# scottbrian_utils

*Release 4.0.1*

**Scott Tuttle**

**May 15, 2024**

# API REFERENCE:

CHAPTER

ONE

# INTRO

This is a collection of generally useful functions for use with any application.

1. The diag_msg function allows you to print a message with the time and caller sequence added for you.
2. The doc_checker module provides an easy way to do a doctest.
3. The etrace decorator provide entry and exit tracing including passed args and returned values.
4. The FileCatalog item allows you to map file names to their paths.
5. The print_flower_box_msg function allows you to print text in a flower box (i.e., surrounded by asterisks).
6. The log_verifier allows you to verify that expected log messages have been issued.
7. The msgs item is a simple facility you can use in test cases to send messages between threads.
8. The Pauser class provides a pause function similar to the python sleep function, but with improved accuracy.
9. The stop_watch item is a simple timing function that you can use in test cases.
10. The @time_box decorator allows you to print start, stop, and execution times.
11. The timer item provides a way to keep track of time to determine when a function has timed out.
12. The UniqueTS class provides a way to obtain a unique timestamp.

CHAPTER

# TWO

# EXAMPLES:

With **diag_msg** you can print messages with the time and caller info added automatically.

**Example**
print a diagnostic message (<input> appears as the module name when run from the console)

```
>>> from scottbrian_utils.diag_msg import diag_msg
>>> diag_msg('this is a diagnostic message')
16:20:05.909260 <input>:1 this is a diagnostic message
```

With **FileCatalog**, you can code your application with file names and retrieve their paths at run time from a catalog. This allows you to use different catalogs for the same set of files, such as one catalog for production and another for testing. Here's as example:

```
>>> from scottbrian_utils.file_catalog import FileCatalog
>>> from pathlib import Path
>>> prod_cat = FileCatalog({'file1': Path('/prod_files/file1.csv')})
>>> print(prod_cat.get_path('file1'))
/prod_files/file1.csv
```

```
>>> test_cat = FileCatalog({'file1': Path('/test_files/test_file1.csv')})
>>> print(test_cat.get_path('file1'))
/test_files/test_file1.csv
```

With **@time_box**, you can decorate a function to be sandwiched between start time and end time messages like this:

```
>>> from scottbrian_utils.time_hdr import time_box
```

```
>>> @time_box
... def func2() -> None:
...     print('2 * 3 =', 2*3)
```

```
>>> func2()

********************************************
* Starting func2 on Mon Jun 29 2020 18:22:50 *
********************************************
2 * 3 = 6

******************************************
* Ending func2 on Mon Jun 29 2020 18:22:51 *
```

(continues on next page)

(continued from previous page)

```
* Elapsed time: 0:00:00.001204             *
*******************************************
```

With **Pauser**, you can pause execution for a specified number of seconds like this:

```
from scottbrian_utils.pauser import Pauser
pauser = Pauser()
pauser.pause(1.5)  # pause for 1.5 seconds
```

CHAPTER

THREE

# INSTALLATION

```
pip install scottbrian-utils
```

CHAPTER

FOUR

# DEVELOPMENT SETUP

See tox.ini

CHAPTER

FIVE

# RELEASE HISTORY

- **1.0.0**
  - Initial release
- **1.0.1**
  - Added doc link to setup.py
  - Added version number to __init__.py
  - Added code in setup.py to get version number from __init__.py
  - Added licence to setup.py classifiers
- **1.1.0**
  - Added FileCatalog
- **1.2.0**
  - Added diag_msg
- **2.0.0**
  - changed get_formatted_call_sequence and diag_msg (both in diag_msg.py) to get class name in additional cases
  - dropped support for python 3.6, 3.7, and 3.8
- **2.1.0**
  - added pauser
  - support for python 3.10
- **2.2.0**
  - added repr for LogVer
- **2.3.0**
  - added is_specified method in Timer
  - added timeout_value in Timer
  - support for python 3.11
- **2.4.0**
  - added fullmatch parm to add_msg in log_ver.py
  - added print_matched parm to print_match_results in log_ver.py
- **3.0.0**

  - added unique_ts
  - added doc_checker
  - support python 3.12
  - drop support python < 3.12

- **4.0.0**
  - added timedelta_match_string to time_hdr.py
  - added entry_trace.py
  - **restructured log_verifier:**
    - performance improvements
    - changes to clarify that regex patterns are used
    - changed report format
    - method add_pattern replaces deprecated method add_msg
    - method verify_match_results replaces deprecated verify_log_results
- **4.0.1**
  - fix etrace to put 2 colons between file name and func

CHAPTER

SIX

# META

Scott Tuttle

CHAPTER

SEVEN

# CONTRIBUTING

1. Fork it (<https://github.com/yourname/yourproject/fork>)
2. Create your feature branch (*git checkout -b feature/fooBar*)
3. Commit your changes (*git commit -am 'Add some fooBar'*)
4. Push to the branch (*git push origin feature/fooBar*)
5. Create a new Pull Request

diag_msg.py module.

## 7.1 diag_msg

With **diag_msg** you can print messages with the time and caller info added automatically. The default time format is H:M:S.f. The caller info includes the module name, class name (or null), method name (or null), and the line number relative to the start of the module.

**Example**

print a diagnostic message

```
>>> from scottbrian_utils.diag_msg import diag_msg
>>> diag_msg('this is a diagnostic message')
16:20:05.909260 <input>:1 this is a diagnostic message
```

Note that the examples are done as if entered in a python session from the console. As such, the module name will show as <input>. When coded in a module, however, you will see the module name instead of <input>.

**class** diag_msg.**CallerInfo**(*mod_name: str*, *cls_name: str*, *func_name: str*, *line_num: int*)

NamedTuple for the caller info used in diag_msg.

Create new instance of CallerInfo(mod_name, cls_name, func_name, line_num)

**cls_name: str**

Alias for field number 1

**func_name: str**

Alias for field number 2

**line_num: int**

Alias for field number 3

**mod_name: str**

Alias for field number 0

diag_msg.**diag_msg**(**args*, *depth=1*, *dt_format='%H:%M:%S.%f'*, ***kwargs*)

Print diagnostic message.

**Parameters**

- **args** (Any) – the text to print as part of the diagnostic message
- **depth** (int) – specifies how many callers to include in the call sequence
- **dt_format** (str) – datetime format to use
- **kwargs** (Any) – keyword args to pass along to the print statement

**Example**

print a diagnostic message from a method with a seq depth of 2

```
>>> from scottbrian_utils.diag_msg import diag_msg
>>> class Cls1:
...     @classmethod
...     def f1(cls, x):
...         # limit to two calls
:rtype: :sphinx_autodoc_typehints_type:`\:py\:obj\:\`None\``
...         diag_msg('diagnostic info', x, depth=2)
>>> Cls1.f1(42)
16:20:05.909260 <input>:1 -> <input>::Cls1.f1:5 diagnostic info 42
```

**Example**

print a diagnostic message with different datetime format

```
>>> from scottbrian_utils.diag_msg import diag_msg
>>> class Cls1:
...     def f1(self, x):
...         # use different datetime format
...         diag_msg('diagnostic info',
...                  x,
...                  dt_format='%a %b-%d %H:%M:%S')
>>> Cls1().f1(24)
Tue Feb-16 10:38:32 <input>::Cls1.f1:4 diagnostic info 24
```

diag_msg.**get_caller_info**(*frame*)

Return caller information from the given stack frame.

**Parameters**

**frame** (FrameType) – the frame from which to extract caller info

**Return type**

*CallerInfo*

**Returns**

The caller module name, class name (or null), function name (or null), and the line number within the module source

**Example**

get caller info for current frame

```
>>> from scottbrian_utils.diag_msg import get_caller_info
>>> import inspect
```

(continues on next page)

(continued from previous page)

```
>>> from os import fspath
>>> from pathlib import Path
>>> def f1():
...     a_frame = inspect.currentframe()
...     caller_info = get_caller_info(a_frame)
...     print(f'{caller_info.mod_name=}')
...     print(f'{caller_info.cls_name=}')
...     print(f'{caller_info.func_name=}')
...     print(f'{caller_info.line_num=}')
>>>
>>> f1()
caller_info.mod_name='<input>'
caller_info.cls_name=''
caller_info.func_name='f1'
caller_info.line_num=3
```

diag_msg.**get_formatted_call_sequence**(*latest=0*, *depth=3*)

Return a formatted string showing the callers.

**Parameters**

- **latest** (int) – specifies the stack position of the most recent caller to be included in the call sequence
- **depth** (int) – specifies how many callers to include in the call sequence

**Return type**

str

**Returns**

Formatted string showing for each caller the module name, possibly a function name or a class name/method_name pair, and the source code line number. There are three basic scenarios:

- A call from a script will appear as: mod_name:lineno
- A call from a function will appear as: mod_name::func_name:lineno
- A call from a class method will appear as:: mod_name::cls_name.func_name:lineno

This function is useful if, for example, you want to include the call sequence in a log.

**Example**

get call sequence for three callers

```
>>> from scottbrian_utils.diag_msg import (
...     get_formatted_call_sequence)
>>> def f1():
...     # f1 now on stack
...     # call f2
...     f2()
>>> def f2():
...     # call f3
...     f3()
>>> def f3():
...     # f3 now latest entry in call sequence
...     # default is to get three most recent calls
```

(continues on next page)

(continued from previous page)

```
...     print(get_formatted_call_sequence())
>>> f1()
<input>::f1:4 -> <input>::f2:3 -> <input>::f3:4
```

Note that when coded in a module, you will get the module name in the sequence instead of <input>, and the line numbers will be relative from the start of the module instead of from each of the function definition sections.

**Example**

get call sequence for last two callers

```
>>> from scottbrian_utils.diag_msg import (
...     get_formatted_call_sequence)
>>> def f1():
...     # f1 now on stack
...     # call f2
...     f2()
>>> def f2():
...     # call f3
...     f3()
>>> def f3():
...     # f3 now latest entry in call sequence
...     # specify depth to get two most recent calls
...     call_seq = get_formatted_call_sequence(depth=2)
...     print(call_seq)
>>> f1()
<input>::f2:3 -> <input>::f3:4
```

**Example**

get call sequence for two callers, one caller back

```
>>> from scottbrian_utils.diag_msg import (
...     get_formatted_call_sequence)
>>> def f1():
...     # f1 now on stack
...     # call f2
...     f2()
>>> def f2():
...     # call f3
...     f3()
>>> def f3():
...     # f3 now latest entry in call sequence
...     # specify latest to go back 1 and thus ignore f3
...     # specify depth to get two calls from latest going back
...     call_seq = get_formatted_call_sequence(latest=1, depth=2)
...     print(call_seq)
>>> f1()
<input>::f1:4 -> <input>::f2:3
```

**Example**

get sequence for script call to class method

```
>>> from scottbrian_utils.diag_msg import (
...     get_formatted_call_sequence)
>>> class Cls1:
...     def f1(self):
...         # limit to two calls
...         call_seq = get_formatted_call_sequence(depth=2)
...         print(call_seq)
>>> a_cls1 = Cls1()
>>> a_cls1.f1()
<input>:1 -> <input>::Cls1.f1:4
```

Module doc_checker.

## 7.2 DocChecker

The DocChecker class is used to help verify the documentation code examples. It builds upon Sybil and provides the ability to adjust the doc examples so that they verify correctly. If, for instance, a code example involves a timestamp, the expected output written at the time of the example will fail to match the timestamp generated when the example is tested. Example 1 below shows a way to make the adjustment to the timestamp so that it will match.

The DocChecker also has a way to print messages for troubleshooting cases that fail to verify. In example 1 below, the line

```
self.msgs.append([old_want, want, old_got, got])
```

will show the input and output values to help solve problems. Using messages is optional and can be customized for any purpose.

**Example 1**

This example shows how to make an adjustment to accommodate doc examples in a module called time_hdr that have timestamps in the output. The code needs to replace the timestamps in the 'want' variable so that it will match the timestamp from running the example at the time of the doctest. Place the following code into a conftest.py file in the project top directory (see scottbrian-utils for an example).

```
from doctest import ELLIPSIS
from doctest import OutputChecker as BaseOutputChecker

import re

from sybil import Sybil
from sybil.parsers.rest import PythonCodeBlockParser

from scottbrian_utils.time_hdr import get_datetime_match_string
from scottbrian_utils.doc_checker import DocCheckerTestParser

from typing import Any

class SbtDocCheckerOutputChecker(DocCheckerOutputChecker):
    def __init__(self) -> None:
        super().__init__()
```

(continues on next page)

(continued from previous page)

```
    def check_output(self, want, got, optionflags):
        old_want = want
        old_got = got

        def repl_dt(match_obj: Any) -> str:
            return found_items.__next__().group()

        if self.mod_name == 'time_hdr' or self.mod_name == 'README':
            # find the actual occurrences and replace in want
            for time_hdr_dt_format in ["%a %b %d %Y %H:%M:%S",
                                       "%m/%d/%y %H:%M:%S"]:
                match_str = get_datetime_match_string(
                                time_hdr_dt_format)

                match_re = re.compile(match_str)
                found_items = match_re.finditer(got)
                want = match_re.sub(repl_dt, want)

            # replace elapsed time in both want and got
            match_str = 'Elapsed time: 0:00:00.[0-9| ]{6,6}'
            replacement = 'Elapsed time: 0:00:00       '
            want = re.sub(match_str, replacement, want)
            got = re.sub(match_str, replacement, got)

        self.msgs.append([old_want, want, old_got, got])
        return super().check_output(want, got, optionflags)


pytest_collect_file = Sybil(
    parsers=[
        DocCheckerTestParser(optionflags=ELLIPSIS,
                             doc_checker_output_checker=SbtDocCheckerOutputChecker()
                             ),
        PythonCodeBlockParser(),],
    patterns=['*.rst', '*.py'],
    ).pytest()
```

**Example 2**

For standard doctest checking with no special cases, place the following code into a conftest.py file in the project top directory (see scottbrian-locking for an example).

```
from doctest import ELLIPSIS

from sybil import Sybil
from sybil.parsers.rest import PythonCodeBlockParser

from scottbrian_utils.doc_checker import DocCheckerTestParser


pytest_collect_file = Sybil(
    parsers=[
        DocCheckerTestParser(optionflags=ELLIPSIS,
```

(continues on next page)

(continued from previous page)

```
                              ),
        PythonCodeBlockParser(),],
    patterns=['*.rst', '*.py'],
    ).pytest()
```

**class** doc_checker.**DocCheckerOutputChecker**

Initialize the output checker object.

**check_output**(*want*, *got*, *optionflags*)

Check the output of the example against expected value.

**Parameters**

- **want** (str) – the expected value of the example output
- **got** (str) – the actual value of the example output
- **optionflags** (int) – doctest option flags for the Sybil BaseOutPutChecker check_output method

**Return type**
bool

**Returns**
True if the want and got values match, False otherwise

**class** doc_checker.**DocCheckerTestEvaluator**(*doc_checker_output_checker*, *optionflags=0*)

Initialize the evaluator object.

**Parameters**

- **doc_checker_output_checker** (*DocCheckerOutputChecker*) – invocation of the output check to use
- **optionflags** (int) – flags passed along to the Sybil DocTestEvaluator

Module entry_trace.

## 7.3 entry_trace

The etrace decorator can be used on a function or method to add a debug log item upon entry and exit. The entry trace log item will include the filename, function or method name, the line number where it is defined, and the specified args and/or kwargs. The exit trace will include the return value.

The decorator can be controlled via the following parameters:

1) enable_trace: boolean value that when True will enable the trace. The default is True.
2) omit_parms: list of parameter names whose argument values should appear in the trace as ellipses. This can help reduce the size of the trace entry for large arguments. The default is None.
3) omit_return_value: if True, do not trace the return value in the exit trace entry. The default is False.

**Example 1**
Decorate a function with no args nor kwargs.

```
from scottbrian_utils.entry_trace import etrace

@etrace
def f1() -> None:
    pass

f1()
```

Expected trace output for Example 1:

```
test_entry_trace.py::f1:69 entry: caller: test_entry_trace.py::TestEntryTraceExamples.
↪test_etrace_example1:77
test_entry_trace.py::f1:69 exit: return_value=None
```

**Example 2**

Decorate a function that has 1 positional arg and 1 keyword arg.

```
from scottbrian_utils.entry_trace import etrace

@etrace
def f1(a1: int, kw1: str = "42") -> str:
    return f"{a1=}, {kw1=}"

f1(42, kw1="forty two")
```

Expected trace output for Example 2:

```
test_entry_trace.py::f1:122 entry: a1=42, kw1='forty two', caller: test_entry_trace.
↪py::TestEntryTraceExamples.test_etrace_example2:130
test_entry_trace.py::f1:122 exit: return_value="a1=42, kw1='forty two'"
```

**Example 3**

Decorate two functions, the first with etrace enabled and the second with etrace disabled.

```
from scottbrian_utils.entry_trace import etrace

do_trace: bool = True

@etrace(enable_trace=do_trace)
def f1(a1: int, kw1: str = "42") -> str:
    return f"{a1=}, {kw1=}"

do_trace: bool = False

@etrace(enable_trace=do_trace)
def f2(a1: int, kw1: str = "42"):
    return f"{a1=}, {kw1=}"

f1(42, kw1="forty two")
f2(24, kw1="twenty four")
```

Expected trace output for Example 3:

```
test_entry_trace.py::f1:180 entry: a1=42, kw1='forty two', caller: test_entry_trace.
↪py::TestEntryTraceExamples.test_etrace_example3:194
test_entry_trace.py::f1:180 exit: return_value="a1=42, kw1='forty two'"
```

**Example 4**

Decorate a function with the positional arg omitted.

```
from scottbrian_utils.entry_trace import etrace

@etrace(omit_parms=["a1"])
def f1(a1: int, kw1: str = "42") -> str:
    return f"{a1=}, {kw1=}"

f1(42, kw1="forty two")
```

Expected trace output for Example 4:

```
test_entry_trace.py::f1:244 entry: a1='...', kw1='forty two', caller: test_entry_trace.
↪py::TestEntryTraceExamples.test_etrace_example4:252
test_entry_trace.py::f1:244 exit: return_value="a1=42, kw1='forty two'"
```

**Example 5**

Decorate a function with the first keyword arg omitted.

```
from scottbrian_utils.entry_trace import etrace

@etrace(omit_parms="kw1")
def f1(a1: int, kw1: str = "42", kw2: int = 24) -> str:
    return f"{a1=}, {kw1=}, {kw2=}"

f1(42, kw1="forty two", kw2=84)
```

Expected trace output for Example 5:

```
test_entry_trace.py::f1:300 entry: a1=42, kw1='...', kw2=84, caller: test_entry_trace.
↪py::TestEntryTraceExamples.test_etrace_example5:308
test_entry_trace.py::f1:300 exit: return_value="a1=42, kw1='forty two', kw2=84"
```

**Example 6**

Decorate a function with the return value omitted.

```
from scottbrian_utils.entry_trace import etrace

@etrace(omit_return_value=True)
def f1(a1: int, kw1: str = "42", kw2: int = 24) -> str:
    return f"{a1=}, {kw1=}, {kw2=}"

f1(42, kw1="forty two", kw2=84)
```

Expected trace output for Example 6:

```
  test_entry_trace.py::f1:347 entry: a1=42, kw1='forty two', kw2=84, caller: test_entry_
↪trace.py::TestEntryTraceExamples.test_etrace_example6:356
  test_entry_trace.py::f1:347 exit: return value omitted
```

(continues on next page)

(continued from previous page)

```
.. # noqa: E501, W505
```

entry_trace.**etrace**(*wrapped=None*, *, *enable_trace=True*, *omit_parms=None*, *omit_return_value=False*, *latest=1*, *depth=1*)

Decorator to produce entry/exit log.

**Parameters**

- **wrapped** (Optional[TypeVar(F, bound= Callable[..., Any])]) – function to be decorated
- **enable_trace** (Union[bool, Callable[..., bool]]) – if True, trace the entry and exit for the decorated function or method.
- **omit_parms** (Optional[Iterable[str]]) – list of parameter names whose argument values should appear in the trace as ellipses. This can help reduce the size of the trace entry for large arguments.
- **omit_return_value** (bool) – if True, do not place the return value into the exit trace entry.
- **latest** (int) – specifies the position in the call sequence that is to be designated as the caller named in the trace output. A value of 1, the default, specifies that the caller is one call back in the sequence and is the normal case. A value greater than 1 is useful when decorators are stacked and the caller of interest is thus further back in the sequence.
- **depth** (int) – specifies the depth of the call sequence to include in the trace output. A value of 1, the default, species that only the latest caller is to be included. Values greater than 1 will include the latest caller and its callers.

**Return type**
TypeVar(F, bound= Callable[..., Any])

**Returns**
funtools partial (when wrapped is None) or decorated function

### Notes

1) In both the entry and exit trace, the line number following the decorated function or method will be the line number of the etrace decorator. The trace message itself, however, will include the name of the traced function or method along with the line number where it is defined.
2) The positional and keyword arguments (if any) will appear after "entry:".
3) The caller of the traced function or method will appear after "caller:" and will include the line number of the call.
4) The exit trace will include the return value unless *omit_return_value* specifies True.

Module file_catalog.

## 7.4 FileCatalog

With **file_catalog**, you can set up a mapping of file names to their paths. An application can then use the catalog to retrieve the paths based on the file name. By keeping different catalogs with the same file names but different paths, different runs of the application can use each of the different catalogs as appropriate. For example, one catalog could be used for testing purposes and another for normal production.

**Example**

instantiate production and test catalogs for one file

```
>>> from scottbrian_utils.file_catalog import FileCatalog
>>> prod_cat = FileCatalog({'file1': Path('/prod_files/file1.csv')})
>>> print(prod_cat.get_path('file1'))
/prod_files/file1.csv
```

```
>>> test_cat = FileCatalog(
...     {'file1': Path('/test_files/test_file1.csv')})
>>> print(test_cat.get_path('file1'))
/test_files/test_file1.csv
```

**Example**

instantiate a catalog for two files:

```
>>> a_cat = FileCatalog({'sales': Path('/home/T/files/file1.csv'),
...                      'inventory': Path('/home/T/files/file2.csv')})
>>> print(a_cat)
FileCatalog({'sales': Path('/home/T/files/file1.csv'),
             'inventory': Path('/home/T/files/file2.csv')})
```

```
>>> print(a_cat.get_path('inventory'))
/home/T/files/file2.csv
```

```
>>> from os import fspath
>>> print(fspath(a_cat.get_path('sales')))
/home/T/files/file1.csv
```

The file_catalog module contains:

1) FileCatalog class with add_paths, del_paths, get_path, save_catalog, and load_catalog methods
2) FileSpec, FileSpecs type aliases that you can use for type hints
3) Error exception classes:
   a. FileNameNotFound
   b. FileSpecIncorrect
   c. IllegalAddAttempt
   d. IllegalDelAtempt

**class** file_catalog.**FileCatalog**(*file_specs=None*)

Provides a mapping of file names to paths.

This is useful for cases where an application is to be used in various environments with files that are in different places. Another use is where one set of files is used for normal processing and another set is used for testing purposes.

Store the input file specs to a data frame.

**Parameters**

**file_specs** (Optional[Dict[str, Path]]) – A dictionary of one or more entries. The key is the file name and the value is the path. The file name must be a sting and the path must be a pathlib Path.

**Example**

instantiate a catalog with two files

```
>>> from scottbrian_utils.file_catalog import FileCatalog
>>> a_catalog = FileCatalog(
...     {'file_1': Path('/run/media/file1.csv'),
...      'file_2': Path('/run/media/file2.pdf')})
>>> print(a_catalog.get_path('file_2'))
/run/media/file2.pdf
```

**add_paths**(*file_specs*)

Add one or more paths to the catalog.

**Parameters**

**file_specs** (Dict[str, Path]) – A dictionary of one or more entries. The key is the file name and the value is the path. The file name must be a sting and the path must be a pathlib Path.

**Raises**

- **FileSpecIncorrect** – The input path is not a string
- **IllegalAddAttempt** – Entry already exists with different path

**Return type**

None

The entries to be added are specified in the file_specs argument. For each file_spec, the specified file name is used to determine whether the entry already exists in the catalog. If the entry already exists, the specified path is compared against the found entry. If they do not match, an IllegalAddAttempt exception is raised and no entries for the add_paths request will be added. Otherwise, if the path matches, there is no need to add it again so processing continues with the next file_spec. If no errors are detected for any of the file_specs, any file names that do not yet exist in the catalog are added.

**Example**

add some paths to the catalog

```
>>> from scottbrian_utils.file_catalog import FileCatalog
>>> from pathlib import Path
>>> a_catalog = FileCatalog()
>>> a_catalog.add_paths({'file1': Path('/run/media/file1.csv')})
>>> print(a_catalog)
FileCatalog({'file1': Path('/run/media/file1.csv')})
>>> a_catalog.add_paths({'file2': Path('/run/media/file2.csv'),
...                      'file3': Path('path3')})
>>> print(a_catalog)
FileCatalog({'file1': Path('/run/media/file1.csv'),
             'file2': Path('/run/media/file2.csv'),
             'file3': Path('path3')})
```

**del_paths**(*file_specs*)

Delete one or more paths from the catalog.

**Parameters**
**file_specs** (Dict[str, Path]) – A dictionary of one or more entries. The key is the file name and the value is the path. The file name must be a sting and the path must be a pathlib Path.

**Raises**

- **FileSpecIncorrect** – The input path is not a string
- **IllegalDelAttempt** – Attempt to delete entry with different path

**Return type**
None

The entries to be deleted are specified in the file_specs argument. For each file_spec, the specified file name is used to find the entry in the catalog. If not found, processing continues with the next file_spec. Otherwise, if the entry is found, the specified path from the file_spec is compared against the path in the found entry. If not equal, an IllegalDelAttempt exception is raised and no entries for the del_paths request will be deleted. Otherwise, if the path matches, the entry will be deleted provided no errors are detected for any of the preceeding or remaining file_specs.

**Example**
add and then delete paths from the catalog

```
>>> from scottbrian_utils.file_catalog import FileCatalog
>>> a_catalog = FileCatalog()
>>> a_catalog.add_paths({'file1': Path('/run/media/file1.csv'),
...                      'file2': Path('/run/media/file2.csv'),
...                      'file3': Path('path3'),
...                      'file4': Path('path4')})
>>> print(a_catalog)
FileCatalog({'file1': Path('/run/media/file1.csv'),
             'file2': Path('/run/media/file2.csv'),
             'file3': Path('path3'),
             'file4': Path('path4')})
```

```
>>> a_catalog.del_paths({'file1': Path('/run/media/file1.csv'),
...                      'file3': Path('path3')})
>>> print(a_catalog)
FileCatalog({'file2': Path('/run/media/file2.csv'),
             'file4': Path('path4')})
```

**get_path**(*file_name*)

Obtain a path given a file name.

**Parameters**
**file_name** (str) – The name of the file whose path is needed

**Return type**
Path

**Returns**
A pathlib Path object for the input file name

**Raises**
**FileNameNotFound** – The input file name is not in the catalog

**Example**
instantiate a catalog with two files and get their paths

```
>>> from scottbrian_utils.file_catalog import FileCatalog
>>> a_catalog = FileCatalog(
...     {'file1': Path('/run/media/file1.csv'),
...      'file2': Path('/run/media/file2.pdf')})
>>> path1 = a_catalog.get_path('file1')
>>> print(path1)
/run/media/file1.csv
```

```
>>> from os import fspath
>>> fspath(a_catalog.get_path('file2'))
'/run/media/file2.pdf'
```

**classmethod load_catalog**(*saved_cat_path*)

Load catalog from a csv file.

**Parameters**
**saved_cat_path** (Path) – The path from where the catalog is to be loaded

**Return type**
*FileCatalog*

**Returns**
A FileCatalog instance

**save_catalog**(*saved_cat_path*)

Save catalog as a csv file.

**Parameters**
**saved_cat_path** (Path) – The path to where the catalog is to be saved

**Return type**
None

flower_box.py module.

## 7.5 flower_box

With **print_flower_box_msg** you can print messages in a flower_box, meaning a box of asterisks.

**Example**
print a one line message in a flower box

```
>>> from scottbrian_utils.flower_box import print_flower_box_msg
>>> print_flower_box_msg('This is my message to the world')

**********************************
* This is my message to the world *
**********************************
```

flower_box.**print_flower_box_msg**(*msgs*, ***kwargs*)

Print a single or multi-line message inside a flower box.

**Parameters**

- **msgs** (Union[str, List[str]]) – single message or list of messages to print
- **kwargs** (Any) – Specifies the print arguments to use on the print statement, such as *end*, *file*, or *flush*.

**Example**
print a two line message in a flower box

**Return type**
None

```
>>> from scottbrian_utils.flower_box import print_flower_box_msg
```

```
>>> msg_list = ['This is my first line test message',
...             'and my second line']
>>> print_flower_box_msg(msg_list)

****************************************
* This is my first line test message *
* and my second line                 *
****************************************
```

Module log_verifier.

## 7.6 LogVer

The LogVer class is intended to be used during testing to allow a pytest test case to verify that the code under test issued log messages as expected. This is done by providing a collection of regex patterns that the LogVer will match up to the issued log messages. A report is printed to the syslog to show any patterns or messages that failed to match, and optionally the log messages that did match.

**Example1**
pytest test case logs a message and verifies

```
from scottbrian_utils.log_verifier import LogVer
import logging
def test_example1(caplog: pytest.LogCaptureFixture) -> None:
    t_logger = logging.getLogger("example_1")
    log_ver = LogVer(log_name="example_1")
    log_msg = "hello"
    log_ver.add_pattern(pattern=log_msg)
    t_logger.debug(log_msg)
    match_results = log_ver.get_match_results(caplog)
    log_ver.print_match_results(match_results, print_matched=True)
    log_ver.verify_match_results(match_results)
```

The output from `LogVer.print_match_results()` for test_example1:

```
************************************************
*             log verifier results             *
************************************************
Start: Thu Apr 11 2024 19:24:28
End: Thu Apr 11 2024 19:24:28
Elapsed time: 0:00:00.006002
```

(continues on next page)

(continued from previous page)

```
************************************************
*                 summary stats                *
************************************************
    type  records  matched  unmatched
patterns        1        1          0
log_msgs        1        1          0

************************
* unmatched patterns: *
************************
*** no unmatched patterns found ***

************************
* unmatched log_msgs: *
************************
*** no unmatched log messages found ***

************************
*  matched log_msgs:  *
************************
log_name  level log_msg records matched unmatched
example_1    10 hello         1       1         0
```

**Example2**

pytest test case expects two log records, only one is issued

```
from scottbrian_utils.log_verifier import LogVer
import logging

def test_example2(caplog: pytest.LogCaptureFixture) -> None:
    t_logger = logging.getLogger("example_2")
    log_ver = LogVer(log_name="example_2")
    log_msg1 = "hello"
    log_ver.add_pattern(pattern=log_msg1)
    log_msg2 = "goodbye"
    log_ver.add_pattern(pattern=log_msg2)
    t_logger.debug(log_msg1)
    match_results = log_ver.get_match_results(caplog)
    log_ver.print_match_results(match_results, print_matched=True)
    with pytest.raises(UnmatchedPatterns):
        log_ver.verify_match_results(match_results)
```

The output from `LogVer.print_match_results()` for test_example2:

```
************************************************
*             log verifier results             *
************************************************
Start: Thu Apr 11 2024 19:24:28
End: Thu Apr 11 2024 19:24:28
Elapsed time: 0:00:00.006002
```

(continues on next page)

(continued from previous page)

```
************************************************
*                 summary stats                *
************************************************
    type  records  matched  unmatched
patterns        2        1          1
log_msgs        1        1          0

************************
* unmatched patterns: *
************************
log_name  level pattern fullmatch records matched unmatched
example_2    10 goodbye True            1       0         1

************************
* unmatched log_msgs: *
************************
*** no unmatched log messages found ***

************************
*  matched log_msgs:  *
************************
log_name  level log_msg records matched unmatched
example_2    10 hello         1       1         0
```

**Example3**

pytest test case expects one log record, two were issued

```
from scottbrian_utils.log_verifier import LogVer
import logging
def test_example3(caplog: pytest.LogCaptureFixture) -> None:
    t_logger = logging.getLogger("example_3")
    log_ver = LogVer(log_name="example_3")
    log_msg1 = "hello"
    log_ver.add_pattern(pattern=log_msg1)
    log_msg2 = "goodbye"
    t_logger.debug(log_msg1)
    t_logger.debug(log_msg2)
    log_ver.print_match_results(
        match_results := log_ver.get_match_results(caplog),
        print_matched=True
    )
    with pytest.raises(UnmatchedLogMessages):
        log_ver.verify_match_results(match_results)
```

The output from `LogVer.print_match_results()` for test_example3:

```
************************************************
*              log verifier results            *
************************************************
Start: Thu Apr 11 2024 19:24:28
End: Thu Apr 11 2024 19:24:28
Elapsed time: 0:00:00.006002
```

(continues on next page)

(continued from previous page)

```
************************************************
*                 summary stats                *
************************************************
    type  records  matched  unmatched
patterns        1        1          0
log_msgs        2        1          1

************************
* unmatched patterns: *
************************
*** no unmatched patterns found ***

************************
* unmatched log_msgs: *
************************
log_name  level log_msg records matched unmatched
example_3    10 goodbye       1       0         1

************************
*  matched log_msgs:  *
************************
log_name  level log_msg records matched unmatched
example_3    10 hello         1       1         0
```

**Example4**

pytest test case expect two log records, two were issued, one different

```
from scottbrian_utils.log_verifier import LogVer
import logging
def test_example4(caplog: pytest.LogCaptureFixture) -> None:
    t_logger = logging.getLogger("example_4")
    log_ver = LogVer(log_name="example_4")
    log_msg1 = "hello"
    log_ver.add_pattern(pattern=log_msg1)
    log_msg2a = "goodbye"
    log_ver.add_pattern(pattern=log_msg2a)
    log_msg2b = "see you soon"
    t_logger.debug(log_msg1)
    t_logger.debug(log_msg2b)
    log_ver.print_match_results(
        match_results := log_ver.get_match_results(caplog),
        print_matched=True
    )
    with pytest.raises(UnmatchedPatterns):
        log_ver.verify_match_results(match_results)
```

The output from `LogVer.print_match_results()` for test_example4:

```
************************************************
*             log verifier results             *
************************************************
```

(continues on next page)

(continued from previous page)

```
Start: Thu Apr 11 2024 19:24:28
End: Thu Apr 11 2024 19:24:28
Elapsed time: 0:00:00.006002

************************************************
*                 summary stats                *
************************************************
    type  records  matched  unmatched
patterns        2        1          1
log_msgs        2        1          1

************************
* unmatched patterns: *
************************
log_name  level pattern fullmatch records matched unmatched
example_4    10 goodbye True            1       0         1

************************
* unmatched log_msgs: *
************************
log_name  level log_msg      records matched unmatched
example_4    10 see you soon       1       0         1

************************
*  matched log_msgs:  *
************************
log_name  level log_msg records matched unmatched
example_4    10 hello         1       1         0
```

The log_verifier module contains:

1) LogVer class with methods:
    a. add_call_seq
    b. get_call_seq
    c. add_pattern
    d. get_match_results
    e. print_match_results
    f. verify_match_results

**class** log_verifier.**LogVer**(*log_name='root'*, *str_col_width=None*)

Log Message Verification Class.

Initialize a LogVer object.

**Parameters**

- **log_name** (str) – name of the logger
- **str_col_width** (Optional[int]) – If specifief, limits the maximum width for string values in the display produced by method print_match_results. The string values that are limited are for columns *log_name*, *log_msg*, *pattern*, and *fullmatch*. The specified limit must be an int with a value of 9 or greater.

Example: create a logger and a LogVer instance >>> logger = logging.getLogger('example_logger') >>> log_ver = LogVer('example_logger')

**add_call_seq**(*name*, *seq*)

Add a call sequence for a given name.

**Parameters**

- **name** (str) – name for whom the call sequence represents
- **seq** (str) – the call sequence in a format as described by get_formatted_call_sequence in diag_msg.py from the scottbrian_utils package

**Return type**
None

**add_msg**(*log_msg*, *log_level=10*, *log_name=None*, *fullmatch=False*)

Add a message to the expected log messages.

**Parameters**

- **log_msg** (str) – expected message to add
- **log_level** (int) – expected logging level
- **log_name** (Optional[str]) – expected logger name
- **fullmatch** (bool) – if True, use regex fullmatch instead of match in method get_match_results

**Return type**
None

Deprecated since version 3.0.0: Use method *add_pattern()* instead.

**add_pattern**(*pattern*, *level=10*, *log_name=None*, *fullmatch=True*)

Add a pattern to be matched to a log message.

**Parameters**

- **pattern** (str) – pattern to use to find log_msg in the log
- **level** (int) – logging level to use
- **log_name** (Optional[str]) – logger name to use
- **fullmatch** (bool) – if True, use regex fullmatch in method get_match_results, otherwise use regex match

**Return type**
None

Added in version 3.0.0: Method *add_pattern()* replaces method *add_msg()*.

Example: add two patterns, each at a different level

```
def test_example(caplog: pytest.LogCaptureFixture) -> None:
    t_logger = logging.getLogger("example_5")
    log_ver = LogVer("example_5")
    log_msg1 = "hello"
    log_msg2 = "goodbye"
    log_ver.add_pattern(pattern=log_msg1)
    log_ver.add_pattern(pattern=log_msg2,
                        level=logging.ERROR)
```

(continues on next page)

(continued from previous page)

```
    t_logger.debug(log_msg1)
    t_logger.error(log_msg2)
    match_results = log_ver.get_match_results(caplog=caplog)
    log_ver.print_match_results(match_results,
                                print_matched=True)
    log_ver.verify_match_results(match_results)
```

The output from `LogVer.print_match_results()` for test_example:

```
************************************************
*              log verifier results            *
************************************************
Start: Thu Apr 11 2024 19:24:28
End: Thu Apr 11 2024 19:24:28
Elapsed time: 0:00:00.006002

************************************************
*                 summary stats                *
************************************************
    type  records  matched  unmatched
patterns        2        2          0
log_msgs        2        2          0

************************
* unmatched patterns: *
************************
*** no unmatched patterns found ***

************************
* unmatched log_msgs: *
************************
*** no unmatched log messages found ***

************************
*  matched log_msgs:  *
************************
log_name  level log_msg records matched unmatched
example_5    10 hello         1       1         0
example_5    40 goodbye       1       1         0
```

**get_call_seq**(*name*)

Retrieve a call sequence by name.

**Parameters**

**name** (str) – name for whom the call sequence represents

**Return type**

str

**Returns**

**the call sequence in a format as described by**
get_formatted_call_sequence in diag_msg.py with the regex string ":[0-9]*" appended to represent the source line number to match

get_match_results(*caplog*)

Match the patterns to log records.

**Parameters**
**caplog** (LogCaptureFixture) – pytest fixture that captures log messages

**Return type**
MatchResults

**Returns**
MatchResults object, which contain the results of the matching operation. This will include a summary for the patterns and messages, and the data frames containing the patterns and messaged used to print and verify the results.

print_df(*df_to_print*, *col_names*, *left_justify_col_names*)

Prin the data set to screen.

**Parameters**

- **df_to_print** (DataFrame) – the dataframe to print
- **col_names** (list[str]) – list of column names to be printed
- **left_justify_col_names** (list[str]) – list of column names to be printed left justified

**Return type**
None

print_match_results(*match_results*, *print_matched=False*)

Print the match results.

**Parameters**

- **match_results** (MatchResults) – contains the results to be printed
- **print_matched** (bool) – if True, print the matched records, otherwise skip printing the matched records

**Return type**
None

Changed in version 3.0.0: *print_matched* keyword default changed to False

static search_df(*avail_df*, *search_arg_df*, *search_targ_df*, *targ_work_grp*, *min_potential_matches*)

Search the data frames for matches.

**Parameters**

- **avail_df** (DataFrame) – data frame of available entries
- **search_arg_df** (DataFrame) – data frame that has the search arg
- **search_targ_df** (DataFrame) – data frame that has the search target
- **targ_work_grp** (DataFrame) – work group dataframe that has the target avail count
- **min_potential_matches** (int) – the currently known minimum number of non-zero potential matches that need to be processed

**Return type**
None

**test_msg**(*log_msg*, *level=10*)

Issue a log msg and add its pattern.

**Parameters**

- **log_msg** (str) – log message to issue
- **level** (int) – logging level to use

**Return type**
None

### Notes

1) This method makes it easier to issue a log message in a test case by also adding the pattern.

Added in version 3.0.0.

Example: issue a test msg

```
def test_example(caplog: pytest.LogCaptureFixture) -> None:
    log_ver = LogVer("example_6")
    log_ver.test_msg("my test message")

    match_results = log_ver.get_match_results(caplog=caplog)
    log_ver.print_match_results(match_results,
                                print_matched=True)
    log_ver.verify_match_results(match_results)
```

The output from LogVer.print_match_results() for test_example:

```
************************************************
*              log verifier results            *
************************************************
Start: Thu Apr 11 2024 19:24:28
End: Thu Apr 11 2024 19:24:28
Elapsed time: 0:00:00.006002

************************************************
*                 summary stats                *
************************************************
    type  records  matched  unmatched
patterns        1        1          0
log_msgs        1        1          0

************************
* unmatched patterns: *
************************
*** no unmatched patterns found ***

************************
* unmatched log_msgs: *
************************
*** no unmatched log messages found ***
```

(continues on next page)

(continued from previous page)

```
************************
*  matched log_msgs:  *
************************
log_name  level log_msg     records matched unmatched
example_6    10 my test msg       1       1         0
```

**static verify_log_results**(*match_results*)

Verify that each log message issued is as expected.

**Parameters**
**match_results** (MatchResults) – contains the results to be verified

**Return type**
None

Deprecated since version 3.0.0: Use method *verify_match_results()* instead.

**Raises**

- **UnmatchedExpectedMessages** – There are expected log messages that failed to match actual log messages.
- **UnmatchedActualMessages** – There are actual log messages that failed to match expected log messages.

**static verify_match_results**(*match_results*)

Verify that each log message issued is as expected.

**Parameters**
**match_results** (MatchResults) – contains the results to be verified

**Raises**

- **UnmatchedPatterns** – One or more patterns failed to match their intended log messages. The patterns and/or the log messages may have been incorrectly specified.
- **UnmatchedLogMessages** – One or more log messages failed to be matched by corresponding patterns. The patterns and/or the log messages may have been incorrectly specified.

**Return type**
None

Module msgs.

## 7.7 Msgs

The Msgs class is intended to be used during testing to send and receive messages between threads.

**Example**
send a message to remote thread

```
>>> import threading
>>> from scottbrian_utils.msgs import Msgs
>>> def f1(msgs) -> None:
...     print('f1 beta entered')
...     my_msg = msgs.get_msg('beta')
...     print(my_msg)
```

(continues on next page)

(continued from previous page)

```
...     print('f1 beta exiting')
>>> def example1() -> None:
...     print('mainline entered')
...     msgs = Msgs()
...     f1_thread = threading.Thread(target=f1, args=(msgs,))
...     f1_thread.start()
...     msgs.queue_msg('beta', 'hello beta')
...     f1_thread.join()
...     print('mainline exiting')
>>> example1()
mainline entered
f1 beta entered
hello beta
f1 beta exiting
mainline exiting
```

**Example**

a command loop using Msgs

```
>>> import threading
>>> from scottbrian_utils.msgs import Msgs
>>> import time
>>> def main() -> None:
...     def f1() -> None:
...         print('f1 beta entered')
...         while True:
...             my_msg = msgs.get_msg('beta')
...             print(f'beta received msg: {my_msg}')
...             if my_msg == 'exit':
...                 break
...             else:
...                 # handle message
...                 msgs.queue_msg('alpha', f'msg "{my_msg}" completed')
...         print('f1 beta exiting')
...     print('mainline entered')
...     msgs = Msgs()
...     f1_thread = threading.Thread(target=f1)
...     f1_thread.start()
...     msgs.queue_msg('beta', 'do command a')
...     print(f"alpha received response: {msgs.get_msg('alpha')}")
...     msgs.queue_msg('beta', 'do command b')
...     print(f"alpha received response: {msgs.get_msg('alpha')}")
...     msgs.queue_msg('beta', 'exit')
...     f1_thread.join()
...     print('mainline exiting')
>>> main()
mainline entered
f1 beta entered
beta received msg: do command a
alpha received response: msg "do command a" completed
beta received msg: do command b
alpha received response: msg "do command b" completed
```

(continues on next page)

(continued from previous page)

```
beta received msg: exit
f1 beta exiting
mainline exiting
```

The msgs module contains:

1) Msgs class with methods:
    a. get_msg
    b. queue_msg

**class** msgs.**Msgs**

Msgs class for testing.

The Msgs class is used to assist in the testing of multi-threaded functions. It provides a set of methods that help with test case coordination and verification. The test case setup involves a mainline thread that starts one or more remote threads. The queue_msg and get_msg methods are used for inter-thread communications.

Initialize the object.

**get_msg**(*recipient*, *timeout=3.0*)

Get the next message in the queue.

**Parameters**

- **recipient** (str) – arbitrary name that designates the target of the message and which will be used with the queue_msg method to identify the intended recipient of the message
- **timeout** (Union[int, float, None]) – number of seconds allowed for msg response. A negative value, zero, or None means no timeout will happen. If timeout is not specified, then the default timeout value will be used.

**Return type**
Any

**Returns**
the received message

**Raises**
**GetMsgTimedOut** – {recipient} timed out waiting for msg

**queue_msg**(*target*, *msg='go'*)

Place a msg on the msg queue for the specified target.

**Parameters**

- **target** (str) – arbitrary name that designates the target of the message and which will be used with the get_msg method to retrieve the message
- **msg** (Optional[Any]) – message to place on queue

**Return type**
None

Module pauser.

## 7.8 Pauser

The *Pauser* class provides a *pause* method that delays execution for a specified interval. When possible, it uses `time.sleep()` for the initial part of the delay, and then completes the delay by looping while checking the elapsed time using `time.perf_counter_ns()`.

**Example**
pause execution for 1.5 seconds

```
>>> from scottbrian_utils.pauser import Pauser
>>> import time
>>> pauser = Pauser()
>>> start_time = time.time()
>>> pauser.pause(1.5)
>>> stop_time = time.time()
>>> print(f'paused for {stop_time - start_time:.1f} seconds')
paused for 1.5 seconds
```

While `time.sleep()` is useful for a rough delay, at very small intervals it can delay for more time than requested. For example, `time.sleep(0.001)` might return after 0.015 seconds.

`Pauser.pause()` provides accuracy at the expense of looping for some portion of the delay, perhaps the entire delay when a small interval is requested. `time.sleep()` should be preferred for applications that do not require acccracy since it will give up the processor and allow other work to make progress.

The Pauser module contains:

1) Pauser class with methods:
   a. calibrate
   b. get_metrics
   c. pause

**class** pauser.**MetricResults**(*pause_ratio: float*, *sleep_ratio: float*)

Results for get_metrics method.

Create new instance of MetricResults(pause_ratio, sleep_ratio)

**pause_ratio: float**
Alias for field number 0

**sleep_ratio: float**
Alias for field number 1

**class** pauser.**Pauser**(*min_interval_secs=0.03*, *part_time_factor=0.4*)

Pauser class to pause execution.

Initialize the instance.

The Pauser instance is initially created with the specified or defaulted values for *min_interval_secs* and *part_time_factor*. These values should be chosen based on the observed behavior of **time.sleep()** at low enough intervals to get an idea of where the sleep time fails to reflect the requested sleep time. Note that `Pauser.calibrate()` can be used to find and set the best value for *min_interval_secs*.

**Parameters**

- **min_interval_secs** (`float`) – the minimum interval that will use `time.sleep()` for a portion of the pause interval.

- **part_time_factor** (float) – the value to multiply the sleep interval by to reduce the sleep time

**Raises**

- **IncorrectInput** – The *min_interval_secs* argument is not valid - it must be a positive non-zero float.
- **IncorrectInput** – The *part_time_factor* argument is not valid - it must be a non-zero float no greater than 1.0.

Example: create an instance of Pauser with defaults

```
>>> from scottbrian_utils.pauser import Pauser
>>> pauser = Pauser()
>>> pauser
Pauser(min_interval_secs=0.03, part_time_factor=0.4)
```

**calibrate**(*min_interval_msecs=1*, *max_interval_msecs=300*, *increment=5*, *part_time_factor=0.4*, *max_sleep_late_ratio=1.0*, *iterations=8*)

Calibrate the Pauser for the current environment.

The *calibrate* method determines and sets the correct value for *min_interval_secs*, the lowest value for which the *pause* method will use `time.sleep()` to achieve a portion of the delay. This helps to ensure that `time.sleep()` is used for as much of the delay as possible to help free up the processor to perform other work. Note that setting the correct value for *min_interval_secs* helps ensure delay accuracy be preventing `time.sleep()` from being used at intervals that it is unable to accurately process.

**Parameters**

- **min_interval_msecs** (int) – starting interval of span to calibrate
- **max_interval_msecs** (int) – ending interval of span to calibrate
- **increment** (int) – number of milliseconds to skip for the next interval
- **part_time_factor** (float) – factor of sleep time to try
- **max_sleep_late_ratio** (float) – allowed error threshold
- **iterations** (int) – number of iteration per interval

**Raises**

- **IncorrectInput** – The *min_interval_msecs* argument is not valid - it must be a positive non-zero integer.
- **IncorrectInput** – The *max_interval_msecs* argument is not valid - it must be a positive non-zero integer equal to or greater than min_interval_msecs.
- **IncorrectInput** – The *increment argument* is not valid - it must be a positive non-zero integer.
- **IncorrectInput** – The *part_time_factor* argument is not valid - it must be a positive non-zero float no greater than 1.0.
- **IncorrectInput** – The *max_sleep_late_ratio* argument is not valid - it must be a positive non-zero float no greater than 1.0.
- **IncorrectInput** – The *iterations* argument is not valid - it must be a positive non-zero integer.

Example: calibrate the Pauser for a specific range of intervals

```
>>> from scottbrian_utils.pauser import Pauser
>>> pauser = Pauser(min_interval_secs=1.0)
>>> pauser.calibrate(min_interval_msecs=5,
...                  max_interval_msecs=100,
...                  increment=5)
:rtype: :sphinx_autodoc_typehints_type:\:py\:obj\:\`None\``
>>> print(f'{pauser.min_interval_secs=}')
pauser.min_interval_secs=0.015
```

**get_metrics**(*min_interval_msecs=1*, *max_interval_msecs=300*, *iterations=8*)

Get the pauser metrics.

**Parameters**

- **min_interval_msecs** (int) – starting number of milliseconds for scan
- **max_interval_msecs** (int) – ending number of milliseconds for scan
- **iterations** (int) – number of iterations to run each interval

**Return type**

*MetricResults*

**Returns**

The pause interval ratio (actual/requested) is returned in Metrics.pause_ratio and is an indication of the accuarcy of the requested delay with the value 1 being perfect. The sleep ratio (sleep/requested) is returned in Metrics.sleep_ratio and is the portion of the delay accomplished using `time.sleep()`.

**Raises**

- **IncorrectInput** – The *min_interval_msecs* argument is not valid - it must be a positive non-zero integer.
- **IncorrectInput** – The *max_interval_msecs* argument is not valid - it must be a positive non-zero integer equal to or greater than min_interval_msecs.
- **IncorrectInput** – The *iterations* argument is not valid - it must be a positive non-zero integer.

**Example: get the metrics for a pause of 0.1 seconds.**

Note that the accuracy is very good at 1.0 and the portion of the delay provided by `time.sleep()` is about 60%.

```
>>> from scottbrian_utils.pauser import Pauser
>>> pauser = Pauser()
>>> metrics = pauser.get_metrics(min_interval_msecs=100,
...                              max_interval_msecs=100,
...                              iterations=3)
>>> print(f'{metrics.pause_ratio=:.1f}, '
...       f'{metrics.sleep_ratio=:.1f}')
metrics.pause_ratio=1.0, metrics.sleep_ratio=0.6
```

**Example: get the metrics for a pause range of 0.98 to 1.0**

seconds. Note that the accuracy is very good at 1.0 but the portion of the delay provided by `time.sleep()` is zero. This is so because the Pauser was instantiated with a

*min_interval_secs* of 1.0 which effectively prevents `time.sleep()` from being used for any requested delays af 1 second or less.

```
>>> from scottbrian_utils.pauser import Pauser
>>> pauser = Pauser(min_interval_secs=1.0)
>>> metrics = pauser.get_metrics(min_interval_msecs=980,
...                              max_interval_msecs=1000,
...                              iterations=3)
>>> print(f'{metrics.pause_ratio=:.1f}, '
...       f'{metrics.sleep_ratio=:.1f}')
metrics.pause_ratio=1.0, metrics.sleep_ratio=0.0
```

**pause**(*interval*)

Pause for the specified number of seconds.

**Parameters**
**interval** (Union[int, float]) – number of seconds to pause

**Raises**
**IncorrectInput** – The interval arg is not valid - please provide a non-negative value.

**Return type**
None

Example: pause for 1 second

```
>>> from scottbrian_utils.pauser import Pauser
>>> pauser = Pauser()
>>> pauser.pause(1)
```

Example: pause for 1 half second and verify using time.time

```
>>> from scottbrian_utils.pauser import Pauser
>>> import time
>>> pauser = Pauser()
>>> start_time = time.time()
>>> pauser.pause(0.5)
>>> stop_time = time.time()
>>> interval = stop_time - start_time
>>> print(f'paused for {interval:.1f} seconds')
paused for 0.5 seconds
```

**Example: pause for 1 quarter second and verify using**

time.perf_counter_ns

```
>>> from scottbrian_utils.pauser import Pauser
>>> import time
>>> pauser = Pauser()
>>> start_time = time.perf_counter_ns()
>>> pauser.pause(0.25)
>>> stop_time = time.perf_counter_ns()
>>> interval = (stop_time - start_time) * Pauser.NS_2_SECS
>>> print(f'paused for {interval:.2f} seconds')
paused for 0.25 seconds
```

Module stop_watch.

## 7.9 StopWatch

The StopWatch class can be used during testing to start a clock, pause for a certain amount of time relative to the started clock, and then stop the clock and get the elapsed time.

**Example**
verify timing of event

```
>>> import threading
>>> import time
>>> from scottbrian_utils.stop_watch import StopWatch
>>> def main() -> None:
...     def f1():
...         print('f1 entered')
...         stop_watch.start_clock(clock_iter=1)
...         print('f1 about to wait')
...         f1_event.wait()
...         print('f1 back from wait')
...         assert 2.5 <= stop_watch.duration() <= 2.6
...         print('f1 exiting')
...     print('mainline entered')
...     stop_watch = StopWatch()
...     f1_thread = threading.Thread(target=f1)
...     f1_event = threading.Event()
...     print('mainline about to start f1')
...     f1_thread.start()
...     stop_watch.pause(2.5, clock_iter=1)
...     print('mainline about to set f1_event')
...     f1_event.set()
...     f1_thread.join()
...     print('mainline exiting')
>>> main()
mainline entered
mainline about to start f1
f1 entered
f1 about to wait
mainline about to set f1_event
f1 back from wait
f1 exiting
mainline exiting
```

The stop_watch module contains:

1) StopWatch class with methods:
    a. duration
    b. pause
    c. start_clock

**class** stop_watch.**StopWatch**

StopWatch class for testing.

The StopWatch class is used to assist in the testing of multi-threaded functions. It provides a set of methods that help with verification of timed event. The test case setup involves a mainline thread that starts one or more remote threads. The start_clock and duration methods are used to verify event times.

Initialize the object.

**duration**()

Return the number of seconds from the start_time.

**Return type**
float

**Returns**
number of seconds from the start_time

**pause**(*seconds*, *clock_iter*)

Sleep for the number of input seconds relative to start_time.

**Parameters**

- **seconds** (Union[int, float]) – number of seconds to pause from the start_time for the given clock_iter.
- **clock_iter** (int) – clock clock_iter to pause on

**Return type**
None

### Notes

1) The clock_iter is used to identify the clock that is currently in use. A remote thread wants to pause for a given number of seconds relative to the StopWatch start_time for a given iteration of the clock. We will do a sleep loop until the given clock_iter matches the StopWatch clock_iter.

**start_clock**(*clock_iter*)

Set the start_time to the current time.

**Parameters**
**clock_iter** (int) – clock_iter to set for the clock

**Return type**
None

Module time_hdr.

## 7.10 time_hdr

With **@time_box**, you can decorate a function to be sandwiched between start time and end time messages like this:

**Example**
decorate a function with time_box

```
>>> from scottbrian_utils.time_hdr import time_box
```

```
>>> @time_box
... def func2() -> None:
...     print('2 * 3 =', 2*3)
```

```
>>> func2()

*******************************************
* Starting func2 on Mon Jun 29 2020 18:22:50 *
*******************************************
2 * 3 = 6

*****************************************
* Ending func2 on Mon Jun 29 2020 18:22:51 *
* Elapsed time: 0:00:00.001204           *
*****************************************
```

The time_hdr module contains two items:

1) StartStopHeader class with two functions that will repectively print a starting time and ending time messages in a flower box (see flower_box module in scottbrian_utils package).

2) a time_box decorator that wraps a function and uses the StartStopHeader to print the starting and ending time messages.

time_box imports functools, sys, datetime, wrapt, and types from typing

time_hdr.**get_datetime_match_string**(*format_str*)

Return a regex string to match a datetime string.

**Parameters**
**format_str** (str) – string used to format a datetime that is to be used to create a match string

Changed in version 3.0.0: *format* keyword changed to *format_str*

**Return type**
str

**Returns**
string that is to be used in a regex expression to match a datetime string

time_hdr.**time_box**(*wrapped=None*, *, *dt_format='%a %b %d %Y %H:%M:%S'*, *time_box_enabled=True*, ***kwargs*)

Decorator to wrap a function in start time and end time messages.

The time_box decorator can be invoked with or without arguments, and the function being wrapped can optionally take arguments and optionally return a value. The wrapt.decorator is used to preserve the wrapped function introspection capabilities, and functools.partial is used to handle the case where decorator arguments are specified. The examples further below will help demonstrate the various ways in which the time_box decorator can be used.

**Parameters**

- **wrapped** (Optional[TypeVar(F, bound= Callable[..., Any])]) – Any callable function that accepts optional positional and/or optional keyword arguments, and optionally returns a value. The default is None, which will be the case when the pie decorator version is used with any of the following arguments specified.
- **dt_format** (NewType(DT_Format, str)) – Specifies the datetime format to use in the start time message. The default is StartStopHeader.default_dt_format.
- **time_box_enabled** (Union[bool, Callable[..., bool]]) – Specifies whether the start and end messages should be issued (True) or not (False). The default is True.

- **kwargs** (Any) – Specifies the print arguments to use on the print statement, such as *end*, *file*, or *flush*.

**Return type**
TypeVar(F, bound= Callable[..., Any])

**Returns**
A callable function that issues a starting time message, calls the wrapped function, issues the ending time message, and returns any return values that the wrapped function returns.

**Example**
statically wrapping function with time_box

```
>>> from scottbrian_utils.time_hdr import time_box
```

```
>>> _tbe = False
```

```
>>> @time_box(time_box_enabled=_tbe)
... def func4a() -> None:
...     print('this is sample text for _tbe = False static '
...           'example')
```

```
>>> func4a()  # func4a is not wrapped by time box
this is sample text for _tbe = False static example
```

```
>>> _tbe = True
```

```
>>> @time_box(time_box_enabled=_tbe)
... def func4b() -> None:
...     print('this is sample text for _tbe = True static example')
```

```
>>> func4b()  # func4b is wrapped by time box

************************************************
* Starting func4b on Mon Jun 29 2020 18:22:51 *
************************************************
this is sample text for _tbe = True static example

********************************************
* Ending func4b on Mon Jun 29 2020 18:22:51 *
* Elapsed time: 0:00:00.000133             *
********************************************
```

**Example**
dynamically wrapping function with time_box:

```
>>> from scottbrian_utils.time_hdr import time_box
```

```
>>> _tbe = True
>>> def tbe() -> bool: return _tbe
```

```
>>> @time_box(time_box_enabled=tbe)
... def func5() -> None:
...     print('this is sample text for the tbe dynamic example')
```

```
>>> func5()  # func5 is wrapped by time box

********************************************
* Starting func5 on Mon Jun 29 2020 18:22:51 *
********************************************
this is sample text for the tbe dynamic example

******************************************
* Ending func5 on Mon Jun 29 2020 18:22:51 *
* Elapsed time: 0:00:00.000130           *
******************************************
```

```
>>> _tbe = False
>>> func5()  # func5 is not wrapped by time_box
this is sample text for the tbe dynamic example
```

**Example**

specifying a datetime format:

```
>>> from scottbrian_utils.time_hdr import time_box
```

```
>>> aDatetime_format: DT_Format = DT_Format('%m/%d/%y %H:%M:%S')
>>> @time_box(dt_format=aDatetime_format)
... def func6() -> None:
...     print('this is sample text for the datetime format example')
```

```
>>> func6()

**************************************
* Starting func6 on 06/30/20 17:07:48 *
**************************************
this is sample text for the datetime format example

************************************
* Ending func6 on 06/30/20 17:07:48 *
* Elapsed time: 0:00:00.000073     *
************************************
```

Module timer.

## 7.11 Timer

The Timer class can be used to detect when a process has exceeded a specified amount of time.

**Example**

create a timer and use in a loop

```
>>> from scottbrian_utils.timer import Timer
>>> import time
>>> def example1() -> None:
...     print('example1 entered')
...     timer = Timer(timeout=3)
...     for idx in range(10):
...         print(f'idx = {idx}')
...         time.sleep(1)
...         if timer.is_expired():
...             print('timer has expired')
...             break
...     print('example1 exiting')
>>> example1()
example1 entered
idx = 0
idx = 1
idx = 2
timer has expired
example1 exiting
```

The timer module contains:

1) Timer class with methods:

   a. is_expired

**class** timer.**Timer**(*timeout=None*, *default_timeout=None*)

Timer class.

Initialize a timer object.

**Parameters**

- **timeout** (Union[int, float, None]) – value to use for timeout
- **default_timeout** (Union[int, float, None]) – value to use if timeout is None

Example: class with a method that uses Timer

```
>>> class A:
...     def __init__(self):
...         self.a = 1
...     def m1(self, sleep_time: float) -> bool:
...         timer = Timer(timeout=1)
...         time.sleep(sleep_time)
...         if timer.is_expired():
...             return False
...         return True
>>> def example2() -> None:
...     print('example2 entered')
```

(continues on next page)

(continued from previous page)

```
...     my_a = A()
...     print(my_a.m1(0.5))
...     print(my_a.m1(1.5))
...     print('example2 exiting')
>>> example2()
example2 entered
True
False
example2 exiting
```

Example: class with a method that uses Timer and timeout parm

```
>>> class A:
...     def __init__(self):
...         self.a = 1
...     def m1(self, sleep_time: float, timeout: float) -> bool:
...         timer = Timer(timeout=timeout)
...         time.sleep(sleep_time)
...         if timer.is_expired():
...             return False
...         return True
>>> def example3() -> None:
...     print('example3 entered')
...     my_a = A()
...     print(my_a.m1(sleep_time=0.5, timeout=0.7))
...     print(my_a.m1(sleep_time=1.5, timeout=1.2))
...     print(my_a.m1(sleep_time=1.5, timeout=1.8))
...     print('example3 exiting')
>>> example3()
example3 entered
True
False
True
example3 exiting
```

Example: class with default timeout and method with timeout parm

```
>>> class A:
...     def __init__(self, default_timeout: float):
...         self.a = 1
...         self.default_timeout = default_timeout
...     def m1(self,
...            sleep_time: float,
...            timeout: Optional[float] = None) -> bool:
...         timer = Timer(timeout=timeout,
...                       default_timeout=self.default_timeout)
...         time.sleep(sleep_time)
...         if timer.is_expired():
...             return False
...         return True
>>> def example4() -> None:
...     print('example4 entered')
```

(continues on next page)

(continued from previous page)

```
...     my_a = A(default_timeout=1.2)
...     print(my_a.m1(sleep_time=0.5))
...     print(my_a.m1(sleep_time=1.5))
...     print(my_a.m1(sleep_time=0.5, timeout=0.3))
...     print(my_a.m1(sleep_time=1.5, timeout=1.8))
...     print(my_a.m1(sleep_time=1.5, timeout=0))
...     print('example4 exiting')
>>> example4()
example4 entered
True
False
False
True
True
example4 exiting
```

### Notes

1) The reason for having both a timeout parm and a default_timeout parm is needed as follows: Some classes are instantiated with a default timeout for use in methods that have a timeout parm. The method timeout arg has priority and can be specified as negative or zero to indicate no timeout is to be in effect, or with a positive value to be used as the timeout value, or as None (explicitly or implicitly). When None is detected, then the method will use the default timeout for the class. The methods can thus simply instantiate a Timer and pass in both the timeout and the default timeout values, and Timer will sort things out and use the intended value as described above.

**is_expired**()

Return either True or False for the timer.

**Return type**
bool

**Returns**
True if the timer has expired, False if not

Example: using is_expired

```
>>> import time
>>> from scottbrian_utils.timer import Timer
>>> def example6() -> None:
...     print('example6 entered')
...     timer = Timer(timeout=2.5)
...     time.sleep(1)
...     print(f'timer expired = {timer.is_expired()}')
...     time.sleep(1)
...     print(f'timer expired = {timer.is_expired()}')
...     time.sleep(1)
...     print(f'timer expired = {timer.is_expired()}')
...     print('example6 exiting')
>>> example6()
example6 entered
timer expired = False
```

(continues on next page)

(continued from previous page)

```
timer expired = False
timer expired = True
example6 exiting
```

**is_specified**()

Return True or False for timeout being specified.

**Return type**
bool

**Returns**
True if a positive timeout value was specified, False if not

### Notes

1) This method simply indicates whether a positive timeout value was initially specified during instatiation, either via a timeout or default_timeout. It does not provide any indication whether the time has expired.

Example: using is_specified

```
>>> import time
>>> from scottbrian_utils.timer import Timer
>>> def example7() -> None:
...     print('example7 entered')
...     timer_a = Timer()
...     print(f'timer_a specified = {timer_a.is_specified()}')
...     timer_b = Timer(timeout=None)
...     print(f'timer_b specified = {timer_b.is_specified()}')
...     timer_c = Timer(timeout=-1)
...     print(f'timer_c specified = {timer_c.is_specified()}')
...     timer_d = Timer(timeout=0)
...     print(f'timer_d specified = {timer_d.is_specified()}')
...     timer_e = Timer(timeout=1)
...     print(f'timer_e specified = {timer_e.is_specified()}')
...     timer_f = Timer(default_timeout=None)
...     print(f'timer_f specified = {timer_f.is_specified()}')
...     timer_g = Timer(default_timeout=-1)
...     print(f'timer_g specified = {timer_g.is_specified()}')
...     timer_h = Timer(default_timeout=0)
...     print(f'timer_h specified = {timer_h.is_specified()}')
...     timer_i = Timer(default_timeout=1)
...     print(f'timer_i specified = {timer_i.is_specified()}')
...     timer_j = Timer(timeout=None, default_timeout=None)
...     print(f'timer_j specified = {timer_j.is_specified()}')
...     timer_k = Timer(timeout=None, default_timeout=-1)
...     print(f'timer_k specified = {timer_k.is_specified()}')
...     timer_l = Timer(timeout=None, default_timeout=0)
...     print(f'timer_l specified = {timer_l.is_specified()}')
...     timer_m = Timer(timeout=None, default_timeout=1)
...     print(f'timer_m specified = {timer_m.is_specified()}')
...     timer_n = Timer(timeout=-1, default_timeout=None)
```

(continues on next page)

(continued from previous page)

```
...     print(f'timer_n specified = {timer_n.is_specified()}')
...     timer_o = Timer(timeout=-1, default_timeout=-1)
...     print(f'timer_o specified = {timer_o.is_specified()}')
...     timer_p = Timer(timeout=-1, default_timeout=0)
...     print(f'timer_p specified = {timer_p.is_specified()}')
...     timer_q = Timer(timeout=-1, default_timeout=1)
...     print(f'timer_q specified = {timer_q.is_specified()}')
...     timer_r = Timer(timeout=0, default_timeout=None)
...     print(f'timer_r specified = {timer_r.is_specified()}')
...     timer_s = Timer(timeout=0, default_timeout=-1)
...     print(f'timer_s specified = {timer_s.is_specified()}')
...     timer_t = Timer(timeout=0, default_timeout=0)
...     print(f'timer_t specified = {timer_t.is_specified()}')
...     timer_u = Timer(timeout=0, default_timeout=1)
...     print(f'timer_u specified = {timer_u.is_specified()}')
...     timer_v = Timer(timeout=1, default_timeout=None)
...     print(f'timer_v specified = {timer_v.is_specified()}')
...     timer_w = Timer(timeout=1, default_timeout=-1)
...     print(f'timer_w specified = {timer_w.is_specified()}')
...     timer_x = Timer(timeout=1, default_timeout=0)
...     print(f'timer_x specified = {timer_x.is_specified()}')
...     timer_y = Timer(timeout=1, default_timeout=1)
...     print(f'timer_y specified = {timer_y.is_specified()}')
...     print('example7 exiting')
>>> example7()
example7 entered
timer_a specified = False
timer_b specified = False
timer_c specified = False
timer_d specified = False
timer_e specified = True
timer_f specified = False
timer_g specified = False
timer_h specified = False
timer_i specified = True
timer_j specified = False
timer_k specified = False
timer_l specified = False
timer_m specified = True
timer_n specified = False
timer_o specified = False
timer_p specified = False
timer_q specified = False
timer_r specified = False
timer_s specified = False
timer_t specified = False
timer_u specified = False
timer_v specified = True
timer_w specified = True
timer_x specified = True
timer_y specified = True
example7 exiting
```

**remaining_time**()

Return the remaining timer time.

**Return type**
Union[int, float, None]

**Returns**
remaining time of the timer

Example: using remaining_time and is_expired

```
>>> import threading
>>> import time
>>> from scottbrian_utils.timer import Timer
>>> def f1(f1_event):
...     print('f1 entered')
...     time.sleep(1)
...     f1_event.set()
...     time.sleep(1)
...     f1_event.set()
...     time.sleep(1)
...     f1_event.set()
...     print('f1 exiting')
>>> def example5() -> None:
...     print('example5 entered')
...     timer = Timer(timeout=2.5)
...     f1_event = threading.Event()
...     f1_thread = threading.Thread(target=f1,
...                                  args=(f1_event,))
...     f1_thread.start()
...     wait_result = f1_event.wait(
...         timeout=timer.remaining_time())
...     print(f'wait1 result = {wait_result}')
...     f1_event.clear()
...     print(f'remaining time = {timer.remaining_time():0.1f}')
...     print(f'timer expired = {timer.is_expired()}')
...     wait_result = f1_event.wait(
...         timeout=timer.remaining_time())
...     print(f'wait2 result = {wait_result}')
...     f1_event.clear()
...     print(f'remaining time = {timer.remaining_time():0.1f}')
...     print(f'timer expired = {timer.is_expired()}')
...     wait_result = f1_event.wait(
...         timeout=timer.remaining_time())
...     print(f'wait3 result = {wait_result}')
...     f1_event.clear()
...     print(f'remaining time = {timer.remaining_time():0.4f}')
...     print(f'timer expired = {timer.is_expired()}')
...     f1_thread.join()
...     print('example5 exiting')
>>> example5()
example5 entered
f1 entered
wait1 result = True
remaining time = 1.5
```

(continues on next page)

(continued from previous page)

```
timer expired = False
wait2 result = True
remaining time = 0.5
timer expired = False
wait3 result = False
remaining time = 0.0001
timer expired = True
f1 exiting
example5 exiting
```

### Notes

1) The remaining time is calculated by subtracting the elapsed time from the timeout value originally supplied when the timer was instantiated. Depending on when the remaining time is requested, it could be such that the timeout value has been reached or exceeded and the true remaining time becomes zero or negative. The returned value, however, will never be zero or negative - it will be at least a value of 0.0001. Thus, the timeout value can not be used to determine whether the timer has expired since it will always show at least 0.0001 seconds remaining after the timer has truly expired. The is_expired method should be used to determine whether the timer has expired. The reason for this is that the intended use of remaining_time is to use it as a timeout arg that is passed into a service such as wait or a lock obtain. Many services interpret a value of zero or less as an unlimited timeout value, meaning it will never timeout. It is thus better to pass in a very small value and get an immediate timeout in these cases than to possibly wait forever.

2) None is returned when the timer was originally instantiated with unlimited time.

**timeout_value**()

Return the timeout value that was specified.

**Return type**
Union[int, float, None]

**Returns**

**The timeout value that was specified, either via the timeout**
value or the default timeout value, or None.

Example: using timeout_value

```
>>> import time
>>> from scottbrian_utils.timer import Timer
>>> def example8() -> None:
...     print('example8 entered')
...     timer_a = Timer()
...     print(f'timer_a timeout = {timer_a.timeout_value()}')
...     timer_b = Timer(timeout=None)
...     print(f'timer_b timeout = {timer_b.timeout_value()}')
...     timer_c = Timer(timeout=-1)
...     print(f'timer_c timeout = {timer_c.timeout_value()}')
...     timer_d = Timer(timeout=0)
...     print(f'timer_d timeout = {timer_d.timeout_value()}')
...     timer_e = Timer(timeout=1)
...     print(f'timer_e timeout = {timer_e.timeout_value()}')
```

(continues on next page)

(continued from previous page)

```
...     timer_f = Timer(default_timeout=None)
...     print(f'timer_f timeout = {timer_f.timeout_value()}')
...     timer_g = Timer(default_timeout=-1)
...     print(f'timer_g timeout = {timer_g.timeout_value()}')
...     timer_h = Timer(default_timeout=0)
...     print(f'timer_h timeout = {timer_h.timeout_value()}')
...     timer_i = Timer(default_timeout=2.2)
...     print(f'timer_i timeout = {timer_i.timeout_value()}')
...     timer_j = Timer(timeout=None, default_timeout=None)
...     print(f'timer_j timeout = {timer_j.timeout_value()}')
...     timer_k = Timer(timeout=None, default_timeout=-1)
...     print(f'timer_k timeout = {timer_k.timeout_value()}')
...     timer_l = Timer(timeout=None, default_timeout=0)
...     print(f'timer_l timeout = {timer_l.timeout_value()}')
...     timer_m = Timer(timeout=None, default_timeout=2.2)
...     print(f'timer_m timeout = {timer_m.timeout_value()}')
...     timer_n = Timer(timeout=-1, default_timeout=None)
...     print(f'timer_n timeout = {timer_n.timeout_value()}')
...     timer_o = Timer(timeout=-1, default_timeout=-1)
...     print(f'timer_o timeout = {timer_o.timeout_value()}')
...     timer_p = Timer(timeout=-1, default_timeout=0)
...     print(f'timer_p timeout = {timer_p.timeout_value()}')
...     timer_q = Timer(timeout=-1, default_timeout=2.2)
...     print(f'timer_q timeout = {timer_q.timeout_value()}')
...     timer_r = Timer(timeout=0, default_timeout=None)
...     print(f'timer_r timeout = {timer_r.timeout_value()}')
...     timer_s = Timer(timeout=0, default_timeout=-1)
...     print(f'timer_s timeout = {timer_s.timeout_value()}')
...     timer_t = Timer(timeout=0, default_timeout=0)
...     print(f'timer_t timeout = {timer_t.timeout_value()}')
...     timer_u = Timer(timeout=0, default_timeout=2.2)
...     print(f'timer_u timeout = {timer_u.timeout_value()}')
...     timer_v = Timer(timeout=1, default_timeout=None)
...     print(f'timer_v timeout = {timer_v.timeout_value()}')
...     timer_w = Timer(timeout=1, default_timeout=-1)
...     print(f'timer_w timeout = {timer_w.timeout_value()}')
...     timer_x = Timer(timeout=1, default_timeout=0)
...     print(f'timer_x timeout = {timer_x.timeout_value()}')
...     timer_y = Timer(timeout=1, default_timeout=2.2)
...     print(f'timer_y timeout = {timer_y.timeout_value()}')
...     print('example8 exiting')
>>> example8()
example8 entered
timer_a timeout = None
timer_b timeout = None
timer_c timeout = None
timer_d timeout = None
timer_e timeout = 1
timer_f timeout = None
timer_g timeout = None
timer_h timeout = None
timer_i timeout = 2.2
```

(continues on next page)

(continued from previous page)

```
timer_j timeout = None
timer_k timeout = None
timer_l timeout = None
timer_m timeout = 2.2
timer_n timeout = None
timer_o timeout = None
timer_p timeout = None
timer_q timeout = None
timer_r timeout = None
timer_s timeout = None
timer_t timeout = None
timer_u timeout = None
timer_v timeout = 1
timer_w timeout = 1
timer_x timeout = 1
timer_y timeout = 1
example8 exiting
```

Module unique_ts.

## 7.12 UniqueTS

The UniqueTS class can be used to obtain a unique time stamp.

**Example1**
obtain unique time stamps

**This example shows that obtaining two time stamps in quick**
succession using get_unique_time_ts() guarantees they will be unique.

```
>>> from scottbrian_utils.unique_ts import UniqueTS, UniqueTStamp
>>> first_time_stamp: UniqueTStamp = UniqueTS.get_unique_ts()
>>> second_time_stamp: UniqueTStamp = UniqueTS.get_unique_ts()
>>> print(second_time_stamp > first_time_stamp)
True
```

**class** unique_ts.**UniqueTS**

Unique Time Stamp class.

**classmethod get_unique_ts()**

Return a unique time stamp.

**Return type**
NewType(UniqueTStamp, float)

**Returns**
a unique time stamp

CHAPTER

EIGHT

# INDICES AND TABLES

- genindex
- modindex
- search

# PYTHON MODULE INDEX

# INDEX