Inform Documentation

Release 1.30

Ken Kundert

Jun 07, 2024

CONTENTS

Version: 1.30
Released: 2024-06-07
Please post all bugs and suggestions at Inform Issues (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, error, os_error
>>> display('This is a plain message.')
This is a plain message.

>>> try:
...     with open('config') as f:
...         config = f.read()
... except OSError as e:
...     error(os_error(e))
error: config: no such file or directory.
```

The *display* and *error* functions are referred to as informants. They 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

ONE

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

TWO

INSTALLATION

Install the latest stable version with:

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

Requires Python2.7 or Python3.3 or better.

The source code is available from GitHub. You can download the repository and install using:

```
git clone https://github.com/KenKundert/inform.git
pip3 install --user inform
```

CHAPTER

THREE

ISSUES

Please ask questions or report problems on Inform Issues.

CHAPTER

FOUR

QUICK TOUR

4.1 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:

```
>>> from inform import display, error, fatal, warn

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

>>> display(*lines, sep='\n')
 blue = 3346ff
```

(continues on next page)

(continued from previous page)

```
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.

4.2 Controlling Informants

For more control of the informants, you can import and instantiate the *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
```

4.3 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 like 'and' or 'or' 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.
Info Class	A base class that can be used to create helper classes.
is_collection	Is object a collection (i.e., is it iterable and not a string)?
is_iterable	Is object iterable (includes strings).
is_mapping	Is object a mapping (i.e., is it a dictionary or is it dictionary like)?
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.
ProgressBar Class	Used to generate progress bars.
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 as e:
...     error(os_error(e))
error: config: no such file or directory.
```

4.4 Generic Exception

Inform also provides a generic exception, *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 as e:
...         raise Error(os_error(e))
```

(continues on next page)

(continued from previous page)

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

CHAPTER

FIVE

DOCUMENTATION

5.1 User's Guide

The main purpose of *Inform* is to present messages to the user of command line programs. With it you can easily output attractive, informative, and consistent messages. In addition, the error messages are given in a style that is consistent with the long traditions of Unix.

It does so by providing a collection of print functions, referred to as informants, to print different types of messages: output text, narration, warnings, errors, etc. All of the informants have a similar use model and share a common control class. Also included are a collection of utilities that are often used to construct messages.

5.1.1 Message Structure

Messages produced by *Inform* have a common structure:

```
>>> from inform import error
>>> error('this is the message', culprit='culprit', codicil='codicil')
error: culprit: this is the message
    codicil
```

There are several parts to this message:

severity:

> The severity indicates the type of the message. Examples include *warning*, *error*, etc. The severity is optional.

culprit:

> The culprit identifies the subject of the message.

header:

> If a severity is included with a message, it is placed into a header along with the culprit. If both the header and message are short, they are combined into a single line. Otherwise, the header is given alone on a line. The message starts on the next line and is indented.
>
> The header often starts with the program name. This makes it possible to identify the source of the message when your program is called as part of a pipeline or script.

message:

> When reporting an issue the message describes the issue. Otherwise it is arbitrary bit of text.

codicil:

An additional bit of text that gives additional context or perhaps helpful suggestions. It follows the message and is indented.

If the message does not have a severity, and so does not have a header, the culprit is merged into the message and the codicil is not indented:

```
>>> from inform import display
>>> display('this is the message', culprit='culprit', codicil='codicil')
culprit: this is the message
codicil
```

There can be multiple culprits and codicils. When multiple culprits are given, they generally represent a hierarchical path to the actual culprit:

```
>>> from inform import error
>>> error(
...     'this is line one of the message',
...     'this is line two of the message',
...     culprit=('culprit-1', 'culprit-2'),
...     codicil=('codicil one', 'codicil two'),
...     sep='\n'
... )
error: culprit-1, culprit-2:
    this is line one of the message
    this is line two of the message
        codicil one
        codicil two
```

Here is a typical message that employs multiple culprits and codicils:

```
nt2json error: address.nt, 4: invalid indentation.
    An indent may only follow a dictionary or list item that does not
    already have a value.
        3     address: Home
        4         > 3636 Buffalo Ave
```

In this case the culprit is "address.nt, 4", the filename and line number where the error was found. It has a brief single line message ("invalid indentation."). It also has two codicils. The first is two lines long and gives a more detailed explanation of the issue. The second is three lines long and gives the relevant lines in the source file with a pointer to the error. Notice that the header also includes the name of the program that detected the error ("nt2json").

5.1.2 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 are 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 are 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 be a 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 ', ').

codicil = *None*:
A string or a collection of strings that contains messages that are printed after the primary message.

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 a simple example:

```
>>> from inform import error

>>> error('file not found.', culprit='now-playing')
error: now-playing: file not found.
```

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

```
>>> 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 *set_culprit()* or *add_culprit()* and then recall them when needed using *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)
...     return parameters

>>> filename = 'parameters'
```

(continues on next page)

(continued from previous page)

```
>>> 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}
↪'))
 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()):
```

(continues on next page)

(continued from previous page)

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

5.1.3 Predefined Informants

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

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

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. Generally, one does not specify a culprit on codicils.

```
>>> 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 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.

When using notify you may pass in the *urgency* named argument to specify the urgency of the notification. Its value must 'low', 'normal', or 'critical' or it will be ignored.

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.

```
>> from inform import fatal, os_error
>> try:
..     with open('config') as f:
..         read_config(f.read())
.. except OSError as e:
..     fatal(os_error(e), codicil='Cannot continue.')
myprog error: config: file not found
    Cannot continue.
```

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.

Modifying Existing Informants

You may adjust the behavior of existing informants by overriding the attributes that were passed in when they were created. 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
```

Any attribute that can be passed into *InformantFactory* when creating an informant can be overridden. However, when overriding a color you must use a colorizer rather than a color name:

```
from inform import comment, Color
comment.message_color=Color('cyan')
```

5.1.4 Informant Control

For more control of the informants, you can import and instantiate the *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()
...         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.strip().split('\n')[-1])
'running test'
```

Logfiles

To configure *Inform* to generate a logfile you can specify the logfile to *Inform* or to *Inform.set_logfile()*. The logfile can be specified as a string, a *pathlib.Path*, an open stream, or as a Boolean. If *True*, a logfile is created and named *./<prog_name>.log*. If *False*, no logfile is created.

You may want to defer the decision on what should be the logfile without losing the log messages that occur before the ultimate destination of those messages is set. You can do so using an instance of *LoggingCache*, which simply saves the messages in memory until it is replaced, at which point they are transferred to the new logfile. For example:

```
>>> from inform import Inform, LoggingCache, log, indent
>>> with Inform(logfile=LoggingCache()) as inform:
...     log("This message is cached.")
...     inform.set_logfile(".mylog")
```

(continues on next page)

(continued from previous page)

```
...     log("This message is not cached.")

>>> with open(".mylog") as f:
...     print("Contents of logfile:")
...     print(indent(f.read()), end='')  # +ELLIPSIS
Contents of logfile:
    ...
    This message is cached.
    This message is not cached.
```

An existing logfile will be renamed before creating the logfile if you specify *prev_logfile_suffix* to *Inform*. In many cases, this does not provide enough persistence for the logged information. In that case you can use ntlog, which accumulates the contents of multiple logfiles into a NestedText file. It allows you to place limits on how many logs to retain in order to keep the logfile reasonably sized. Visit the *accessories page* for examples on how to use *ntlog*.

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.

5.1.5 User Defined Informants

You can create your own informants using *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*. 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, display, output

>>> succeed = InformantFactory(message_color='green', clone=display)
>>> fail = InformantFactory(message_color='red', clone=output)

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

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

A common use for this would be to have success and failure messages. For example, if your program runs a series of tests, the successes could be printed in green and the failures in red.

In this example, the two informants are first cloned from existing informants before applying any additional arguments. In this way the *success* informant inherits the qualities of *display* and the *fail* informant inherits the qualities of *output* before applying the color. The result is that the *success* informant suppresses the messages if the user asks for quiet, but the *fail* informant does not.

5.1.6 Exceptions

An exception, *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 *Error.report()* and *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.
```

Besides *culprit*, you can use any of the named arguments accepted by informants. In addition, you can also use *informant* as a named argument. *informant* changes the informant that is used when reporting the error. It is often used to convert an exception to a warning or to a fatal error. For example:

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

>>> Inform(prog_name='myprog')
<...>
>>> def read_files(filenames):
...     files = {}
...     for filename in filenames:
...        try:
...            with open(filename) as f:
...                files[filename] = f.read()
...        except FileNotFoundError:
...            raise Error('missing.', culprit=filename, informant=warn)
...     return files

>>> filenames = 'parameters swallows worlds'.split()
>>> try:
...     files = read_files(filenames)
... except Error as e:
...     files = None
...     e.report()
myprog warning: worlds: missing.
```

Error also provides *Error.get_message()* and *Error.get_culprit()* methods, which return the message and the culprit. You can also cast the exception to a string or call the *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 *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 inform import Error, did_you_mean

>>> 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 = did_you_mean(e.args[0], e.choices)
...     e.report(codicil=f"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 *Error.get_message()* or *Error.render()*. With *Error.report()* or *Error.terminate()* you can override any named argument, such as *template* or *culprit*. 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.")
```

(continues on next page)

(continued from previous page)

```
... except Error as e:
...     e.report(template="'{}' ist nicht definiert.")
myprog error: 'alfa' ist nicht definiert.
```

You can catch an *Error* exception and then reraise it after modifying its named arguments using *Error.reraise()*. This is helpful when all the information needed for the error message is not available where the initial exception is detected. Typically new culprits or codicils are added. For example, in the following the filename is added to the exception using *reraise* in *parse_file*:

```
>>> def parse_lines(lines):
...     values = {}
...     for i, line in enumerate(lines):
...         try:
...             k, v = line.split()
...         except ValueError:
...             raise Error('syntax error.', culprit=i+1)
...         values[k] = v
...     return values

>>> def parse_file(filename):
...     try:
...         with open(filename) as f:
...             return parse_lines(f.read().splitlines())
...     except Error as e:
...         e.reraise(culprit=e.get_culprit(filename))

>>> try:
...     unladen_airspeed = parse_file('swallows')
... except Error as e:
...     e.report()
myprog error: swallows, 2: syntax error.
```

This example uses *Error.get_culprit()* to access the existing culprit or culprits of the exception. Regardless of how many there are, they are always returned as a culprit. It also accepts a culprit as an argument, which is returned along with and before the culprit from the exception.

Also available is *Error.get_codicil()*, which behaves similarly except with codicils rather than culprits and the argument is added after the codicil from the exception rather than before.

Subclassing Error

When creating subclasses of *Error* you can add a template to the subclass as a way of specifying the error message or messages that are to be used for that exception. For example:

```
>>> class InvalidValueError(Error):
...     template = 'invalid value.'

>>> try:
...     raise InvalidValueError()
... except Error as e:
...     e.report()
myprog error: invalid value.
```

You can include named and unnamed arguments of the exception in the template:

```
>>> class InvalidValueError(Error):
...     template = 'must not be {}.'

>>> try:
...     raise InvalidValueError('negative', culprit='rate')
... except Error as e:
...     e.report()
myprog error: rate: must not be negative.
```

You can also specify a list of templates that are tried in order, the first for which all arguments are available is used:

```
>>> class InvalidValueError(Error):
...     template = [
...         '{} must fall between {min} and {max}.',
...         '{} must be greater than {min}.',
...         '{} must be less than {max}.',
...         '{} must not be {illegal}.',
...         '{} must be {legal}.',
...         '{} is invalid.',
...         'invalid value.',
...     ]

>>> rate = -1.0
>>> try:
...     if rate < 0:
...         raise InvalidValueError(rate, illegal='negative', culprit='rate')
... except Error as e:
...     e.report()
myprog error: rate: -1.0 must not be negative.
```

5.1.7 Utilities

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

Color Class

The *Color* class creates colorizers, which are functions used to render text in a particular color. They combine their arguments in a manner very similar to an *informant* and returns the result as a string, except the string is coded for the chosen color. Uses the *sep*, *template* and *wrap* keyword arguments to combine the arguments.

```
>> 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
```

(continues on next page)

(continued from previous page)

```
..     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 four supported schemes: *None*, *True*, 'light', and 'dark'. With *None* the text is not colored, with *True* the colorscheme of the currently active informer is used. In general it is best to use the 'light' colorscheme on dark backgrounds and the 'dark' colorscheme on light backgrounds. You can pass in the colorscheme using the *scheme* argument either to the color class or to the colorizer.

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 *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

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

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

This example 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

inform.**conjoin**(*iterable*, *conj=' and '*, *sep=', '*, *fmt=None*)

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'

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

If you prefer the use of the Oxford comma, you can add it as follow:

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

You can specify a format string that is applied to every item in the list before they are joined:

```
>>> conjoin([10.1, 32.5, 16.9], fmt='${:0.2f}')
'$10.10, $32.50 and $16.90'
```

cull

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

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, *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
```

dedent

inform.**dedent**(*text*, *strip_nl=None*, ***, *bolm=None*, *wrap=False*)

Without its named arguments, *dedent* behaves just like, and is as equivalent replacement for, *textwrap.dedent*.

Args:

strip_nl = None:
strip_nl is used to strip off a single leading or trailing newline. strip_nl may be None, 's', 'e', or 'b' representing neither, start, end, or both. True may also be passed, which is equivalent to 'b'.

bolm = None:
The beginning of line mark (bolm) is replaced by a space after the indent is removed. It must be the first non-space character after the initial newline. Normally bolm is a single character, often '|', but it may be contain multiple characters, all of which are replaced by spaces.

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 dedent
```

```
>>> print(dedent('''
...         Diaspar
...          Lys
... ''', bolm=''))

    Diaspar
    Lys
```

```
>>> print(dedent('''
...     |   Diaspar
...     |   Lys
... ''', bolm='|', strip_nl='e'))

    Diaspar
|   Lys
```

```
>>> print(dedent('''
...     ||  Diaspar
...         Lys
... ''', bolm='||', strip_nl='s'))
    Diaspar
    Lys
```

```
>>> print(dedent('''
...         Diaspar
...         Lys
... ''', strip_nl='b'))
Diaspar
Lys
```

```
>>> print(dedent('''
...         Diaspar
...         Lys
... ''', strip_nl='b', wrap=True))
Diaspar Lys
```

did_you_mean

inform.**did_you_mean**(*candidate*, *choices*)

Given a candidate string from the user, return the closest valid choice.

Args:

candidate (string):
The string given by the user.

choices (iterable):
The set of valid strings that the user was expected to choose from.

Examples:

```
>>> from inform import did_you_mean
>>> did_you_mean('cat', ['cat', 'dog'])
'cat'
>>> did_you_mean('car', ['cat', 'dog'])
'cat'
>>> did_you_mean('car', {'cat': 1, 'dog': 2})
'cat'
```

fmt

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

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

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

>>> filenames = ['a', 'b', 'c', 'd']
>>> filetype = 'CSV'
>>> display(
...     fmt(
...         'Reading {filetype} files: {names}.',
...         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.

format_range

inform.**format_range**(*items*)

format_range() can be used to create a succinct, readable string representing a set of numbers.

```
>>> from inform import format_range
>>> format_range({1, 2, 3, 5})
'1-3,5'
```

full_stop

inform.**full_stop**(*string*)

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

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

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))
```

(continues on next page)

(continued from previous page)

```
.   a
.   .   b
```

Info Class

The *Info* class is intended to be used as a helper class. When instantiated, it converts provided keyword arguments to attributes. Unknown attributes evaluate to None. *Info* can be used directly, or it can be used as a base class.

```
>>> from inform import display, Info
>>> class Orwell(Info):
...     pass

>>> george = Orwell(peace='war', truth='lies')
>>> display(str(george))
Orwell(peace='war', truth='lies')

>>> display(george.peace)
war

>>> display(george.happiness)
None
```

is_collection

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

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('')  # string
False

>>> is_collection([])  # list
True

>>> is_collection(())  # tuple
True

>>> is_collection({})  # dictionary
True
```

is_iterable

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

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_mapping

inform.**is_mapping**(*obj*)

is_collection() returns *True* if its argument is a mapping. This includes dictionary and other dictionary-like objects.

```
>>> from inform import is_mapping

>>> is_mapping('')  # string
False

>>> is_mapping([])  # list
False

>>> is_mapping(())  # tuple
False

>>> is_mapping({})  # dictionary
True
```

is_str

inform.**is_str**(*obj*)

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

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

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 in sorted(accounts):
...     lines.append(join(name, accounts[name], template='{:>10s}: ${:,.2f}'))

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

os_error

inform.**os_error**(*exception*)

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 as e:
...     error(os_error(e))
myprog error: temperatures.csv: no such file or directory.
```

parse_range

inform.**parse_range**(*items*)

parse_range() can be used to parse sets of numbers from user-inputted strings.

```
>>> from inform import parse_range
>>> parse_range('1-3,5')
{1, 2, 3, 5}
```

ProgressBar Class

The *ProgressBar* class is used to draw a progress bar as a single text line. The line counts down as progress is made and reaches 0 as the task completes. Interruptions are handled with grace.

There are three typical ways to use the progress bar. The first is used to illustrate the progress of an iterator. The iterator must have a length. For example:

```
>>> from inform import ProgressBar

>>> processed = []
>>> def process(item):
...     # this function would implement some expensive operation
...     processed.append(item)
>>> items = ['i1', 'i2', 'i3', 'i4', 'i5', 'i6', 'i7', 'i8', 'i9', 'i10']

>>> for item in ProgressBar(items, prefix='Progress: ', width=60):
...     process(item)
Progress: 9876543210

>>> display('Processed:', conjoin(processed), end='.\n')
Processed: i1, i2, i3, i4, i5, i6, i7, i8, i9 and i10.
```

The second is similar to the first, except you just give an integer to indicate how many iterations you wish:

```
>>> for i in ProgressBar(50, prefix='Progress: '):
...     process(i)
Progress: 9876543210
```

Finally, the third illustrates progress through a continuous range:

```
>>> stop = 1e-6
>>> step = 1e-9

>>> with ProgressBar(stop) as progress:
...     display('Progress:')
...     value = 0
...     while value <= stop:
...         progress.draw(value)
...         value += step
Progress:
9876543210
```

In this case, you need to notify the progress bar if you decide to exit the loop before its complete unless an exception is raised that causes the *with* block to exit:

```
>>> with ProgressBar(stop) as progress:
...     display('Progress:')
...     value = 0
...     while value <= stop:
...         progress.draw(value)
...         value += step
...         if value > stop/2:
...             progress.escape()
```

(continues on next page)

(continued from previous page)

```
...             break
Progress:
9876
```

Without calling escape, the bar would have been terminated with a 0 upon exiting the *with* block. Using *escape()* is not necessary if the *with* block is exited via an exception:

```
>>> try:
...     with ProgressBar(stop) as progress:
...         display('Progress:')
...         value = 0
...         while value <= stop:
...             progress.draw(value)
...             value += step
...             if value > stop/2:
...                 raise Error('early exit.')
... except Error as e:
...     e.report()
Progress:
9876
myprog error: early exit.
```

It is possible to pass a second argument to *ProgressBar.draw()* that indicates the desired marker to use when updating the bar. This is usually used to signal that there was a problem with the update. To do so, you define the desired markers when instantiating *ProgressBar*. Each marker consists of a fill character and a color. The color can be specified by giving its name, with a *Color* object, or with None. For example, the following example uses markers to distinguish four types of results: *okay*, *warn*, *fail*, *error*.

```
>>> results = 'okay okay okay fail okay fail okay error warn okay'.split()

>>> def process(index):
...     # this function would implement some expensive operation
...     return results[index]

>>> markers = dict(
...     okay=('', None),
...     warn=('', None),
...     fail=('+', None),
...     error=('×', None)
... )

>>> with ProgressBar(len(results), prefix="progress: ", markers=markers) as progress:
...     for i in range(len(results)):
...         status = results[i]
...         progress.draw(i+1, status)
progress: 987++++++65++++++43×××××210
```

In this case color was not used, but you could specify the following to render the markers in color:

```
>>> markers = dict(
...     okay=('', 'green'),
...     warn=('-', 'yellow'),
...     fail=('+', 'magenta'),
```

(continues on next page)

(continued from previous page)

```
...     error=('×', 'red')
... )
```

You can also use the *Color* class:

```
>>> markers = dict(
...     okay=('', Color('green', enable=Color.isTTY())),
...     warn=('-', Color('yellow', enable=Color.isTTY())),
...     fail=('+', Color('magenta', enable=Color.isTTY())),
...     error=('×', Color('red', enable=Color.isTTY()))
... )
```

The progress bar generally handles interruptions with grace. For example:

```
>>> for item in ProgressBar(items, prefix='Progress: ', width=60):
...     if item == 'i4':
...         warn('bad value.', culprit=item)
Progress: 987
myprog warning: i4: bad value.
Progress: 9876543210
```

Notice that the warning started on a new line and the progress bar was restarted from the beginning after the warning.

Generally the progress bar is not printed if no tasks were performed. In some cases you would like to associate a progress bar with an iterator, and then decide later whether there are any tasks that require processing. That could be handled as follows:

```
>>> with ProgressBar(items, prefix='Progress: ') as progress:
...     for i, item in enumerate(items):
...         if item.startswith('i'):
...             continue
...         progress.draw(i)
...         process(item)
```

In this example, every item starts with 'i' and so is skipped. The result is that no items are processed and so the progress bar is not printed.

plural

class inform.**plural**(*value*, *, *render_num=str*, *num='#'*, *invert='!'*, *slash='/'*)

Used with Python format strings to conditionally format a phrase depending on the number of items that *value* represents.

The format specification has multiple sections, separated by '/'. The first section is always included and the remaining sections specify what should be added based on whether there is one, many, or no items. If there is only one section, then an 's' is appended if there are many or no items. If there are two sections, then the second section is appended if there are many or no items. If any of the sections contain a '#', it is replaced by the number of items.

You may provide either a number (e.g. 0, 1, 2, ...) or any object that implements *__len__()* (e.g. list, dict, set, ...) as the value. In the latter case, the length of the object then decides which form to use.

If the format string starts with a '!' it is removed and the sense of plurality is reversed (the plural form is used for one thing and the singular form is used otherwise). This is useful when pluralizing verbs.

Here is a typical usage:

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

>>> astronauts = plural(['John Glenn'])
>>> f"{astronauts:/One astronaut/# astronauts/No astronauts} {astronauts:!train}."
'One astronaut trains.'

>>> f"The {astronauts:astronaut}: {conjoin(astronauts.value)}"
'The astronaut: John Glenn'

>>> astronauts = plural(['Neil Armstrong', 'Buzz Aldrin', 'Michael Collins'])
>>> f"{astronauts:/One astronaut/# astronauts/No astronauts} {astronauts:!train}."
'3 astronauts train.'

>>> f"The {astronauts:astronaut}: {conjoin(astronauts.value)}"
'The astronauts: Neil Armstrong, Buzz Aldrin and Michael Collins'

>>> astronauts = plural([])
>>> f"{astronauts:/One astronaut/# astronauts/No astronauts} {astronauts:!sleep}."
'No astronauts sleep.'
```

You change the way the count is rendered by passing a function to *render_num*:

```
>>> from num2words import num2words
```

```
>>> f"He has {plural(1, render_num=num2words):# /wife/wives}."
'He has one wife.'
```

```
>>> f"He has {plural(42, render_num=num2words):# /wife/wives}."
'He has forty-two wives.'
```

You can also use the *format* method to directly produce a descriptive string:

```
>>> plural(2).format("/a goose/# geese")
'2 geese'
```

render

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

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 *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}
```

(continues on next page)

(continued from previous page)

```
>>> 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,
    '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',
}
```

In addition, you can add support for *render* to your classes by adding one or both of these methods:

_inform_get_args(): returns a list of argument values.

_inform_get_kwargs(): returns a dictionary of keyword arguments.

```
>>> class Chimera:
...     def __init__(self, *args, **kwargs):
...         self.args = args
...         self.kwargs = kwargs
...
...     def _inform_get_args(self):
```

(continues on next page)

(continued from previous page)

```
...         return self.args
...
...     def _inform_get_kwargs(self):
...         return self.kwargs

>>> lycia = Chimera('Lycia', front='lion', middle='goat', tail='snake')
>>> display(render(lycia))
Chimera(
    'Lycia',
    front='lion',
    middle='goat',
    tail='snake',
)
```

render_bar

inform.**render_bar**(*normalized_value*, *width=72*)

render_bar() produces a graphic representation of a normalized value in the form of a bar. *normalized_value* is the value to render; it is expected to be a value between 0 and 1. *width* specifies the maximum width of the line in characters.

```
>>> from inform import render_bar, display
>>> for i in range(10):
...     value = 1 - i/9.02
...     display('{:0.3f}: {}'.format(value, render_bar(value, 70)))
1.000:
0.889:
0.778:
0.667:
0.557:
0.446:
0.335:
0.224:
0.113:
0.002:
```

If you would like to add delimiters to the bar, you can make each bar fixed width by adding `fullwidth=True`:

```
>>> assets = {'property': 13_194, 'cash': 2846, 'equities': 19_301}
>>> total = sum(assets.values())
>>> for key, value in assets.items():
...     display(f"{key:>8}: {render_bar(value/total, full_width=True)}")
property:
    cash:
equities:
```

title_case

inform.**title_case**(*string*, *exceptions=...*)

title_case() converts the initial letters in the words of a string to upper case while maintaining any letters that are already upper case, such as acronyms. Common 'small' words are excepted and words within quotes are handled properly.

```
>>> from inform import title_case
>>> headline = 'CDC warns about "aggressive" rats as coronavirus shuts down restaurants'
>>> display(title_case(headline))
CDC Warns About "Aggressive" Rats as Coronavirus Shuts Down Restaurants
```

truth

class inform.**truth**(*value*, *, *interpolate='%'*, *slash='/'*)

Used with python format strings to conditionally format a phrase depending on whether *value* is true or false.

The format string has two sections, separated by '/'. The first section is included only if the given value is true and the last section is included only if the given value is false.

Both sections are optional. If the last section is not given it is left blank. If both sections are not given, 'yes' is returned for true and 'no' for false.

If either section contains %, it is replaced by the value.

Converting truth to a string returns 'yes' or 'no'. Converting truth to a Boolean returns True or False.

Examples:

```
>>> from inform import truth

>>> f"account is {truth(True):past due/current}."
'account is past due.'

>>> f"account is {truth(False):past due/current}."
'account is current.'

>>> paid = truth("20 July 1969")
>>> is_overdue = truth(True)
>>> f"last payment: {paid:%/not received}{is_overdue: - overdue}"
'last payment: 20 July 1969 - overdue'

>>> paid.format('%')
'20 July 1969'

>>> paid = truth(None)
>>> f"last payment: {paid:%/not received}{is_overdue: - overdue}"
'last payment: not received - overdue'

>>> paid.format('%')
''

>>> f"in arrears: {is_overdue}"
```

(continues on next page)

(continued from previous page)

```
'in arrears: yes'

>>> bool(is_overdue)
True

>>> str(is_overdue)
'yes'
```

If '/', or '%' are inconvenient, you can change them by passing the *slash* and *interpolate* arguments to truth().

5.1.8 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

inform.**aaa**(*arg*)

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

>>> 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[...]>, 2, __main__: int1: 2
myprog DEBUG: <doctest user.rst[...]>, 2, __main__: int2: 2
vout = 1.4
myprog DEBUG: <doctest user.rst[...]>, 2, __main__: int1: 1.6
myprog DEBUG: <doctest user.rst[...]>, 2, __main__: int2: 3.6
vout = 2.52
```

ddd

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

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[...]>, 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[...]>, 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[...]>, 4, __main__.Info.__init__():
    self = Info object containing {
        'email': 'ted@ledbelly.com',
        'name': 'Ted Ledbelly',
    }
```

ppp

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

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[...]>, 1, __main__: 1 this is a test (2, 3)
```

sss

inform.**sss**(*ignore_exceptions*)

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[...]>:2, __main__.foo():
    Traceback (most recent call last):
        ...
CONTINUING
```

vvv

inform.**vvv**(**args*)

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[...]>, 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[...]>, 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

try:
    from inform import aaa, ddd, ppp, sss, vvv
    builtins.aaa = aaa
    builtins.ddd = ddd
    builtins.ppp = ppp
    builtins.sss = sss
    builtins.vvv = vvv
except ImportError:
    pass
```

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

5.1.9 Inform Helper Functions

An informer (an *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 *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

inform.**done**(*exit=True*)

done() terminates the program with the normal exit status. It calls *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

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

terminate() terminates the program with specified exit status or message. It calls *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

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

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

inform.**errors_accrued**(*reset=False*)

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

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

get_prog_name

inform.**get_prog_name**()

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

get_informer

inform.**get_informer**()

get_informer() returns the currently active informer.

set_culprit

inform.**set_culprit**(*culprit*)

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.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.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 *set_culprit()* use.

add_culprit

inform.**add_culprit**(*culprit*)

add_culprit() appends a culprit to any existing saved culprit. It calls *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.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 *add_culprit()* use.

get_culprit

inform.**get_culprit**(*culprit=None*)

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.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 *get_culprit()* use.

5.2 Examples

In general whenever you write a command line utility it is worthwhile importing *Inform*. At a minimum you would use it to report errors to the user in a way that stands out from the normal output of your program because it is in color. From there you can expand your use of *Inform* in many different directions as appropriate. For example, you can use *Inform* for all of your textual output to the user so that you can easily turn on logging or implement verbose and quiet modes. You can also use *Error* directly or you can subclass it to access *Inform's* rich exception handling. You can use the various *Inform* utilities such as the text colors, multiple columns lists, and progress bars.

You can the source text for these examples on GitHub.

5.2.1 Find Debug Functions

This utility examines all python files in the current directory and all subdirectories looking for files that contain the various debug functions (*aaa()*, *ddd()*, *ppp()*, *sss()*, and *vvv()*) etc.) and then it opens those files in the Vim editor. This allows you to easily remove these functions after you are finished debugging your code.

The places where *Inform* is used are marked with the *inform* comment at the end of the line.

To get the prerequisites for this example, run:

```
> pip3 install --user --upgrade docopt inform shlib
```

```
#!/usr/bin/env python3
# Description
"""
fdb

Search through all python files in current working directory and all
subdirectories and edit those that contain any Inform debug functions (aaa,
ddd, ppp, sss, vvv).  Use *n* to search for debug functions, and *^n* to go
to next file.  Going to the next file automatically writes the current file
if any changes were made.

Usage:
    fdb [options]

Options:
    -l, --list   list the files rather than edit them
"""

# Imports
from docopt import docopt
from inform import display, os_error, terminate                    ## inform
import re
from shlib import lsf, Run

# Globals
debug_functions = 'aaa ddd ppp sss vvv'.split()
finder = re.compile(r'\b({})\('.format('|'.join(debug_functions)))
vim = 'vim'
vim_search = r'\<\({}\)('.format(r'\|'.join(debug_functions))
vim_flags = 'aw nofen'.split()   # autowrite, disable folds
```

(continues on next page)

(continued from previous page)

```
vim_options = 'set {}'.format(' '.join(vim_flags))
# Configure ctrl-N to move to first occurrence of search string in next file
# while suppressing the annoying 'press enter' message and echoing the
# name of the new file so you know where you are.
next_file_map = 'map <C-N> :silent next +//<CR> :file<CR>'
search_pattern = 'silent /{}'.format(vim_search)

# Main
cmdline = docopt(__doc__)

# determine which files contains any debug function
matches = []
for filepath in lsf(select='**/*.py', reject='inform.py'):
    try:
        contents = filepath.read_text()
        if finder.search(contents):
            matches.append(filepath)
    except OSError as e:
        error(os_error(e))                                          ## inform
if not matches:
    terminate()                                                     ## inform

if cmdline['--list']:
    display(*matches, sep='\n')                                     ## inform
    terminate()                                                     ## inform

# edit the files
cmd = [
    vim,
    '+{}'.format('|'.join([vim_options, next_file_map, search_pattern]))
] + matches
editor = Run(cmd, modes='soeW*')
terminate(editor.status)                                            ## inform
```

5.2.2 Add Keys to SSH Agent

Imagine you have multiple SSH keys, such as your personal keys, work keys, github key, key for your remote backups, etc. For convenience, you might want to add all of these keys to your SSH agent when you first login. This can become quite tedious. This script could be used load all of the keys to your agent in one simple action. It assumes the use of the Avendesora Collaborative Password Manager to securely hold the pass phrases of the keys.

You would put the name of your SSH keys in *SSHkeys*. The program steps through each key, accessing the passphrase and key file name from *Avendesora*, then pexpect interacts with *ssh-add* to add the passphrase to the SSH agent.

The places where *Inform* is used are marked with the *inform* comment at the end of the line. *Avendesora* uses *Inform*, and its *PasswordError* is a subclass of *Error*.

To get the prerequisites for this example, run:

```
> pip3 install --user --upgrade avendesora docopt inform pathlib pexpect
```

You will also have to update the *SSHkeys* variable below and add the requisite alias and keyfile attributes to the Avendesora accounts that contain your SSH pass phrases.

```
#!/usr/bin/env python3
"""
Add SSH keys

Add SSH keys to SSH agent.
The following keys are added: {keys}.

Usage:
    addsshkeys [options]

Options:
    -v, --verbose    list the keys as they are being added

A description of how to configure and use this program can be found at
`<https://avendesora.readthedocs.io/en/latest/api.html#example-add-ssh-keys>_.
"""
# Assumes that the Avendesora account that contains the ssh key's passphrase
# has a name or alias of the form <name>-ssh-key. It also assumes that the
# account contains a field named 'keyfile' or 'keyfiles' that contains an
# absolute path or paths to the ssh key files in a string.

from avendesora import PasswordGenerator, PasswordError
from inform import Inform, codicil, conjoin, error, narrate        ## inform
from docopt import docopt
from pathlib import Path
import pexpect

SSHkeys = 'personal work github backups'.split()
SSHadd = 'ssh-add'

cmdline = docopt(__doc__.format(keys = conjoin(SSHkeys)))          ## inform
Inform(narrate=cmdline['--verbose'])                               ## inform

try:
    pw = PasswordGenerator()
except PasswordError as e:                                         ## inform
    e.terminate()                                                  ## inform

for key in SSHkeys:
    name = key + '-ssh-key'
    try:
        account = pw.get_account(name)
        passphrase = str(account.get_passcode().value)
        if account.has_field('keyfiles'):
            keyfiles = account.get_value('keyfiles').value
        else:
            keyfiles = account.get_value('keyfile').value
        for keyfile in keyfiles.split():
            path = Path(keyfile).expanduser()
            narrate('adding.', culprit=keyfile)                    ## inform
            try:
                sshadd = pexpect.spawn(SSHadd, [str(path)])
                sshadd.expect('Enter passphrase for %s: ' % (path), timeout=4)
```

(continues on next page)

(continued from previous page)

```
                sshadd.sendline(passphrase)
                sshadd.expect(pexpect.EOF)
                sshadd.close()
                response = sshadd.before.decode('utf-8')
                if 'identity added' in response.lower():
                    continue
            except (pexpect.EOF, pexpect.TIMEOUT):
                pass
            error('failed.', culprit=path)                          ## inform
            response = sshadd.before.decode('utf8')
            if response:
                codicil('response:', response, culprit=SSHadd)      ## inform
            if sshadd.exitstatus:
                codicil('exit status:', sshadd.exitstatus , culprit=SSHadd)
                                                                    ## inform
    except PasswordError as e:
        e.report(culprit=name)                                      ## inform
```

5.2.3 Status of Solar Energy System

This utility prints the current status of an Enphase home solar array.

The places where *Inform* is used are marked with the *inform* comment at the end of the line.

To get the prerequisites for this example, run:

```
> pip3 install --user --upgrade docopt inform quantiphy arrow requests
```

You will also have to tailor the values of the *system*, *api_key* and *user_id* variables to your account.

```
#!/usr/bin/env python3
"""Solar Production

Displays current production of my solar panels.

Usage:
    solar [options]

Options:
    -f, --full   give full report
    -q, --quiet  no text output, exit status is zero if array status is normal
    -r, --raw    output the raw data
"""

# Imports
from docopt import docopt
from inform import display, fatal, render, terminate, Color        ## inform
from quantiphy import Quantity
from textwrap import dedent
import arrow
import requests
date_keys = 'operational_at last_report_at last_interval_end_at'.split()
```

(continues on next page)

(continued from previous page)

```
power_keys = 'size_w current_power'.split()
energy_keys = 'energy_today energy_lifetime'.split()
status_key = 'status'
normal = Color('green')                                          ## inform
abnormal = Color('red')                                          ## inform
Quantity.set_prefs(prec=2)

# Parameters
system = '1736719'
api_key = '6ff307fb00660f4c030b45b2fc1dabc5'
user_id = '24e03c5d24c2d0a7fb43b2ef68'
base_url = f'https://api.enphaseenergy.com/api/v2/systems/{system}'
keys = dict(key = api_key, user_id = user_id)

# Program
try:
    cmdline = docopt(__doc__)
    command = 'summary'
    keys = '&'.join(f'{k}={v}' for k, v in keys.items())
    url = f'{base_url}/{command}?{keys}'
    response = requests.get(url)
    data = response.json()

    # output the raw data and terminate
    if cmdline['--raw']:
        display(render(data))                                    ## inform
        terminate(data[status_key] != 'normal')                  ## inform

    # process dates
    for each in date_keys:
        if each in data:
            date_utc = arrow.get(data[each])
            date_local = date_utc.to('US/Pacific')
            data[each] = date_local.format('dddd, YYYY-MM-DD @ hh:mm:ss A')

    # process powers
    for each in power_keys:
        if each in data:
            data[each] = Quantity(data[each], 'W')
    data['utilization'] = Quantity(100*data['current_power']/data['size_w'], '%')

    # process energies
    for each in energy_keys:
        if each in data:
            data[each] = Quantity(data[each], 'Wh')

    # process status
    raw_status = data.get(status_key)
    if raw_status == 'normal':
        data[status_key] = normal(raw_status)                    ## inform
    elif raw_status:
        data[status_key] = abnormal(raw_status)                  ## inform
```

(continues on next page)

(continued from previous page)

```
    # display information
    if cmdline['--quiet']:
        # do not display anything, instead return status through exit code
        pass
    elif cmdline['--full']:
        for k, v in data.items():
            display(k, v, template='{}: {}')                    ## inform
    else:
        display(dedent('''                                      ## inform
            date: {last_report_at}
            status: {status}
            power: {current_power} ({utilization:.1p})
            energy today: {energy_today}
            energy lifetime: {energy_lifetime}
        '''.format(**data)).strip())

except requests.RequestException as e:
    fatal(e)                                                    ## inform
except KeyboardInterrupt:
    terminate()                                                 ## inform
terminate(raw_status != 'normal')                               ## inform
```

A typical output of the utility is:

```
date: Friday, 2018-10-12 @ 03:36:45 PM
status: normal
power: 1.48 kW (44 %)
energy today: 15.2 kWh
energy lifetime: 2.71 MWh
```

5.2.4 Run Command

This function runs a command and captures it output. It uses *Inform's* rich exceptions. If something goes wrong while invoking the command then all relevant information is attached to the exception and so is available to help build the most informative error message. In this way, the code that is responsible for reporting the problem to the user can adapt to the errant command reports its errors (some commands just return an exit status, some output the error in stderr, some in stdout).

```
from inform import Error, narrate, os_error
from subprocess import Popen, PIPE

def run(cmd, stdin='', accept=0):
    "Run a command and capture its output."
    narrate('running:', cmd)

    try:
        process = Popen(cmd, shell=True, stdin=PIPE, stdout=PIPE, stderr=PIPE)
        stdout, stderr = process.communicate(stdin.encode('utf8'))
        stdout = stdout.decode('utf8')
        stderr = stderr.decode('utf8')
```

(continues on next page)

(continued from previous page)

```
        status = process.returncode
    except OSError as e:
        raise Error(msg=os_error(e), cmd=cmd, template = '{msg}')

    # check exit status
    narrate('completion status:', status)
    if status < 0 or status > accept:
        raise Error(
            msg = 'unexpected exit status',
            status = status,
            stdout = stdout.rstrip(),
            stderr = stderr.rstrip(),
            cmd = cmd,
            template = '{msg} ({status}).'
        )
    return status, stdout, stderr

try:
    status, stdout, stderr = run('unobtanium')
except Error as e:
    e.terminate(culprit=e.cmd, codicil=e.stderr)
```

The output to this command would be something like this:

```
error: unobtanium: unexpected exit status (127).
    /bin/sh: unobtanium: command not found
```

5.2.5 Networth

This utility use the Avendesora Collaborative Password Manager to keep track of the value of assets and liabilities that together make up ones networth.

To get the prerequisites for this example, run:

```
> pip3 install --user --upgrade docopt inform quantiphy arrow requests appdirs
```

```
#!/usr/bin/env python3
# Description
"""Networth

Show a summary of the networth of the specified person.

Usage:
    networth [options] [<profile>]

Options:
    -u, --updated           show the account update date rather than breakdown

{available_profiles}
Settings can be found in: {settings_dir}.
Typically there is one file for generic settings named 'config' and then one
```

(continues on next page)

(continued from previous page)

```
file for each profile whose name is the same as the profile name with a '.prof'
suffix.  Each of the files may contain any setting, but those values in 'config'
override those built in to the program, and those in the individual profiles
override those in 'config'. The following settings are understood. The values
are those before an individual profile is applied.

Profile values:
    default_profile = {default_profile}

Account values:
    avendesora_fieldname = {avendesora_fieldname}
    value_updated_subfieldname = {value_updated_subfieldname}
    date_formats = {date_formats}
    max_account_value_age = {max_account_value_age}  (in days)
    aliases = {aliases}
        (aliases is used to fix account names to make them more readable)

Cryptocurrency values:
    coins = {coins}
    prices_filename = {prices_filename}
    max_coin_price_age = {max_coin_price_age}  (in seconds)

Bar graph values:
    screen_width = {screen_width}
    asset_color = {asset_color}
    debt_color = {debt_color}

The prices and log files can be found in {cache_dir}.

A description of how to configure and use this program can be found at
<https://avendesora.readthedocs.io/en/latest/api.html#example-net-worth>`_
"""

# Imports
from avendesora import PasswordGenerator, PasswordError
from avendesora.gpg import PythonFile
from inform import (
    conjoin, display, done, error, fatal, is_str, join, narrate, os_error,
    render_bar, terminate, warn, Color, Error, Inform,
)
from quantiphy import Quantity
from docopt import docopt
from appdirs import user_config_dir, user_cache_dir
from pathlib import Path
import arrow

# Settings
# These can be overridden in ~/.config/networth/config
prog_name = 'networth'
config_filename = 'config'

# Avendesora settings
```

(continues on next page)

(continued from previous page)

```
default_profile = 'me'
avendesora_fieldname = 'estimated_value'
value_updated_subfieldname = 'updated'
aliases = {}

# cryptocurrency settings (empty coins to disable cryptocurrency support)
proxy = None
prices_filename = 'prices'
coins = None
max_coin_price_age = 86400  # refresh cache if older than this (seconds)

# bar settings
screen_width = 79
asset_color = 'green'
debt_color = 'red'
    # currently we only colorize the bar because ...
    # - it is the only way of telling whether value is positive or negative
    # - trying to colorize the value really messes with the column widths and is
    #     not attractive

# date settings
date_formats = [
    'MMMM YYYY',
    'YYMMDD',
]
max_account_value_age = 120  # days

# Utility functions
# get the age of an account value
def get_age(date, profile):
    if date:
        for fmt in date_formats:
            try:
                then = arrow.get(date, fmt)
                age = arrow.now() - then
                return age.days
            except:
                pass
    warn(
        'could not compute age of account value',
        '(updated missing or misformatted).',
        culprit=profile
    )

# colorize text
def colorize(value, text = None):
    if text is None:
        text = str(value)
    return debt_color(text) if value < 0 else asset_color(text)

try:
    # Initialization
```

(continues on next page)

(continued from previous page)

```
    settings_dir = Path(user_config_dir(prog_name))
    cache_dir = user_cache_dir(prog_name)
    Quantity.set_prefs(prec=2)
    Inform(logfile=Path(cache_dir, 'log'))
    display.log = False   # do not log normal output

    # Read generic settings
    config_filepath = Path(settings_dir, config_filename)
    if config_filepath.exists():
        narrate('reading:', config_filepath)
        settings = PythonFile(config_filepath)
        settings.initialize()
        locals().update(settings.run())
    else:
        narrate('not found:', config_filepath)

    # Read command line and process options
    available=set(p.stem for p in settings_dir.glob('*.prof'))
    available.add(default_profile)
    if len(available) > 1:
        choose_from = f'Choose <profile> from {conjoin(sorted(available))}.'
        default = f'The default is {default_profile}.'
        available_profiles = f'{choose_from} {default}\n'
    else:
        available_profiles = ''

    cmdline = docopt(__doc__.format(
        **locals()
    ))
    show_updated = cmdline['--updated']
    profile = cmdline['<profile>'] if cmdline['<profile>'] else default_profile
    if profile not in available:
        fatal(
            'unknown profile.', choose_from, template=('{} {}', '{}'),
            culprit=profile
        )

    # Read profile settings
    config_filepath = Path(user_config_dir(prog_name), profile + '.prof')
    if config_filepath.exists():
        narrate('reading:', config_filepath)
        settings = PythonFile(config_filepath)
        settings.initialize()
        locals().update(settings.run())
    else:
        narrate('not found:', config_filepath)

    # Process the settings
    if is_str(date_formats):
        date_formats = [date_formats]
    asset_color = Color(asset_color)
    debt_color = Color(debt_color)
```

(continues on next page)

(continued from previous page)

```
    # Get cryptocurrency prices
    if coins:
        import requests

        cache_valid = False
        cache_dir = Path(cache_dir)
        cache_dir.mkdir(parents=True, exist_ok=True)
        prices_cache = Path(cache_dir, prices_filename)
        if prices_cache and prices_cache.exists():
            now = arrow.now()
            age = now.timestamp - prices_cache.stat().st_mtime
            cache_valid = age < max_coin_price_age
        if cache_valid:
            contents = prices_cache.read_text()
            prices = Quantity.extract(contents)
            narrate('coin prices are current:', prices_cache)
        else:
            narrate('updating coin prices')
            # download latest asset prices from cryptocompare.com
            currencies = dict(
                fsyms=','.join(coins),     # from symbols
                tsyms='USD',               # to symbols
            )
            url_args = '&'.join(f'{k}={v}' for k, v in currencies.items())
            base_url = f'https://min-api.cryptocompare.com/data/pricemulti'
            url = '?'.join([base_url, url_args])
            try:
                r = requests.get(url, proxies=proxy)
            except Exception as e:
                # must catch all exceptions as requests.get() can generate
                # a variety based on how it fails, and if the exception is not
                # caught the thread dies.
                raise Error('cannot access cryptocurrency prices:', codicil=str(e))
            except KeyboardInterrupt:
                done()

            try:
                data = r.json()
            except:
                raise Error('cryptocurrency price download was garbled.')
            prices = {k: Quantity(v['USD'], '$') for k, v in data.items()}

            if prices_cache:
                contents = '\n'.join('{} = {}'.format(k,v) for k,v in
                prices.items())
                prices_cache.write_text(contents)
                narrate('updating coin prices:', prices_cache)
        prices['USD'] = Quantity(1, '$')
    else:
        prices = {}
```

(continues on next page)

(continued from previous page)

```
    # Build account summaries
    narrate('running avendesora')
    pw = PasswordGenerator()
    totals = {}
    accounts = {}
    total_assets = Quantity(0, '$')
    total_debt = Quantity(0, '$')
    grand_total = Quantity(0, '$')
    width = 0
    for account in pw.all_accounts():

        # get data
        data = account.get_composite(avendesora_fieldname)
        if not data:
            continue
        if type(data) != dict:
            error(
                'expected a dictionary.',
                culprit=(account_name, avendesora_fieldname)
            )
            continue

        # get account name
        account_name = account.get_name()
        account_name = aliases.get(account_name, account_name)
        account_name = account_name.replace('_', ' ')
        width = max(width, len(account_name))

        # sum the data
        updated = None
        contents = {}
        total = Quantity(0, '$')
        odd_units = False
        for k, v in data.items():
            if k == value_updated_subfieldname:
                updated = v
                continue
            if k in prices:
                value = Quantity(v*prices[k], prices[k])
                k = 'cryptocurrency'
            else:
                value = Quantity(v, '$')
            if value.units == '$':
                total = total.add(value)
            else:
                odd_units = True
            contents[k] = value.add(contents.get(k, 0))
            width = max(width, len(k))
        for k, v in contents.items():
            totals[k] = v.add(totals.get(k, 0))

        # generate the account summary
```

(continues on next page)

(continued from previous page)

```
        age = get_age(data.get(value_updated_subfieldname), account_name)
        if show_updated:
            desc = updated
        else:
            desc = ', '.join('{}={}'.format(k, v) for k, v in contents.items() if v)
            if len(contents) == 1 and not odd_units:
                desc = k
            if age and age > max_account_value_age:
                desc += f' ({age//30} months old)'
        accounts[account_name] = join(
            total, desc.replace('_', ' '),
            template=('{:7q} {}', '{:7q}'), remove=(None,'')
        )

        # sum assets and debts
        if total > 0:
            total_assets = total_assets.add(total)
        else:
            total_debt = total_debt.add(-total)
        grand_total = grand_total.add(total)

    # Summarize by account
    display('By Account:')
    for name in sorted(accounts):
        summary = accounts[name]
        display(f'{name:>{width+2}s}: {summary}')

    # Summarize by investment type
    display('\nBy Type:')
    largest_share = max(v for v in totals.values() if v.units == '$')
    barwidth = screen_width - width - 18
    for asset_type in sorted(totals, key=lambda k: totals[k], reverse=True):
        value = totals[asset_type]
        if value.units != '$':
            continue
        share = value/grand_total
        bar = render_bar(value/largest_share, barwidth)
        asset_type = asset_type.replace('_', ' ')
        display(f'{asset_type:>{width+2}s}: {value:>7s} ({share:>5.1%}) {bar}')
    display(
        f'\n{"TOTAL":>{width+2}s}:',
        f'{grand_total:>7s} (assets = {total_assets}, debt = {total_debt})'
    )

# Handle exceptions
except OSError as e:
    error(os_error(e))
except KeyboardInterrupt:
    terminate('Killed by user.')
except (PasswordError, Error) as e:
    e.terminate()
done()
```

The output of this program should look something like this:

```
By Account:
          ameritrade:   $705k equities=$315k, cash=$389k
            pnc bank:  $21.3k cash
        john hancock:    $80k equities
              praxis:  $55.7k equities
         oppenheimer:   $134k equities
           tiaa cref:    $93k retirement
          black rock:  $98.4k equities
               pimco:   $211k equities
            jpmorgan:  $12.9k equities
            hartford:    $31k equities
    american century:   $914k equities

By Type:
             equities:  $1.85M (78.6%)
                 cash:   $411k (17.4%)
           retirement:    $93k ( 3.9%)

                TOTAL:  $2.36M (assets = $2.36M, debt = $0)
```

5.3 Classes and Functions

5.3.1 Inform

The Inform class controls the active informants.

class inform.**Inform**(*mute=False*, *quiet=False*, *verbose=False*, *narrate=False*, *logfile=False*, *prev_logfile_suffix=None*, *error_status=1*, *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. It may also be an instance of *LoggingCache*, which caches the log messages until it is replaced with *Inform.set_logfile()*.

- **prev_logfile_suffix** (*string*) – If specified, the previous logfile will be moved aside before creating the new logfile.

- **error_status** (*int*) – The default exit status to return to the shell when terminating the program due to an error. The default is 1.

- **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. This function is called before the logfile is closed and is only called if Inform processes the program termination. If you want to register a function to run regardless of how the program exit is processed, use the atexit module.

- **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 that 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 that act as 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.

'errors':
stderr is used for all errors, stdout is used otherwise. This is also commonly used for programs that act as filters.

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.

close_logfile(*status=None*)

Close logfile

If status is given, it is taken to be the exit message or exit status.

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.

`join_culprit`(*culprit*)

Join the specified culprits with the current culprit separators.

Culprits are returned from the informer or for exceptions as a tuple. This function allows you to join those culprits into a string.

Parameters
culprit (*tuple of strings or numbers*)

Returns
The culprit tuple joined into a string.

`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.

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+1))

>>> filename = 'pyproject.toml'
>>> with open(filename) as f, set_culprit(filename):
...     lines = f.read().splitlines()
...     num_lines = count_lines(lines)
warning: pyproject.toml, 25: empty line.
warning: pyproject.toml, 33: empty line.
warning: pyproject.toml, 39: empty line.
```

set_logfile(*logfile*, *prev_logfile_suffix=None*, *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.

  Directory containing the logfile must exist.

- **prev_logfile_suffix** – If specified, the existing logfile will be renamed before creating the new logfile. This only occurs the first time the logfile is specified.

- **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 inform.error_status if True is passed in. When None, return inform.error_status if errors occurred and 0 otherwise. Status may also be a string, in which case it is printed to stderr without a header and the exit status is inform.error_status.

- **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

Of, if your program naturally want to signal pass or failure using its exit status:

0: success
1: failure
2: error
3: panic

`terminate_if_errors`(*status=None*, *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=None*, *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.

5.3.2 InformantFactory

class inform.**InformantFactory**(***kwargs*)

Create informants.

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.
- **clone** (*informant*) – Clone the attributes of the given informer. Any explicitly specified arguments override those acquired through cloning.

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, display

>>> success = InformantFactory(
...     clone = display,
...     severity = 'Pass',
...     header_color = 'green'
... )
>>> failure = InformantFactory(
...     clone = display,
...     severity = 'FAIL',
...     header_color = 'red'
... )
```

success and *failure* are both informants. Once created, they 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 = failure
...     else:
...         report = success
...     report(
...         measured, expected, measured-expected,
...         template='measured = {:.3f}V, expected = {:.3f}V, diff = {:.3f}V
↪'
...     )
Pass: measured = 0.005V, expected = 0.000V, diff = 0.005V
Pass: measured = 0.512V, expected = 0.500V, diff = 0.012V
FAIL: measured = 0.875V, expected = 1.000V, diff = -0.125V
```

In the console 'Pass' is rendered in green and 'FAIL' in red.

5.3.3 Inform Utilities

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.

Example:

```
>>> 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.

class inform.**LoggingCache**

Use as logfile if you cannot know the desired location of the logfile until after log messages have been emitted. It holds the log messages in memory until you establish a logfile. At that point the messages are copied into the logfile.

Example:

```
>>> from inform import Inform, LoggingCache, log, indent
>>> with Inform(logfile=LoggingCache()) as inform:
...     log("This message is cached.")
...     inform.set_logfile(".mylog")
...     log("This message is not cached.")

>>> with open(".mylog") as f:
...     print("Contents of logfile:")
...     print(indent(f.read()), end='')  # +ELLIPSIS
Contents of logfile:
    ...: invoked as: ...
    ...: log opened on ...
    This message is cached.
    This message is not cached.
```

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():
...     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.**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.**is_collection**(*obj*)

Identifies objects that can be iterated over, excluding strings.

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

Example:

```
>>> from inform import is_collection
>>> is_collection('')  # string
False

>>> is_collection([])  # list
True

>>> is_collection(())  # tuple
True

>>> is_collection({})  # dictionary
True
```

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_mapping**(*obj*)

Identifies objects that are mappings (are dictionary like).

Returns *True* if argument is a mapping.

Example:

```
>>> from inform import is_mapping
>>> is_mapping('')  # string
False

>>> is_mapping([])  # list
False

>>> is_mapping(())  # tuple
False

>>> is_mapping({})  # dictionary
True
```

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
```

5.3.4 User Utilities

class inform.**Info**(***kwargs*)

Generic Data Structure Class

When instantiated, it converts the provided keyword arguments to attributes. Unknown attributes evaluate to None.

Example:

```
>>> class Orwell(Info):
...     pass

>>> george = Orwell(peace='war', freedom='slavery', ignorance='strength')
>>> print(str(george))
Orwell(
    peace='war',
    freedom='slavery',
    ignorance='strength',
)

>>> george.peace
'war'
```

(continues on next page)

(continued from previous page)

```
>>> george.happiness
```

render(*template*)

Render class to a string

Parameters

template (*str*) – The template string is returned with any instances of {name} replaced by the value of the corresponding attribute.

Example:

```
>>> george.render('Peace is {peace}. Freedom is {freedom}. Ignorance is
↪{ignorance}.')
'Peace is war. Freedom is slavery. Ignorance is strength.'
```

class inform.**ProgressBar**(*stop*, *start=0*, ***, *log=False*, *prefix=None*, *width=79*, *informant=True*, *markers={}*)

Draw a progress bar.

Parameters

- **stop** (*float, iterable*) – The last expected value. May also be an iterable (list, tuple, iterator, etc), in which case the ProgressBar becomes an interable and start and log are ignored.
- **start** (*float*) – The first expected value. May be greater than or less than stop, but it must not equal stop. Must be specified and must be nonzero and the same sign as stop if log is True.
- **log** (*bool*) – Report the logarithmic progress (start and stop must be positive and nonzero).
- **prefix** (*str*) – A string that is output before the progress bar on the same line.
- **width** (*int*) – The maximum width of the bar, the largest factor of 10 that is less than or equal to this value is used.
- **informant** (*informant*) – Which informant to use when outputting the progress bar. By default, `inform.display()` is used. Passing *None* or *False* as *informant* suppresses the display of the progress bar.
- **markers** (*dict*) – This argument is used to associate a marker name with a pair of values, a character and a color. If a known marker name is passed to draw(), the resulting update is rendered using the matching fill character and color. The color may be specified as a string (the color name), a Color object, or None (uncolored).

  Markers should be given in order of increasing priority. If two different markers appear on non-printing updates, the one that is closer to the end of the dictionary is used on the next printing update.

There are three typical use cases. First, use to illustrate the progress through an iterator:

```
for item in ProgressBar(items):
    process(item)
```

Second, use to illustrate the progress through a fixed number of items:

```
for i in ProgressBar(50):
    process(i)
```

Lastly, to illustrate the progress through a continuous range:

```
stop = 1e-6
step = 1e-9
with ProgressBar(stop) as progress:
    value = 0
    while value <= stop:
        progress.draw(value)
        value += step
```

It produces a bar that grows in order to indicate progress. After progress is complete, it will have produced the following:

```
9876543210
```

It coordinates with the informants so that interruptions are handled cleanly:

```
987
warning: the sky is falling.
9876543210
```

This last version can be used to indicate the nature of individual updates. This is usually used to signal that there was a problem with the update. For example, the following example uses both color and fill character to distinguish four types of results: okay, warn, fail, error:

```
results = 'okay okay okay fail okay fail okay error warn okay'.split()

markers = dict(
    okay=('', 'green'),
    warn=('', 'yellow'),
    fail=('×', 'magenta'),
    error=('!', 'red')
)
with ProgressBar(len(results), markers=markers) as progress:
    for i in range(len(repos)):
        result = results[i]
        progress.draw(i+1, result)
```

It produces the following, where each of the types is rendered in the appropriate color:

```
987××××××65××××××43!!!!!!210
```

done()

Complete the progress bar.

Not needed if *ProgressBar* is used with the Python *with* statement.

draw(*abscissa*, *marker=None*)

Draw the progress bar.

escape()

Terminate the progress bar without completing it.

inform.**columns**(*array*, *pagewidth=79*, *alignment='<'*, *leader=' '*, *min_sep_width=2*, *min_col_width=1*)

Distribute array over enough columns to fill the screen.

Returns a multiline string.

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 ('<'), center ('^'), or right ('>') align the *array* items in their columns.
- **leader** (*str*) – The string to prepend to each line.
- **min_sep_width** (*int*) – The minimum number of spaces between columns. Default is 2.
- **min_col_width** (*int*) – The minimum width of a column. Default is 1.

Example:

```
>>> from inform import columns, display, full_stop
>>> title = '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(title, columns(words), sep=newline)
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
```

inform.**conjoin**(*iterable*, *conj=' and '*, *sep=', '*, *end=''*, *fmt=None*)

Conjunction join.

Parameters

- **iterable** (*list or generator of strings*) – The collection of items to be joined. All items are converted to strings.
- **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*.
- **end** (*string*) – Is added to the end of the returned string.
- **fmt** (*string*) – A format string used to convert each item in *iterable* to a string. May be a function, in which case it called on each member of *iterable* and must return a string. If *fmt* is not given, str() is used.

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, Info
>>> display(conjoin([], ' or '))


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

(continues on next page)

(continued from previous page)

```
a

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

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

>>> display(conjoin([10.1, 32.5, 16.9], fmt='${:0.2f}'))
$10.10, $32.50 and $16.90

>>> characters = dict(
...     bob = 'bob@btca.com',
...     ted = 'ted@btca.com',
...     carol = 'carol@btca.com',
...     alice = 'alice@btca.com',
... )
>>> display(conjoin(characters.items(), fmt='{0[0]:>7} : <{0[1]}>', conj='\n', sep=
↪'\n'))
    bob : <bob@btca.com>
    ted : <ted@btca.com>
  carol : <carol@btca.com>
  alice : <alice@btca.com>

>>> characters = [
...     dict(name='bob', email='bob@btca.com'),
...     dict(name='ted', email='ted@btca.com'),
...     dict(name='carol', email='carol@btca.com'),
...     dict(name='alice', email='alice@btca.com'),
... ]
>>> display(conjoin(characters, fmt="{0[name]:>7} : <{0[email]}>", conj=', or\n',␣
↪sep=',\n', end='.'))
    bob : <bob@btca.com>,
    ted : <ted@btca.com>,
  carol : <carol@btca.com>, or
  alice : <alice@btca.com>.

>>> characters = [
...     Info(name='bob', email='bob@btca.com'),
...     Info(name='ted', email='ted@btca.com'),
...     Info(name='carol', email='carol@btca.com'),
...     Info(name='alice', email='alice@btca.com'),
... ]
>>> display(conjoin(characters, fmt='{0.name:>7} : <{0.email}>', conj='; &\n', sep=
↪';\n', end='.'))
    bob : <bob@btca.com>;
    ted : <ted@btca.com>;
  carol : <carol@btca.com>; &
  alice : <alice@btca.com>.

>>> display(conjoin(characters, fmt=lambda a: f'{a.name:>7} : <{a.email}>', conj='\n
↪', sep='\n'))
```

(continues on next page)

(continued from previous page)

```
    bob : <bob@btca.com>
    ted : <ted@btca.com>
  carol : <carol@btca.com>
  alice : <alice@btca.com>
```

inform.**dedent**(*text*, *strip_nl=None*, ***, *bolm=None*, *wrap=False*)

Removes indentation that is common to all lines.

Without its named arguments, dedent behaves just like, and is a equivalent replacement for, textwrap.dedent.

bolm (str):

The beginning of line mark (bolm) is replaced by a space after the indent is removed. It must be the first non-space character after the initial newline. Normally bolm is a single character, often '|', but it may be contain multiple characters, all of which are replaced by spaces.

strip_nl = None:

strip_nl is used to strip off a single leading or trailing newline. strip_nl may be None, 'l', 't', or 'b' representing neither, leading, trailing, or both. True may also be passed, which is equivalent to 'b'. Can also use 's' (start) as synonym for 'l' and 'e' (end) as synonym for 't'.

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.

```
>>> from inform import dedent
```

```
>>> print(dedent('''
...         Diaspar
...          Lys
... ''', bolm=''))

    Diaspar
    Lys
```

```
>>> print(dedent('''
...     |   Diaspar
...     |   Lys
... ''', bolm='|', strip_nl='e'))

    Diaspar
|   Lys
```

```
>>> print(dedent('''
...     ||  Diaspar
...         Lys
... ''', bolm='||', strip_nl='s'))
    Diaspar
    Lys
```

```
>>> print(dedent('''
...         Diaspar
...         Lys
... ''', strip_nl='b'))
```

(continues on next page)

(continued from previous page)

```
Diaspar
Lys
```

```
>>> print(dedent('''
...         Diaspar
...         Lys
... ''', strip_nl='b', wrap=True))
Diaspar Lys
```

inform.**did_you_mean**(*invalid_str*, *valid_strs*)

Given an invalid string from the user, return the valid string with the most similarity.

Parameters

- **invalid_str** (*string*) – The invalid string given by the user.
- **valid_strs** (*iterable*) – The set of valid strings that the user was expected to choose from.

Examples:

```
>>> from inform import did_you_mean
>>> did_you_mean('cat', ['cat', 'dog'])
'cat'
>>> did_you_mean('car', ['cat', 'dog'])
'cat'
>>> did_you_mean('car', {'cat': 1, 'dog': 2})
'cat'
```

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.**format_range**(*items*, *diff=<function <lambda>>*, *key=None*, *str=<class 'str'>*, *block_delim=', '*, *range_delim='-'*)

Create a string that succinctly represents the given set of items. Groups of consecutive items are succinctly displayed as a range, and other items are listed individually.

Parameters

- **items** – An iterable containing the values to format. Any type of iterable can be given, but it will always be treated as a set (e.g. order doesn't matter, duplicates are ignored). By default, the items in the iterable must be non-negative integers, but by customizing the other arguments, it is possible to support any discrete, ordered type.
- **key** (*callable or None*) – A key function used to sort the given values, or None if the values can be sorted directly.
- **str** (*callable*) – A function that can be used to convert an individual value from *items* into a string.
- **block_delim** (*str*) – The character used to separate individual items and ranges in the formatted string.
- **range_delim** (*str*) – The character used to indicate ranges in the formatted string.

Examples:

```
>>> from inform import format_range
>>> format_range([1, 2, 3, 5])
'1-3,5'
>>> abc_diff = lambda a, b: ord(b) - ord(a)
>>> format_range('ACDE', diff=abc_diff)
'A,C-E'
```

inform.**full_stop**(*sentence*, *end='.'*, *allow='.?!'*)

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. The argument is first converted to a string, and then any white space at the end of the string is removed before looking for terminal punctuation. The return value is always a 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?'
```

You can override the allowed and desired endings:

```
>>> cases = '1, 3 9, 12.'.split()
>>> print(*[full_stop(c, end=',', allow=',.') for c in cases])
1, 3, 9, 12.
```

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.**parse_range**(*items_str*, *cast=<class 'int'>*, *range=<function <lambda>>*, *block_delim=', '*, *range_delim='-'*)

Parse a set of values from a string where commas can be used to separate individual items and hyphens can be used to specify ranges of items.

Parameters

- **items_str** (*str*) – The string to parse.
- **cast** (*callable*) – A function that converts items from the given string to the type that will be returned. The function will be given a single argument, which will be a string, and should return that same value casted into the desired type. Note that the casted values will also be used as the inputs for the *range()* function.
- **range** (*callable*) – A function that produces the values implied by a range. It will be given two arguments: the start and end of a range. Both arguments will have already been transformed by the *cast()* function, and the first argument is guaranteed to be less than the second. The function should return an iterable containing all the values in that range, including the start and end values.

- **block_delim** (*str*) – The character used to separate items and ranges.
- **range_delim** (*str*) – The character used to indicate a range.

Returns
All of the values specified by the given string.

Return type
set

Examples:

```
>>> from inform import parse_range
>>> parse_range('1-3,5')
{1, 2, 3, 5}
>>> abc_range = lambda a, b: [chr(x) for x in range(ord(a), ord(b) + 1)]
>>> parse_range('A-C,E', cast=str, range=abc_range)
{'B', 'E', 'C', 'A'}
```

inform.**os_error**(*e*)

Generates clean messages for operating system errors.

Parameters
e (*exception*) – The value of an *OSError* exception.

Example:

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

class inform.**plural**(*value*, *, *render_num=<class 'str'>*, *num='#'*, *invert='!'*, *slash='/'*)

Conditionally format a phrase depending on the number of things.

You may provide the count directly by specifying a number (e.g. 0, 1, 2, ...) for the value. Or the value may be an object that implements *__len__()* (e.g. list, dict, set, ...) in which case the count is the length is taken to be the count.

The format string has one to four sections separated by '/' with the various section being included in the output depending on the count.

ALWAYS ALWAYS/MANY ALWAYS/ONE/MANY ALWAYS/ONE/MANY/NONE

The first section, ALWAYS, is always included, the rest are appended to ALWAYS as appropriate based on the count. If no other sections are given, then an 's' appended to ALWAYS except when the count is 1. Otherwise MANY is added if the count is two or more. It is also used if NONE is not given and the count is zero. ONE is added if available and the count is 1. Finally NONE is added if available and the value is zero.

If any of the sections contain a '#', it is replaced by the number of things.

If the format string starts with an '!' then it is removed and the sense of plurality reverses. The plural form is used if the value is 1 and the singular form is used otherwise. In this situation, NONE is ignored. This is useful when pluralizing verbs.

Examples:

```
>>> from inform import plural

>>> f"{plural(1):thing}, {plural(2):thing}"
'thing, things'

>>> f"{plural(1):bush/es}, {plural(2):bush/es}"
'bush, bushes'

>>> f"{plural(1):/goose/geese}, {plural(2):/goose/geese}"
'goose, geese'

>>> f"{plural(1):# ox/en}, {plural(2):# ox/en}"
'1 ox, 2 oxen'

>>> none = plural(0)
>>> one = plural(1)
>>> many = plural(9)
>>> f"{none:/a cactus/# cacti/no cacti}, {one:/a cactus/# cacti/no cacti}, {many:/a␣
↪cactus/# cacti/no cacti}"
'no cacti, a cactus, 9 cacti'

>>> f"{none:!# run}, {one:!# run}, {many:!# run}"
'0 run, 1 runs, 9 run'

>>> pronoun = 'He'
>>> singers = plural(['John'])
>>> print(f"{singers:/{pronoun}/They} {singers:!sing}.".capitalize())
He sings.

>>> singers = plural(['John', 'Paul', 'George', 'Ringo'])
>>> print(f"{singers:/{pronoun}/They} {singers:!sing}.".capitalize())
They sing.
```

You can specify a function for *render_num* to customize the conversion of the count to a string.

```
>>> from num2words import num2words
```

```
>>> f"He has {plural(1, render_num=num2words):# /wife/wives}."
'He has one wife.'
```

```
>>> f"He has {plural(42, render_num=num2words):# /wife/wives}."
'He has forty-two wives.'
```

You can access the originally specified value using the *value* attribute.

```
>>> agreement = "{tenants:Tenant} ({names}) {tenants:!agree} to ..."
```

```
>>> tenants = plural(["Hayden Fair"])
>>> agreement.format(tenants=tenants, names=conjoin(tenants.value))
'Tenant (Hayden Fair) agrees to ...'
```

```
>>> tenants = plural(["Tawna", "Barbara"])
>>> agreement.format(tenants=tenants, names=conjoin(tenants.value))
'Tenants (Tawna and Barbara) agree to ...'
```

You can access the number of items using the *count* attribute.

```
>>> plural(5).count
5
>>> plural(['a', 'b', 'c']).count
3
>>> plural(1/2).count
0.5
```

If '/', '#', or '!' are inconvenient, you can change them by passing the *slash*, *num* and *invert* arguments to plural().

The original implementation is from Veedrac on Stack Overflow: http://stackoverflow.com/questions/21872366/plural-string-formatting

format(*formatter*)

Expand plural to a string.

You can use this method to directly expand plural to a string without needing to use f-strings or the string format method.

Examples:

```
>>> plural(1).format('thing')
'thing'
>>> plural(3).format('/a cactus/# cacti')
'3 cacti'
```

class inform.**truth**(*value*, *, *interpolate='%'*, *slash='/'*)

Conditionally format a phrase depending on whether it is true or false.

The format string has two sections, separated by '/'. The first section is included only if the given value is true and the last section is included only if the given value is false.

Both sections are optional. If the last section is not given it is left blank. If both sections are not given, 'yes' is returned for true and 'no' for false.

If either section contains %, it is replaced by the value.

Converting truth to a string returns 'yes' or 'no'. Converting truth to a Boolean returns True or False.

Examples:

```
>>> from inform import truth

>>> f"account is {truth(True):past due/current}."
'account is past due.'

>>> f"account is {truth(False):past due/current}."
'account is current.'

>>> paid = truth("20 July 1969")
>>> is_overdue = truth(True)
>>> f"last payment: {paid:%/not received}{is_overdue: - overdue}"
```

(continues on next page)

(continued from previous page)

```
'last payment: 20 July 1969 - overdue'

>>> paid.format('%')
'20 July 1969'

>>> paid = truth(None)
>>> f"last payment: {paid:%/not received}{is_overdue: - overdue}"
'last payment: not received - overdue'

>>> paid.format('%')
''

>>> f"in arrears: {is_overdue}"
'in arrears: yes'

>>> bool(is_overdue)
True

>>> str(is_overdue)
'yes'
```

If '/', or '%' are inconvenient, you can change them by passing the *slash* and *interpolate* arguments to truth().

format(*formatter*)

Expand truth to a string.

You can use this method to directly expand truth to a string without needing to use f-strings or the string format method.

Examples:

```
>>> truth(True).format('yes/no')
'yes'
```

inform.**render**(*obj*, *sort=None*, *level=None*, *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.
- **level** (*int*) – The indent level. If not specified and render is called recursively the indent will be incremented, otherwise the indent is 0.
- **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]}
```

In addition, you can add support for render to your classes by adding one or both of these methods:

_inform_get_args(): returns a list of argument values.

_inform_get_kwargs(): returns a dictionary of keyword arguments.

Example:

```
>>> class Chimera:
...     def __init__(self, *args, **kwargs):
...         self.args = args
...         self.kwargs = kwargs
...
...     def _inform_get_args(self):
...         return self.args
...
...     def _inform_get_kwargs(self):
...         return self.kwargs

>>> lycia = Chimera('Lycia', front='lion', middle='goat', tail='snake')
>>> display(render(lycia))
Chimera(
    'Lycia',
    front='lion',
    middle='goat',
    tail='snake',
)
```

inform.**render_bar**(*value*, *width=72*, *full_width=False*)

Render graphic representation of a value in the form of a bar

Parameters

- **value** (*real*) – Should be normalized (fall between 0 and 1)
- **width** (*int*) – The width of the bar in characters when value is 1.
- **full_width** (*bool*) – Whether bar should be rendered to fill the whole width using trailing spaces,. This is useful if you plan to mark the end of the bar.

Examples:

```
>>> from inform import render_bar

>>> assets = {'property': 13_194, 'cash': 2846, 'equities': 19_301}
>>> total = sum(assets.values())
>>> for key, value in assets.items():
...     display(f"{key:>8}: {render_bar(value/total, full_width=True)}")
property:
    cash:
equities:
```

inform.**title_case**(*s*, *exceptions=('and', 'or', 'nor', 'but', 'a', 'an', 'and', 'the', 'as', 'at', 'by', 'for', 'in', 'of', 'on', 'per', 'to')*)

Convert to title case

This is an attempt to provide an alternative to ‘’.title() that works with acronyms.

There are several tricky cases to worry about in typical order of importance:

0. Upper case first letter of each word that is not an ‘minor’ word.
1. Always upper case first word.
2. Do not down case acronyms
3. Quotes
4. Hyphenated words: drive-in
5. Titles within titles: 2001 A Space Odyssey
6. Maintain leading spacing
7. Maintain given spacing: This is a test. This is only a test.

The following code addresses 0-3 & 7. It was felt that addressing the others would add considerable complexity. Case 2 was handled by simply maintaining all upper case letters in the specified string.

Example:

```
>>> from inform import title_case
>>> cases = '''
...     CDC warns about "aggressive" rats as coronavirus shuts down restaurants
...     L.A. County opens churches, stores, pools, drive-in theaters
...     UConn senior accused of killing two men was looking for young woman
...     Giant asteroid that killed the dinosaurs slammed into Earth at ‘deadliest␣
↪possible angle,’ study reveals
...     Maintain given spacing: This is a test.  This is only a test.
... '''.strip()

>>> for case in cases.splitlines():
...     print(title_case(case))
CDC Warns About "Aggressive" Rats as Coronavirus Shuts Down Restaurants
L.A. County Opens Churches, Stores, Pools, Drive-in Theaters
UConn Senior Accused of Killing Two Men Was Looking for Young Woman
Giant Asteroid That Killed the Dinosaurs Slammed Into Earth at ‘Deadliest Possible␣
↪Angle,’ Study Reveals
Maintain Given Spacing: This Is a Test.  This Is Only a Test.
```

5.3.5 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.**ccc**(**args*, ***kwargs*)

Print the class name for all arguments.

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**(*ignore_exceptions=True*)

Print a stack trace

Parameters
ignore_exceptions – (bool) If true, the stack trace will exclude the path through exceptions.

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.

5.3.6 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.

template may be added to the class as an attribute, in which case it acts as the default template for the exception (used to format the exception arguments into an error message).

The idea of allowing *template* to be an attribute to *Error* was originally proposed on the Python Ideas mailing list by Ryan Fox (https://pypi.org/project/exception-template/).

get_codicil(*codicil=None*)

Get the codicils.

A codicil is extra text attached to an error that can clarify the error message or to give extra context.

Return the codicil as a tuple. If a codicil is specified as an argument, it is appended to the exception's codicil without modifying it.

Parameters
codicil (*string or tuple of strings*) – A codicil or collection of codicils that is appended to the return value without modifying the cached codicil.

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

get_culprit(*culprit=None*)

Get the culprits.

Culprits are extra pieces of information attached to an error that help to identify the source of the error. For example, file name and line number where the error was found are often attached as culprits.

Return the culprit as a tuple. If a culprit is specified as an argument, it is appended to the exception's 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 prepended to the exception's culprit and the combination is returned. The return value is always in the form of a tuple even if there is only one component.

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.

Returns
The formatted message without the culprits.

render(*template=None*, *include_codicil=True*)

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.

- **include_codicil** (*bool*) – Include the codicil in the rendered message.

Returns
The formatted message with any culprits.

report(***new_kwargs*)

Report exception to the user.

Prints the error message on the standard output.

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

Parameters
****kwargs** – *report()* takes any of the normal keyword arguments normally allowed on an informant (culprit, template, etc.). Any keyword argument specified here overrides those that were specified when the exception was first raised.

reraise(***new_kwargs*)

Re-raise the exception with replaced arguments.

terminate(***new_kwargs*)

Report exception and terminate.

Prints the error message on the standard output and exits the program.

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

Parameters

****kwargs** – *report()* takes any of the normal keyword arguments normally allowed on an informant (culprit, template, etc.). Any keyword argument specified here overrides those that were specified when the exception was first raised.

5.4 Accessories

5.4.1 Logging with ntLog

ntLog is a log file aggregation utility.

Unlike daemons, *Inform* based applications tend to run on demand. If the application generates a log file, each run over-writes over a previously generated logfile. This can be problematic if you are interested in keeping a log of events that do not occur during each run.

ntlog is a utility that accumulates logfiles into NestedText file. It provides *NTlog*, a class whose instances provide a output file stream interface. They can be specified to *Inform* as the *logfile*, and in doing so, provide an accumulating logfile. *NTlog* allows you to specify trimming parameters to keep the logfile from getting too big.

Here are two examples that use *ntlog* with *inform*. The first is used with a short-lived processes:

```
from ntlog import NTlog
from inform import Inform, display, error, log

with (
    NTlog('appname.log.nt', keep_for='7d') as ntlog,
    Inform(logfile=ntlog) as inform,
):
    display('status message')
    log('log message')
    if there_is_a_problem:
        error('error message')
    ...
```

The next example demonstrates how to use *ntlog* with long-lived processes. The difference from the above example is that *ntlog* is configured to create a temporary log file and *Inform* is configured to flush after each write. The temporary logfile is intended to allow you to monitor the progress of the process as it runs.

```
with (
    NTlog('appname.log.nt', 'appname.log', keep_for='7d') as ntlog,
    Inform(logfile=ntlog, flush=True) as inform,
):
    display('status message')
    log('log message')
    if there_is_a_problem:
        error('error message')
    ...
```

5.5 Releases

5.5.1 Latest development release

Version: 1.30
Released: 2024-06-07

- Added `truth` utility class.
- Added *render_num* argument to `plural`.
- Added *errors stream_policy*.
- Improved handling of *BrokenPipeErrors*.

5.5.2 1.29 (2024-04-27)

- Change *strip_nl* keys for `dedent()`.
- Added *include_codicil* argument to `Error.render()`.

5.5.3 1.28 (2023-03-20)

- Use critical urgency by default with notifier if message is an error.
- Ignore BrokenPipeError.
- Add *full_width* argument to `render_bar()`.
- Require secondary arguments to `Color` be keyword arguments.
- Require secondary arguments to `plural` be keyword arguments.
- Add *max_col_width* parameter to `columns()`.
- Added type hints.

Warning: `Color` and `plural` now requires secondary arguments to be specified by name.

5.5.4 1.27 (2022-09-15)

- Add markers to `ProgressBar`.
- Change order of arguments to `dedent()`.
- Drop support for Python 2.

5.5.5 1.26 (2021-09-15)

- Added *dedent()*.
- Added *LoggingCache*.

5.5.6 1.25 (2021-07-07)

- Allow culprits to be falsy.

5.5.7 1.24 (2021-05-18)

- Defer evaluation of *stdout* and *stderr*.

5.5.8 1.23 (2020-08-26)

- Strip out empty culprits and codicils.

5.5.9 1.22 (2020-08-24)

- Added *clone* argument to *InformantFactory*.

5.5.10 1.21 (2020-07-20)

- Allow *ProgressBar* output to be suppressed.
- Allow / to be overridden in *plural*
- Various enhancements to *conjoin()* and *full_stop()*.
- Added *parse_range()* and *format_range()* functions.
- Added *title_case()* function.

5.5.11 1.20 (2020-01-08)

- Add *format* method to *plural*.

5.5.12 1.19 (2019-09-25)

- Minor fixes.

5.5.13 1.18 (2019-08-10)

- Wrap now applies to codicils passed as arguments.
- Enhance *plural* (now supports pluralizing verbs).
- Add *fmt* argument to *conjoin()*.
- Support *template* attribute on subclasses of *Error*.

5.5.14 1.17 (2019-05-16)

- Added *is_mapping()*

5.5.15 1.16 (2019-04-27)

- Add end support to *join()*.
- Allow previous logfile to be saved.
- Allow urgency to be specified on notifications.
- Allow *render()* support in user-defined classes with addition of special methods.

5.5.16 1.15 (2019-01-16)

- Added *error_status* argument to *Inform*.
- **Enhanced *plural*. This enhancement is not backward** compatible.
- Enhance for *render()* to allow it to be used in a __repr__ function.

5.5.17 1.14 (2018-12-03)

- Added *render_bar()* utility function.
- Added *ProgressBar* class.
- Added *Info* class.
- **Added *Inform.join_culprit()* method and** `join_culprit()`.
- **Allow culprit to be passed into *Error.report()* and** *Error.terminate()*.
- Added *Error.reraise()* method.
- Allow a codicil or codicils to be added to any informant.
- Added *codicil* named argument to informants and *Error*.
- Added *informant* named argument to *Error*.
- Use colorscheme of active informer as default for colorizers.
- *Error.get_culprit()* now returns a tuple rather than a string.
- Added `Error.join_culprit()`.

- Added *Error.get_codicil()*.

5.5.18 1.13 (2018-08-11)

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

5.5.19 1.12 (2018-02-18)

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

5.5.20 1.11 (2017-12-25)

- Released the documentation.
- Added ability to override template in *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 *cull()*.
- Add support for multiple templates.
- Added *join()* function.
- genindex

INDEX

S

T

V