an environment by subclassing the `Environment` and defining a `state`. The `state` is all variables
that change per step and we want to keep together. It will be accessible in your tools, so you can use it to store
information that you want to persist between steps and between tools.

```
from pydantic import BaseModel
from aviary.core import Environment


class ExampleState(BaseModel):
    reward: float = 0
    done: bool = False


class ExampleEnv(Environment[ExampleState]):
    state: ExampleState
```

We do not have other variables aside from `state` for this environment. We could have things like configuration, a name,
tasks, etc. attached to it.

Common environments

We expose a simple interface to some commonly-used environments that are included in the aviary codebase. You can instantiate one by referring to its name and passing keyword arguments:

```
from aviary.core import Environment

env = Environment.from_name(
    "calculator",
    problem_id="example-problem",
    problem="What is 2+3?",
    answer=5,
)
```

Included with some environments are collections of problems that define training or evaluation datasets.
We refer to these as `TaskDataset`s, and expose them with a similar interface:

```
from aviary.core import TaskDataset

dataset = TaskDataset.from_name("hotpotqa", split="dev")
```

Tool

Now let's define our functions that will make up our tools. We'll just have one tool. Tools can optionally have their
last argument be `state` which is the environment state. This is how you can access the state. This argument will not be
exposed to the agent as a possible parameter and will be injected by the environment (if part of the function
signature).

```
def print_story(story: str, state: ExampleState):
    """Print a story.

    Args:
        story: Story to print.
        state: Environment state (hidden from agent - can put this
string to shutup linter).
    """
    print(story)
    state.reward = 1
    state.done = True
```

There is special syntax we use for defining a tool. The tool is built from the following parts of the function: its
name, its arguments names, the arguments types, and the docstring. The docstring is parsed to get a description of the
function and its arguments, so match the syntax carefully.

Setting the `state.done = True` is how we indicate completion. This example terminates immediately. You can use other

ways to decide to terminate.

You can make the function `async` - the environment will account for that when the tool is called.

Advanced tool descriptions

We support more sophisticated signatures, for those who want to use them:

- Multiline docstrings
- Non-primitive type hints (e.g. type unions)
- Default values
- Exclusion of info below `\f` (see below)

If you have summary-level information that belongs in the docstring,
but you don't want it part of the `Tool.info.description`,
add a `r` prefix to the docstring
and inject `\f` before the summary information to exclude.
This convention was created by FastAPI (docs ↗).

```
def print_story(story: str | bytes, state: ExampleState):
    r"""Print a story.

    Extra information that is part of the tool description.

    \f

    This sentence is excluded because it's an implementation detail.

    Args:
        story: Story to print, either as a string or bytes.
        state: Environment state.
    """
    print(story)
    state.reward = 1
    state.done = True
```

Environment `reset` method

Now we'll define the `reset` function which should set-up the tools,
and return one or more initial observations and the tools.
The `reset` function is `async` to allow for database interactions or HTTP requests.

```
from aviary.core import Message, Tool


async def reset(self):
    self.tools = [Tool.from_function(ExampleEnv.print_story)]
    start = Message(content="Write a 5 word story and call print")
    return [start], self.tools
```

Environment `step` method

Now we can define the `step` function which should take an action and return the next observation, reward, done, and if
the episode was truncated.

```
from aviary.core import Message


async def step(self, action: Message):
    msgs = await self.exec_tool_calls(action, state=self.state)
    return msgs, self.state.reward, self.state.done, False
```

You will probably often use this specific syntax for calling the tools - calling `exec_tool_calls` with the action.

Environment `export_frame` method

Optionally, we can define a function to export a snapshot of the environment
and its state for visualization or debugging purposes.

```
from aviary.core import Frame


def export_frame(self):
    return Frame(
        state={"done": self.state.done, "reward": self.state.reward},
        info={"tool_names": [t.info.name for t in self.tools]},
    )
```

View Environment Tools

If an environment can be instantiated without anything other than a task (i.e., it implements `from_task`), you can start a server to view its tools:

```
pip install fhaviary[server]
aviary tools [env name]
```

This will start a server that allows you to view the tools and call them, viewing the descriptions/types and output that an agent would see when using the tools.

Environments

Here are a few environments implemented with aviary:

Environment	PyPI	Extra	README
GSM8k	`aviary.gsm8k` ↗	`fhaviary[gsm8k]`	`README.md`
HotPotQA	`aviary.hotpotqa` ↗	`fhaviary[hotpotqa]`	`README.md`
LitQA	`aviary.litqa` ↗	`fhaviary[litqa]`	`README.md`
LFRQA	`aviary.lfrqa` ↗	`fhaviary[lfrqa]`	`README.md`

Contributing to aviary

Repo Structure

aviary is a monorepo using `uv`'s workspace layout ↗.

Installation

1. Git clone this repo
2. Install the project manager `uv`: https://docs.astral.sh/uv/getting-started/installation/
3. Run `uv sync`

This will editably install the full monorepo in your local environment.

Testing

To run tests, please just run `pytest` in the repo root.

Note you will need OpenAI and Anthropic API keys configured.

packages

aviary.gsm8k

GSM8k environment implemented with aviary, allowing agents to solve math word problems from the GSM8k dataset.

Citation

The citation for GSM8k is given below:

```
@article{gsm8k-paper,
    title = {Training verifiers to solve math word problems},
    author = {Cobbe, Karl and Kosaraju, Vineet and Bavarian, Mohammad
and Chen, Mark and Jun, Heewoo and Kaiser, Lukasz and
 Plappert, Matthias and Tworek, Jerry and Hilton, Jacob and Nakano,
Reiichiro and others},
    journal = {arXiv preprint arXiv:2110.14168},
    year = {2021}
}
```

References

[1] Cobbe, K., Kosaraju, V., Bavarian, M., Chen, M., Jun, H., Kaiser, L., Plappert, M., Tworek, J., Hilton, J., Nakano, R. and Hesse, C., 2021. Training verifiers to solve math word problems ↗. arXiv preprint arXiv:2110.14168.

Installation

To install the GSM8k environment, run the following command:

```
pip install fhaviary[gsm8k]
```

aviary.hotpotqa

HotPotQA environment implemented with aviary, allowing agents to perform multi-hop question answering on the HotPotQA dataset.

References

[1] Yang et al. HotpotQA: A Dataset for Diverse, Explainable Multi-Hop Question Answering ↗. EMNLP, 2018.

[2] Yao et al., ReAct: Synergizing Reasoning and Acting in Language Models ↗. In The Eleventh International Conference on Learning Representations. 2023

Installation

To install the HotPotQA environment, run the following command:

```
pip install fhaviary[hotpotqa]
```

aviary.lfrqa

An environment designed to utilize PaperQA for answering questions from the LFRQATaskDataset. Long-form RobustQA (LFRQA) is a human-annotated dataset introduced in the RAG-QA-Arena, featuring over 1400 questions from various categories, including science.

Installation

To install the LFRQA environment, run:

```
pip install fhaviary[lfrqa]
```

Usage

Refer to this tutorial ↗ for instructions on how to run the environment.

References

[1] RAG-QA Arena (https://arxiv.org/pdf/2407.13998)

aviary.litqa

LitQA2 environment implemented with aviary, allowing agents to perform question answering on the LitQA dataset.

LitQA ↗ (now legacy) is a dataset composed from 50 multiple-choice questions from recent literature. It is designed to test the LLM's the ability to retrieve information outside of the pre-training corpus. To ensure the questions are not in the pre-training corpus, the questions were collected from scientific papers published after September 2021 -- cut-off date of GPT-4's training data.

LitQA2 is part of the LAB-Bench dataset ↗. LitQA2 contains 248 multiple-choice questions from the literature and was created ensuring that the questions cannot be answered by recalling from the pre-training corpus only. It considered scientific paper published within 36 months from the data of its publication. Therefore, LitQA2 is considered a scientific RAG dataset.

Installation

To install the LitQA environment, run:

```
pip install fhaviary[litqa]
```

Usage

In `litqa/env.py` ↗, you will find:

`GradablePaperQAEnvironment`: an environment that can grade answers given an evaluation function.

And in `litqa/task.py` ↗, you will find:

`LitQAv2TaskDataset`: a task dataset designed to pull LitQA v2 from Hugging Face, and create one `GradablePaperQAEnvironment` per question

Here is an example of how to use them:

```
import os

from ldp.agent import SimpleAgent
from ldp.alg import Evaluator, EvaluatorConfig, MeanMetricsCallback
from paperqa import Settings

from aviary.env import TaskDataset
from aviary.envs.litqa.task import TASK_DATASET_NAME


async def evaluate(folder_of_litqa_v2_papers: str | os.PathLike) ->
None:
    settings = Settings(paper_directory=folder_of_litqa_v2_papers)
    dataset = TaskDataset.from_name(TASK_DATASET_NAME,
settings=settings)
    metrics_callback = MeanMetricsCallback(eval_dataset=dataset)

    evaluator = Evaluator(
        config=EvaluatorConfig(batch_size=3),
        agent=SimpleAgent(),
        dataset=dataset,
        callbacks=[metrics_callback],
    )
    await evaluator.evaluate()

    print(metrics_callback.eval_means)
```

References

[1] Lála et al. PaperQA: Retrieval-Augmented Generative Agent for Scientific Research ↗. ArXiv:2312.07559, 2023.

[2] Skarlinski et al. Language agents achieve superhuman synthesis of scientific knowledge ↗. ArXiv:2409.13740, 2024.

[3] Laurent et al. LAB-Bench: Measuring Capabilities of Language Models for Biology Research ↗. ArXiv:2407.10362, 2024.