stakk

Aaron Stopher

Aug 15, 2023

CONTENTS:

CHAPTER

ONE

USAGE

1.1 register functions with stakk

Using the register decorator *@stakk.register('stack_id')* on your functions will register it with a stack in *meta_handler.Stack*. Functions in a stack are available for utilities to register and create a link to the stack.

```
import stakk

@stakk.register('stack_id')
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 stakk
import asyncio

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

1.2 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. The cli initialization will require 1 argument a stack_id, optionally you can provide a description it is suggested to use doc strings for this and init as shown.

```
import stakk

# registered functions...

# module level function calls...

if __name__ == '__main__':
```

(continues on next page)

(continued from previous page)

```
    # main code (will run even when using cli commands)...
    stakk.cli(stack_id = 'stack_id', desc = __doc__) # desc is optional
    # main code (will NOT run when using cli commands)...
```

1.3 cli - args and optional args

Adding a default value will create an optional arg for the command. The keyword and default value will be displayed in the help docs along with the type if one is given.

```
"""This module does random stuff."""
import stakk

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

# module level function calls...

if __name__ == '__main__':
    # main code (will run even when using cli commands)...
    stakk.cli(stack_id = '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 that 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

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

command help output:

```
usage: module meet [-gr GREETING] [-fa FAREWELL] [-h] name

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

positional arguments:
name                  type: str

options:
-gr GREETING, --greeting GREETING
                        type: str, default: hello
-fa FAREWELL, --farewell FAREWELL
                        type: str, default: goodbye
-h, --help            Show this help message and exit.
```

1.4 cli - choice arguments

Adding an iterable as the type annotation will define a choices argument. A custom type checker is defined based on the option types in the iterable provided. This will allow you to define mixed types in your choices lists.

```python
"""This module does random stuff."""
import stakk

foo_choices = ['foo', 'fooo','foooo']
bar_choices = ('bar', 1, 'baar', 2)

@stakk.register('cli')
def foo_choices(foo: foo_choices, bar: bar_choices = 2) -> tuple:
    '''foo bar'''
    return foo, bar

# module level function calls...

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

command usage:

```
python module.py foo_choices fooo --bar 1
```

output:

```
('fooo', 1)
```

command help output:

```
usage: module foo_choices [-ba BAR] [-h] foo

foo_choices(foo: choice_type, bar: choice_type = 2) -> tuple
```

(continues on next page)

(continued from previous page)

```
positional arguments:
foo                 choices: (foo, fooo, foooo)

options:
-ba BAR, --bar BAR  choices: (bar, 1, baar, 2), default: 2
-h, --help          Show this help message and exit.
```

1.5 cli - list annotation

Using list as a type annotation has special context. This will prompt the cli to define a custom list type which returns *re.split(r'[;,|]', value)*. This allows you to specify string with delimiters which are converted to lists before being passed to the function. You are welcome to create and pass your own type functions but lists are built in for ease of use.

```python
"""This module does random stuff."""
import stakk

@stakk.register('cli')
def foo_lists(foo : list, bar : list = ['foo','bar']) -> tuple:
        return foo, bar

# module level function calls...

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

command usage:

```
python module.py foo_lists 'foo,bar;foo|bar foo'
```

output:

```
(['foo', 'bar', 'foo', 'bar', 'foo'], ['foo', 'bar'])
```

command help output:

```
usage: module foo_lists [-ba BAR] [-h] foo

foo_lists(foo: type_list, bar: type_list = ['foo', 'bar']) -> tuple

positional arguments:
foo                 type: list

options:
-ba BAR, --bar BAR  type: list, default: ['foo', 'bar']
-h, --help          Show this help message and exit.
```

1.6 cli - using variadic functions

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

NOTE: providing type annotations will enforce type, however this will apply to all **args* or ***kwargs*, if custom logic is needed you can create and pass a custom type function.

```python
"""This module does random stuff."""
import stakk

@stakk.register('cli')
def variadic(*args: str, **kwargs: int):

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

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

# module level function calls...

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

command usage:

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

output:

```
Positional arguments:
foo
bar
foo

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

command help output:

```
usage: dev variadic [-h] [*args ...] [**kwargs ...]

variadic(args, kwargs)

positional arguments:
*args       ex: command arg1 arg2
**kwargs    ex: command key=value
```

(continues on next page)

(continued from previous page)

```
options:
-h, --help  Show this help message and exit.
```

1.7 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 *@stakk.register*, the decorators are order independent.

```
import stakk
import asyncio

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

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

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

@stakk.register('test_stack')
@stakk.benchy
async def async_example():
    '''An example async function'''
    await asyncio.sleep(1)
    print("Async function completed.")

add(1,2)
add(2,2)
subtract(1,2)
calc(2,3, atype='-')
asyncio.get_event_loop().run_until_complete(async_example())
```

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

```
# print the benchmark report
print(stakk.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}}],
'subtract': [{'args': [{'type': 'int', 'value': 1}, {'type': 'int', 'value': 2}],
        'benchmark': 5.2053947001695 63e-05,
        'kwargs': None,
        'result': {'type': 'int', 'value': -1}}],
'async_example': [{'args': None,
        'benchmark': 1.001522845996078,
        'kwargs': None,
        'result': {'type': 'NoneType', 'value': None}}],
```

The output of the benchmark report will adhere to the following format. *function : call report*. Call reports 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

TWO

HELPER CLASSES

class stakk.meta_handler.**Stack**

Bases: object

internal object for storing function dictionary

add_cli(*cli_obj*)

adds a cli object to the stack

add_func(*stack: str*, *func*)

registers a function to the function dictionary

get_stack(*stack: str*) → dict

retrieve functions from specific stack

class stakk.cli_handler.**CLI**(*desc: str*)

Bases: object

object designed for swift module CLI configuration

add_funcs(*func_dict*)

add registered functions to the cli

static choice_type(*value*, *choices*)

custom type for checking types on provided choices.

static custom_partial(*func*, ***partial_kwargs*)

partial function wrapper which retains the original function's name and doc string.

parse()

initialize parsing args

static type_list(*value*)

custom type for list annotation

class stakk.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

THREE

INDICES AND TABLES

- genindex
- modindex
- search

INDEX