ewc-commons

Release 0.0.1.10

Evil Wizard Creations

Nov 24, 2021

CONTENTS:

1 EWC Commons Library **3**
- 1.1 Installing . . . 3
- 1.2 Getting Started . . . 3
- 1.3 How do I get set up? . . . 4
- 1.4 Sophie Module . . . 4
- 1.5 Dice Module . . . 6
- 1.6 Dice Rolls Module . . . 7
- 1.7 Card Deck Module . . . 7
- 1.8 Contribution guidelines . . . 8
- 1.9 Who do I talk to? . . . 8

2 ewccommons **9**
- 2.1 ewccommons package . . . 9

3 ewccommons **11**

4 Sophie Module **13**

5 Dice Module **17**

6 Dice Rolls Module **19**

7 Card Deck Module **21**

8 Indices and tables **23**

9 Code Examples **25**

Contents

- *EWC Commons Library*
  - *Installing*
  - *Getting Started*
  - *How do I get set up?*
  - *Sophie Module*
  - *Dice Module*
  - *Dice Rolls Module*
  - *Card Deck Module*
  - *Contribution guidelines*
  - *Who do I talk to?*

CHAPTER

ONE

EWC COMMONS LIBRARY

A collection of common & useful things to make other things easier.

1.1 Installing

Use pip to install the **EWC Commons Library**, which can also be used from the command line to trigger certain inbuilt functionality.

```
$ pip install ewc-commons[dev]
```

Veryfy the library installed and check the version or help.

```
$ ewccommons -v
...
$ ewccommons -h
```

1.2 Getting Started

- Sophie Module
- Dice Module
- Dice Rolls Module
- Card Deck Module

1.3 How do I get set up?

- Summary of set up
- Configuration
- Dependencies
- Database configuration
- How to run tests
- Deployment instructions

1.4 Sophie Module

The **sophie** module is used to provide convenient CLI script launcher framework

```
# Import the typing library so variables can be type cast
from typing import List

# Import the sys for args
import sys

# Import the Sophie launcher system needed
from ewccommons import sophie

def cli_argument_handler(
    opts: List,
    args: List,
    show_usage_help: sophie._TNGN_,
    show_version: sophie._TNGN_,
) -> sophie._CLI_Arg_Handled_:
    """Define a function handle the processing of cli arguments"""
    my_arg: str = None
    opt: str
    arg: str
    for opt, arg in opts:
        if opt in ("-h", "--help"):
            # Just show the usage help & halt
            show_version()
            show_usage_help()
            sys.exit()
        elif opt in ("-v", "--version"):
            # Just show the version & halt
            show_version()
            sys.exit()
        elif opt == "--my_arg":
            # Assign the my_arg value
            my_arg = arg
    return my_arg, args

def help_text() -> str:
```

(continues on next page)

(continued from previous page)

```
    """Define a function to generate the help text to display"""
    return "\n".join(
        [
            "Dynamically invoked help from show_usage_help() call",
            "  -h --help (Show this help)",
            "  -v --version (Show version)",
            "  --my_arg <my-arg-value>",
        ])

# With most of the setup done above, let's start creating the launcher
# Define the returned object variables for use
my_app: sophie.SophieLauncher = None
usage: sophie.CLIUsageHandler = None
# Create a new launcher based on the settings provided
my_app, usage = launcher(
    # This doesn't need to be an instance of the usage handler
    # Just a reference to a variable that matches the CLIUsageHandler
    # The handler is created internally based of the settings provided
    _usage=sophie.CLIUsageHandler,
    # Name the script
    app_name="My New Script",
    # Set the script version, defaults to the version of the sophie launcher
    version="0.0.1",
    # Set any short CLI argument option names.
    # These are a string of single character values for the option names
    # Characters immediately followed with a : (colon) will expect a value
    # E.G. f: would expect -f=<value> or -f <value>
    # Let's just use the inbuilt functionality to show help & version
    short_options="hv:",
    # Set any long CLI argument option names
    # Options with immediately followed with a = (equals) will expect a value
    # E.G. my_arg= would expect -my_arg=<value> or -my_arg <value>
    # Let's include the inbuilt functionality to show help & version
    # Let's also add a option of our own that expects user supplied value
    long_options=["help", "version", "my_arg="],
    # This is the script help text
    # Displayed when show_usage_help() is invoked from sophie._CLI_Arg_Handler_
    # Acessible as a string through the returned CLIUsageHandler usage object
    # Also accessible a string via SophieLauncher with app name & version included
    # N.B.
    # This doesn't have to be a callback as long as it can be converted to string
    help_text=help_text,
)
# Example of the help_text option in use
# my_app.show_usage_help()
# usage.help()

# Let's have a look at what we got
print(
    "Let's have a look:",
    "##############",
    "my_app",
```

(continues on next page)

(continued from previous page)

```
    my_app,
    "usage",
    usage,
    "##############",
    sep="\n",
)
# Define the usage & types of variables returned from the launch
my_arg: str
# Launch the script using the cli_argument_handler function to process the CLI options
my_arg, args = my_app.launch(cli_argument_handler)
# Lets have a look at what came back
print(my_arg, args, sep="\n")
```

1.5 Dice Module

The **dice** module provides an OOP solution for creating various sided die simulators and a conveniant way of rolling multiple (varying) Dice objects

```
from ewccommons.dice import Dice, DiceShaker

# Set a number of sides for the dice
number_of_die_sides: int = 4
# Create a n sided, named Dice object
dice: Dice = Dice(number_of_die_sides, name="My New Die", val=None)
# Have a look at what is created
print(
    f"Dice({number_of_die_sides})",
    dice,
    dice < number_of_die_sides,
    d > 0,
    sep="\n",
)
# Roll the dice and get the rolled value
dice_roll: int = dice.roll()
# Get the last rolled value
dice_rolled: int = dice.rolled
# The dice object can be compared like an int, using the rolled value
if dice == number_of_die_sides:
    # The dice object can be converted to string
    # This gives a readable version of the last rolled value
    print("Highest Roll", dice, sep="\n")
# The dice object also supports new object copy
# The copy will have the same number of sides/faces and dice name
# The copy will also have the starting value of the last rolled
dice2: Dice = dice.copy()
# Roll both dice to hopefully get 2 different numbers
# It is possible for both objects to independantly choose the same random value
dice.roll()
dice2.roll()
# Verify the 2 dice could be different in value
```

(continues on next page)

(continued from previous page)

```
print(dice, dice2, dice.rolled, dice2.rolled, sep="\n")

# TODO Add the DiceShaker example code
```

1.6 Dice Rolls Module

The **dicerolls** module can be used in multiple ways, firstly it can be used in the standard way of being imported into python scripts. **dicerolls** can also be invoked as a module script and will output the dice rolls for the dice specified. The same dice rolling functionality can also be invoked via the main **ewccommons** package executable.

```
$ python -m ewccommons.dicerolls d3 d4 d6 D8 D10 d12 d20 D100
...
$ ewccommons --dicerolls "d3 d4 d6 D8 D10 d12 d20 D100"
...
```

```
from ewccommons.dicerolls import roll_d6, D6

# Impliment the functional approach to rolling a standard Dn sided die
roll:int = roll_d6()
# Returns a new D6 die roll
# Impliment the OOP approach to rolling a standard Dn sided die
dice_roll:int = D6.roll()
# Returns the last rolled value of OOP Dn Dice instance
dice_rolled:int = D6.rolled()
```

1.7 Card Deck Module

The **carddeck** module provides a convenient means of creating card deck list which can be used as a basis for card based games.

```
from ewccommons.carddeck import (
    _Deck_,
    _Hand_,
    shuffle_deck,
    draw_card,
    new_deck,
    new_shuffled_deck,
)

deck: _Deck_ = new_deck()
shuffled_deck: _Deck_ = shuffle_deck(deck)
# alternitively create a new shuffled deck
shuffled_deck_alt: _Deck_ = new_shuffled_deck()

hand_size:int = 5
drawn: _Hand_
deck_remaining:_Deck_
drawn, deck_remaining = draw_card(deck=shuffled_deck, cards=hand_size)
```

1.8 Contribution guidelines

- Writing tests
- Code review
- Other guidelines

1.9 Who do I talk to?

- Repo owner or admin
- Other community or team contact

CHAPTER

TWO

EWCCOMMONS

2.1 ewccommons package

2.1.1 Subpackages

ewccommons.sophie package

Submodules

ewccommons.sophie.errors module

ewccommons.sophie.launcher module

ewccommons.sophie.processors module

ewccommons.sophie.usage module

Module contents

2.1.2 Submodules

2.1.3 ewccommons.carddeck module

2.1.4 ewccommons.dice module

2.1.5 ewccommons.dicerolls module

2.1.6 ewccommons.errors module

2.1.7 ewccommons.typing_extensions module

2.1.8 Module contents

CHAPTER

THREE

EWCCOMMONS

CHAPTER

FOUR

SOPHIE MODULE

The **sophie** module is used to provide convenient CLI script launcher framework

```
# Import the typing library so variables can be type cast
from typing import List

# Import the sys for args
import sys

# Import the Sophie launcher system needed
from ewccommons import sophie

def cli_argument_handler(
    opts: List,
    args: List,
    show_usage_help: sophie._TNGN_,
    show_version: sophie._TNGN_,
) -> sophie._CLI_Arg_Handled_:
    """Define a function handle the processing of cli arguments"""
    my_arg: str = None
    opt: str
    arg: str
    for opt, arg in opts:
        if opt in ("-h", "--help"):
            # Just show the usage help & halt
            show_version()
            show_usage_help()
            sys.exit()
        elif opt in ("-v", "--version"):
            # Just show the version & halt
            show_version()
            sys.exit()
        elif opt == "--my_arg":
            # Assign the my_arg value
            my_arg = arg
    return my_arg, args

def help_text() -> str:
    """Define a function to generate the help text to display"""
    return "\n".join(
        [
```

(continues on next page)

(continued from previous page)

```
            "Dynamically invoked help from show_usage_help() call",
            "  -h --help (Show this help)",
            "  -v --version (Show version)",
            "  --my_arg <my-arg-value>",
        ])

# With most of the setup done above, let's start creating the launcher
# Define the returned object variables for use
my_app: sophie.SophieLauncher = None
usage: sophie.CLIUsageHandler = None
# Create a new launcher based on the settings provided
my_app, usage = launcher(
    # This doesn't need to be an instance of the usage handler
    # Just a reference to a variable that matches the CLIUsageHandler
    # The handler is created internally based of the settings provided
    _usage=sophie.CLIUsageHandler,
    # Name the script
    app_name="My New Script",
    # Set the script version, defaults to the version of the sophie launcher
    version="0.0.1",
    # Set any short CLI argument option names.
    # These are a string of single character values for the option names
    # Characters immediately followed with a : (colon) will expect a value
    # E.G. f: would expect -f=<value> or -f <value>
    # Let's just use the inbuilt functionality to show help & version
    short_options="hv:",
    # Set any long CLI argument option names
    # Options with immediately followed with a = (equals) will expect a value
    # E.G. my_arg= would expect -my_arg=<value> or -my_arg <value>
    # Let's include the inbuilt functionality to show help & version
    # Let's also add a option of our own that expects user supplied value
    long_options=["help", "version", "my_arg="],
    # This is the script help text
    # Displayed when show_usage_help() is invoked from sophie._CLI_Arg_Handler_
    # Acessible as a string through the returned CLIUsageHandler usage object
    # Also accessible a string via SophieLauncher with app name & version included
    # N.B.
    # This doesn't have to be a callback as long as it can be converted to string
    help_text=help_text,
)
# Example of the help_text option in use
# my_app.show_usage_help()
# usage.help()

# Let's have a look at what we got
print(
    "Let's have a look:",
    "##############",
    "my_app",
    my_app,
    "usage",
    usage,
```

(continues on next page)

(continued from previous page)

```
    "##############",
    sep="\n",
)
# Define the usage & types of variables returned from the launch
my_arg: str
# Launch the script using the cli_argument_handler function to process the CLI options
my_arg, args = my_app.launch(cli_argument_handler)
# Lets have a look at what came back
print(my_arg, args, sep="\n")
```

CHAPTER

FIVE

DICE MODULE

The **dice** module provides an OOP solution for creating various sided die simulators and a conveniant way of rolling multiple (varying) Dice objects

```python
from ewccommons.dice import Dice, DiceShaker

# Set a number of sides for the dice
number_of_die_sides: int = 4
# Create a n sided, named Dice object
dice: Dice = Dice(number_of_die_sides, name="My New Die", val=None)
# Have a look at what is created
print(
    f"Dice({number_of_die_sides})",
    dice,
    dice < number_of_die_sides,
    d > 0,
    sep="\n",
)
# Roll the dice and get the rolled value
dice_roll: int = dice.roll()
# Get the last rolled value
dice_rolled: int = dice.rolled
# The dice object can be compared like an int, using the rolled value
if dice == number_of_die_sides:
    # The dice object can be converted to string
    # This gives a readable version of the last rolled value
    print("Highest Roll", dice, sep="\n")
# The dice object also supports new object copy
# The copy will have the same number of sides/faces and dice name
# The copy will also have the starting value of the last rolled
dice2: Dice = dice.copy()
# Roll both dice to hopefully get 2 different numbers
# It is possible for both objects to independantly choose the same random value
dice.roll()
dice2.roll()
# Verify the 2 dice could be different in value
print(dice, dice2, dice.rolled, dice2.rolled, sep="\n")

# TODO Add the DiceShaker example code
```

CHAPTER

SIX

DICE ROLLS MODULE

The **dicerolls** module can be used in multiple ways, firstly it can be used in the standard way of being imported into python scripts. **dicerolls** can also be invoked as a module script and will output the dice rolls for the dice specified. The same dice rolling functionality can also be invoked via the main **ewccommons** package executable.

```
$ python -m ewccommons.dicerolls d3 d4 d6 D8 D10 d12 d20 D100
...
$ ewccommons --dicerolls "d3 d4 d6 D8 D10 d12 d20 D100"
...
```

```
from ewccommons.dicerolls import roll_d6, D6

# Impliment the functional approach to rolling a standard Dn sided die
roll:int = roll_d6()
# Returns a new D6 die roll
# Impliment the OOP approach to rolling a standard Dn sided die
dice_roll:int = D6.roll()
# Returns the last rolled value of OOP Dn Dice instance
dice_rolled:int = D6.rolled()
```

CHAPTER

SEVEN

CARD DECK MODULE

The **carddeck** module provides a convenient means of creating card deck list which can be used as a basis for card based games.

```
from ewccommons.carddeck import (
    _Deck_,
    _Hand_,
    shuffle_deck,
    draw_card,
    new_deck,
    new_shuffled_deck,
)

deck: _Deck_ = new_deck()
shuffled_deck: _Deck_ = shuffle_deck(deck)
# alternitively create a new shuffled deck
shuffled_deck_alt: _Deck_ = new_shuffled_deck()

hand_size:int = 5
drawn: _Hand_
deck_remaining:_Deck_
drawn, deck_remaining = draw_card(deck=shuffled_deck, cards=hand_size)
```

CHAPTER

EIGHT

INDICES AND TABLES

- genindex
- modindex
- search

CHAPTER

NINE

CODE EXAMPLES

- *Sophie Module*
- *Dice Module*
- *Dice Rolls Module*
- *Card Deck Module*

Note: This project is under active development.