clichain Documentation

Release 0.1.0

Loïc Peron

Mar 12, 2019

documentation:

CHAPTER 1

documentation

here's a practical documentation, for source documentation please see *source documentation*

1.1 quickstart

to create a command line tool with `clichain` you need:

- to create a factory:

```
from clichain import cli
tasks = cli.Tasks()
```

- to implement task types using coroutine functions:

  See also:

  http://www.dabeaz.com/coroutines

  The easiest way of implementing a task type is to use the *task* decorator:

```
from clichain import pipeline
import logging
import ast


@pipeline.task
def add_offset(ctrl, offset):
    logger = logging.getLogger(f'{__name__}.{ctrl.name}')
    logger.info(f'starting, offset = {offset}')

    with ctrl as push:
        while True:
            value = yield
            push(value + offset)
```

(continues on next page)

(continued from previous page)

```
    logger.info('offset task finished, no more value')


@pipeline.task
def parse(ctrl):
    _parse = ast.literal_eval
    with ctrl as push:
        while True:
            push(_parse((yield)))
```

See also:

clichain.pipeline.task

- to register task types into the factory:

  tasks are integrated into the command line tool using `click` commands

  The simplest way of registering a task type is to decorate it with the factory:

```
import click


@tasks
@click.command(name='offset')
@click.argument('offset')
def offset_cli(offset):
    "add offset to value"
    offset = ast.literal_eval(offset)
    return add_offset(offset)


@tasks
@click.command(name='parse')
def parse_cli():
    "parse input data with ast.literal_eval"
    return parse()
```

See also:

`click` documentation for more details about commands

Note: it's up to you to determine where and how you want the tasks to be registered into the factory, one way of doing this is to make the factory a module attribute and use it into separate scripts...

- to start the main command from your main entry point:

```
if __name__ == '__main__':
    cli.app(tasks)
```

If we combine all the previous code into a single script, we get this:

```
#! /usr/bin/env python
# -*- coding: utf-8 -*-
from clichain import cli, pipeline
import click
```

(continues on next page)

(continued from previous page)

```
import logging
import ast


tasks = cli.Tasks()


# ---------------------------------------------------------------------- #
# implement tasks                                                        #
# ---------------------------------------------------------------------- #
@pipeline.task
def add_offset(ctrl, offset):
    logger = logging.getLogger(f'{__name__}.{ctrl.name}')
    logger.info(f'starting, offset = {offset}')

    with ctrl as push:
        while True:
            value = yield
            push(value + offset)

    logger.info('offset task finished, no more value')


@pipeline.task
def parse(ctrl):
    _parse = ast.literal_eval
    with ctrl as push:
        while True:
            push(_parse((yield)))


# ---------------------------------------------------------------------- #
# register tasks                                                         #
# ---------------------------------------------------------------------- #
@tasks
@click.command(name='offset')
@click.argument('offset')
def offset_cli(offset):
    "add offset to value"
    offset = ast.literal_eval(offset)
    return add_offset(offset)


@tasks
@click.command(name='parse')
def parse_cli():
    "parse input data with ast.literal_eval"
    return parse()


# ---------------------------------------------------------------------- #
# run cli                                                                #
# ---------------------------------------------------------------------- #
if __name__ == '__main__':
    cli.app(tasks)
```

if our script is called '**dummy.py**', we can use **–help** option to get a full description:

```
$ ./dummy.py --help
Usage: dummy.py [OPTIONS] COMMAND1 [ARGS]... [COMMAND2 [ARGS]...]...

  create a pipeline of tasks, read text data from the standard input
  and send results to the standard output: ::

              stdin(text) --> tasks... --> stdout(text)


[...]


Options:
  -l, --logfile PATH  use a logfile instead of stderr
  -v, --verbose       set the log level: None=WARNING, -v=INFO, -vv=DEBUG
  --help              Show this message and exit.

Commands:
  offset  add offset to value
  parse   parse input data with ast.literal_eval
  [       begin fork
  ]       end fork
  ,       new branch
  {       begin debug name group
  }       end debug name group
```

we can see our two task types are availables, we can use **–help** option as well on it:

```
$ ./dummy.py offset --help
Usage: dummy.py offset [OPTIONS] OFFSET

  add offset to value

Options:
  --help  Show this message and exit.
```

See also:

click

assuming we want to run this:

```
                          +--> +1 --+
              +--> +10 --|          +------+
              |          +--> +2 --+      |
inp >> parse--|                           +--> >> out
              +--> +100 --> +1 -----------+
```

we can use our tool as followings (*sh*):

```
$ PIPELINE="parse [ offset 10 [ offset 1 , offset 2 ] , offset 100 offset 1 ]"
$ python -c 'print("\n".join("123456789"))' | ./dummy.py $PIPELINE
12
13
102
13
14
103
```

(continues on next page)

(continued from previous page)

```
14
15
104
15
16
105
16
17
106
17
18
107
18
19
108
19
20
109
20
21
110
```

Note: everything is run into a single process and thread

1.2 creating a factory

Task types are integrated into the command line tool using `click` commands.

In order to achieve this we register commands into a *factory* and then use that factory when running the main command line interface.

```
from clichain import cli
tasks = cli.Tasks()
```

The created factory will register all the commands into a `dict`, which can be accessed via the **commands** attribute.

See also:

clichain.cli.Tasks

It's up to the user to define a strategy about where to create the factory and how to access it from different parts of the program.

1.3 implementing a task

Task types are implemented using coroutine functions:

See also:

http://www.dabeaz.com/coroutines

Though you can implement a coroutine by yourself, the framework provides two ways of implementing a coroutine as expected by the framework:

- *clichain.pipeline.coroutine* decorator

  this is simply a trivial decorator which creates a coroutine function that will be *primed* when called, i.e advanced to the first *yield*.

  example:

```
from clichain import pipeline

@pipeline.coroutine
def cr(*args, **kw):
    print('starting...')
    try:
        while True:
            item = yield
            print(f'processing: {item}')
    except GeneratorExit:
        print('ending...')
```

  When used in a pipeline, the coroutine function will be called with specific keyword arguments:

  See also:

  clichain.pipeline.create for more details

  - **context**: a *clichain.pipeline.Context* object shared by all the coroutines of the pipeline.
  - **targets**: an iterable containing the following coroutines in the pipeline (default is no targets).
  - **debug**: an optional name (used by *clichain.pipeline.task*, see below)

    Note: Default value is the coroutine's key in the pipeline definition (default will be used if value is `None` or an empty string).

- *clichain.pipeline.task* decorator

  this is the easiest way of implementing a task type because the decorated function won't have to worry about the input args and `GeneratorExit` handling, in addition automatic exception handling will be performed if an exception occurs (see *clichain.pipeline.task* for details).

  The *clichain.pipeline.Control* object will provide a **push** function to directly send data to next stages of the pipeline, and a **name** attribute can be used to identify the coroutine instance (when logging for example). The name is optionally given when creating the pipeline object using *clichain.pipeline.create*.

  example:

```
from clichain import pipeline

@pipeline.task
def add_offset(ctrl, offset):
    with ctrl as push:
        while True:
            value = yield
            push(value + offset)
```

1.4 registering a task

Task types are integrated into the command line tool using `click` commands.

See also:

`click` documentation for more details about commands

The factory (see *creating a factory*) is a callable object meant to be used as a decorator to register a new `click` command into its *commands* dictionary.

```
import click

@tasks
@click.command(name='offset')
@click.argument('offset')
def offset_cli(offset):
    "add offset to value"
    offset = ast.literal_eval(offset)
    return add_offset(offset)
```

The `click` command function is expected to return a coroutine function that can be integrated into the created pipeline, see *implementing a task* section for details.

Note: in the previous example we can access the registered task through the *commands* attribute of the factory:

```
assert tasks.commands['offset'] is offset_cli
```

Note the *offset_cli* callback function is a decorated version of the original callback function (defined by the user).

1.5 running the command line tool

The main command is executed by `click` framework. Use the *clichain.cli.app* function to run it with the factory, example:

```
if __name__ == '__main__':
    cli.app(tasks)
```

Note: additional *args* and *kwargs* will be kept in the `click` context object, see *clichain.cli.app* for details.

1.6 testing

In order to perform automated tests you can run the *clichain.cli.test* function, which will run the main command using `click.testing` framework.

example:

```
from clichain import cli

tasks = cli.Tasks()

# register the 'compute' task
[...]
```

(continues on next page)

(continued from previous page)

```
# test
args = ['compute', '--help']
inputs = [1, 2, 3]
result = cli.test(tasks, args, inputs=inputs)
assert result.output == "foo"
assert result.exit_code == 0
assert not result.exception
```

Note: the test function supports additional arguments, see *clichain.cli.test* for details.

1.7 logging

Automatic logging is performed for registered tasks by *clichain.pipeline* framework when an unhandled exception occurs.

In this case the exception will be logged at **ERROR** level with exception info (see `logging.error`), using the optional *name* of the coroutine to determine the logger path.

Todo: custom kwargs cannot be passed to *clichain.pipeline.create* when using *clichain.cli* framework (*clichain.cli.app* or *clichain.cli.test*), such as custom root logger.

Note: when creating the pipeline the root logger to use can be specified, see *clichain.pipeline.create* for details. The default root logger will be `clichain.pipeline.logger`.

Note: an optional name can be given by the user (using a specific command defined in *clichain.cli*) to coroutines when creating the pipeline, see *clichain.pipeline.create* and *implementing a task* for more details.

Using the optional name to perform logging in tasks implementation is advised, example:

```
from clichain import pipeline
import logging

logger = logging.getLogger(__name__)

@pipeline.task
def add_offset(ctrl, offset):
    log = logger.getChild(ctrl.name)
    log.info(f'starting, offset = {offset}')

    with ctrl as push:
        while True:
            value = yield
            push(value + offset)

    log.info('offset task finished, no more value')
```

1.8 exceptions handling

Exceptions in `click` commands should be handled using `click` exception handling framework.

example:

```
@tasks
@click.command(name='offset')
@click.argument('offset')
def offset_cli(offset):
    "add offset to value"
    try:
        offset = ast.literal_eval(offset)
    except:
        raise click.BadParameter(f'wrong value: {offset}')

    return add_offset(offset)
```

If an unhandled exception occurs in a task when the pipeline is running, then the exception will be logged (see *logging*) and the main command will abort (using `click.Abort`) after all the coroutines have been **closed**.

example:

```
@pipeline.task
def add_offset(ctrl, offset):
    with ctrl as push:
        while True:
            value = yield
            if value > 0:
                push(value + offset)
            else:
                raise NotImplementedError(value)
```

Note: In the above example, all the tasks after 'add_offset' in the pipeline will be terminated, all the tasks before 'add_offset' will fail. This behaviour is the native behaviour of coroutines, since coroutines following 'add_offset' will have no more values and coroutines before 'add_offset' will face a StopIteration.

Whatever the exit state of the process (fail or completed), all the coroutines of the pipeline will be **closed** (i.e *coroutine.close()* will be called), that means the following coroutine will close the file **as soon as the pipeline stops** anyways:

```
@coroutine
def cr(*args, **kw):
    with open('foo/bar') as f:
        while True:
            data = yield
            [...]
```

See also:

clichain.pipeline.Pipeline

CHAPTER 2

source documentation

2.1 cli

`cli` focuses on the command line tool aspect using `click`

See also:

`Tasks`, *`usage`*, *`clichain.pipeline`*

class clichain.cli.**Cli**(*name=None*, *invoke_without_command=False*, *no_args_is_help=None*, *subcommand_metavar=None*, *chain=False*, *result_callback=None*, ***attrs*)

Implements root command using `click.MultiCommand`

See also:

`click`

`Cli` provides implementation for the root command by extending `click.MultiCommand` (Then the click command is created specifying *`Cli`* as "cls" parameter in the `click.command` decorator.

See also:

`app`, *`Tasks`*

get_command(*ctx*, *name*)

Given a context and a command name, this returns a `Command` object if it exists or returns `None`.

list_commands(*ctx*)

Returns a list of subcommand names in the order they should appear.

class clichain.cli.**Tasks**(*commands=None*)

Defines a factory to register tasks types.

`Tasks` provides a factory to register and hold implemented tasks so they get available to the user.

Tasks are implemented by coroutines.

See also:

`clichain.pipeline`

The command line interface is implemented using `click`. In order to make a task usable from the command line interface, you need to define a `click` command (for most cases using `click.command` decorator). example:

```
@click.command(name='compute')
@click.option('--approximate', '-a',
              help='compute with approximation')
@click.argument('x')
def my_compute_task(approximate, x):
    " the task doc that will appear as help "
    # process inputs parameters and options...
```

See also:

`click` for details on how to implements commands

The command is expected to return a coroutine function such as *`clichain.pipeline.coroutine`*, see *`clichain.pipeline.create`* for details.

full example:

```
from clichain import pipeline, cli
import ast

# creates the factory here but should be
# common to all task types...
tasks = cli.Tasks()

@pipeline.task
def divide(ctrl, den):
    print(f'{ctrl.name} is starting')

    with ctrl as push:
        while True:
            value = yield
            push(value / den)

    print(f'{ctrl.name} has finished with no error')


# the task will be made available as 'compute' in the
# command line interface

@tasks
@click.command(name='compute')
@click.option('--approximate', '-a',
              help='compute with approximation')
@click.argument('den')
def compute_task_cli(approximate, den):
    " the task doc that will appear as help "
    if den == 0:
        raise click.BadParameter("can't devide by 0")
    if approximate:
        den = round(den)
    return devide(den)

@pipeline.task
def parse(ctrl):
    _parse = ast.literal_eval
```

(continues on next page)

(continued from previous page)

```
    with ctrl as push:
        while True:
            try:
                push(_parse((yield)))
            except (SyntaxError, ValueError):
                pass

@tasks
@click.command(name='parse')
def parse_cli():
    " performs literal_eval on data "
    return parse()
```

See also:

usage

__call__(*cmd*)

wraps and register a `click.command` object into the factory

Tasks object is intended to be used as a decorator:

```
tasks = cli.Tasks()

# register 'compute' command into 'tasks'
@tasks
@click.command(name='compute')
[...]
```

The command is expected to return a coroutine function such as *clichain.pipeline.coroutine*, see *clichain.pipeline.create* for details.

Note: a log message will be emitted to indicate this task is created every time the command is called

---

__init__(*commands=None*)

initializes new factory with `commands`

`commands` is a dict containing all registered commands, and will be set to a new empty dict if None.

`context` is the current click context, it's set to None until the main command is called (see *app*), which will set `context` to the current context value.

_prepare_cmd(*cmd*)

wraps the `click.command` callback function and replace it

the wrapper function will:

- log a 'create' message (using `logger.info`)
- use the callback function result to create the next coroutine in the pipeline. The coroutine function created by the callback function is stored in the current `click` context. A stack is used to process pipeline's branches.

  See also:

  clichain.pipeline for details on how the pipeline is specified

The wrapper function does not return anything

See also:

this method is called by *Tasks.__call__*

clichain.cli.**_get_obj**(*tasks*, *args*, *kwargs*)
get *obj* parameter for click context

The created *obj* is a dict, it's used internally when processing commands.

- tasks is the *Tasks* factory to use (containing user commands)
- optional args and kwargs will be send to the click context (in context.obj['args'] and context.obj['kwargs']), they will not be used by the framework.

This function is used by *app* and *test*.

clichain.cli.**app**(*tasks*, **args*, ***kw*)
run click main command: this will start the CLI tool.

See also:

test

tasks is the *Tasks* factory to use (containing user commands)

extra args and kwargs are added to the click context's **obj** (a dict) as **'args'** and **'kwargs'**, they're not used by the framework.

app uses *Cli* which extends click.MultiCommand to create the main command as a multicommand interface. This main command holds all the user defined commands and is the main entry point of the created tool.

The pipeline itself is created and run by the *process* function, which is called when the main command itself returns, i.e when the all the input args have been processed by click.

See also:

Tasks, *process*

See also:

the main command itself only set up logging, see also *usage*

clichain.cli.**process**(*obj*, *rv*, *logfile*, *verbose*)
callback of the main command, called by click

process creates the pipeline (using *clichain.pipeline.create*), then run it with inputs from stdin and sending outputs to stdout (getting stdin and stdout from click.get_text_stream).

if an exception occures then log the exception and raise click.Abort.

See also:

clichain.pipeline

clichain.cli.**test**(*tasks*, *clargs*, *args=None*, *kwargs=None*, ***kw*)
run the CLI using click.testing, intended for automated tests

See also:

app

The main command is then run with click.testing.CliRunner

this is roughly equivalent to:

```
>>> runner = click.testing.CliRunner()
>>> obj = cli._get_obj(tasks, args, kwargs)
>>> result = runner.invoke(cli._app, clargs, obj=obj, **kw)
```

- `tasks` is the *`Tasks`* factory to use (containing user commands)
- `clargs` is a list containing the command line arguments, i.e what the user would send in interactive mode.
- optional `args` and `kwargs` will be sent to the `click` context (in context.obj['args'] and context.obj['kwargs']), they will not be used by the framework.
- extra `kw` will be forwarded to `click.testing.CliRunner.invoke`, for example:

```
input=[1,2,3], catch_exceptions=False
```

creates a `click.testing.CliRunner` to invoke the main command and returns *runner.invoke* result.

See also:

`click.testing`

`clichain.cli.`**`usage`**`()`

create a pipeline of tasks, read text data from the standard input and send results to the standard output:

```
stdin(text) --> tasks... --> stdout(text)
```

The overall principle is to run a data stream processor by chaining different kinds of tasks (available tasks depending on the implementation, see list below).

you can create a single branch pipeline as a sequence of tasks, for instance:

```
inp >>> A -> B -> C >>> out
```

or you can create a more complex pipeline defining multiple branches, for instance:

```
           +--> B --> C --+
inp >>> A--|               +--> F >>> out
           +--> D --> E --+
```

tasks are implemented by *coroutines* functions as described by David Beazle (see http://www.dabeaz.com/coroutines/ for details).

- Specifying pipeline workflow:

  basic syntax allows you to specify the worflow of the pipeline.

  A single sequence of tasks as the following:

  ```
  inp >>> A -> B -> C >>> out
  ```

  is specified as:

  ```
  A B C
  ```

  Note: plus parameters and options of the tasks themselves, i.e:

  ```
  A -x -y arg1 B -z ...
  ```

  Creating branches requires 'workflow commands', for instance the following example:

  ```
             +--> B --> C --+
  inp >>> A--|               +--> F >>> out
             +--> D --> E --+
  ```

would be specified as:

```
A [ B C , D E ] F
```

the same way we can define sub branches, for instance:

```
                    +--> C1 --+
           +--> B --|         +-----+
           |        +--> C2 --+     |
inp >>> A--|                        +--> F >>> out
           +--> D --> E ------------+
```

would be specified as:

```
A [ B [ C1 , C2 ] , D E ] F
```

- Execution order:

  When *parallel* branches are defined (as 'C1' and 'C2' in the previous example) they are processed in the same order as they are defined in the command line arguments, that means in this example:

```
A [ B [ C1 , C2 ] , D E ] F
```

  If the input data is:

```
1
2
[...]
```

  Then the workflow will be such as:

```
# data will go through C1 then C2
1 -> A -> B -> C1 -> F
1 -> A -> B -> C2 -> F
2 -> A -> B -> C1 -> F
2 -> A -> B -> C2 -> F
[...]
```

  And the order is reproducible

- Attaching a name to branches or tasks:

  You can attach a name to coroutines when defining the pipeline, which will be used as a suffix to get the logger if an exception occurs in the coroutne, i.e:

```
base_logger.getChild(<name>)
```

  Note: This is useful in particular if you're using the same task type in several branches. The name could be used as well in the coroutine, depending on its implementation (see *clichain.pipeline.create* for more details).

  example:

```
           +--> B --> C --+
inp >>> A--|              +--> B >>> out
           +--> B --> D --+
```

you could specify the name of the branches (i.e all the coroutines of those branches) with:

```
A [ { 'b1' B C } , { 'b2' B D } ] { 'b3' B )
```

Note: the name specification is valid for every coroutine whose definition starts within the parenthesis, for example:

```
A { 'b1' [ B C , B D ] } B
```

is equivalent to:

```
A [ { 'b1' B C , B D } ] B
```

which is also equivalent to:

```
A [ { 'b1' B C , B D ] } B
```

which is equivalent to:

```
A [ { 'b1' B C } , { 'b1' B D } ] B
```

note the output 'B' coroutine will have no name

And using the following specification:

```
A [ { 'b1' B } C , { 'b2' B } D ] B
```

will only give 'b1' and 'b2' names to the 'B' coroutines (and not to the 'C' and 'D' coroutines as in the previous example).

Then note the following:

```
A { 'b1' [ B C , { 'b2' B } D ] } B
```

is equivalent to:

```
A [ { 'b1' B C } , { 'b2' B } { 'b1' D } ] B
```

Then note that the following:

```
A [ { 'b1' } B , { 'b2' } B ] B
```

will have no effect at all.

2.2 pipeline

`pipeline` module provides tools to create a pipeline of tasks

a pipeline can be composed of one or several branches, but everything runs in a single thread. The initial goal of this framework is to provide a simple and direct way of defining task types and reuse them in different pipeline configurations.

The motivation is not to parallelise tasks yet tasks could be parallelized in some configurations, depending on the exact use case and the context...

tasks are implemented by *coroutines* functions as described by David Beazle (see http://www.dabeaz.com/coroutines/ for details).

This module is used by *clichain.cli* module.

class clichain.pipeline.**Context**(*logger=<Logger clichain.pipeline (WARNING)>*, *obj=None*)
will be passed to all 'coroutine's in the pipeline

Context object is a common object shared by all coroutines in the pipeline.

attributes:

- exceptions is a list which remains empty until an exception occurs within a *task* and is handled by the module. Then exception contains each exception caught by the module. Each exception is logged only one time with its traceback when it's caught by *Control* context manager.

  Note: if an exception is caught by the module it will be "re-raised" thus terminate the process but user code could still raise another exception(s) for example if a coroutine is not implemented using *task* or GeneratorExit is handled within the user loop...

- logger will be used for every message logged by the module, and can be used by the user. The default is to use the module's logger.
- obj attribute is an arbitrary object provided by the user when creating the pipeline. The default is None.

See also:

create

__init__(*logger=<Logger clichain.pipeline (WARNING)>*, *obj=None*)
init the *Context* which will be shared by coroutines

class clichain.pipeline.**Control**(*context*, *name*, *targets*)
Internal, 'control' obj received by *task* decorated functions

Control is a context manager

See also:

Control.__init__

__enter__()
return push function

See also:

Control.push, *Control.__exit__*

__exit__(*tpe*, *value*, *tb*)
handle GeneratorExit exception and log unhandled exceptions

Control object is created by *task* decorator, the decorated function gets the *Control* object as first arg, and is expected to use it as a context manager, which will handle GeneratorExit and log any unhandled exception.

context attribute (*Context* object) will be used if the exception is not None or GeneratorExit, in order to:

- determine if the exception traceback should be logged, if the exception has already been logged by another *Control* object (i.e in another coroutine), then only an error message will be logged, otherwise the full exception will be recorded.
- get the base logger to use

See also:

task, *Control.__init__*, *Context*

__init__(*context*, *name*, *targets*)
initialize new *Control* object

- `context` is a *Context* object, *Control* object will use it to log exception if an exception occurs in the coroutine while used as a context manager. The *Context* object is also accessible through the `context` attribute.
- `name` will be accessible as `name` attribute (rw)

  ex:

```
logger = logging.getLogger(f'{__name__}.{ctrl.name}')
```

- *targets* will be accessible through *targets* property (ro)

  See also:

  Control.targets

targets is read only to ensure consistency with *push* function returned by *Control.__enter__*: *Control* object is expected to be used as a context manager:

```
with ctrl as push:
    while True:
        data = yield
        push(data)  # send data to next coroutines
```

a *push* function is defined and returned when *Control* is used as a context manager, but can actually be created using *Control.push* property.

the purpose is to force using an efficient solution avoiding attributes lookup (using *self*) for every call, which has an impact given this function is likely to be called a lot (usually for each processed item). This way we define the function and reference it once in the user function (as 'push' in the example).

See also:

task decorator

push
return a 'push' function sending data to next coroutines

See also:

Control.__init__

targets
next coroutines in the pipeline

class clichain.pipeline.**Pipeline**(*targets*)
User interface returned by *create* function

Pipeline object contains the 'coroutine's of a pipeline.

When used as a context manager, it ensures that coroutines will be closed immediately at the end of the process.

See also:

Pipeline.__enter__, *Pipeline.__exit__*

Pipeline also has an additional *Pipeline.run* method which can be called to run the pipeline over an input stream and wait until the process complete.

__enter__()
return a function sending data thtough the pipeline

ex:

```
with pipeline as process:
    for data in stream:
        process(data)
```

Note: this is equivalent to:

```
with pipeline:
    target = pipeline.target
    for data in stream:
        target.send(data)
```

See also:

Pipeline.__exit__

__exit__(*tpe*, *value*, *tb*)
close all the coroutines of the pipeline, raise exc if any

The purpose of using the *Pipeline* object as a context manager is essentially to make sure all the coroutines will be terminated (closed) at the end of the process.

This can be critical if user functions are expected to do some teardown jobs after processing data, for instance:

```
# file won't be closed until the coroutine is closed
# (see while loop...)

@coroutine
def cr(targets, *args, file, **kw):
    with open(file) as f:
        while True:
            data = yield
            [...]
```

See also:

Pipeline.__enter__

__init__(*targets*)
initialize a new pipeline with ‘coroutine‘s

`targets` is an iterable containing the coroutines of the pipeline, the first item must be the input coroutine.

See also:

Pipeline.run

run(*inputs*)
run the pipeline over `inputs` iterator

send data from `inputs` to the pipeline until their is no more data in inputs or an exception occurs.

clichain.pipeline.**_listify**(*obj*)
makes sure `obj` is a `list`

clichain.pipeline.**coroutine**(*func*)
coroutine decorator, 'prime' the coroutine function

this function is intended to be used as a decorator to create a basic *coroutine* function, for instance:

```
@coroutine
def cr(*args, **kw):
    print('starting...')
    try:
        while True:
            item = yield
            print(f'processing: {item}')
    except GeneratorExit:
        print('ending...')
```

calling the decorated function will automatically get it to the first *yield* statement.

```
>>> cr()
starting...
```

Note: the decorated function is wrapped using `functools.wraps`

See also:

http://www.dabeaz.com/coroutines/

clichain.pipeline.**create**(*tasks*, *output=<built-in function print>*, ***kw*)
create a pipeline of coroutines from a specification

a pipeline is a succession of coroutines organized into one or several branches.

`output` is a strategy to use for the pipeline output, the default strategy is `print`. `output` will be called for each data reaching the pipeline's output, it takes a single argument.

extra keyword args will be used to initialize the *`Context`* object that will be send to all the coroutines of the pipeline.

`tasks` argument describes the pipeline and consists of a mapping of coroutines as key: value pairs, where each single key identifies a single coroutine.

each coroutine is defined either by a single *`coroutine`* function (see **task** field below) or a dictionnay, which contains the following fields:

- **task**: the coroutine function to use

  See also:

  `task` decorator

  the coroutine function will be called with the following keyword arguments:

  - **context**: the *`Context`* object shared by all the coroutines of the pipeline.
  - **targets**: an iterable containing the following coroutines in the pipeline (default is no targets).
  - **debug**: an optional name, used by *`task`* to get a child logger from the *`Context`* logger, which will be used to log error if an exception occurs. The exception will be logged at error level and the exc_info will be passed to the log record. The value will be accessible (and writeable) through `Control.name` attribute, which can be usefull for logging:

    ex:

```
logger = logging.getLogger(f'{__name__}.{ctrl.name}')
```

Note: Default value is the coroutine's key in the pipeline definition (default will be used if value is `None` or an empty string).

- **input**: (optional) set this coroutine as a 'target' of the coroutine(s) defined by **input**. **input** can be a single key or an iterable containing keys of other coroutines defined in the pipeline dictionnary.

  Note: `None` value will be interpreted as the pipeline's main input. No value or an empty list is equivalent as None if this coroutine is not specified as **output** of an other coroutine in the pipeline.

- **output**: (optional) set the coroutine(s) whose keys are defined in **output** as a 'target' of this coroutine. **output** can be a single key or an iterable containing keys of other coroutines defined in the pipeline dictionnary.

  Note: `None` value will be interpreted as the pipeline's main output. No value or an empty list is equivalent as None if this coroutine is not specified as **input** of an other coroutine in the pipeline.

- **debug**: (optional) a debug name to use in logging if an unhandled exception occurs. see above description.

Note: specifying a coroutine by a *coroutine* function is equivalent as providing a dictionnary containing only the **task** field.

examples:

See also:

task and *coroutine* decorators

given we have the following declarations:

```
@coroutine
def output(targets, **kw):
    try:
        while True:
            for t in targets:
                t.send('RESULT: {}'.format((yield)))
    except GeneratorExit:
        return

@task
def parse(ctrl):
    with ctrl as push:
        while True:
            try:
                value = ast.literal_eval((yield))
            except (SyntaxError, ValueError):
                continue
            push(value)

@task
def mytask(ctrl, param):
```

(continues on next page)

(continued from previous page)

```
    logger = logging.getLogger(f'{__name__}.{ctrl.name}')
    logger.info('starting task')
    with ctrl as push:
        while True:
            [...]
    logger.info('finishing task')
```

- defining a pipeline composed of a single sequence:

  example:

```
inp >>> a --> b --> c --> d >>> out
```

  here's how we could define it:

```
pipeline = pipeline.create({
    'a': parse(),
    'b': {'task': mytask(1), 'input': 'a'},
    'c': {'task': mytask(2), 'input': 'b'},
    'd': {'task': output, 'input': 'c'},
})
```

  the created pipeline is a *Pipeline* object, it can be run over any input generator using its 'Pipeline.run' method, sending data to stdout by default.

  See also:

  Pipeline.run

- define a pipeline with branches:

  example:

```
           +--> B --> C >>> out
inp >>> A--|
           +--> D --> E >>> out
```

  here's how we could define it:

```
pipeline = pipeline.create({
    'a': {'task': A, 'output': ('b', 'd')},
    'b': B,
    'd': D,
    'c': {'task': C, 'input': 'b'},
    'e': {'task': E, 'input': 'd'},
})
```

  redoundant specification is not an issue, and the following example is equivalent to the previous one:

```
pipeline = pipeline.create({
    'a': {'task': A, 'output': ('b', 'd')},
    'b': {'task': B, 'input': 'a', 'output': 'c'},
    'd': {'task': D, 'input': 'a', 'output': 'e'},
    'c': {'task': C, 'input': 'b', 'output': None},
    'e': {'task': E, 'input': 'd', 'output': ()},
})
```

- join branches

example: given we want to implement this:

```
           +--> B --> C --+
inp >>> A--|              +--> N >>> out
           +--> D --> E --+
```

here's how we could define it:

```
pipeline = pipeline.create({
    'a': {'task': A, 'output': ('b', 'd')},
    'b': B,
    'c': {'task': C, 'input': 'b', 'output': 'f'},
    'd': D,
    'e': {'task': E, 'input': 'd', 'output': 'f'},
    'f': F,
})
```

- control branches order

  the order in which coroutines are initialized, called and closed is reproducible.

  to control the data flow order between several branches just use the keys in the pipeline definition, as they will be sorted, like in the following example:

```
           +--> (1) X --+
           +--> (2) X --+
inp >>> A--+--> (3) X --+--> B >>> out
           +--> (4) X --+
           +--> (5) X --+
```

  here's how we could define it:

```
pipeline = pipeline.create({
    'a': A,
    1: {'task': X, 'input': 'a', 'output': 'b'},
    2: {'task': X, 'input': 'a', 'output': 'b'},
    3: {'task': X, 'input': 'a', 'output': 'b'},
    4: {'task': X, 'input': 'a', 'output': 'b'},
    5: {'task': X, 'input': 'a', 'output': 'b'},
    'b': B,
})
```

  the 'X' coroutines will be initialized and processed in the expected order: 1, 2, 3, 4, 5 (they will be closed, if no exception occurs, in the opposite order).

- loop back

  Warning: looping is currently not implemented and will raise a `NotImplementedError` when creating the pipeline.

  example: given we want to implement this:

```
           +--> B --> C --+          + >>> out
inp >>> A--|              +--> N -- +
           +--> D --> E --+          |
                          |          |
                 +--> F --+          |
```

(continues on next page)

(continued from previous page)

```
                  |                   |
                  +-------------------+
```

here's how we could define it:

```
pipeline = pipeline.create({
    'a': {'task': A, 'output': ('b', 'd')},
    'b': {'task': B, 'output': 'c'},
    'c': {'task': C},
    'd': {'task': D, 'output': 'e'},
    'e': {'task': E},
    'n': {'task': N, 'input': ('c', 'e'), 'output': None},
    'f': {'task': F, 'input': 'n', 'output': 'n'},
})
```

Warning: defining a loop can end up in an infinite recursion , no control is done on this, so it's up to the tasks implementation to handle this...

- specify coroutines name

  in some contexts we may want to define a name for a coroutine which is different from its key.

  example: the previous example with ordered branches was:

```
           +--> (1) X --+
           +--> (2) X --+
inp >>> A--+--> (3) X --+--> B >>> out
           +--> (4) X --+
           +--> (5) X --+
```

  here's how we could define it:

```
pl = {
    'a': A,
    'b': B,
}

pl.update({
   i: {'task': X, 'input': 'a', 'output': 'b',
       'debug': f"the X task number {i}"}
   for i in range(1, 6)
})

pl = pipeline.create(pl)
```

clichain.pipeline.**task**(*func*)

make "partial" `coroutines` expected to be used with *`create`*.

`task` will create a "partial" function, which when called with args and keyword args will actually return a *`coroutine`* function designed to be used with *`create`* function.

example:

a basic coroutine adding offset to input data could be defined as follows using *`task`*:

```
@task
def offset(ctrl, offset):
    print('pre-processing')

    with ctrl as push:
        while True:
            value = yield
            push(value + offset)

    # will be executed unless an exception occurs in
    # the 'while' loop
    print('post_processing')
```

- `ctrl` will handle the `GeneratorExit` exception and log any unhandled exception.
- the `push` method send data to the next coroutines in the pipeline.

the resulting function is called with the original function's args and keyword args:

```
off = offset(offset=1)
```

off is a partial *coroutine* function expected to be used in a pipeline defintion with *create*.

the coroutine will eventually be created calling this new function with specific arguments depending on the pipeline specification (see *create* for details), ex:

```
# create the coroutine
off = off(targets=[t1, t2...])
```

Note: as for *coroutine*, all the functions (partial or final functions) are wrapped using `functools.wraps`

example:

```
@task
def output(ctrl):
    with ctrl:
        while True:
            print((yield))

@task
def parse(ctrl):
    with ctrl as push:
        while True:
            try:
                value = ast.literal_eval((yield))
            except (SyntaxError, ValueError):
                continue
            push(value)

@task
def offset(ctrl, offset):
    offset = int(offset)
    logger = logging.getLogger(f'{__name__}.{ctrl.name}')
    logger.info(f'offset: {offset}')
```

(continues on next page)

(continued from previous page)

```
    with ctrl as push:
        while True:
            value = yield
            push(value + offset)

    logger.info('offset task finished, no more value')

if __name__ == '__main__':
    out = output()
    off1 = offset(10)
    off2 = offset(offset=100)
    parse = parse()

    # the previous results (out, off1, off2, proc) should
    # be used in the pipeline definition and the followings
    # should be performed by "create"
    out = out()
    off1 = off1((out,))
    off2 = off2((out,))
    parse = parse([off1, off2])

    with open('foo.txt') as inputs:
        for data in inputs:
            parse.send(data)

    out.close()
    off1.close()
    off2.close()
    parse.close()
```

See also:

coroutine, *create*

CHAPTER 3

clichain

Create a command line interface to chain tasks as a pipeline

clichain is a framework to easily define task types and chain them from a command line interface.

The goal of this framework is to use David Beazle's idea to implement task types as coroutines and use them to create and run a pipeline.

The goal is **not** to parallelize tasks but to be able to reuse task types in different configurations without need for coding and in some cases reuse a result from a long computational task for different purposes without running it several times.

CHAPTER 4

install and test

4.1 install from pypi

using pip:

```
$ pip install clichain
```

4.2 dev install

There is a makefile in the project root directory:

```
$ make dev
```

Using pip, the above is equivalent to:

```
$ pip install -r requirements-dev.txt
$ pip install -e .
```

4.3 run the tests

Use the makefile in the project root directory:

```
$ make test
```

This runs the tests generating a coverage html report

4.4 build the doc

The documentation is made with sphinx, you can use the makefile in the project root directory to build html doc:

```
$ make doc
```

CHAPTER 5

Documentation

Documentation on Read The Docs.

CHAPTER 6

Meta

loicpw - peronloic.us@gmail.com

https://github.com/loicpw

CHAPTER 7

Indices and tables

- genindex
- modindex
- search

Python Module Index

C

Index

Symbols

A

C

G

L

P

R

T

U