sutools

Release 0.1

Aaron Stopher

Jun 29, 2023

CONTENTS:

CHAPTER

ONE

SUTOOLS

su (Super User) tools

Per module utilities, designed to be lightweight, easy to configure, and reduce boilerplate code.

info

This package is intended to create an lower barrier for entry for logging and module level cli with sensible defaults; sutools is not intended to replace click, argparse, logging or other utility libraries. If your project requires a more flexible configuration please use the appropriate tooling.

`sutools.cli(`*desc=None*, *logs=False*`)`

init cli and register to store

Parameters

- **desc** – description of the CLI
- **logs** – enable logging in CLI

`sutools.log()`

retrieve loggers namespace from store

`sutools.logger(`*name='<frozen run'*, *loggers=None*, *loglvl=20*, *filename='2023-06-29_17-11-33'*, *filepath=None*, *filefmt=<logging.Formatter object>*, *fhandler=None*, *filecap=None*, *filetimeout=None*, *file=True*, *streamfmt=<logging.Formatter object>*, *shandler=<StreamHandler <stderr> (NOTSET)>*, *stream=False*`)`

init logger object and register to store

Parameters

- **name** – name of the logger
- **loggers** – list of names for the logger to create
- **loglvl** – logging level to use
- **filename** – name of the log file
- **filepath** – name of folder to create logs in
- **filefmt** – format of the file logger
- **fhandler** – file handler to use for logging
- **filecap** – maximum number of log files to keep
- **filetimeout** – define a timeout period for log files to be removed
- **file** – toggle file logging
- **streamfmt** – format of the stream logger

- **shandler** – stream handler to use for logging
- **stream** – toggle stream logging

Timeout String
<int><time_unit> - '10d': represents 10 days

Time Units
{ "m": "minutes", "h": "hours", "d": "days", "o": "months", "y": "years" }

sutools.**register**(*func*)

register a function to the store

Parameters
func – the function to register

1.1 store

sutools `store` instance is a global instance of the `meta_handler.Bucket` class. This instance is used to store functions, cli objects, and logger objects for access across utilities.

1.2 benchy

sutools `benchy` instance is a global instance of the `bench_handler.Benchy` class. This instance is used as a decorator to collect benchmarking stats for selected functions.

CHAPTER

TWO

USAGE

2.1 register functions with sutools

Using the register decorator *@su.register* on your functions will register it with sutools *meta_handler*. Stored functions are available across tools. This registry is intended to be used by logger and cli utilities.

```
import sutools as su

@su.register
def add(x : int, y : int):
    '''add two integers'''
    return x + y
```

You can also register async functions, these will be executed using *asyncio.run()* given a valid coroutine function

```
import sutools as su
import asyncio

@su.register
async def delay_add(x : int, y : int):
    '''add two integers after 1 sec'''
    await asyncio.sleep(1)
    return x + y
```

NOTE: Adding type hinting to your functions enforces types in the cli and adds positional arg class identifiers in the help docs for the command.

2.2 cli - initialization standard

It is suggested to define the command line interface after *if __name__ == '__main__'*. Any code before the cli will run even if a cli command is used; code after the cli definition will not run when passing a cli command.

```
import sutools as su

# registered functions...

su.logger(*args, **kwargs) # optional

# module level function calls...
```

(continues on next page)

(continued from previous page)

```
if __name__ == '__main__':
    # main code (will run even when using cli commands)...
    su.cli(*args, **kwargs)
    # main code (will NOT run when using cli commands)...
```

NOTE: The CLI should be defined after the logger if you choose to use the two utilities in parallel.

2.3 cli - usage example

The logger utility should be instantiated after any registered functions but before any module level functions.

```
"""This module does random stuff."""
import sutools as su

@su.register
def meet(name : str, greeting : str = 'hello', farewell : str = 'goodbye') -> str:
    "meet a person"
    output = f'\n{greeting} {name}\n{farewell} {name}'
    su.log().meeting.info(output)
    return output

    su.logger() # optional

# module level function calls...

if __name__ == '__main__':
    # main code (will run even when using cli commands)...
    su.cli(desc = __doc__)
    # main code (will NOT run when using cli commands)...
```

NOTE: Adding type hinting to your functions enforces types in the cli and adds positional arg class identifiers in the help docs for the command.

command usage:

```
python module.py meet foo
```

output:

```
hello foo
goodbye foo
```

module help output:

```
usage: module [-h] {meet} ...

This module does random stuff.

options:
-h, --help  show this help message and exit
```

(continues on next page)

(continued from previous page)

```
commands:
{meet}
    meet      meet a person
```

command help output:

```
usage: module meet [-gr <class 'str'>] [-fa <class 'str'>] [-h] name

meet(name: str, greeting: str = 'hello', farewell: str = 'goodbye') -> str

positional arguments:
name                  <class 'str'>

options:
-gr <class 'str'>, --greeting <class 'str'>
                        default: hello
-fa <class 'str'>, --farewell <class 'str'>
                        default: goodbye
-h, --help            Show this help message and exit.
```

2.4 cli - using variadic functions

Variadic functions are compatible with sutools cli utility. When calling kwargs from the cli; *key*=*value* should be used instead of – and -, these are reserved for default arguments.

NOTE: since input is from *stdin* it will always be of type <*string*>, sutools will not infer the data type you must infer your needed type within the function.

```python
"""This module does random stuff."""
import sutools as su

@su.register
def variadic(*args, **kwargs):

    print("Positional arguments:")
    for arg in args:
        print(arg)

    print("Keyword arguments:")
    for key, value in kwargs.items():
        print(f"{key} = {value}")

    su.logger() # optional

# module level function calls...

if __name__ == '__main__':
    # main code (will run even when using cli commands)...
    su.cli(desc = __doc__)
    # main code (will NOT run when using cli commands)...
```

command usage:

```
python module.py variadic 1 2 3 foo=1 bar=2
```

output:

```
Positional arguments:
1
2
3
Keyword arguments:
foo = 1
bar = 2
```

2.5 logger - initialization standard

The logger utility should be instantiated after any registered functions but before any module level functions.

```
import sutools as su

# registered functions...

su.logger(*args, **kwargs)

# module level function calls...

if __name__ == '__main__':
    # main code (will run even when using cli commands)...
    su.cli(*args, **kwargs) # optional
    # main code (will NOT run when using cli commands)...
```

2.6 logger - usage examples

accessing defined loggers is done with a *log()* helper function. Note the use of *su.log()* in the below functions to access a specified logger before defining the log level and message.

using registered function names

```
import sutools as su

@su.register
def add(x : int, y : int):
    '''add two integers'''
    su.log().add.info(f'{x} + {y} = {x+y}')
    return x + y

@su.register
def subtract(x : int, y : int):
    '''subtract two integers'''
    su.log().subtract.info(f'{x} - {y} = {x-y}')
```

(continues on next page)

(continued from previous page)

```
    return x - y

su.logger() # logger definition

# module level function calls
add(1,2)
subtract(1,2)

if __name__ == '__main__':
    # main code (will run even when using cli commands)...
    su.cli() # optional
    # main code (will NOT run when using cli commands)...
```

log output

```
16:16:34, 961 add INFO 1 + 2 = 3
16:16:34, 961 subtract INFO 1 - 2 = -1
```

using custom logger names

```
import sutools as su

@su.register
def add(x : int, y : int):
    '''add two integers'''
    su.log().logger1.info(f'{x} + {y} = {x+y}')
    return x + y

@su.register
def subtract(x : int, y : int):
    '''subtract two integers'''
    su.log().logger2.info(f'{x} - {y} = {x-y}')
    return x - y

su.logger(loggers=['logger1','logger2']) # logger definition

# module level function calls
add(1,2)
subtract(1,2)

if __name__ == '__main__':
    # main code (will run even when using cli commands)...
    su.cli() # optional
    # main code (will NOT run when using cli commands)...
```

log output

```
16:16:34, 961 add INFO 1 + 2 = 3
16:16:34, 961 subtract INFO 1 - 2 = -1
```

2.7 benchy - usage example

The *benchy* decorator is designed to collect performance timing and call info for selected functions. This can be used in combination with *@su.register*, the decorators are order independent.

```
import sutools as su

@su.benchy
@su.register
def add(x : int, y : int):
    '''add two integers'''
    return x + y

@su.register
@su.benchy
def subtract(x : int, y : int):
    '''subtract two integers'''
    return x - y

@su.benchy
@su.register
def calc(x : int, y : int, atype : str = '+') -> int:
    '''caclulates a thing'''
    if atype == '+':
        res = add(x, y)
    elif atype == '-':
        res = subtract(x, y)
    return res

add(1,2)
add(2,2)
subtract(1,2)
calc(2,3, atype='-')
```

After the functions have been executed, the benchmark report can be accessed with *su.benchy.report*.

```
# print the benchmark report
print(su.benchy.report)
```

example output

```
{'add': [{'args': [{'type': 'int', 'value': 1}, {'type': 'int', 'value': 2}],
        'benchmark': 0.00015466799959540367,
        'kwargs': None,
        'result': {'type': 'int', 'value': 3}},
        {'args': [{'type': 'int', 'value': 2}, {'type': 'int', 'value': 2}],
        'benchmark': 6.068096263334155e-05,
        'kwargs': None,
        'result': {'type': 'int', 'value': 4}}],
'calc': [{'args': [{'type': 'int', 'value': 2}, {'type': 'int', 'value': 3}],
        'benchmark': 4.855601582676172e-05,
        'kwargs': {'atype': {'length': 1, 'type': 'str'}},
        'result': {'type': 'int', 'value': 5}}],
```

(continues on next page)

(continued from previous page)

```
'subtract': [{'args': [{'type': 'int', 'value': 1}, {'type': 'int', 'value': 2}],
        'benchmark': 5.205394700169563e-05,
        'kwargs': None,
        'result': {'type': 'int', 'value': -1}}]}
```

The output of the benchmark report will adhere to the following format. *function* > *call records*. Call records consist of *{args, kwargs, result, benchmark}* there will be a record for each call of a given function.

NOTE: given an iterable for *arg*, *kwarg*, or *result* the object will be summarized in terms of vector length.

```
{'function_name': [{'args': [{'type': 'arg_type', 'value': int}]
                    'benchmark': float,
                    'kwargs': {'kwarg_name': {'type': 'arg_type', 'length': int, }}
                    'result': {'type': 'arg_type', 'value': float}}]}
```

CHAPTER

THREE

HELPER CLASSES

class sutools.cli_handler.**CLI**(*desc*, *logs*, *log_obj=None*)

Bases: object

object designed for swift module CLI configuration

add_funcs(*func_dict*)

add registered functions to the cli

parse()

initialize parsing args

class sutools.log_handler.**Logger**(*name*, *loggers*, *loglvl*, *filename*, *filepath*, *filefmt*, *fhandler*, *filecap*, *filetimeout*, *file*, *streamfmt*, *shandler*, *stream*)

Bases: object

object designed for swift granular logging configuration

cap(*filecap*)

delete any file outside of range based on file age

out()

Check all loggers in the loggers namespace object for existing logs. If none exist, close the file fhandlers and remove the empty file

timeout(*filetimeout*)

delete any file outside given time range

class sutools.meta_handler.**Store**

Bases: object

internal object for storing function dictionary

add_cli(*cli_obj*)

adds a cli object to the store

add_func(*func*)

registers a function to the function dictionary

add_log(*log_obj*)

adds a logger object to the store

class sutools.bench_handler.**Benchy**

Bases: object

decorator class for collecting benchmark reports

func_meta(*data*)

collect args / kwargs meta info & summarize inputs

static summarize(*data*)

summarize iterable data

CHAPTER

FOUR

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

S

INDEX