fret Documentation

Release 0.3.5

yxonic

Apr 06, 2022

Contents

Welcome to `fret`'s documentation. `fret` stands for "Framework for Reproducible ExperimenTs". For detailed reference, see *API* section.

CHAPTER 1

User's Guide

1.1 Installation

From pip:

```
pip install fret
```

From source: clone the repository and then run: `python setup.py install`.

1.2 Tutorial

1.2.1 Basic Usage

Create a file named `app.py` with content:

```
import fret

@fret.command
def run(ws):
    model = ws.build()
    print(model)

@fret.configurable
class Model:
    def __init__(self, x=3, y=4):
        ...
```

Then under the same directory, you can run:

```
$ fret config Model
[ws/_default] configured "main" as "Model" with: x=3, y=4
```

(continues on next page)

(continued from previous page)

```
$ fret run
Model(x=3, y=4)
$ fret config Model -x 5 -y 10
[ws/_default] configured "main" as "Model" with: x=5, y=10
$ fret run
Model(x=5, y=10)
```

1.2.2 Using Workspace

You can specify different configuration in different workspace:

```
$ fret -w ws/model1 config Model
[ws/model1] configured "main" as "Model" with: x=3, y=4
$ fret -w ws/model2 config Model -x 5 -y 10
[ws/model2] configured "main" as "Model" with: x=5, y=10
$ fret -w ws/model1 run
Model(x=3, y=4)
$ fret -w ws/model2 run
Model(x=5, y=10)
```

1.2.3 Save/Load

```
import fret

@fret.command
def train(ws):
    model = ws.build()
    model.train()
    ws.save(model, 'trained')

@fret.command
def test(ws):
    model = ws.load('ws/best/snapshot/main.trained.pt')
    print(model.weight)

@fret.configurable(states=['weight'])
class Model:
    def __init__(self):
        self.weight = 0
    def train(self):
        self.weight = 23
```

```
$ fret -w ws/best config Model
[ws/_default] configured "main" as "Model"
$ fret -w ws/best train
$ fret test
23
```

1.2.4 An Advanced Workflow

In `app.py`:

```
import time
import fret

@fret.configurable(states=['value'])
class Model:
    def __init__(self):
        self.value = 0

@fret.command
def resumable(ws):
    model = ws.build()
    with ws.run('exp-1') as run:
        run.register(model)
        cnt = run.acc()
        for e in fret.nonbreak(run.range(5)):
            # with `nonbreak`, the program always finish this loop before exit
            model.value += e
            time.sleep(0.5)
            cnt += 1
            print('current epoch: %d, sum: %d, cnt: %d' %
                  (e, model.value, cnt))
```

Then you can stop and restart this program anytime, with consistent results:

```
$ fret resumable
current epoch: 0, sum: 0, cnt: 1
current epoch: 1, sum: 1, cnt: 2
^CW SIGINT received. Delaying KeyboardInterrupt.
current epoch: 2, sum: 3, cnt: 3
Traceback (most recent call last):
    ...
KeyboardInterrupt
W cancelled by user
$ fret resumable
current epoch: 3, sum: 6, cnt: 4
current epoch: 4, sum: 10, cnt: 5
```

1.2.5 Dynamic commands

You can specify commands inside configurables, and run them depending on current workspace setup:

```
import fret

@fret.configurable
class App1:
    @fret.command
    def check(self):
        print('running check from App1')

@fret.configurable
class App2:
    @fret.command
    def check(self, msg):
        print('running check from App2 with message: ' + msg)
```

Then run:

```
$ fret config App1
[ws/_default] configured "main" as "App1"
$ fret check
running check from App1
$ fret config App2
[ws/_default] configured "main" as "App2"
$ fret check -m hello
running check from App2 with message: hello
```

1.2.6 Submodule

```
@fret.configurable
class A:
    def __init__(self, foo):
        ...

@fret.configurable(submodules=['sub'], build_subs=False)
class B:
    def __init__(self, sub, bar=3):
        self.sub = sub(foo='bar')    # call sub to build submodule
```

```
$ fret config sub A
[ws/_default] configured "sub" as "A"
$ fret config B
[ws/_default] configured "main" as "B" with: sub='sub', bar=3
$ fret run
B(sub=A(), bar=3)
```

1.2.7 Inheritance

```
@fret.configurable
class A:
    def __init__(self, foo='bar', sth=3):
        ...

@fret.configurable
class B(A):
    def __init__(self, bar=3, **others):
        super().__init__(**others)
        ...
```

```
$ fret config B -foo baz -bar 0
[ws/_default] configured "main" as "B" with: bar=0, foo='baz', sth=3
$ fret run
B(bar=0, foo='baz', sth=3)
```

1.2.8 Internals

```
>>> config = fret.Configuration({'foo': 'bar'})
>>> config
foo='bar'
```

1.3 Programming API

1.3.1 Workspace

Simplist Case

```
# ws/test/config.toml
[main]
__module = "Model"
param = 1
```

```
# app.py
import fret

@fret.configurable
class Model:
    def __init__(self, param=0):
        self.param = param

ws = fret.workspace('ws/test')
model = ws.build()  # or equivalently, ws.build('main')
print(model.param)  # 1
```

You can call a funtion on a workspace, even if it is wrapped as a CLI command:

```
# app.py
import fret

@fret.configurable
class Model:
    def __init__(self, param=0):
        self.param = param

@fret.command
def check_model(ws):
    print(ws.build().param)

if __name__ == '__main__':
    ws = fret.workspace('ws/test')
    check_model(ws)
```

Then the following lines are equivalent:

```
$ python app.py
1
$ fret -w ws/test check_model
1
```

1.3.2 App

Package can be organized to form a fret app. If you want to have access to the app, just `import fret.app`:

```
# app.py
import fret
```

(continues on next page)

(continued from previous page)

```
@fret.configurable
class Model:
    def __init__(self, param=0):
        self.param = param

@fret.command
def check_model(ws):
    print(ws.build().param)
```

```
# main.py
import fret
import fret.app

if __name__ == '__main__':
    ws = fret.workspace('ws/test')
    fret.app.check_model(ws)
```

1.3.3 CLI

```
import fret.cli
if __name__ == '__main__':
    fret.cli.main()
```

1.3.4 Package Structure

```
fret/
  __init__.py
  __main__.py
  common.py      # common public APIs
  util.py        # util types and functions
  workspace.py   # workspace related
  app.py         # app related
  cli.py         # CLI related
```

CHAPTER 2

API Reference

2.1 API

2.1.1 Main Functionalities

fret.**workspace**(*path*, *config=None*, *config_dict=None*)
Workspace utilities. One can save/load configurations, build models with specific configuration, save snapshots, open results, etc., using workspace objects.

fret.**configurable**(*wraps=None*, *submodules=None*, *build_subs=True*, *states=None*)
Class decorator that registers configurable module under current app.

Parameters

- **wraps** (*class or None*) – object to be decorated; could be given later
- **submodules** (*list*) – submodules of this module
- **build_subs** (*bool*) – whether submodules are built before building this module (default: True)
- **states** (*list*) – members that would appear in state_dict

fret.**command**(*wraps=None*, *help=None*, *description=None*)
Function decorator that would turn a function into a fret command.

class fret.**argspec**(**args*, ***kwargs*)
In control of the behavior of commands. Replicates arguments for argparse.ArgumentParser.add_argument().

fret.**nonbreak**(*f=None*)
Make sure a loop is not interrupted in between an iteration.

fret.**stateful**(**states*)
Decorator for building stateful classes.

2.1.2 Common Components

exception fret.common.**DuplicationError**

exception fret.common.**NoAppError**

exception fret.common.**NoWorkspaceError**

exception fret.common.**NotConfiguredError**

class fret.common.**Module**(***kwargs*)
Interface for configurable modules.

Each module class should have an `add_arguments` class method to define model arguments along with their types, default values, etc.

Parameters **config** (*dict*) – module configuration

class fret.common.**Plugin**
Interface for external plugin

Pour new commands, or modify workspace object.

class fret.common.**argspec**(**args*, ***kwargs*)
In control of the behavior of commands. Replicates arguments for `argparse.ArgumentParser.add_argument()`.

class fret.common.**funcspec**(*f*)
Utility to generate argument specification from function signature.

fret.common.**command**(*wraps=None*, *help=None*, *description=None*)
Function decorator that would turn a function into a fret command.

fret.common.**configurable**(*wraps=None*, *submodules=None*, *build_subs=True*, *states=None*)
Class decorator that registers configurable module under current app.

Parameters

- **wraps** (*class or None*) – object to be decorated; could be given later
- **submodules** (*list*) – submodules of this module
- **build_subs** (*bool*) – whether submodules are built before building this module (default: `True`)
- **states** (*list*) – members that would appear in state_dict

class fret.workspace.**Workspace**(*path*, *config=None*, *config_dict=None*)
Workspace utilities. One can save/load configurations, build models with specific configuration, save snapshots, open results, etc., using workspace objects.

build(*name='main'*, ***kwargs*)
Build module according to the configurations in current workspace.

load(*name='main'*, *tag=None*, *path=None*)
Load module from a snapshot.

Parameters **tag** (*str or pathlib.Path*) – snapshot tag or path.

log(**filename*)
Get log file path within current workspace.

Parameters **filename** (*str or list*) – relative path to file; if ommited, returns root path of logs.

logger(*name: str*)
Get a logger that logs to a file under workspace.

Notice that same logger instance is returned for same names.

Parameters **name** (*str*) – logger name

register(*name*, *module*, ***kwargs*)
Register and save module configuration.

result(**filename*)
Get result file path within current workspace.

Parameters **filename** (*str or list*) – relative path to file; if ommited, returns root path of results.

run(*tag*, *resume=True*)
Initiate a context manager that provides a persistent running environment. Mainly used to suspend and resume a time consuming process.

save(*obj*, *tag*)
Save module as a snapshot.

Parameters **tag** (*str or pathlib.Path*) – snapshot tag or path.

snapshot(**filename*)
Get snapshot file path within current workspace.

Parameters **filename** (*str or list*) – relative path to file; if ommited, returns root path of snapshots.

write()
Save module configuration of this workspace to file.

config_path
Workspace configuration path.

path
Workspace root path.

class fret.workspace.**Run**(*ws*, *tag*, *resume*)
Class designed for running state persistency.

brange(**args*, *name=None*)
Breakable range. Works like normal range but with position recorded. Next time start from current position, as this loop isn't finished.

range(**args*, *name=None*)
Works like normal range but with position recorded. Next time start from next loop, as current loop is finished.

class fret.workspace.**Accumulator**
A stateful accumulator.

class fret.workspace.**Range**(**args*, *breakable=False*)
A stateful range object that mimics built-in `range`.

class fret.workspace.**Builder**(*ws*, *name*)
Class for building a specific module, with preset ws configuration.

2.1.3 Application Object and CLI

class fret.cli.**ParserBuilder**(*parser*, *style='java'*)
Utility to generate CLI arguments in different styles.

add_opt(*name*, *spec*)
Add option with specification.

Parameters

- **name** (*str*) – option name
- **spec** (argspec) – argument specification

class fret.cli.**_ArgumentParser**(*prog=None*, *usage=None*, *description=None*, *epilog=None*, *parents=[]*, *formatter_class=<class 'argparse.HelpFormatter'>*, *prefix_chars='-'*, *fromfile_prefix_chars=None*, *argument_default=None*, *conflict_handler='error'*, *add_help=True*, *allow_abbrev=True*)

error(*message: string*)
Prints a usage message incorporating the message to stderr and exits.

If you override this in a subclass, it should not return – it should either exit or raise an exception.

fret.cli.**clean**(*ws*, *config=(False*, *'remove workspace configuration')*, *log=(False*, *'clear workspace logs')*, *snapshot=(False*, *'clear snapshots')*, *everything=<fret.common.argspec object>*, *all=<fret.common.argspec object>*, *force=(False*, *'do without confirmation')*)
Command clean.

Remove all snapshots in specific workspace. If --all is specified, clean the entire workspace

fret.cli.**fork**(*ws*, *path*, *mods=[]*, *'modifications (in format: NAME.ARG=VAL)'*)
Command fork,

Fork from existing workspace and change some of the arguments.

Example

```
$ fret fork ws/test main.foo=6
In [ws/test]: as in [ws/_default], with modification(s): main.foo=6
```

2.1.4 Utilities

class fret.util.**ColoredFormatter**(*fmt=None*, *datefmt=None*, *style='%'*)
Formatter for colored log.

Initialize the formatter with specified format strings.

Initialize the formatter either with the specified format string, or a default as described above. Allow for specialized date formatting with the optional datefmt argument. If datefmt is omitted, you get an ISO8601-like (or RFC 3339-like) format.

Use a style parameter of '%', '{' or '$' to specify that you want to use one of %-formatting, str.format() ({}) formatting or string.Template formatting in your format string.

Changed in version 3.2: Added the style parameter.

format(*record*)
Format the specified record as text.

The record's attribute dictionary is used as the operand to a string formatting operation which yields the returned string. Before formatting the dictionary, a couple of preparatory steps are carried out. The message attribute of the record is computed using LogRecord.getMessage(). If the formatting string uses the time (as determined by a call to usesTime(), formatTime() is called to format the event time. If there is exception information, it is formatted using formatException() and appended to the message.

class fret.util.**Configuration**(**args*, ***kwargs*)
Easy to construct, use and read configuration class.

class fret.util.**Iterator**(*data*, **label*, *prefetch=False*, *length=None*, *batch_size=None*, *shuffle=True*, *full_shuffle=False*)
Iterator on data and labels, with states for save and restore.

class fret.util.**classproperty**(*f*)
Class property decorator.

fret.util.**colored**(*fmt*, *fg=None*, *bg=None*, *style=None*)
Return colored string.

List of colours (for fg and bg):

- k: black
- r: red
- g: green
- y: yellow
- b: blue
- m: magenta
- c: cyan
- w: white

List of styles:

- b: bold
- i: italic
- u: underline
- s: strike through
- x: blinking
- r: reverse
- y: fast blinking
- f: faint
- h: hide

Parameters

- **fmt** (*str*) – string to be colored
- **fg** (*str*) – foreground color
- **bg** (*str*) – background color

- **style** (*str*) – text style

fret.util.**nonbreak**(*f=None*)
Make sure a loop is not interrupted in between an iteration.

fret.util.**stateful**(**states*)
Decorator for building stateful classes.

Python Module Index

f

Index

Symbols

A

B

C

D

E

F

I

L

M

N

P

R

S

W