Inform Documentation

Release 1.13.0

Ken Kundert

Aug 11, 2018

Contents

Version: 1.13.0
Released: 2018-08-11
Please post all bugs and suggestions at Github (or contact me directly at inform@nurdletech.com).

Inform is designed to display messages from programs that are typically run from a console. It provides a collection of 'print' functions that allow you to simply and cleanly print different types of messages. For example:

```
>>> from inform import display, warn, error
>>> display('This is a plain message.')
This is a plain message.

>>> error('this is an error message.')
error: this is an error message.
```

These functions behave in a way that is very similar to the *print* function that is built-in to Python3, but they also provide some additional features as well. For example, they can be configured to log their messages and they can be disabled en masse.

Finally, *Inform* provides a generic exception and a collection of small utilities that are useful when creating messages.

CHAPTER 1

Alternatives

The Python standard library provides the logging package. This package differs from *Inform* in that it is really intended to log events to a file. It is more intended for daemons that run in the background and the logging is not meant to communicate directly to the user in real time, but rather record enough information into a log file for an administrator to understand how well the program is performing and whether anything unusual is happening.

In contrast, *Inform* is meant to used to provide information from command line utilities directly to the user in real time. It is not confined to only logging events, but instead can be used anywhere the normal Python *print* function would be used. In effect, *Inform* allows you to create and use multiple print functions each of which is tailored for a specific situation or task. This of course is something you could do yourself using the built-in *print* function, but with *Inform* you will not have to embed your print functions in complex condition statements, every message is formatted in a consistent manner that follows normal Unix conventions, and you can control all of your print functions by configuring a single object.

CHAPTER 2

Installation

Install with:

```
pip3 install --user inform
```

Requires Python2.7 or Python3.3 or better.

CHAPTER 3

Quick Tour

3.1 Informants

Inform defines a collection of *print*-like functions that have different roles. These functions are referred to as 'informants' and include *display*, *warn*, *error*, and *fatal*. All of them take arguments in the same manner as Python's built-in print function and all of them write the desired message to standard output, with the last three adding a header to the message that indicates the type of message. For example:

```
>>> display('ice', 9)
ice 9

>>> warn('cannot write to file, logging suppressed.')
warning: cannot write to file, logging suppressed.

>>> filename = 'config'
>>> error('%s: file not found.' % filename)
error: config: file not found.

>>> fatal('defective input file.', culprit=filename)
error: config: defective input file.
```

Notice that in the error message the filename was explicitly added to the front of the message. This is an extremely common idiom and it is provided by *Inform* using the *culprit* named argument as shown in the fatal message. *fatal* is similar to *error* but additionally terminates the program. To make the error messages stand out, the header is generally rendered in a color appropriate to the message, so warnings use yellow and errors use red. However, they are not colored above because messages are only colored if they are being written to the console (a TTY).

In a manner similar to Python3's built-in *print* function, unnamed arguments are converted to strings and then joined using the separator, which by default is a single space but can be specified using the *sep* named argument.

```
>>> colors = dict(red='ff5733', green='4fff33', blue='3346ff')

>>> lines = []
>>> for key in sorted(colors.keys()):
```

(continues on next page)

(continued from previous page)

```
...     val = colors[key]
...     lines.append('{key:>5s} = {val}'.format(key=key, val=val))

>>> display(*lines, sep='\n')
 blue = 3346ff
green = 4fff33
  red = ff5733
```

Alternatively, you can specify an arbitrary collection of named and unnamed arguments, and form them into a message using the *template* argument:

```
>>> for key in sorted(colors.keys()):
...     val = colors[key]
...     display(val, k=key, template='{k:>5s} = {}')
 blue = 3346ff
green = 4fff33
  red = ff5733
```

You can even specify a collection of templates. The first one for which all keys are known is used. For example;

```
>>> colors = dict(
...     red = ('ff5733', 'failure'),
...     green = ('4fff33', 'success'),
...     blue = ('3346ff', None),
... )

>>> for name in sorted(colors.keys()):
...     code, desc = colors[name]
...     templates = ('{:>5s} = {}  -- {}', '{:>5s} = {}')
...     display(name, code, desc, template=templates)
 blue = 3346ff
green = 4fff33  -- success
  red = ff5733  -- failure

>>> for name in sorted(colors.keys()):
...     code, desc = colors[name]
...     templates = ('{k:>5s} = {v}  -- {d}', '{k:>5s} = {v}')
...     display(k=name, v=code, d=desc, template=templates)
 blue = 3346ff
green = 4fff33  -- success
  red = ff5733  -- failure
```

All informants support the *culprit* named argument, which is used to identify the object of the message. The *culprit* can be a scalar, as above, or a collection, in which case the members of the collection are joined together:

```
>>> line = 5
>>> display('syntax error.', culprit=(filename, line))
config, 5: syntax error.
```

Besides the four informants already described, *Inform* provides several others, including *log*, *codicil*, *comment*, *narrate*, *output*, *notify*, *debug* and *panic*. Informants in general can write to the log file, to the standard output, or to a notifier. They can add headers and specify the color of the header and the message. They can also continue the previous message or they can terminate the program. Each informant embodies a predefined set of these choices. In addition, they are affected by options passed to the active informer (described next), which is often used to enable or disable informants based on various verbosity options.

3.2 Controlling Informants

For more control of the informants, you can import and instantiate the *inform.Inform* class yourself along with the desired informants. This gives you the ability to specify options:

```
>>> from inform import Inform, display, error
>>> Inform(logfile=True, prog_name="teneya", quiet=True)
<...>
>>> display('Initializing ...')

>>> error('file not found.', culprit='data.in')
teneya error: data.in: file not found.
```

Notice that in this case the call to *display* did not print anything. That is because the *quiet* argument was passed to *Inform*, which acts to suppress all but error messages. However, a logfile was specified, so the message would be logged. In addition, the program name was specified, with the result in it being added to the header of the error message.

An object of the *Inform* class is referred to as an informer (not to be confused with the print functions, which are referred to as informants). Once instantiated, you can use the informer to change various settings, terminate the program, or return a count of the number of errors that have occurred.

```
>>> from inform import Inform, error
>>> informer = Inform(prog_name=False)
>>> error('file not found.', culprit='data.in')
error: data.in: file not found.
>>> informer.errors_accrued()
1
```

3.3 Utility Functions

Inform provides a collection of utility functions that are often useful when constructing messages.

aaa	Pretty prints, then returns, its argument; used when debugging code.
Color Class	Used to color messages sent to the console.
columns	Distribute an array over enough columns to fill the screen.
conjoin	Like join, but adds a conjunction between the last two items.
cull	Strips uninteresting value from collections.
ddd	Pretty prints its arguments, used when debugging code.
fmt	Similar to format(), but can pull arguments from the local scope.
full_stop	Add a period to end of string if it has no other punctuation.
indent	Adds indentation.
is_collection	Is object a collection (i.e., is it iterable and not a string)?
is_iterable	Is object iterable (includes strings).
is_str	Is object a string?
join	Combines arguments into a string in the same way as an informant.
os_error	Generates clean messages for operating system errors
plural	Pluralizes a word if needed.
ppp	Print function, used when debugging code.
render	Converts many of the built-in Python data types into attractive, compact, and easy to read strings.
sss	Prints stack trace, used when debugging code.
vvv	Print all variables that have given value, used when debugging code.

One of the most used is *os_error*. It converts *OSError* exceptions into a simple well formatted string that can be used to describe the exception to the user.

```
>>> from inform import os_error, error
>>> try:
...     with open(filename) as f:
...         config = f.read()
... except (OSError, IOError) as e:
...     error(os_error(e))
error: config: no such file or directory.
```

3.4 Generic Exception

Inform also provides a generic exception, *inform.Error*, that can be used directly or can be subclassed to create your own exceptions. It takes arguments in the same manner as informants, and provides some useful methods used when reporting errors:

```
>>> from inform import Error

>>> def read_config(filename):
...     try:
...         with open(filename) as f:
...             config = f.read()
...     except (OSError, IOError) as e:
...         raise Error(os_error(e))

>>> try:
...     read_config('config')
... except Error as e:
...     e.report()
error: config: no such file or directory.
```

CHAPTER 4

Documentation

4.1 User's Guide

4.1.1 Using Informants

This package defines a collection of 'print' functions that are referred to as informants. They include include *log*, *comment*, *codicil*, *narrate*, *display*, *output*, *notify*, *debug*, *warn*, *error*, *fatal* and *panic*.

They all take arguments in a manner that is a generalization of Python's built-in print function. Each of the informants is used for a specific purpose, but they all take and process arguments in the same manner. These functions will be distinguished in the *Predefined Informants* section. In this section, the manner in which they process their arguments is presented.

With the simplest use of the program, you simply import the informants you need and call them, placing those things that you wish to print in the argument list as unnamed arguments:

```
>>> from inform import display
>>> display('ice', 9)
ice 9
```

Informant Arguments

By default, all of the unnamed arguments converted to strings and then joined together using a space between each argument. However, you can use named arguments to change this behavior. The following named arguments are used to control the informants:

sep = ' ': Specifies the string used to join the unnamed arguments.

end = '\n': Specifies a string to append to the message.

file: The destination stream (a file pointer).

flush = *False*: Whether the message should flush the destination stream (not available in python2).

culprit = *None*: A string that is added to the beginning of the message that identifies the culprit (the object for which the problem being reported was found). May also number or a tuple that contains strings and numbers. If *culprit* is a tuple, the members are converted to strings and joined with *culprit_sep* (default is ', ').

wrap = False: Specifies whether message should be wrapped. *wrap* may be True, in which case the default width of 70 is used. Alternately, you may specify the desired width. The wrapping occurs on the final message after the arguments have been joined.

template = None: A template that if present interpolates the arguments to form the final message rather than simply joining the unnamed arguments with *sep*. The template is a string, and its *format* method is called with the unnamed and named arguments of the message passed as arguments. *template* may also be a collection of strings, in which case the first template for which all the necessary arguments are available is used.

remove: Specifies the argument values that are unavailable to the template.

The first four are also accepted by Python's built-in *print* function and have the same behavior.

This example makes use of the *sep* and *end* named arguments:

```
>>> from inform import display

>>> actions = ['r: rewind', 'p: play/pause', 'f: fast forward']
>>> display('The choices include', *actions, sep=',\n    ', end='.\n')
The choices include,
    r: rewind,
    p: play/pause,
    f: fast forward.
```

Culprits

culprit is used to identify the target of the message. If the message is pointing out a problem the *culprit* is generally the source of the problem.

Here is an example that demonstrates the wrap and composite culprit features:

```
>>> from inform import error

>>> value = -1
>>> error(
...     'Encountered illegal value',
...     value,
...     'when filtering.  Consider regenerating the dataset.',
...     culprit=('input.data', 32), wrap=True,
... )
error: input.data, 32:
    Encountered illegal value -1 when filtering.  Consider regenerating
    the dataset.
```

Occasionally the actual culprits are not available where the messages are printed. In this case you can use culprit caching. Simply cache the culprits in you informer using *inform.set_culprit()* or *inform.add_culprit()* and then recall them when needed using *inform.get_culprit()*. Both *set_culprit* and *add_culprit* are designed to be used with Python's *with* statement.

The following example illustrates the used of culprit caching. Here, the code is spread over several functions, and the various culprits are known locally but are not passed directly into the function that may report the error. Rather than explicitly passing the culprits into the various functions, which would clutter up their argument lists, the culprits are cached in case they are needed.

```
>>> from inform import add_culprit, get_culprit, set_culprit, error

>>> def read_param(line, parameters):
...     name, value = line.split(' = ')
...     try:
...         parameters[name] = float(value)
...     except ValueError:
...         error(
...             'expected a number, found:', value,
...             culprit=get_culprit(name)
...         )

>>> def read_params(lines):
...     parameters = {}
...     for lineno, line in enumerate(lines):
...         with add_culprit(lineno+1):
...             read_param(line, parameters)

>>> filename = 'parameters'
>>> with open(filename) as f, set_culprit(filename):
...     lines = f.read().splitlines()
...     parameters = read_params(lines)
error: parameters, 3, c: expected a number, found: ack
```

Templates

The *template* strings are the same as one would use with Python's built-in format function and string method (as described in Format String Syntax). The *template* string can interpolate either named or unnamed arguments. In this example, named arguments are interpolated:

```
>>> colors = {
...     'red': ('ff5733', 'failure'),
...     'green': ('4fff33', 'success'),
...     'blue': ('3346ff', None),
... }

>>> for key in sorted(colors.keys()):
...     val = colors[key]
...     display(k=key, v=val, template='{k:>5s} = {v[0]}')
 blue = 3346ff
green = 4fff33
  red = ff5733
```

You can also specify a collection of templates. The first one for which all keys are available is used. For example;

```
>>> for name in sorted(colors.keys()):
...     code, desc = colors[name]
...     display(name, code, desc, template=('{:>5s} = {}  -- {}', '{:>5s} = {}'))
 blue = 3346ff
green = 4fff33  -- success
  red = ff5733  -- failure

>>> for name in sorted(colors.keys()):
...     code, desc = colors[name]
...     display(k=name, v=code, d=desc, template=('{k:>5s} = {v}  -- {d}', '{k:>5s} =
↪{v}'))
```

(continues on next page)

(continued from previous page)

```
 blue = 3346ff
green = 4fff33  -- success
  red = ff5733  -- failure
```

The first loop interpolates positional (unnamed) arguments, the second interpolates the keyword (named) arguments.

By default, the values that are considered unavailable and so will invalidate a template are those that would be False when cast to a Boolean. So, by default, the following values are considered unavailable: 0, False, None, '', (), [], {}, etc. You can use the *remove* named argument to control this. *remove* may be a function, a collection, or a scalar. The function would take a single argument that is the value to consider and return True if the value should be unavailable. The scalar or the collection simply specifies the value or values that should be unavailable.

```
>>> accounts = dict(checking=1100, savings=0, brokerage=None)

>>> for name, amount in sorted(accounts.items()):
...     display(name, amount, template=('{:>10s} = ${}', '{:>10s} = NA'), remove=None)
 brokerage = NA
  checking = $1100
   savings = $0
```

4.1.2 Predefined Informants

The following informants are predefined in *Inform*. You can create custom informants using *inform.InformantFactory*.

All of the informants except *panic* and *debug* do not produce any output if *mute* is set.

If you do not care for the default behavior for the predefined informants, you can customize them by overriding their attributes. For example, in many cases you might prefer that normal program output is not logged, either because it is voluminous or because it is sensitive. In that case you can simply override the *log* attributes for the *display* and *output* informants like so:

```
from inform import display, output
display.log = False
output.log = False
```

log

```
log = InformantFactory(
    output=False,
    log=True,
)
```

Saves a message to the log file without displaying it.

comment

```
comment = InformantFactory(
    output=lambda informer: informer.verbose and not informer.mute,
    log=True,
    message_color='cyan',
)
```

Displays a message only if *verbose* is set. Logs the message. The message is displayed in cyan when writing to the console.

Comments are generally used to document unusual occurrences that might warrant the user's attention.

codicil

```
codicil = InformantFactory(is_continuation=True)
```

Continues a previous message. Continued messages inherit the properties (output, log, message color, etc) of the previous message. If the previous message had a header, that header is not output and instead the message is indented.

```
>>> from inform import Inform, warn, codicil
>>> informer = Inform(prog_name="myprog")
>>> warn('file not found.', culprit='ghost')
myprog warning: ghost: file not found.

>>> codicil('skipping')
    skipping
```

narrate

```
narrate = InformantFactory(
    output=lambda informer: informer.narrate and not informer.mute,
    log=True,
    message_color='blue',
)
```

Displays a message only if *narrate* is set. Logs the message. The message is displayed in blue when writing to the console.

Narration is generally used to inform the user as to what is going on. This can help place errors and warnings in context so that they are easier to understand. Distinguishing narration from comments allows them to colored differently and controlled separately.

display

```
display = InformantFactory(
    output=lambda informer: not informer.quiet and not informer.mute,
    log=True,
)
```

Displays a message if *quiet* is not set. Logs the message.

```
>>> from inform import display
>>> display('We the people ...')
We the people ...
```

output

```
output = InformantFactory(
    output=lambda informer: not informer.mute,
    log=True,
)
```

Displays and logs a message. This is used for messages that are not errors and that are noteworthy enough that they need to get through even though the user has asked for quiet.

```
>>> from inform import output
>>> output('The sky is falling!')
The sky is falling!
```

notify

```
notify = InformantFactory(
    notify=True,
    log=True,
)
```

Temporarily display the message in a bubble at the top of the screen. Also prints the message on the standard output and sends it to the log file. This is used for messages that the user is otherwise unlikely to see because they have no access to the standard output.

debug

```
debug = InformantFactory(
    severity='DEBUG',
    output=True,
    log=True,
    header_color='magenta',
)
```

Displays and logs a debugging message. A header with the label *DEBUG* is added to the message and the header is colored magenta.

```
>>> from inform import Inform, debug
>>> informer = Inform(prog_name="myprog")
>>> debug('HERE!')
myprog DEBUG: HERE!
```

Generally one does not use the *debug* informant directly. Instead one uses the available debugging functions: `aaa()`, `ddd()`, `ppp()`, `sss()` and `vvv()`.

warn

```
warn = InformantFactory(
    severity='warning',
    header_color='yellow',
    output=lambda informer: not informer.quiet and not informer.mute,
    log=True,
)
```

Displays and logs a warning message. A header with the label *warning* is added to the message. The header is colored yellow when writing to the console.

```
>>> from inform import Inform, warn
>>> informer = Inform(prog_name="myprog")
>>> warn('file not found, skipping.', culprit='ghost')
myprog warning: ghost: file not found, skipping.
```

error

```
error = InformantFactory(
    severity='error',
    is_error=True,
    header_color='red',
    output=lambda informer: not informer.mute,
    log=True,
)
```

Displays and logs an error message. A header with the label *error* is added to the message. The header is colored red when writing to the console.

```
>>> from inform import Inform, error
>>> informer = Inform(prog_name="myprog")
>>> error('invalid value specified, expected a number.', culprit='count')
myprog error: count: invalid value specified, expected a number.
```

fatal

```
fatal = InformantFactory(
    severity='error',
    is_error=True,
    terminate=1,
    header_color='red',
    output=lambda informer: not informer.mute,
    log=True,
)
```

Displays and logs an error message. A header with the label *error* is added to the message. The header is colored red when writing to the console. The program is terminated with an exit status of 1.

panic

```
panic = InformantFactory(
    severity='internal error (please report)',
    is_error=True,
    terminate=3,
    header_color='red',
    output=True,
    log=True,
)
```

Displays and logs a panic message. A header with the label *internal error* is added to the message. The header is colored red when writing to the console. The program is terminated with an exit status of 3.

4.1.3 Informant Control

For more control of the informants, you can import and instantiate the *inform.Inform* class along with the desired informants. This gives you the ability to specify options:

```
>>> from inform import Inform, display, error
>>> Inform(logfile=False, prog_name=False, quiet=True)
<...>

>>> display('hello')

>>> error('file not found.', culprit='data.in')
error: data.in: file not found.
```

In this example the *logfile* argument disables opening and writing to the logfile. The *prog_name* argument stops *Inform* from adding the program name to the error message. And *quiet* turns off non-essential output, and in this case it causes the output of *display* to be suppressed.

An object of the Inform class is referred to as an informer (not to be confused with the print functions, which are referred to as informants). Once instantiated, you can use the informer to change various settings, terminate the program, return a count of the number of errors that have occurred, etc.

```
>>> from inform import Inform, error
>>> informer = Inform(prog_name="prog")

>>> error('file not found.', culprit='data.in')
prog error: data.in: file not found.

>>> informer.errors_accrued()
1
```

You can also use a *with* statement to invoke the informer. This activates the informer for the duration of the *with* statement, returning to the previous informer when the *with* statement terminates. This is useful when writing tests. In this case you can provide your own output streams so that you can access the normally printed output of your code:

```
>>> from inform import Inform, display
>>> import sys
>>> if sys.version[0] == '2':
...     # io assumes unicode, which python2 does not provide by default
...     # so use StringIO instead
...     from StringIO import StringIO
...     # Add support for with statement by monkeypatching
...     StringIO.__enter__ = lambda self: self
...     StringIO.__exit__ = lambda self, exc_type, exc_val, exc_tb: self.close()
... else:
...     from io import StringIO

>>> def run_test():
...     display('running test')

>>> with StringIO() as stdout, \
...      StringIO() as stderr, \
...      StringIO() as logfile, \
...      Inform(stdout=stdout, stderr=stderr, logfile=logfile) as msg:
...         run_test()
...
...         num_errors = msg.errors_accrued()
```

(continues on next page)

(continued from previous page)

```
...         output_text = stdout.getvalue()
...         error_text = stderr.getvalue()
...         logfile_text = logfile.getvalue()

>>> num_errors
0

>>> str(output_text)
'running test\n'

>>> str(error_text)
''

>>> str(logfile_text[:10]), str(logfile_text[-13:])
('Invoked as', 'running test\n')
```

Message Destination

You can specify the output stream when creating an informant. If you do not, then the stream uses is under the control of *Inform's stream_policy* argument.

If *stream_policy* is set to 'termination', then all messages are sent to the standard output except the final termination message, which is set to standard error. This is suitable for programs whose output largely consists of status messages rather than data, and so would be unlikely to be used in a pipeline.

If *stream_policy* is 'header'. then all messages with headers (those messages produced from informants with *severity*) are sent to the standard error stream and all other messages are sent to the standard output. This is more suitable for programs whose output largely consists of data and so would likely be used in a pipeline.

It is also possible for *stream_policy* to be a function that takes three arguments, the informant and the standard output and error streams. It should return the desired stream.

If *True* is passed to the *notify_if_no_tty Inform* argument, then error messages are sent to the notifier if the standard output is not a TTY.

4.1.4 User Defined Informants

You can create your own informants using *inform.InformantFactory*. One application of this is to support multiple levels of verbosity. To do this, an informant would be created for each level of verbosity, as follows:

```
>>> from inform import Inform, InformantFactory

>>> verbose1 = InformantFactory(output=lambda m: m.verbosity >= 1)
>>> verbose2 = InformantFactory(output=lambda m: m.verbosity >= 2)

>>> with Inform(verbosity=0):
...     verbose1('First level of verbosity.')
...     verbose2('Second level of verbosity.')

>>> with Inform(verbosity=1):
...     verbose1('First level of verbosity.')
...     verbose2('Second level of verbosity.')
First level of verbosity.
```

(continues on next page)

(continued from previous page)

```
>>> with Inform(verbosity=2):
...     verbose1('First level of verbosity.')
...     verbose2('Second level of verbosity.')
First level of verbosity.
Second level of verbosity.
```

The argument *verbosity* is not an explicitly supported argument of *inform.Inform*. In this case *Inform* simply saves the value and makes it available as an attribute, and it is this attribute that is queried by the lambda function passed to *InformantFactory* when creating the informants.

Another use for user-defined informants is to create print functions that output is a particular color:

```
>>> from inform import InformantFactory

>>> succeed = InformantFactory(message_color='green')
>>> fail = InformantFactory(message_color='red')

>>> succeed('This message would be green.')
This message would be green.

>>> fail('This message would be red.')
This message would be red.
```

4.1.5 Exceptions

An exception, *inform.Error*, is provided that takes the same arguments as an informant. This allows you to catch the exception and handle it if you like. Any arguments you pass into the exception are retained and are available when processing the exception. The exception provides the *inform.Error.report()* and *inform.Error.terminate()* methods that processes the exception as an error or fatal error if you find that you can do nothing else with the exception.

```
>>> from inform import Inform, Error

>>> Inform(prog_name='myprog')
<...>
>>> try:
...     raise Error('must not be zero.', culprit='naught')
... except Error as e:
...     e.report()
myprog error: naught: must not be zero.
```

inform.Error also provides *inform.Error.get_message()* and *inform.Error.get_culprit()* methods, which return the message and the culprit. You can also cast the exception to a string or call the *inform.Error.render()* method to get a string that contains both the message and the culprit formatted so that it can be shown to the user.

All positional arguments are available in *e.args* and any keyword arguments provided are available in *e.kwargs*.

One common approach to using *inform.Error* is to pass all the arguments that make up the error message as arguments and then assemble them into the message by providing a template. In that way the arguments are directly available to the handler if needed. For example:

```
>>> from difflib import get_close_matches
>>> from inform import Error, codicil, conjoin, fmt

```

(continues on next page)

(continued from previous page)

```
>>> known_names = 'alpha beta gamma delta epsilon'.split()
>>> name = 'alfa'

>>> try:
...     if name not in known_names:
...         raise Error(name, choices=known_names, template="name '{}' is not defined.
↪")
... except Error as e:
...     candidates = get_close_matches(e.args[0], e.choices, 1, 0.6)
...     candidates = conjoin(candidates, conj=' or ')
...     e.report()
...     codicil(fmt('Did you mean {candidates}?'))
myprog error: name 'alfa' is not defined.
    Did you mean alpha?
```

Notice that useful information (*choices*) is passed into the exception that may be useful when processing the exception even though it is not incorporated into the message.

You can override the template by passing a new one to *inform.Error.get_message()*, *inform.Error.render()*, *inform.Error.report()*, or *inform.Error.terminate()*. This can be helpful if you need to translate a message or change it to make it more meaningful to the end user:

```
>>> try:
...     raise Error(name, template="name '{}' is not defined.")
... except Error as e:
...     e.report("'{}' ist nicht definiert.")
myprog error: 'alfa' ist nicht definiert.
```

4.1.6 Utilities

Several utility functions are provided for your convenience. They are often helpful when creating messages.

Color Class

The *inform.Color* class creates colorizers, which are functions used to render text in a particular color. They are like the informants in that they take any number of unnamed arguments that are converted to strings and then joined into a single string, though the result is not printed. Instead, the string is then coded for the chosen color and returned. For example:

```
>> from inform import Color, display

>> green = Color('green')
>> red = Color('red')
>> success = green('pass:')
>> failure = red('FAIL:')

>> failures = {'outrigger': True, 'signalman': False}
>> for name, fails in failures.items():
..     result = failure if fails else success
..     display(result, name)
FAIL: outrigger
pass: signalman
```

When the messages print, the 'pass:' will be green and 'FAIL:' will be red.

The Color class has the concept of a colorscheme. There are three supported schemes: *None*, 'light', and 'dark'. With *None* the text is not colored. In general it is best to use the 'light' colorscheme on 'dark' backgrounds and the 'dark' colorscheme on light backgrounds.

Colorizers have one user settable attribute: *enable*. By default *enable* is *True*. If you set it to *False* the colorizer no longer renders the text in color:

```
>> warning = Color('yellow')
>> warning('This will be yellow on the console.')
This will be yellow on the console.

>> warning.enable = False
>> warning('This will not be yellow.')
This will not be yellow.
```

Alternatively, you can enable or disable the colorizer when creating it. This example uses the *inform.Color.isTTY()* method to determine whether the output stream, the standard output by default, is a console.

```
>> warning = Color('yellow', enable=Color.isTTY())
>> warning('Cannot find precursor, ignoring.')
Cannot find precursor, ignoring.
```

columns

columns(*array*, *pagewidth=79*, *alignment='<'*, *leader=' '*)

inform.columns() distributes the values of an array over enough columns to fill the screen.

This example uses prints out the phonetic alphabet:

```
>>> from inform import columns

>>> title = 'Display the NATO phonetic alphabet.'
>>> words = """
...     Alfa Bravo Charlie Delta Echo Foxtrot Golf Hotel India Juliett Kilo
...     Lima Mike November Oscar Papa Quebec Romeo Sierra Tango Uniform
...     Victor Whiskey X-ray Yankee Zulu
... """.split()

>>> display(title, columns(words), sep='\n')
Display the NATO phonetic alphabet.
    Alfa      Echo      India     Mike      Quebec    Uniform   Yankee
    Bravo     Foxtrot   Juliett   November  Romeo     Victor    Zulu
    Charlie   Golf      Kilo      Oscar     Sierra    Whiskey
    Delta     Hotel     Lima      Papa      Tango     X-ray
```

conjoin

conjoin(*iterable*, *conj=' and '*, *sep=', '*)

inform.conjoin() is like ''.join(), but allows you to specify a conjunction that is placed between the last two elements. For example:

```
>>> from inform import conjoin
>>> conjoin(['a', 'b', 'c'])
'a, b and c'
```

(continues on next page)

(continued from previous page)

```
>>> conjoin(['a', 'b', 'c'], conj=' or ')
'a, b or c'
```

cull

cull (*collection*[, *remove*])

inform.cull() strips items from a collection that have a particular value. The collection may be list-like (*list*, *tuple*, *set*, etc.) or a dictionary-like (*dict*, *OrderedDict*). A new collection of the same type is returned with the undesirable values removed.

By default, *inform.cull()* strips values that would be *False* when cast to a Boolean (0, *False*, *None*, '', (), [], etc.). A particular value may be specified using the *remove* as a keyword argument. The value of remove may be a collection, in which case any value in the collection is removed, or it may be a function, in which case it takes a single item as an argument and returns *True* if that item should be removed from the list.

```
>>> from inform import cull, display
>>> display(*cull(['a', 'b', '', 'd']), sep=', ')
a, b, d

>>> accounts = dict(checking=1100.16, savings=13948.78, brokerage=0)
>>> for name, amount in sorted(cull(accounts).items()):
...     display(name, amount, template='{:>10s}: ${:,.2f}')
  checking: $1,100.16
   savings: $13,948.78
```

fmt

fmt (*msg*, **args*, ***kwargs*)

inform.fmt() is similar to ''.format(), but it can pull arguments from the local scope.

```
>>> from inform import conjoin, display, fmt, plural

>>> filenames = ['a', 'b', 'c', 'd']
>>> filetype = 'CSV'
>>> display(
...     fmt(
...         'Reading {filetype} {files}: {names}.',
...         files=plural(filenames, 'file'),
...         names=conjoin(filenames),
...     )
... )
Reading CSV files: a, b, c and d.
```

Notice that *filetype* was not explicitly passed into *fmt()* even though it was explicitly called out in the format string. *filetype* can be left out of the argument list because if *fmt* does not find a named argument in its argument list, it will look for a variable of the same name in the local scope.

full_stop

full_stop (*string*)

inform.full_stop() adds a period to the end of the string if needed (if the last character is not a period, question mark or exclamation mark). It applies str() to its argument, so it is generally a suitable replacement for str in str(exception) when trying extract an error message from an exception.

This is generally useful if you need to print a string that should have punctuation, but may not.

```
>>> from inform import Error, error, full_stop

>>> found = 0
>>> try:
...     if found is False:
...         raise Error('not found', culprit='marbles')
...     elif found < 3:
...         raise Error('insufficient number.', culprit='marbles')
...     raise Error('not found', culprit='marbles')
... except Error as e:
...     error(full_stop(e))
myprog error: marbles: insufficient number.
```

indent

indent(*text*, *leader=' '*, *first=0*, *stops=1*, *sep='\n'*)

inform.indent() indents *text*. Multiples of *leader* are added to the beginning of the lines to indent. *first* is the number of indentations used for the first line relative to the others (may be negative but (first + stops) should not be. *stops* is the default number of indentations to use. *sep* is the string used to separate the lines.

```
>>> from inform import display, indent
>>> text = 'a b'.replace(' ', '\n')
>>> display(indent(text))
    a
    b

>>> display(indent(text, first=1, stops=0))
    a
b

>>> display(indent(text, leader='.   ', first=-1, stops=2))
.   a
.   .   b
```

is_collection

is_collection(*obj*)

inform.is_collection() returns *True* if its argument is a collection. This includes objects such as lists, tuples, sets, dictionaries, etc. It does not include strings.

```
>>> from inform import is_collection

>>> is_collection('abc')
False

>>> is_collection(['a', 'b', 'c'])
True
```

is_iterable

is_iterable(*obj*)

inform.is_iterable() returns *True* if its argument is a collection or a string.

```
>>> from inform import is_iterable

>>> is_iterable('abc')
True

>>> is_iterable(['a', 'b', 'c'])
True
```

is_str

is_str(*obj*)

inform.is_str() returns *True* if its argument is a string-like object.

```
>>> from inform import is_str

>>> is_str('abc')
True

>>> is_str(['a', 'b', 'c'])
False
```

join

join(**args*, ***kwargs*)

inform.join() combines the arguments in a manner very similar to an *informant* and returns the result as a string. Uses the *sep*, *template* and *wrap* keyword arguments to combine the arguments.

```
>>> from inform import display, join

>>> accounts = dict(checking=1100.16, savings=13948.78, brokerage=0)
>>> lines = []
>>> for name, amount in accounts.items():
...     lines.append(join(name, amount, template='{:>10s}: ${:,.2f}'))

display(lines, sep='\n')
 brokerage: $0.00
  checking: $1,100.16
   savings: $13,948.78
```

os_error

os_error(*exception*)

inform.os_error() generates clean messages for operating system errors.

```
>>> from inform import error, os_error

>>> try:
...     with open('temperatures.csv') as f:
...         contents = f.read()
... except (OSError, IOError) as e:
...     error(os_error(e))
myprog error: temperatures.csv: no such file or directory.
```

plural

plural(*count*, *singular_form*, *plural_form=*None**)

Produces either the singular or plural form of a word based on a count. The count may be an integer, or an iterable, in which case its length is used. If the plural form is not given, the singular form is used with an 's' added to the end.

```
>>> from inform import conjoin, display, plural

>>> filenames = ['a', 'b', 'c', 'd']
>>> display(
...     files=plural(filenames, 'file'), names=conjoin(filenames),
...     template='Reading {files}: {names}.'
... )
Reading files: a, b, c and d.
```

render

render(*obj*, *sort=None*, *level=0*, *tab=' '*)

inform.render() recursively converts an object to a string with reasonable formatting. Has built in support for the base Python types (*None*, *bool*, *int*, *float*, *str*, *set*, *tuple*, *list*, and *dict*). If you confine yourself to these types, the output of *inform.render()* can be read by the Python interpreter. Other types are converted to string with *repr()*. The dictionary keys and set values are sorted if sort is *True*. Sometimes this is not possible because the values are not comparable, in which case render reverts to the natural order.

This example prints several Python data types:

```
>>> from inform import render, display
>>> s1='alpha string'
>>> s2='beta string'
>>> n=42
>>> S={s1, s2}
>>> L=[s1, n, S]
>>> d = {1:s1, 2:s2}
>>> D={'s': s1, 'n': n, 'S': S, 'L': L, 'd':d}
>>> display('D', '=', render(D, True))
D = {
    'L': [
        'alpha string',
        42,
        {'alpha string', 'beta string'},
    ],
    'S': {'alpha string', 'beta string'},
    'd': {1: 'alpha string', 2: 'beta string'},
    'n': 42,
```

(continues on next page)

(continued from previous page)

```
    's': 'alpha string',
}

>>> E={'s': s1, 'n': n, 'S': S, 'L': L, 'd':d, 'D':D}
>>> display('E', '=', render(E, True))
E = {
    'D': {
        'L': [
            'alpha string',
            42,
            {'alpha string', 'beta string'},
        ],
        'S': {'alpha string', 'beta string'},
        'd': {1: 'alpha string', 2: 'beta string'},
        'n': 42,
        's': 'alpha string',
    },
    'L': [
        'alpha string',
        42,
        {'alpha string', 'beta string'},
    ],
    'S': {'alpha string', 'beta string'},
    'd': {1: 'alpha string', 2: 'beta string'},
    'n': 42,
    's': 'alpha string',
}
```

4.1.7 Debugging Functions

The debugging functions are intended to be used when you want to print something out when debugging your program. They are colorful to make it easier to find them among the program's normal output, and a header is added that describes the location they were called from. This makes it easier to distinguish several debug message and also makes it easy to find and remove the functions once you are done debugging.

aaa

aaa(*arg*)

inform.aaa() prints and then returns its argument. The argument may be name or unnamed. If named, the name is used as a label when printing the value of the argument. It can be used to print the value of a term within an expression without being forced to replicate that term.

In the following example, a critical statement is instrumented to show the intermediate values in the computation. In this case it would be difficult to see these intermediate values by replicating code, as calls to the *update* method has the side effect of updating the state of the integrator.

```
>>> from inform import aaa, display
>>> class Integrator:
...     def __init__(self, ic=0):
...         self.state = ic
...     def update(self, vin):
...         self.state += vin
...         return self.state
```

(continues on next page)

(continued from previous page)

```
>>> int1 = Integrator(1)
>>> int2 = Integrator()
>>> vin = 1
>>> vout = 0
>>> for t in range(1, 3):
...     vout = 0.7*aaa(int2=int2.update(aaa(int1=int1.update(vin-vout))))
...     display('vout = {}'.format(vout))
myprog DEBUG: <doctest user.rst[133]>, 2, __main__: int1: 2
myprog DEBUG: <doctest user.rst[133]>, 2, __main__: int2: 2
vout = 1.4
myprog DEBUG: <doctest user.rst[133]>, 2, __main__: int1: 1.6
myprog DEBUG: <doctest user.rst[133]>, 2, __main__: int2: 3.6
vout = 2.52
```

ddd

ddd(**args*, ***kwargs*)

inform.ddd() pretty prints all of both its unnamed and named arguments.

```
>>> from inform import ddd
>>> a = 1
>>> b = 'this is a test'
>>> c = (2, 3)
>>> d = {'a': a, 'b': b, 'c': c}
>>> ddd(a, b, c, d)
myprog DEBUG: <doctest user.rst[139]>, 1, __main__:
    1
    'this is a test'
    (2, 3)
    {
        'a': 1,
        'b': 'this is a test',
        'c': (2, 3),
    }
```

If you give named arguments, the name is prepended to its value:

```
>>> from inform import ddd
>>> ddd(a=a, b=b, c=c, d=d, s='hey now!')
myprog DEBUG: <doctest user.rst[141]>, 1, __main__:
    a = 1
    b = 'this is a test'
    c = (2, 3)
    d = {
        'a': 1,
        'b': 'this is a test',
        'c': (2, 3),
    }
    s = 'hey now!'
```

If an arguments has a __dict__ attribute, it is printed rather than the argument itself.

```
>>> from inform import ddd

>>> class Info:
...     def __init__(self, **kwargs):
...         self.__dict__.update(kwargs)
...         ddd(self=self)

>>> contact = Info(email='ted@ledbelly.com', name='Ted Ledbelly')
myprog DEBUG: <doctest user.rst[143]>, 4, __main__.Info.__init__():
    self = Info object containing {
        'email': 'ted@ledbelly.com',
        'name': 'Ted Ledbelly',
    }
```

ppp

ppp(**args*, ***kwargs*)

inform.ppp() is very similar to the normal Python print function in that it prints out the values of the unnamed arguments under the control of the named arguments. It also takes the same named arguments as `print()`, such as `sep` and `end`.

If given without unnamed arguments, it will just print the header, which good way of confirming that a line of code has been reached.

```
>>> from inform import ppp
>>> a = 1
>>> b = 'this is a test'
>>> c = (2, 3)
>>> d = {'a': a, 'b': b, 'c': c}
>>> ppp(a, b, c)
myprog DEBUG: <doctest user.rst[150]>, 1, __main__: 1 this is a test (2, 3)
```

sss

sss()

inform.sss() prints a stack trace, which can answer the *How did I get here?* question better than a simple print function.

```
>> from inform import sss

>> def foo():
..     sss()
..     print('CONTINUING')

>> foo()
DEBUG: <doctest user.rst[142]>:2, __main__.foo():
    Traceback (most recent call last):
        ...
CONTINUING
```

vvv

vvv(**args*)

inform.vvv() prints variables from the calling scope. If no arguments are given, then all the variables are printed. You can optionally give specific variables on the argument list and only those variables are printed.

```
>>> from inform import vvv

>>> vvv(b, d)
myprog DEBUG: <doctest user.rst[152]>, 1, __main__:
    b = 'this is a test'
    d = {
        'a': 1,
        'b': 'this is a test',
        'c': (2, 3),
    }
```

This last feature is not completely robust. The checking is done by value, so if several variables share the value of one requested, they are all shown.

```
>>> from inform import vvv

>>> aa = 1
>>> vvv(a)
myprog DEBUG: <doctest user.rst[155]>, 1, __main__:
    a = 1
    aa = 1
    vin = 1
```

Site Customization

Many people choose to add the importing of the debugging function to their usercustomize.py file. In this way, the debugging functions are always available without the need to explicitly import them. To accomplish this, create a *usercustomize.py* files that contains the following and place it in your site-packages directory:

```
# Include Inform debugging routines
try:                # python3
    import builtins
except ImportError:  # python2
    import __builtin__ as builtins

from inform import aaa, ddd, ppp, sss, vvv
builtins.aaa = aaa
builtins.ddd = ddd
builtins.ppp = ppp
builtins.sss = sss
builtins.vvv = vvv
```

The path of this file is typically . . . */lib/pythonN.M/site-packages/usercustomize.py* where *M.N* is the version number of your python.

4.1.8 Inform Helper Functions

An informer (an *inform.Inform* object) provides a number of useful methods. However, it is common that the informer is not locally available. To avoid the clutter that would be created by passing the informer around to where ever it is needed, *Inform* gives you several alternate ways of accessing these methods. Firstly is *inform.get_informer()*, which simply returns the currently active informer. Secondly, *Inform* provides a collection of functions that provide direct access to the corresponding methods on the currently active informer. They are:

done

done(*exit=True*)

inform.done() terminates the program with the normal exit status. It calls *inform.Inform.done()* for the active informer.

If the *exit* argument is False, preparations are made for exiting, but *sys.exit* is not called. Instead, the desired exit status is returned.

terminate

terminate(*status=None*, *exit=True*)

inform.terminate() terminates the program with specified exit status or message. It calls *inform.Inform.terminate()* for the active informer.

status may be an integer, boolean, string, or None. An exit status of 1 is used if True or a string is passed in. If None is passed in then 1 is used for the exit status if an error was reported and 0 otherwise.

If the *exit* argument is False, preparations are made for exiting, but *sys.exit* is not called. Instead, the desired exit status is returned.

terminate_if_errors

terminate_if_errors(*status=None*, *exit=True*)

inform.terminate_if_errors() terminates the program with specified exit status or message if an error was previously reported. It calls *inform.Inform.terminate_if_errors()* for the active informer.

status may be an integer, boolean, or string. An exit status of 1 is used if True or a string is passed in.

If the *exit* argument is False, preparations are made for exiting, but *sys.exit* is not called. Instead, the desired exit status is returned.

errors_accrued

errors_accrued(*reset=False*)

inform.errors_accrued() returns the number of errors that have been reported. It calls *inform.Inform.errors_accrued()* for the active informer.

If the *reset* argument is True, the error count is reset to 0.

get_prog_name

get_prog_name()

inform.get_prog_name() returns the name of the program. It calls *inform.Inform.get_prog_name()* for the active informer.

get_informer

get_informer()

inform.get_informer() returns the currently active informer.

set_culprit

set_culprit(*culprit*)

inform.set_culprit() saves a culprit in the informer for later use. Any existing saved culprit is temporarily moved out of the way. It calls *inform.Inform.set_culprit()* for the active informer.

A culprit is a string, number, or tuple of strings or numbers that would be prepended to a message to indicate the object of the message.

inform.Inform.set_culprit() is used with Python's *with* statement. The original saved culprit is restored when the *with* statement exits.

See *Culprits* for an example of *inform.set_culprit()* use.

add_culprit

add_culprit(*culprit*)

inform.add_culprit() appends a culprit to any existing saved culprit. It calls *inform.Inform.add_culprit()* for the active informer.

A culprit is a string, number, or tuple of strings or numbers that would be prepended to a message to indicate the object of the message.

inform.Inform.add_culprit() is used with Python's *with* statement. The original saved culprit is restored when the *with* statement exits.

See *Culprits* for an example of *inform.add_culprit()* use.

get_culprit

get_culprit(*culprit=None*)

inform.get_culprit() returns the specified culprit, if any, appended to the end of the current culprit that is saved in the informer. The resulting culprit is always returned as a tuple. It calls *inform.Inform.get_culprit()* for the active informer.

A culprit is a string, number, or tuple of strings or numbers that would be prepended to a message to indicate the object of the message.

See *Culprits* for an example of *inform.get_culprit()* use.

4.2 Classes and Functions

4.2.1 Inform

The Inform class controls the active informants.

class inform.**Inform**(*mute=False*, *quiet=False*, *verbose=False*, *narrate=False*, *logfile=False*, *prog_name=True*, *argv=None*, *version=None*, *termination_callback=None*, *colorscheme='dark'*, *flush=False*, *stdout=None*, *stderr=None*, *length_thresh=80*, *culprit_sep=' '*, *stream_policy='termination'*, *notify_if_no_tty=False*, *notifier='notify-send'*, ***kwargs*)

Manages all informants, which in turn handle user messaging. Generally informants copy messages to the logfile while most also send to the standard output as well, however all is controllable.

Parameters

- **mute** (*bool*) – All output is suppressed (it is still logged).

  With the provided informants all output is suppressed when set (it is still logged). This is generally used when the program being run is being run by another program that is generating its own messages and does not want the user confused by additional messages. In this case, the calling program is responsible for observing and reacting to the exit status of the called program.

- **quiet** (*bool*) – Normal output is suppressed (it is still logged).

  With the provided informants normal output is suppressed when set (it is still logged). This is used when the user has indicated that they are uninterested in any conversational messages and just want to see the essentials (generally error messages).

- **verbose** (*bool*) – Comments are output to user, normally they are just logged.

  With the provided informants comments are output to user when set; normally they are just logged. Comments are generally used to document unusual occurrences that might warrant the user's attention.

- **narrate** (*bool*) – Narration is output to user, normally it is just logged.

  With the provided informants narration is output to user when set, normally it is just logged. Narration is generally used to inform the user as to what is going on. This can help place errors and warnings in context so that they are easier to understand.

- **logfile** (*path, string, stream, bool*) – May be a pathlib path or a string, in which case it is taken to be the path of the logfile. May be *True*, in which case ./.<prog_name>.log is used. May be an open stream. Or it may be *False*, in which case no log file is created.

- **prog_name** (*string*) – The program name. Is appended to the message headers and used to create the default logfile name. May be a string, in which case it is used as the name of the program. May be *True*, in which case *basename(argv[0])* is used. May be *False* to indicate that program name should not be added to message headers.

- **argv** (*list of strings*) – System command line arguments (logged). By default, *sys.argv* is used. If *False* is passed in, *argv* is not logged and *argv[0]* is not available to be the program name.

- **version** (*string*) – program version (logged if provided).

- **termination_callback** (*func*) – A function that is called at program termination.

- **colorscheme** (*None*, 'light', or 'dark') – Color scheme to use. *None* indicates that messages should not be colorized. Colors are not used if desired output stream is not a TTY.

- **flush** (*bool*) – Flush the stream after each write. Is useful if your program is crashing, causing loss of the latest writes. Can cause programs to run considerably slower if they produce a lot of output. Not available with python2.

- **stdout** (*stream*) – Messages are sent here by default. Generally used for testing. If not given, *sys.stdout* is used.

- **stderr** (*stream*) – Exceptional messages are sent here by default. Exceptional message include termination messages and possibly error messages (depends on stream_policy). Generally used for testing. If not given, *sys.stderr* is used.

- **length_thresh** (*integer*) – Split header from body if line length would be greater than this threshold.

- **culprit_sep** (*string*) – Join string used for culprit collections. Default is ‘, ‘.
- **stream_policy** (*string or func*) – The default stream policy, which determines which stream each informant uses by default (which stream is used if the stream is not specifically specified when the informant is created).

  The following named policies are available:

  ’termination’: stderr is used for the final termination message. stdout is used otherwise. This is generally used for programs are not filters (the output is largely status rather than data that might be fed into another program through a pipeline).

  ’header’: stderr is used for all messages with headers/severities. stdout is used otherwise. This is generally used for programs are filters (the output is largely data that might be fed into another program through a pipeline). In this case stderr is used for error messages so they do not pollute the data stream.

  May also be a function that returns the stream and takes three arguments: the active informant, Inform’s stdout, and Inform’s stderr.

  If no stream is specified, either explicitly on the informant when it is defined, or through the stream policy, then Inform’s stdout is used.
- **notify_if_no_tty** (*bool*) – If it appears that an error message is expecting to displayed on the console but the standard output is not a TTY send it to the notifier if this flag is True.
- **notifier** (*str*) – Command used to run the notifier. The command will be called with two arguments, the header and the body of the message.
- ****kwargs** – Any additional keyword arguments are made attributes that are ignored by *Inform*, but may be accessed by the informants.

add_culprit(*culprit*)
Add to the currently saved culprit.

Similar to *Inform.set_culprit()* except that this method appends the given culprit to the cached culprit rather than simply replacing it.

Parameters **culprit** (*string, number or tuple of strings and numbers*) – A culprit or collection of culprits that are cached with the intent that they be available to be included in a message upon demand. They generally are used to indicate what a message refers to.

This function is designed to work as a context manager, meaning that it meant to be used with Python’s *with* statement. It temporarily replaces any existing culprit, but that culprit in reinstated upon exiting the *with* statement. Once a culprit is saved, *inform.Inform.get_culprit()* is used to access it.

See *Inform.set_culprit()* for an example of a closely related method.

disconnect()
Disconnect informer, returning to previous informer.

done(*exit=True*)
Terminate the program with normal exit status.

Parameters **exit** (*bool*) – If False, all preparations for termination are done, but sys.exit() is not called. Instead, the exit status is returned.

Returns The desired exit status is returned if exit is False (the function does not return if exit is True).

errors_accrued(*reset=False*)
Returns number of errors that have accrued.

Parameters **reset** (*bool*) – Reset the error count to 0 if *True*.

flush_logfile()
Flush the logfile.

get_culprit(*culprit=None*)
Get the current culprit.

Return the currently cached culprit as a tuple. If a culprit is specified as an argument, it is appended to the cached culprit without modifying it.

Parameters **culprit** (*string, number or tuple of strings and numbers*) – A culprit or collection of culprits that is appended to the return value without modifying the cached culprit.

Returns The culprit argument is appended to the cached culprit and the combination is returned. The return value is always in the form of a tuple even if there is only one component.

See *Inform.set_culprit()* for an example use of this method.

get_prog_name()
Returns the program name.

set_culprit(*culprit*)
Set the culprit while temporarily displacing current culprit.

Squirrels away a culprit for later use. Any existing culprit is moved out of the way.

Parameters **culprit** (*string, number or tuple of strings and numbers*) – A culprit or collection of culprits that are cached with the intent that they be available to be included in a message upon demand. They generally are used to indicate what a message refers to.

This function is designed to work as a context manager, meaning that it meant to be used with Python's *with* statement. It temporarily replaces any existing saved culprit, but that culprit in reinstated upon exiting the *with* statement. Once a culprit is saved, *inform.Inform.get_culprit()* is used to access it. For example:

```
>>> from inform import get_culprit, set_culprit, warn

>>> def count_lines(lines):
...     empty = 0
...     for lineno, line in enumerate(lines):
...         if not line:
...             warn('empty line.', culprit=get_culprit(lineno))

>>> filename = 'setup.py'
>>> with open(filename) as f, set_culprit(filename):
...     lines = f.read().splitlines()
...     num_lines = count_lines(lines)
warning: setup.py, 5: empty line.
warning: setup.py, 8: empty line.
warning: setup.py, 13: empty line.
```

set_logfile(*logfile, encoding='utf-8'*)
Allows you to change the logfile (only available as a method).

Parameters

- **logfile** – May be a pathlib path. May be a string, in which case it is taken to be the path of the logfile. May be *True*, in which case ./.<prog_name>.log is used. May be an open stream. Or it may be *False*, in which case no log file is created.
- **encoding** (*string*) – The encoding to use when writing the file.

set_stream_policy(*stream_policy*)
Allows you to change the stream policy (see *inform.Inform*).

suppress_output(*mute=True*)
Allows you to change the mute flag (only available as a method).

Parameters **mute** (*bool*) – If *mute* is True all output is suppressed (it is still logged).

terminate(*status=None*, *exit=True*)
Terminate the program with specified exit status.

Parameters

- **status** (*int, bool, string, or None*) – The desired exit status or exit message. Exit status is 1 if True is passed in. When None, return 1 if errors occurred and 0 otherwise
- **exit** (*bool*) – If False, all preparations for termination are done, but sys.exit() is not called. Instead, the exit status is returned.

Returns The desired exit status is returned if exit is False (the function does not return if exit is True).

Recommended status codes:

0: success
1: unexpected error
2: invalid invocation
3: panic

Status may also be a string, in which case it is printed to stderr and the exit status is 1.

terminate_if_errors(*status=1*, *exit=True*)
Terminate the program if error count is nonzero.

Parameters

- **status** (*int, bool or string*) – The desired exit status or exit message.
- **exit** (*bool*) – If False, all preparations for termination are done, but sys.exit() is not called. Instead, the exit status is returned.

Returns None is returned if there is no errors, otherwise the desired exit status is returned if exit is False (the function does not return if there is an error and exit is True).

Direct Access Functions

Several of the above methods are also available as stand-alone functions that act on the currently active informer. This make it easy to use their functionality even if you do not have local access to the informer.

inform.**done**(*exit=True*)
Terminate the program with normal exit status.

Calls *inform.Inform.done()* for the active informer.

inform.**terminate**(*status=None*, *exit=True*)
Terminate the program with specified exit status."

Calls *inform.Inform.terminate()* for the active informer.

inform.**terminate_if_errors**(*status=1*, *exit=True*)
Terminate the program if error count is nonzero."

Calls *inform.Inform.terminate_if_errors()* for the active informer.

inform.**errors_accrued**(*reset=False*)
Returns number of errors that have accrued."

Calls *inform.Inform.errors_accrued()* for the active informer.

inform.**get_prog_name**()
Returns the program name.

Calls *inform.Inform.get_prog_name()* for the active informer.

inform.**set_culprit**(*culprit*)
Set the culprit while displacing current culprit.

Calls *inform.Inform.set_culprit()* for the active informer.

inform.**add_culprit**(*culprit*)
Append to the end of the current culprit.

Calls *inform.Inform.add_culprit()* for the active informer.

inform.**get_culprit**(*culprit=None*)
Get the current culprit.

Calls *inform.Inform.get_culprit()* for the active informer.

You can also request the active informer:

inform.**get_informer**()
Returns the active informer.

4.2.2 InformantFactory

class inform.**InformantFactory**(*severity=None*, *is_error=False*, *log=True*, *output=True*, *notify=False*, *terminate=False*, *is_continuation=False*, *message_color=None*, *header_color=None*, *stream=None*)
An object of InformantFactory is referred to as an informant. It is generally treated as a function that is called to produce the desired output.

Parameters

- **severity** (*string*) – Messages with severities get headers. The header consists of the severity, the program name (if desired), and the culprit (if provided). If the message text does not contain a newline it is appended to the header. Otherwise the message text is indented and placed on the next line.
- **is_error** (*bool*) – Message is counted as an error.
- **log** (*bool*) – Send message to the log file. May be a boolean or a function that accepts the informer as an argument and returns a boolean.
- **output** (*bool*) – Send message to the output stream. May be a boolean or a function that accepts the informer as an argument and returns a boolean.

- **notify** (*bool*) – Send message to the notifier. The notifier will display the message that appears temporarily in a bubble at the top of the screen. May be a boolean or a function that accepts the informer as an argument and returns a boolean.
- **terminate** (*bool or integer*) – Terminate the program. Exit status is the value of *terminate* unless *terminate* is *True*, in which case 1 is returned if an error occurred and 0 otherwise.
- **is_continuation** (*bool*) – This message is a continuation of the previous message. It will use the properties of the previous message (output, log, message color, etc) and if the previous message had a header, that header is not output and instead the message is indented.
- **message_color** (*string*) – Color used to display the message. Choose from: *black*, *red*, *green*, *yellow*, *blue*, *magenta*, *cyan* or *white*.
- **header_color** (*string*) – Color used to display the header, if one is produced. Choose from: *black*, *red*, *green*, *yellow*, *blue*, *magenta*, *cyan* or *white*.
- **stream** (*stream*) – Output stream to use. Typically sys.stdout or sys.stderr. If not specified, the stream to use will be determine by stream policy of active informer.

Example

The following generates two informants, *passes*, which prints its messages in green, and *fails*, which prints its messages in red. Output to the standard output for both is suppressed if *quiet* is *True*:

```
>>> from inform import InformantFactory

>>> passes = InformantFactory(
...     output=lambda inform: not inform.quiet,
...     log=True,
...     message_color='green',
... )
>>> fails = InformantFactory(
...     output=lambda inform: not inform.quiet,
...     log=True,
...     message_color='red',
... )
```

pass and *fail* are both informants. Once created, the can be used to give messages to the user:

```
>>> results = [
...     (0,   0.005, 0.025),
...     (0.5, 0.512, 0.025),
...     (1,   0.875, 0.025),
... ]
>>> for expected, measured, tolerance in results:
...     if abs(expected - measured) > tolerance:
...         report, label = fails, 'FAIL'
...     else:
...         report, label = passes, 'Pass'
...     report(
...         label, measured, expected, measured-expected,
...         template='{}: measured = {:.3f}V, expected = {:.3f}V, diff = {:.3f}V'
...     )
Pass: measured = 0.005V, expected = 0.000V, diff = 0.005V
```

(continues on next page)

(continued from previous page)

```
Pass: measured = 0.512V, expected = 0.500V, diff = 0.012V
FAIL: measured = 0.875V, expected = 1.000V, diff = -0.125V
```

In the console the passes are rendered in green and the failures in red.

4.2.3 Inform Utilities

inform.**indent**(*text*, *leader=' '*, *first=0*, *stops=1*, *sep='\n'*)

Add indentation.

Parameters

- **leader** (*string*) – the string added to be beginning of a line to indent it.
- **first** (*integer*) – number of indentations for the first line relative to others (may be negative but (first + stops) should not be).
- **stops** (*integer*) – number of indentations (number of leaders to add to the beginning of each line).
- **sep** (*string*) – the string used to separate the lines

Example:

```
>>> from inform import display, indent
>>> display(indent('And the answer is ...\n42!', first=-1))
And the answer is ...
    42!
```

inform.**cull**(*collection*, ***kwargs*)

Cull items of a particular value from a collection.

Parameters

- **collection** – The collection may be list-like (list, tuples, sets, etc.) or dictionary-like (dict, OrderedDict, etc.). A new collection of the same type is returned with the undesirable values removed.
- **remove** – Must be specified as keyword argument. May be a function, a collection, or a scalar. The function would take a single argument, one of the values in the collection, and return True if the value should be culled. The scalar or the collection simply specified the specific value or values to be culled.

  If remove is not specified, the value is culled if its value would be False when cast to a boolean (0, False, None, '', (), [], {}, etc.)

Example:

```
>>> from inform import cull, display
>>> from collections import OrderedDict
>>> fruits = OrderedDict([
...     ('a','apple'), ('b','banana'), ('c','cranberry'), ('d','date'),
...     ('e',None), ('f',None), ('g','guava'),
... ])
>>> display(*cull(list(fruits.values())), sep=', ')
apple, banana, cranberry, date, guava

>>> for k, v in cull(fruits).items():
```

(continues on next page)

(continued from previous page)

```
...     display('{k} is for {v}'.format(k=k, v=v))
a is for apple
b is for banana
c is for cranberry
d is for date
g is for guava
```

inform.**is_str**(*arg*)

Identifies strings in all their various guises.

Returns *True* if argument is a string.

Example:

```
>>> from inform import is_str
>>> is_str('abc')
True

>>> is_str(['a', 'b', 'c'])
False
```

inform.**is_iterable**(*obj*)

Identifies objects that can be iterated over, including strings.

Returns *True* if argument is a collecton or a string.

Example:

```
>>> from inform import is_iterable
>>> is_iterable('abc')
True

>>> is_iterable(['a', 'b', 'c'])
True
```

inform.**is_collection**(*obj*)

Identifies objects that can be iterated over, excluding strings.

Returns *True* if argument is a collecton (tuple, list, set or dictionary).

Example:

```
>>> from inform import is_collection
>>> is_collection('abc')
False

>>> is_collection(['a', 'b', 'c'])
True
```

class inform.**Color**(*color*, *scheme=True*, *enable=True*)

Used to create colorizers, which are used to render text in a particular color.

Parameters

- **color** (*string*) – The desired color. Choose from: *black red green yellow blue magenta cyan white*.
- **scheme** (*string*) – Use the specified colorscheme when rendering the text. Choose from *None*, 'light' or 'dark', default is 'dark'.

- **enable** (*bool*) – If set to False, the colorizer does not render the text in color.

Example

```
>>> from inform import Color
>>> fail = Color('red')
```

In this example, *fail* is a colorizer. It behave just like *inform.join()* in that it combines its arguments into a string that it returns. The difference is that colorizers add color codes that will cause most terminals to display the string in the desired color.

Like *inform.join()*, colorizers take the following arguments:

unnamed arguments: The unnamed arguments are converted to strings and joined to form the text to be colored.

sep = ' ': The join string, used when joining the unnamed arguments.

template = None: A template that if present interpolates the arguments to form the final message rather than simply joining the unnamed arguments with *sep*. The template is a string, and its *format* method is called with the unnamed and named arguments of the message passed as arguments.

wrap = False: Specifies whether message should be wrapped. *wrap* may be True, in which case the default width of 70 is used. Alternately, you may specify the desired width. The wrapping occurs on the final message after the arguments have been joined.

scheme = *False*: Use to override the colorscheme when rendering the text. Choose from *None*, *False*, 'light' or 'dark'. If you specify *False* (the default), the colorscheme specified when creating the colorizer is used.

static isTTY(*stream=<_io.TextIOWrapper name='<stdout>' mode='w' encoding='UTF-8'>*)

Takes a stream as an argument and returns true if it is a TTY.

Parameters **stream** (*stream*) – Stream to test. If not given, *stdout* is used as the stream.

```
>>> from inform import Color, display
>>> import sys, re
```

```
>>> if Color.isTTY(sys.stdout):
...     emphasize = Color('magenta')
... else:
...     emphasize = str.upper
```

```
>>> def highlight(matchobj):
...     return emphasize(matchobj.group(0))
```

```
>>> display(re.sub('your', highlight, 'Imagine your city without cars.'))
Imagine YOUR city without cars.
```

classmethod strip_colors(*text*)

Takes a string as its input and return that string stripped of any color codes.

4.2.4 User Utilities

inform.**fmt**(*message*, **args*, ***kwargs*)

Similar to ''.format(), but it can pull arguments from the local scope.

Convert a message with embedded attributes to a string. The values for the attributes can come from the argument list, as with ''.format(), or they may come from the local scope (found by introspection).

Examples:

```
>>> from inform import fmt
>>> s = 'str var'
>>> d = {'msg': 'dict val'}
>>> class Class:
...     a = 'cls attr'

>>> display(fmt("by order: {0}, {1[msg]}, {2.a}.", s, d, Class))
by order: str var, dict val, cls attr.

>>> display(fmt("by name: {S}, {D[msg]}, {C.a}.", S=s, D=d, C=Class))
by name: str var, dict val, cls attr.

>> display(fmt("by magic: {s}, {d[msg]}, {c.a}."))
by magic: str var, dict val, cls attr.
```

You can change the level at which the introspection occurs using the _lvl keyword argument.

_lvl=0 searches for variables in the scope that calls fmt(), the default

_lvl=-1 searches in the parent of the scope that calls fmt()

_lvl=-2 searches in the grandparent, etc.

_lvl=1 search root scope, etc.

inform.**join**(**args*, ***kwargs*)

Combines arguments into a string.

Combines the arguments in a manner very similar to an informant and returns the result as a string. Uses the *sep*, *template* and *wrap* keyword arguments to combine the arguments.

If *template* is specified it controls how the arguments are combined and the result returned. Otherwise the unnamed arguments are joined using the separator and returned.

Parameters

- **sep** (*string*) – Use specified string as join string rather than single space. The unnamed arguments will be joined with using this string as a separator. Default is ' '.
- **template** (*string or collection of strings*) – A python format string. If specified, the unnamed and named arguments are combined under the control of the strings format method. This may also be a collection of strings, in which case each is tried in sequence, and the first for which all the interpolated arguments are known is used. By default, an argument is 'known' if it would be True if casted to a boolean.
- **remove** – Used if *template* is a collection.

  May be a function, a collection, or a scalar. The function would take a single argument, one of the values in the collection, and return True if the value should not be considered known. The scalar or the collection simply specified the specific value of values that should not be considered known.

  If remove is not specified, the value should not be considered known if its value would be False when cast to a boolean (0, False, None, '', (), [], {}, etc.)

- **wrap** (*bool or int*) – If true the string is wrapped using a width of 70. If an integer value is passed, is used as the width of the wrap.

Examples:

```
>>> from inform import join
>>> join('a', 'b', 'c', x='x', y='y', z='z')
'a b c'

>>> join('a', 'b', 'c', x='x', y='y', z='z', template='{2} {z}')
'c z'
```

inform.**render**(*obj*, *sort=None*, *level=0*, *tab=' '*)

Recursively convert object to string with reasonable formatting.

Parameters

- **obj** – The object to render
- **sort** (*bool*) – Dictionary keys and set values are sorted if *sort* is *True*. Sometimes this is not possible because the values are not comparable, in which case *render* reverts to using the natural order.
- **tab** (*string*) – The string used when indenting.

render has built in support for the base Python types (*None*, *bool*, *int*, *float*, *str*, *set*, *tuple*, *list*, and *dict*). If you confine yourself to these types, the output of render can be read by the Python interpreter. Other types are converted to string with *repr()*.

Example:

```
>>> from inform import display, render
>>> display('result =', render({'a': (0, 1), 'b': [2, 3, 4]}))
result = {'a': (0, 1), 'b': [2, 3, 4]}
```

inform.**os_error**(*err*)

Generates clean messages for operating system errors.

Parameters **err** (*exception*) – The value of an *OSError* or *IOError* exception (in Python3 *IOError* is a subclass of *OSError*, so you only need to catch *OSError*).

Example:

```
>>> from inform import display, os_error
>>> try:
...     with open('config') as f:
...         contents = f.read()
... except (OSError, IOError) as e:
...     display(os_error(e))
config: no such file or directory.
```

inform.**conjoin**(*iterable*, *conj=' and '*, *sep=', '*)

Conjunction join.

Parameters

- **iterable** (*exception*) – The collection of strings to be joined.
- **conj** (*string*) – The separator used between the next to last and last values.
- **sep** (*string*) – The separator to use when joining the strings in *iterable*.

Return the items of the *iterable* joined into a string, where *conj* is used to join the last two items in the list, and *sep* is used to join the others.

Examples

```
>>> from inform import conjoin, display
>>> display(conjoin([], ' or '))
```

```
>>> display(conjoin(['a'], ' or '))
a
```

```
>>> display(conjoin(['a', 'b'], ' or '))
a or b
```

```
>>> display(conjoin(['a', 'b', 'c']))
a, b and c
```

inform.**plural**(*count*, *singular*, *plural=None*)

Pluralize a word.

If *count* is 1 or has length 1, the *singular* argument is returned, otherwise the *plural* argument is returned. If *plural* is None, then it is created by adding an 's' to the end of *singular* argument.

Example:

```
>>> from inform import display, plural
>>> fruits = 'apple banana cranberry date'.split()
>>> display('I have {} {} of fruit.'.format(len(fruits), plural(fruits, 'piece')))
I have 4 pieces of fruit.
```

inform.**full_stop**(*sentence*)

Add period to end of string if it is needed.

A full stop (a period) is added if there is no terminating punctuation at the end of the string.

Examples:

```
>>> from inform import full_stop
>>> full_stop('The file is out of date')
'The file is out of date.'

>>> full_stop('The file is out of date.')
'The file is out of date.'

>>> full_stop('Is the file is out of date?')
'Is the file is out of date?'
```

inform.**columns**(*array*, *pagewidth=79*, *alignment='<'*, *leader=' '*)

Distribute array over enough columns to fill the screen.

Returns a list of strings, one for each line.

Parameters

- **array** (*collection of strings*) – The array to be printed.
- **pagewidth** (*int*) – The number of characters available for each line.

- **alignment** (*'<' or '>'*) – Whether to left ('<') or right ('>') align the *array* items in their columns.
- **leader** (*str*) – The string to prepend to each line.

Example:

```
>>> from inform import columns, display, full_stop
>>> title = 'Display the NATO phonetic alphabet'
>>> words = '''
...     Alfa Bravo Charlie Delta Echo Foxtrot Golf Hotel India Juliett
...     Kilo Lima Mike November Oscar Papa Quebec Romeo Sierra Tango
...     Uniform Victor Whiskey X-ray Yankee Zulu
... '''.split()
>>> newline = '''
... '''
>>> display(full_stop(title), columns(words), sep=newline)
Display the NATO phonetic alphabet.
    Alfa      Echo      India     Mike      Quebec    Uniform   Yankee
    Bravo     Foxtrot   Juliett   November  Romeo     Victor    Zulu
    Charlie   Golf      Kilo      Oscar     Sierra    Whiskey
    Delta     Hotel     Lima      Papa      Tango     X-ray
```

4.2.5 Debug Utilities

inform.**aaa**(**args*, ***kwargs*)
Print argument, then return it.

Pretty-prints its argument. Argument may be named or unnamed. Allows you to display the value that is only contained within an expression.

inform.**ddd**(**args*, ***kwargs*)
Print arguments function.

Pretty-prints its arguments. Arguments may be named or unnamed.

inform.**ppp**(**args*, ***kwargs*)
Print function.

Mimics the normal print function, but colors printed output to make it easier to see and labels it with the location of the call.

inform.**sss**()
Print a stack trace.

inform.**vvv**(**args*)
Print variables function.

Pretty-prints variables from the calling scope. If no arguments are given, all variables are printed. If arguments are given, only the variables whose value match an argument are printed.

4.2.6 Exceptions

exception inform.**Error**(**args*, ***kwargs*)
A generic exception.

The exception accepts both unnamed and named arguments. All are recorded and available for later use.

get_culprit()
Get exception culprit.

If the *culprit* keyword argument was specified as a string, it is returned. If it was specified as a collection, the members are converted to strings and joined with *culprit_sep*. The resulting string is returned.

get_message(*template=None*)
Get exception message.

Parameters **template** (*str*) – This argument is treated as a format string and is passed both the unnamed and named arguments. The resulting string is treated as the message and returned.

If not specified, the *template* keyword argument passed to the exception is used. If there was no *template* argument, then the positional arguments of the exception are joined using *sep* and that is returned.

Returned: The formatted message without the culprits.

render(*template=None*)
Convert exception to a string for use in an error message.

Parameters **template** (*str*) – This argument is treated as a format string and is passed both the unnamed and named arguments. The resulting string is treated as the message and returned.

If not specified, the *template* keyword argument passed to the exception is used. If there was no *template* argument, then the positional arguments of the exception are joined using *sep* and that is returned.

report(*template=None*)
Report exception.

The `inform.error()` function is called with the exception arguments.

Parameters **template** (*str*) – This argument is treated as a format string and is passed both the unnamed and named arguments. The resulting string is treated as the message and returned.

If not specified, the *template* keyword argument passed to the exception is used. If there was no *template* argument, then the positional arguments of the exception are joined using *sep* and that is returned.

terminate(*template=None*)
Report exception and terminate.

The `inform.fatal()` function is called with the exception arguments.

Parameters **template** (*str*) – This argument is treated as a format string and is passed both the unnamed and named arguments. The resulting string is treated as the message and returned.

If not specified, the *template* keyword argument passed to the exception is used. If there was no *template* argument, then the positional arguments of the exception are joined using *sep* and that is returned.

4.3 Releases

1.11 (2017-12-25):

- Released the documentation.
- Added ability to override template in *inform.Error*.
- Added *stream_policy* option.
- Added *notify_if_no_tty* option.
- Informers now stack, so disconnecting from an existing informer reinstates the previous informer.
- Generalize *inform.cull()*.
- Add support for multiple templates.
- Added *inform.join()* function.

1.12 (2018-02-18):

- do not use notify override on continuations.
- tidied up a bit.

Latest development release:

Version: 1.13.0
Released: 2018-08-11

- Added *inform.aaa()* debug function.
- Added exit argument to *inform.done()*, *inform.terminate()*, and *inform.terminate_if_errors()*.
- *inform.terminate()* now produces an exit status of 0 if there was no errors reported.
- Added *inform.set_culprit()*, *inform.add_culprit()* and *inform.get_culprit()*.

Index

S

T

V