pyfra

Release 0.1.4

Leo Gao

Nov 03, 2022

CONTENTS

CHAPTER

ONE

PYFRA.REMOTE MODULE

class pyfra.remote.**Env**(*ip=None*, *envname=None*, *git=None*, *branch=None*, *force_rerun=False*, *python_version='3.9.4'*, *additional_ssh_config=''*)

Bases: *pyfra.remote.Remote*

An environment is a Remote pointing to a directory that has a virtualenv and a specific version version of python installed, optionally initialized from a git repo. Since environments are also just Remotes, all methods on Remotes work on environments too.

A typical design pattern sees functions accepting remotes as argument and immediately turning it into an env that's used for the rest of the function. Alternatively, functions can take in already-created envs and perform some task inside the env.

See *pyfra.remote.Remote* for more information about methods. Envs can be created from an existing Remote using *pyfra.remote.Remote.env()*.

Example usage:

```
def train_model(rem, ...):
    e = rem.env("neo_experiment", "https://github.com/EleutherAI/gpt-neo", python_
↪version="3.8.10")
    e.sh("do something")
    e.sh("do something else")
    f = some_other_thing(e, ...)
    e.path("goose.txt").write(f.jread()["honk"])

def some_other_thing(env, ...):
    env.sh("do something")
    env.sh("do something else")

    return env.path("output.json")
```

Parameters

- **ip** (*str*) – The host to ssh to. This looks something like `12.34.56.78` or `goose.com` or `someuser@12.34.56.78` or `someuser@goose.com`. You must enable passwordless ssh and have your ssh key added to the server first. If None, the Remote represents localhost.
- **git** (*str*) – The git repo to clone into the fresh env. If None, no git repo is cloned.
- **branch** (*str*) – The git branch to clone. If None, the default branch is used.
- **force_rerun** (*bool*) – If True, all hashing will be disabled and everything will be run every time. Deprecated in favor of *with pyfra.always_rerun()*
- **python_version** (*str*) – The python version to use.

sh(*x*, *quiet=False*, *wrap=True*, *maxbuflen=1000000000*, *ignore_errors=False*, *no_venv=False*, *pyenv_version=<object object>*, *forward_keys=False*)
Run a series of bash commands on this remote. This command shares the same arguments as *pyfra.shell.sh()*. :meta private:

class pyfra.remote.**Remote**(*ip=None*, *wd=None*, *experiment=None*, *resumable=False*, *additional_ssh_config=''*)

Bases: object

Parameters

- **ip** (*str*) – The host to ssh to. This looks something like `12.34.56.78` or `goose.com` or `someuser@12.34.56.78` or `someuser@goose.com`. You must enable passwordless ssh and have your ssh key added to the server first. If None, the Remote represents localhost.
- **wd** (*str*) – The working directory on the server to start out on.
- **python_version** (*str*) – The version of python to use (i.e running `Remote("goose.com", python_version="3.8.10").sh("python --version")` will use python 3.8.10). If this version is not already installed, pyfra will install it.
- **resumable** (*bool*) – If True, this Remote will resume where it left off, with the same semantics as Env.

env(*envname*, *git=None*, *branch=None*, *force_rerun=False*, *python_version='3.9.4'*) → *pyfra.remote.Remote*
Arguments are the same as the `pyfra.experiment.Experiment` constructor.

fingerprint() → str
A unique string for the server that this Remote is pointing to. Useful for detecting if the server has been yanked under you, or if this different ip actually points to the same server, etc.

glob(*pattern: str*) → List[*pyfra.remote.RemotePath*]
Find all files matching the glob pattern.

home() → str
The home directory on the remote.

is_local()
Returns true if this is a local remote/environment.

ls(*x='.'*) → List[str]
Lists files, sorted by natsort.

Parameters **x** (*str*) – The directory to list. Defaults to current directory.

no_hash()
Context manager to turn off hashing temporarily. Example usage:

```
print(env.hash)
with env.no_hash():
    env.sh("echo do something")
print(env.hash) # will be the same as before
```

path(*fname=None*) → *pyfra.remote.RemotePath*
This is the main way to make a *RemotePath* object; see RemotePath docs for more info on what they're used for.

If fname is not specified, this command allocates a temporary path

rm(*x*, *no_exists_ok=True*)
Remove a file or directory.

sh(*x*, *quiet=False*, *wrap=True*, *maxbuflen=1000000000*, *ignore_errors=False*, *no_venv=False*, *pyenv_version=None*, *forward_keys=False*)
Run a series of bash commands on this remote. This command shares the same arguments as *pyfra.shell.sh()*.

class pyfra.remote.**RemotePath**(*remote*, *fname*)
Bases: `object`

A RemotePath represents a path somewhere on some Remote. The RemotePath object can be used to manipulate the file.

Example usage:

```
# write text
rem.path("goose.txt").write("honk")

# read text
print(rem.path("goose.txt").read())

# write json
rem.path("goose.json").jwrite({"honk": 1})

# read json
print(rem.path("goose.json").jread())

# write csv
rem.path("goose.csv").csvwrite([{"col1": 1, "col2": "duck"}, {"col1": 2, "col2":
↪"goose"}])

# read csv
print(rem.path("goose.csv").csvread())

# copy stuff to/from remotes
copy(rem1.path('goose.txt'), 'test1.txt')
copy('test1.txt', rem2.path('goose.txt'))
copy(rem2.path('goose.txt'), rem1.path('testing123.txt'))
```

csvread(*colnames=None*) → List[dict]
Read the contents of this csv file and parses it into an array of dictionaries where the keys are column names.

Parameters **colnames** (*list*) – Optionally specify the names of the columns for csvs without a header row.

csvwrite(*data*, *colnames=None*)
Write a list of dicts object to this csv file.

Parameters **content** (*List[dict]*) – A list of dicts where the keys are column names. Every dicts should have the exact same keys.

exists() → bool
Check if this file exists

expanduser() → *pyfra.remote.RemotePath*
Return a copy of this path with the home directory expanded.

glob(*pattern: str*) → List[*pyfra.remote.RemotePath*]
Find all files matching the glob pattern.

is_dir() → bool
Check if this file exists

jread() → Dict[str, Any]
Read the contents of this json file and parses it. Equivalent to `json.loads(self.read())`

jwrite(*content*)
Write a json object to this file. Equivalent to `self.write(json.dumps(content))`

Parameters **content** (*json*) – The json object to write

quick_hash() → str
Get a hash of this file that catches file changes most of the time by hashing blocks from the file at th beginning, middle, and end. Really useful for getting a quick hash of a really big file, but obviously unsuitable for guaranteeing file integrity.

Uses imohash under the hood.

read() → str
Read the contents of this file into a string

rsyncstr() → str

sh(*cmd*, **args*, ***kwargs*)

sha256sum() → str
Return the sha256sum of this file.

stat() → os.stat_result
Stat a remote file

unlink() → None
Delete this file

write(*content*, *append=False*) → str
Write text to this file.

Parameters

- **content** (*str*) – The text to write
- **append** (*bool*) – Whether to append or overwrite the file contents

pyfra.remote.**always_rerun**()
Use as a context manager to force all Envs to ignore cached results. Also forces stages to run.

pyfra.remote.**stage**(*fn*)
This decorator is used to mark a function as a "stage".

The purpose of this stage abstraction is for cases where you have some collection of operations that accomplish some goal and the way this goal is accomplished is intended to be abstracted away. Some examples would be tokenization, model training, or evaluation. After a stage runs once, the return value will be cached and subsequent calls with the same arguments will return the cached value.

However, there are several subtleties to the usage of stages. First, you might be wondering why we need this if Env already resumes where it left off. The main reason behind this is that since the way a stage accomplises its goal is meant to be abstracted away, it is possible that the stage will have changed in implementation, thus invalidating the hash (for example, the stage is switched to use a more efficient tokenizer that outputs the same thing). In these cases, just using Env hashing would rerun everything even when we know we don't need to. Also, any other expensive operations that are not Env operations will still run every time. Finally, this decorator correctly handles setting all the env hashes to what they should be after the stage runs, whereas using some other generic function caching would not.

Example usage:

```
@stage
def train_model(rem, ...):
    e = rem.env("neo_experiment", "https://github.com/EleutherAI/gpt-neo", python_
↪version="3.8.10")
    e.sh("do something")
    e.sh("do something else")
    f = some_other_thing(e, ...)
    return e.path("checkpoint")

train_model(rem)
```

Deprecated since version Will: be replaced by pyfra.idempotent eventually

CHAPTER

TWO

PYFRA.SHELL MODULE

exception pyfra.shell.**ShellException**(*code*, *rem=False*)

Bases: Exception

pyfra.shell.**copy**(*frm*, *to*, *quiet=False*, *connection_timeout=10*, *symlink_ok=True*, *into=True*, *exclude=[]*) → None

Copies things from one place to another.

Parameters

- **frm** (*str or* RemotePath) – Can be a string indicating a local path, a *pyfra.remote.RemotePath*, or a URL.
- **to** (*str or* RemotePath) – Can be a string indicating a local path or a *pyfra.remote.RemotePath*.
- **quiet** (*bool*) – Disables logging.
- **connection_timeout** (*int*) – How long in seconds to give up after
- **symlink_ok** (*bool*) – If frm and to are on the same machine, symlinks will be created instead of actually copying. Set to false to force copying.
- **into** (*bool*) – If frm is a file, this has no effect. If frm is a directory, then into=True for frm="src" and to="dst" means "src/a" will get copied to "dst/src/a", whereas into=False means "src/a" will get copied to "dst/a".

pyfra.shell.**curl**(*url*, *max_tries=10*, *timeout=30*)

pyfra.shell.**ls**(*x='.'*)

pyfra.shell.**sh**(*cmd*, *quiet=False*, *wd=None*, *wrap=True*, *maxbuflen=1000000000*, *ignore_errors=False*, *no_venv=False*, *pyenv_version=None*)

Runs commands as if it were in a local bash terminal.

This function patches out the non-interactive detection in bashrc and sources it, activates virtualenvs and sets pyenv shell, handles interrupts correctly, and returns the text printed to stdout.

Parameters

- **quiet** (*bool*) – If turned on, nothing is printed to stdout.
- **wd** (*str*) – Working directory to run in. Defaults to ~
- **wrap** (*bool*) – Magic for the bashrc, virtualenv, interrupt-handing, and pyenv stuff. Turn off to make this essentially os.system
- **maxbuflen** (*int*) – Max number of bytes to save and return. Useful to prevent memory errors.
- **ignore_errors** (*bool*) – If set, errors will be swallowed.

- **no_venv** (*bool*) – If set, virtualenv will not be activated
- **pyenv_version** (*str*) – Pyenv version to use. Will be silently ignored if not found.

Returns The standard output of the command, limited to maxbuflen bytes.

CHAPTER

THREE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

p

INDEX