# say Documentation

***Release 1.6.6***

**Jonathan Eunice**

**Mar 14, 2019**

# Contents

It's been *almost fifty years* since `C` introduced `printf()` and the basic formatted printing of positional parameters. Isn't it time for an upgrade? **You betcha!**

`say` evolves Python's `print` statement/function, `format` function/method, and `%` string interpolation operator with simpler, higher-level facilities. For example, it provides direct template formatting, a feature Python *finally* provided in Python 3.6. In addition:

- DRY, Pythonic, inline string templates that piggyback Python's well-proven `format()` method, syntax, and underlying engine.
- A single output mechanism that works the same way across Python 2 or Python 3.
- A companion `fmt()` object for string formatting.
- Higher-order line formatting such as line numbering, indentation, and line-wrapping built in. You can get substantially better output formatting with almost no additional code.
- Convenient methods for common formatting items such as titles, horizontal separators, and vertical whitespace.
- Easy styled output, including ANSI colors and user-defined styles and text transforms.
- Easy output to one or more files, without additional code or complexity.
- Super-duper template/text aggregator objects for easily building, reading, and writing multi-line texts.

CHAPTER 1

# Usage

```
from say import *

x = 12
nums = list(range(4))
name = 'Fred'

say("There are {x} things.")
say("Nums has {len(nums)} items: {nums}")
say("Name: {name!r}")
```

yields:

```
There are 12 things.
Nums has 4 items: [0, 1, 2, 3]
Name: 'Fred'
```

Or if you want the resulting string, rather than to print the string:

```
>>> fmt("{name} has {x} things and {len(nums)} numbers.")
'Fred has 12 things and 4 numbers.'
```

At this level, `say` is basically a simpler, nicer recasting of:

```
from __future__ import print_function

print("There are {0} things.".format(x))
print("Nums has {0} items: {1}".format(len(nums), nums))
print("Name: {0!r}".format(name))
s = "{0} has {1} things and {2} numbers.".format(name, x, len(nums))
```

(The `import` and numerical sequencing of `{}` format specs is required to make pure-Python code work correctly from Python 2.6 forward from a single code base.)

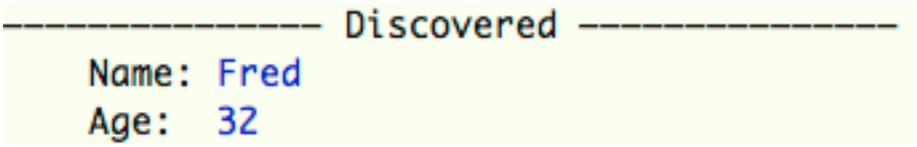
But `say` and `fmt` read so much nicer! They are clear, simplem and direct, and don't separate the place where the value should appear from the value.

Full expressions are are supported within the format braces (`{}`). Whatever variable names or expressions are found therein will be evaluated in the context of the caller.

The more items that are being printed, and the complicated the `format` invocation, the more valuable this simple inline specification becomes.

But `say` isn't just replacing positional templates with inline templates. It also works in a variety of ways to up-level the output-generation task. For example:

```
say.title('Discovered')
say("Name: {name:style=blue}", indent='+1')
say("Age:  {age:style=blue}", indent='+1')
```




Prints a nicely formatted text block, with a proper title and indentation, and just the variable information in blue.

CHAPTER 2

# Indentation and Wrapping

Indentation is a common way to display data hierarchically. `say` will help you manage it. For example:

```
say('ITEMS')
for item in items:
    say(item, indent=1)
```

will indent the items by one indentation level (by default, each indent level is four spaces, but you can change that with the `indent_str` option).

If you want to change the default indentation level:

```
say.set(indent=1)       # to an absolute level
say.set(indent='+1')    # strings => set relative to current level

...

say.set(indent=0)       # to get back to the default, no indent
```

Or you can use a `with` construct:

```
with say.settings(indent='+1'):
    say(...)

    # anything say() emits here will be auto-indented +1 levels

# anything say() emits here, after the with, will not be indented +1
```

**Note:** If using a string to indicate relative indent levels offends your sense of dimensionality or strict typing, there is a class `Relative` that does the same thing in a more formal way. `indent='+2'` and `indent=Relative(+2)` are identical.

If you have a lot of data or text to print, and it would normally create super-long, difficult-to-read lines, you can easily wrap it:

```
say("This is a really long...blah blah blah", wrap=40)
```

Will automatically wrap the text to the given width using Python's standard `textwrap` module. Feel free to use indentation and wrapping together.

# CHAPTER 3

## Prefixes and Suffixes

Every line can be given a prefix or suffix, if desired. For example:

```
with say.settings(prefix='> '):
    say('this')
    say('that')
```

Will give what text email and Markdown consider a quoted block look:

```
> this
> that
```

Or if you'd like some text to be line-quoted with blue marks:

```
say(text, prefix=styled('> ', 'blue'))
```

Or if you'd like output numbered:

```
say.set(prefix=numberer())
say('this\nand\nthat')
```

yields:

```
1: this
2: and
3: that
```

You can instantiate different numberers for different files, and if you like, use the `start` keyword argument to start a `numberer` on a designated value.

Aother common prefixing scenario is needing to use one prefix on the first line, but a second prefix on the remainder of lines. The Python REPL uses this scheme, for example, with the prefix strings `'>>> '` and `'... '`. If you'd like that scheme:

```
say(text, prefix=first_rest('>>> ', '... '))
```

If you want a prefixer to start and run forever, the above is sufficient. If you want a prefixer to reset itself every time it's invoked in a `say()`, initialize it with the kwarg `oneshot=False`.

For example, to put a blue Unicode square as a hanging indicator of where a paragraph starts:

```
bluesquare = first_rest(styled('\u25a0 ', 'blue'), '  ', oneshot=False)
say(text, prefix=bluesquare)
```

## 3.1 Beneath the Covers

Prefixers are essentially generator objects, but with several tweaks. They support `len()` so that layout code can determine how much space to allot for the prefix, and they can be easily reset to their starting point. If they're designed multi-shot (e.g. `oneshot=False`), they'll be auto-reset on each call of `say()`.

# CHAPTER 4

## The Value Proposition

While it's easy enough to add a few spaces to the format string of any `print` statement or function in order to achieve a little indentation, it's easy to mistakenly type too many or too few spaces, or to forget to type them in some format strings. If you're indenting strings that themselves may contain multiple lines, the simple `print` approach breaks because it won't take multi-line strings into account. Nor will it be integrated with line wrapping or numbering or other formatting you also want.

`say`, however, simply and correctly handles these combined formatting operations. Harder cases like multi-line strings are just as nicely and well indented as simple ones–something not otherwise easily accomplished without adding gunky, complexifying string manipulation code to every place in your program that prints anything.

This starts to illustrate `say`'s "do the right thing" philosophy. So many languages' printing and formatting functions "output values" at a low level. They may format basic data types, but they don't provide straightforward ways to do neat text transformations that rapidly yield correct, attractively-formatted output. `say` does. Over time, `say` will provide even more high-level formatting options. For now: indentation, wrapping, and line numbering.

---

**Note:** If you do find any errors in the way `say` handles formatting operations, there's an app for that. Let's fix them once, in a common place, in reusable code–not spread around many different programs.

---

CHAPTER 5

# Titles, Rules, and Spacing

`say` defines a few convenience formatting functions:

```
say.title('Errors', char='-')
for i,e in enumerate(errors, start=1):
    say("{i:3}: {e['name'].upper()}")
```

might yield:

```
--------------- Errors ---------------

  1: I/O ERROR
  2: COMPUTE ERROR
```

A similar method `hr` produces just a horizontal line ("rule"), like the HTML `<hr>` element. For either, one can optionally specify the width (`width`), character repeated to make the line (`char`), and vertical separation/whitespace above and below the item (`vsep`). Good options for the repeated character might be be '-', '=', or parts of the Unicode box drawing character set.

A final method, `sep`, creates a short left-aligned bar with optional following text. It's useful for creating logical subsections.:

```
say.sep("coffee")
say("I prefer coffee")
say.sep("tea", char="=", width=4)
say("I prefer tea")
```

Yields:

```
-- coffee
I prefer coffee

==== tea
I prefer tea
```

You can even define reusable styles for separators (and other say calls):

```
tilde_sep = dict(char="~", width=4)
say.sep("pass one", **tilde_sep)
```

Yields:

```
~~~~ pass one
```

**Note:** The `char` parameter was until recently called `sep`, which conflicted with another use of `sep`. It has since been renamed.

## 5.1 Vertical Spacing

You don't need to add explicit newline characters here and there to achieve good vertical spacing. `say.blank_lines(n)` emits n blank lines. And just about every `say` call also supports a `vsep` (vertical separation) parameter.:

```
say('TITLE', vsep=(2,0)         # add 2 newlines before (none after)
say('=====', vsep=(0,2))        # add 2 newlines after (none before)
say('something else', vsep=1)   # add 1 newline before, 1 after
```

## 5.2 This Just In

A new capability is to differentially set the formatting parameters on a method by method basis. For example, if you want to see titles in green:

```
say.title.set(style='green')
```

You could long set such options on a call-by-call basis, but being able to set the defaults just for specific methods allows you to get more formatting in with fewer characters typed. This capability is available on a limited basis: primarily for format-specific calls (`blank_lines`, `hr`, `sep`, and `title`) for now.

**Note:** `title` and `sep` now print out more vertical whitespace than in previous versions. This is a direct usage of this method-by-method configurability. Basically, `say.title.set(vsep=1)` and `say.sep.set(vsep=(1,0))` now come baked-in.

# CHAPTER 6

## Colors and Styles

`say` has built-in support for style-driven formatting. By default, ANSI terminal colors and styles are automagically supported.

```
answer = 42

say("The answer is {answer:style=bold+red}")
```

This uses the ansicolors module, though with a slightly more permissive syntax. Available colors are 'black', 'blue', 'cyan', 'green', 'magenta', 'red', 'white', and 'yellow'. Available styles are 'bold', 'italic', 'underline', 'blink', 'blink2', 'faint', 'negative', 'concealed', and 'crossed'. These styles can be combined with a + or | character. Note, however, that not all styles are available on every terminal.

---

**Note:** When naming a style within the template braces (`{}`) of format strings, you can quote the style name or not. `fmt("{x:style=red+bold}")` is equivalent to `fmt("{x:style='red+bold'}")`.

---

You can define your own styles:

```
say.style(warning=lambda x: color(x, fg='red'))
```

Because styles are defined through executables (lambdas, usually), they can include decisions or text transformations of arbitrary complexity. For example:

```
say.style(redwarn=lambda n: color(n, fg='red', style='bold') if int(n) < 0 else n)
...
say("Result: {n:style=redwarn}")
```

That will display the number `n` in bold red characters, but only if it's value is negative. For positive numbers, `n` is displayed normally.

Or define a style where a message is surrounded by red stars:

```
say.style(stars=lambda x: fmt('*** ', style='red') + \
                          fmt(x,      style='black') + \
                          fmt(' ***', style='red'))
say.style(redacted=lambda x: 'x' * len(x))

message = 'hey'
say(message, style='stars')
say(message, style='redacted')
```

Yields:

```
*** hey ***
xxx
```

(with red stars)

---

**Note:** Style defining lambdas (or functions) take string arguments. If the string is logically a number, it must be then cast into an `int`, `float`, or whatever. The code must ultimate return a string.

---

You can also apply a style to the entire contents of a `say` or `fmt` invocation:

```
say("There is green everywhere!", style='green|underline')
```

(Whether or not you get underlines, and what shade of green, those depend on the terminal you use.)

Or try:

```
say('a long paragraph with gobs of text',
    style='indigo', prefix=numberer(), wrap=25)
```

Which yields:

```
1: a long paragraph
2: with gobs of text
```

The lines are numbered, they're wrapped to 25 characters, and (on most consoles), the text appears in the color indigo. If you don't think that's an impressive amount of formatting for one function call, you've never tried to implement similar formatting without `say`'s help.

If you want to get really fancy, give the line numbers their own independent styling:

```
linenum = numberer(template=color('{:>3}', fg='hotpink'))
say('a long paragraph with gobs of text',
    style='indigo', prefix=linenum, wrap=25)
```

This is just not something that's practical without `say`, but easy with it.

Styled formatting is an extremely powerful approach, giving the same kind of flexibility and abstraction seen for styles in word processors and CSS-based Web design. It will be further developed.

CHAPTER 7

# Where and Wen You Like

`say` is organized to put output to the place or places you want, and to do so only when you want. The destinations and on/off status can easily be changed.

## 7.1 Where

`say()` writes to a list of files. By default the list is just standard output (`sys.stdout`). But with a simple configuration call, it will write to different–even multiple–files:

```
say.set(files=[sys.stdout, "report.txt"])
say(...)    # now prints to both sys.stdout and report.txt
```

Note that you never even had to open `"report.txt"`. It's okay if you pass in open file objects, but if you pass in strings, they'll be interpreted as file names of intended output files, and opened for you (with UTF-8 encoding, even).

With the above lines, you're now both writing program output as normal *and* capturing it to a file for later inspection and use. Try that with your normal `print` statement/function! It can be done... if you double the number of print calls.

Note however that if you pass a file descriptor that you open yourself, and you're using Python 2, *you* are responsible for opening the file in a way that supports a proper encoding–a detail that Python 3 handles for you. Please see the Encodings and Unicode section for more details and examples.

`say` does, by the by, also support the `file` argument in the same way Python 3's `print()` does. This is a less typical use, but is provided for compatiblity for those converting from `print()` calls.

You can also define your own targeted `Say` instances, for example for error reporting:

```
err = say.clone(files=[sys.stderr, 'error.txt'])
err("Failed with error {errcode}")  # writes in both places
say("something else")   # independent of err
```

## 7.2 When

Output is great, but sometimes you need to go silent. If you want to stop printing for a while:

```
say.set(silent=True)   # no printing until set to False
```

Or transiently:

```
say(...stuff..., silent=not verbose) # prints iff bool(verbose) is True
```

Of course, you don't have to print at all. `fmt()` works exactly like `say()` and inherits most of its options, but doesn't print. (The `C` analogy: `say` **:** `fmt` **::** `printf` **:** `sprintf`.)

## 7.3 How

On occasion it can be valuable to use `say` but not its varaible interpolation. The `say.verbatim()` method does this. All the standard formmating applies, but `{}` variable templates will not be filled in.

# CHAPTER 8

# Encodings and Unicode

Character encodings remain a fractious and often exasperating part of IT.

If you are deal with more than ASCII characters with any regularity whatsoever, for the love of God and all that is holy, use Python 3. It has *greatly* superior support for Unicode characters, and will generally make your life much eaiser.

`say()` and `fmt()` try to avoid encoding gotchas by working with Unicode strings.

In Python 3, all strings are Unicode strings, and all files and I/O streams are inherently smart enough to read and write to reasonable encodings needed to store Unicode on disk. But in Python 2, there is a choice between `str` and `unicode`, and most files are *not* smart enough to use rational encodings. Indeed, files that appear to have an `encoding` attribute will not let you set that attribute, and they will not enforce that encoding when doing file IO. *!@#$%^&!!!*

So if you must use Python 2:

- Seriously reconsider what led you to this point. Work to upgrade to modern Python (e.g. 3.7+) ASAP.
- Use `unicode` strings whenever possible.
- Seriously consider `from __future__ import unicode_literals` to automate the upleveling of string literals to Unicode.
- If you use the basie `str` type, include *only* ASCII characters, not encoded bytes from UTF-8 or whatever. If you don't do this, any trouble results be on your head.
- If `say` opens a file for you, it will do it with the `codecs` module with a default encoding of UTF-8. If you have `say` write to a file that you open, you **must** use `codecs.open()`, `io.open()`, or a similar mechanism that supports proper encoding. Else errors will result.

`say` has a long history of trying to make Python 2 automatically "do the right thing" even when basic Python 2 facilities do not. We have discovered, like so many others before us, that was a fool's errand. Python 2 is simply ill-prepared for day-in, day-out use of Unicode characters that are all around us in the modern global world. While `say` continues some of this with respect to the default standard output (`stdout`) stream, many of the previous back-bends to support auto-encoding have been withdrawn. If you choose to use Python 2, *you* are responsible for opening files in a responsible, encoding-friendly way.:

```
from __future__ import unicode_literals
from codecs import open

contents = u'Contains\u2012Unicode!'
with open('outfile.txt', 'w', encoding='utf-8') as f:
    say(contents, file=f)
```

CHAPTER 9

# Text and Templates

Often the job of output is not about individual text lines, but about creating multi-line files such as scripts and reports. This often leads away from standard output mechanisms toward template packages, but `say` has you covered here as well.

```
from say import Text

# assume `hostname` and `filepath` already defined

script = Text()
script += """
    !#/bin/bash

    # Output the results of a ping command to the given file

    ping {hostname!r} >{filepath!r}
"""

script.write_to("script.sh")
```

Then `script.sh` will contain:

```
!#/bin/bash

# Output the results of a ping command to the given file

ping 'server1234.example.com' >'ping-results.txt'
```

`Text` objects are basically a list of text lines. In most cases, when you add text (either as multi-line strings or lists of strings), `Text` will automatically interpolate variables the same way `say` does. One can simply `print` or `say` `Text` objects, as their `str()` value is the full text you would assume. `Text` objects have both `text` and `lines` properties which can be either accessed or assigned to.

+= incremental assignment automatically removes blank starting and ending lines, and any whitespace prefix that is common to all of the lines (i.e. it will *dedent* any given text). This ensures you don't need to give up nice Python program formatting just to include a template.

While += is a handy way of incrementally building text, it isn't strictly necessary in the simple example above; the `Text(...)` constructor itself accepts a string or set of lines.

Other in-place operators are: `|=` for adding text while preserving leading white space (no dedent) and `&=` adds text verbatim–without dedent or string interpolation.

One can `read_from()` a file (appending the contents of the file to the given text object, with optional interpolation and dedenting). One can also `write_to()` a file. Use the `append` flag if you wish to add to rather than overwrite the file of a given name, and you can set an output encoding if you like (`encoding='utf-8'` is the default).

So far we've discussed `Text` objects almost like strings, but they also act as lists of individual lines (strings). They are, for example, indexable via `[]`, and they are iterable. Their `len()` is the number of lines they contain. One can `append()` or `extend()` them with one or multiple strings, respectively. `append()` takes a keyword parameter `interpolate` that controls whether `{}` expressions in the string are interpolated. `extend()` additionally takes a `dedent` flag that, if true, will automatically remove blank starting and ending lines, and any whitespace prefix that is common to all of the lines.

If `t` is a `Text` instance, `str(t)` will be the full string representing it. If you wish to move from multiple lines to a single-line, joined string, `' '.join(t)` does the trick.

`Text` objects, unlike strings, are mutable. The `replace(x, y)` method will replace all instances of `x` with `y` *in situ*. If given just one argument, a `dict`, all the keys will be replaced with their corresponding values.

`Text` doesn't have the full set of text-onboarding options seen in textdata, but it should suit many circumstances. If you need more, `textdata` can be used alongside `Text`.

Finally, it's possible to use a `Text` object like a file and write to it. So:

```
t = Text()
say.set(files=[sys.stdout, t])

say('something')
```

will now append each thing said to both `sys.stdout` and `t`.

There is a related class `Template` that does not interpolate its format variables when constructed, but rather when explicitly rendered. This suits certain form-filling operations:

```
t = Template("Dear {name},\n\nWelcome to our club!\n")
for name in 'Joe Jane Jeremey'.split():
    print t.render()
```

# CHAPTER 10

## Iterpolators and Exceptions

You may want to write your own functions that take strings and interpolate `{}` format templates in them. The easy way is:

```
from say import caller_fmt

def ucfmt(s):
    return caller_fmt(s).upper()
```

If `ucfmt()` had used `fmt()`, it would not have worked. `fmt()` would look for interpolating values within the context of `ucfmt()` and, not finding any, probably raised an exception. But using `caller_fmt()` it looks into the context of the caller of `ucfmt()`, which is exactly where those values would reside. *Voila!*

And example of how this can work–and a useful tool in its own right–is `FmtException`. If you want to have comprehensible error messages when something goes wrong, you could use `fmt()`:

```
if bad_thing_has_happened:
    raise ValueError(fmt("Parameters {x!r} or {y!r} invalid."))
```

But if you define your own exceptions, consider subclassing `FmtException`:

```
class InvalidParameters(FmtException, ValueError):
    pass

...

if bad_thing_has_happened:
    raise InvalidParameters("Parameters {x!r} or {y!r} invalid.")
```

You'll save a few characters, and the code will be simpler and more comprehensible.

CHAPTER 11

# Python 3

Say works virtually the same way in Python 2 and Python 3. This can simplify software that should work across the versions. For compatibility, `from say import say` is basically a higher-level of `from __future__ import print_function`.

`say` attempts to mask some of the quirky complexities of the 2-to-3 divide, such as string encodings and codec use. In general, things work best if you use Unicode strings any time you need to use non-ASCII characters. In Python 3, this is automatic.

If you are supporting Python 2, recommend you use this import:

```
from __future__ import unicode_literals
```

To default strings to Unicode strings.

And if you are migrating to Python 3.6+'s new f-strings, there is a compatibility shim / polyfill that may be helpful:

    from say import f

    condition = 'good' print(f('this is {condition}'))

While not quite as elegant as the new f-string syntax `f'this is {condition}'`, it has the virtue of working broadly all the way back to Python 2.6.

# CHAPTER 12

# Alternatives

- f-strings. As of Python 3.6, Python finally has formatted strings that are in-place interpolated. Thank you, PEP 498. Some decades after Perl, PHP, Ruby, et al, but *bravo!* nonetheless. In many cases, if all you need are interplated strings, f-strings are grand. Just one of many reasons to upgrade to the latest modern Python builds. Sadly, f-strings lack the easy coloring, wrapping, and other formatting functions `say` builds in. But f-strings are quite compatible with `say`, so feel free to use them together.
- ScopeFormatter provides variable interpolation into strings. It is amazingly compact and elegant. Sadly, it only interpolates Python names, not full expressions. `say` has full expressions, as well as a framework for higher-level printing features beyond `ScopeFormatter`'s...um...scope.
- interpolate is similar to `say.fmt()`, in that it can interpolate complex Python expressions, not just names. Its `i % "format string"` syntax is a little odd, however, in the way that it re-purposes Python's earlier `"C format string" % (values)` style `%` operator. It also depends on the native `print` statement or function, which doesn't help bridge Python 2 and 3.
- Even simpler are invocations of `%` or `format()` using `locals()`. E.g.:

```
name = "Joe"
print "Hello, %(name)!" % locals()
# or
print "Hello, {name}!".format(**locals())
```

  Unfortunately this has even more limitations than `ScopeFormatter`: it only supports local variables, not globals or expressions. And the interpolation code seems gratuitous. Simpler:

```
say("Hello, {name}!")
```

CHAPTER 13

# Notes

- The `say` name was inspired by Perl's say, but the similarity stops there.
- Automated multi-version testing managed with the wonderful pytest, pytest-cov, coverage, and tox. Packaging linting with pyroma.
- Successfully packaged for, and tested against, all late-model versions of Python: 2.7, 3.5, 3.6, and 3.7 as well as late models of PyPy and PyPy3. It may work on Python 2.6 and earlier builds of 3.x (it did historically), but testing can no longer verify that. Also, those are such old Python builds! Upgrade to 3.7 or later ASAP!
- `say` has greater ambitions than just simple template prßinting. It's part of a larger rethinking of how output should be formatted. `say.Text`, show, and quoter are other down-payments on this larger vision. Stay tuned.
- In addition to being a practical module in its own right, `say` is testbed for options, a package that provides high-flexibility option, configuration, and parameter management.
- The author, Jonathan Eunice or @jeunice on Twitter welcomes your comments and suggestions. If you're using `say` in your own work, drop me a note and tell me how you're using it, how you like it, and what you'd like to see!

CHAPTER 14

# To-Dos

- Further formatting techniques for easily generating HTML output and formatting non-scalar values.
- Complete the transition to per-method styling and more refined named styles.
- Provide code that allows `pylint` to see that variables used inside the `say` and `fmt` format strings are indeed thereby used.

# CHAPTER 15

## API Reference

**class** say.**Say**(***kwargs*)

Say provides high-level printing functions. Instances are configurable and callable.

**__call__**(**args*, ***kwargs*)

Primary interface. say(something)

**__init__**(***kwargs*)

Make a Say instance with the given options.

**blank_lines**(*n*, ***kwargs*)

Output N blank lines ("vertical separation"). Unlike other methods, this does not obey normal vertical separation rules, because it is about explicit vertical separation. If it obeyed vsep, it would usually gild the lily (double space).

**but**(***kwargs*)

Create a new instance whose options are chained to this instance's options (and thence to self.__class__.options). kwargs become the cloned instance's overlay options.

**clone**(***kwargs*)

Create a new instance whose options are chained to this instance's options (and thence to self.__class__.options). kwargs become the cloned instance's overlay options.

**static escape**(*s*)

Double { and } characters in a string to 'escape' them so `str.format` doesn't treat them as template characters. NB This is NOT idempotent! Escaping more than once (when { or } are present ) = ERROR.

**fork**(***kwargs*)

Create a new instance whose options are chained to this instance's class's options, then this instance's current values, then any difference values stated in kwwargs.

**hr**(***kwargs*)

Print a horizontal line. Like the HTML hr tag. Optionally specify the width, character repeated to make the line, and vertical separation.

Good options for the separator may be '-', '=', or parts of the Unicode box drawing character set. http://en.wikipedia.org/wiki/Box-drawing_character

**options = Options(styles={}, files=[<open file '<stdout>', mode 'w'>], style=None, end**

**sep**(*text=''*, ***kwargs*)
Print a short horizontal line, possibly with some text following, of the desired width. Useful as a separator for different parts of output.

**set**(***kwargs*)
Permanently change the reciver's settings to those defined in the kwargs. An update-like function.

**setfiles**(*files*)
Set the list of output files. `files` is a list. For each item, if it's a real file like `sys.stdout`, use it. If it's a string, assume it's a filename and open it for writing.

**settings**(***kwargs*)
Open a context manager for a *with* statement. Temporarily change settings for the duration of the with.

**style**(**args*, ***kwargs*)
Define a style.

**title**(*name*, ***kwargs*)
Print a horizontal line with an embedded title.

**verbatim**(**args*, ***kwargs*)
Say, but without interpretation. Useful for just its text decoration features.

**class** say.**Fmt**(***kwargs*)
A type of Say that returns its result, rather than writes it to files.

**__call__**(**args*, ***kwargs*)
Primary interface. say(something)

**__init__**(***kwargs*)
Make a Say instance with the given options.

**blank_lines**(*n*, ***kwargs*)
Output N blank lines ("vertical separation"). Unlike other methods, this does not obey normal vertical separation rules, because it is about explicit vertical separation. If it obeyed vsep, it would usually gild the lily (double space).

**but**(***kwargs*)
Create a new instance whose options are chained to this instance's options (and thence to self.__class__.options). kwargs become the cloned instance's overlay options.

**clone**(***kwargs*)
Create a new instance whose options are chained to this instance's options (and thence to self.__class__.options). kwargs become the cloned instance's overlay options.

**static escape**(*s*)
Double { and } characters in a string to 'escape' them so `str.format` doesn't treat them as template characters. NB This is NOT idempotent! Escaping more than once (when { or } are present ) = ERROR.

**fork**(***kwargs*)
Create a new instance whose options are chained to this instance's class's options, then this instance's current values, then any difference values stated in kwwargs.

**hr**(***kwargs*)
Print a horizontal line. Like the HTML hr tag. Optionally specify the width, character repeated to make the line, and vertical separation.

Good options for the separator may be '-', '=', or parts of the Unicode box drawing character set. http://en.wikipedia.org/wiki/Box-drawing_character

**options = Options(files=Prohibited, styles={}, suffix='', sep=' ', indent_str=' ', pre**

**sep**(*text=''*, ***kwargs*)
Print a short horizontal line, possibly with some text following, of the desired width. Useful as a separator for different parts of output.

**set**(***kwargs*)
Permanently change the reciver's settings to those defined in the kwargs. An update-like function.

**setfiles**(*files*)
Set the list of output files. `files` is a list. For each item, if it's a real file like `sys.stdout`, use it. If it's a string, assume it's a filename and open it for writing.

**settings**(***kwargs*)
Open a context manager for a *with* statement. Temporarily change settings for the duration of the with.

**style**(**args*, ***kwargs*)
Define a style.

**title**(*name*, ***kwargs*)
Print a horizontal line with an embedded title.

**verbatim**(**args*, ***kwargs*)
Say, but without interpretation. Useful for just its text decoration features.

**class** `say.`**Text**(*data=None*, *interpolate=True*, *dedent=True*)

**__init__**(*data=None*, *interpolate=True*, *dedent=True*)
x.__init__(...) initializes x; see help(type(x)) for signature

**append**(*line*, *callframe=None*, *interpolate=True*)

**copy**()
Make a copy.

**extend**(*lines*, *callframe=None*, *interpolate=True*, *dedent=True*)

**insert**(*i*, *data*, *callframe=None*, *interpolate=True*, *dedent=True*)

**lines**

**re_replace**(*target*, *replacement*)
Regular expression replacement. Target is either compiled re object or string that will be compiled into one. Replacement is either string or function that takes re match object as a parameter and returns replacement string.

**read_from**(*filepath*, *interpolate=True*, *dedent=True*, *encoding='utf-8'*)
Reads lines from the designated file, appending them to the end of the given Text. By default, interpolates and dedents any {} expressions.

**render**()
Equivalent to `__str__`. For compatibility with `Template` subclass.

**replace**(*target*, *replacement=None*)
Replace all instances of the target string with the replacement string. Works in situ, contra str.replace().

**text**

**write**(*contents*)
Make it possible for Text objects to operate like file objects, and be written to.

**write_to**(*filepath*, *append=False*, *encoding='utf-8'*)
Write the Text instance's contents to the given filepath. :param filepath: Filepath to write to. :param append: Whether to appened to the file (if it exists) or overwrite (default) :param encoding: How to encode the contents (default utf-8)

CHAPTER 16

# Installation

To install or upgrade to the latest version:

```
pip install -U say
```

To `easy_install` under a specific Python version (3.3 in this example):

```
python3.3 -m easy_install --upgrade say
```

(You may need to prefix these with `sudo` to authorize installation. In environments without super-user privileges, you may want to use `pip`'s `--user` option, to install only for a single user, rather than system-wide.)

## 16.1 Testing

If you wish to run the module tests locally, you'll need to install `pytest` and `tox`. For full testing, you will also need `pytest-cov` and `coverage`. Then run one of these commands:

```
tox                # normal run - speed optimized
tox -e py27        # run for a specific version only (e.g. py27, py34)
tox -c toxcov.ini  # run full coverage tests
```

The provided `tox.ini` and `toxcov.ini` config files do not define a preferred package index / repository. If you want to use them with a specific (presumably local) index, the `-i` option will come in very handy:

```
tox -i INDEX_URL
```

# CHAPTER 17

# Change Log

**1.6.7** (March 14, 2019)

Reworked "prefixers" into more sustainable design that supports both "oneshot" and "multi-shot" variants. A one-shot continues operating indefinitely (e.g. line numbering for an entire document) while multi-shot are reset to their starting position each time `say` is called (for example for numbering multiple examples).

Removed arbitrary encodings (e.g. base64) as target of underlying I/O engine. Too complicated, and the main point.

Tweaked docs. Refreshed dependencies. Tweaked tests. Coverage increased by 1%.

Official support has been removed for Python 3.3 and 3.4 because of problems properly installing dependencies in the testing environment. Say may work on those platforms (it has historically), but regular testing can't verify that. (Also, Python 3.7 is now widely available, and is great. Upgrade if at all possible!)

**1.6.5** (January 17, 2018)

Added new `f` alias to `fmt` as compatibility shim / polyfill for users moving toward Python 3.6+ f-strings, but who have to support prior versions.

**1.6.4** (May 27, 2017)

Now uses the latest version of `ansicolors`, extending to the full set of CSS color names and hex notations, in addition to the traditional small set of ANSI color names. So `say('this', style='peachpuff')` or `say('this', style='#663399')` to your heart's content!

A future release will be needed to extend color name parsing to other notations such as ANSI numeric and CSS `rgb()` spcs.

Also fixed a bug when wrapping, ANSI colors, and colored prefixes are used together.

**1.6.3** (May 26, 2017)

Adds a `say.verbatim` method. It provides all the standard say formatting features, but does NOT interpolate variable expressions in braces. Useful for managing pre-formatted text which might contain expressions without the need for escaping.

Updated Python 2/3 compatibility strategy to be Python 3-centric. Retired _PY3 flag for _PY2 flag, as Python 3 is now the default assumption. That we now exclude 2.x with x < 6 and 3.x with x < 3 helps greatly. 2.6 and 2.7 make great reaches forward toward 3.x, and 3.3 started to make strong reaches backwards.

**1.6.1** (May 23, 2017)

Replaces `textwrap` module with `ansiwrap`. ANSI-colored or styled text can now be correctly wrapped, prefixed, etc. `say` version is bumped only slightly, but this marks a substantial advance in ability to manage colored/styled text in a "just works" way, which is the original premise of the package.

**1.6.0** (May 19, 2017)

Withdrew support for backflip-level attempts to make Python 2 files behave with rational encodings. If `say` opens a file on your behalf, it will do the rigtht thing. It will also try very hard to do the right thing with respect to `sys.stdout`. But for arbitrary files that you open, make sure they're properly encoded. Use `codecs.open` or `io.open` for that.

Reorganized some code. Added and reinstated tests. Bumped coverage +1%, to 97%.

Added `file` parameter to `say()`, to make 1:1 compatible with Python 3's native `print()`.

**1.6.1** (May 15, 2017)

Updated mechanism for method-specific option setting. Still work in progress, but code now much cleaner.

The experimental operator form of `say` has been withdrawn. The operator style isn't consonant with Python philosophy, complicated the code base, and only partially worked. Interesting idea, but experience suggests not worth the trouble.

**1.5.0** (May 14, 2017)

Changed name of parameter `sep` in `hr`, `title`, and `sep` methods because discovered it was conflating and interfering with the `sep` parameter in the main options. The horizontal separator character that is repeated N times is now addressed as `char`.

**1.4.5** (March 22, 2017)

Added `first_rest` prefix helper. First line gets one prefix, (all) subsequent lines get another. Prefix helpers reorganized into their own submodule, `show.prefixes`.

**1.4.4** (March 22, 2017)

Fixed problem with Unicode stream handling under Python 2. It has slipped under the testing radar, given too many mocks and not enough full-out integration testing. Oops!

**1.4.3** (January 23, 2017)

Updates testing for early 2017 Python versions. Successfully packaged for, and tested against, all late-model versions of Python: 2.6, 2.7, 3.3, 3.4, 3.5, and 3.6, as well as PyPy 5.6.0 (based on 2.7.12) and PyPy3 5.5.0 (based on 3.3.5). Python 3.2 removed from official support; no longer a current version of Python and not well-supported by testing matrix.

**1.4.2** (September 15, 2015)

Tested with Python 3.5.0 final.

**1.4.0** (September 8, 2015)

Added ability to set styles for some methods such as `title`, `hr`, and `sep` as an overlay to class, object, and per-call settings. This is a first delivery on what will become a general feature over the next few releases. Added vertical spacing to `title` and `sep` methods for nicer layouts.

Increased testing line coverage to 96%, improving several routines' robustness in the process.

**1.3.12** (September 1, 2015)

Tweaks and testing for new version 1.4 of underlying `options` module.

New `options` version returns support for Python 2.6.

**1.3.9** (August 26, 2015)

Reorganized documentation structure. Updated some setup dependencies.

**1.3.5** (August 17, 2015)

Instituted integrated, multi-version coverage testing with tox, pytest, pytest-cov, and coverage. Initial score: 86%.

**1.3.4** (August 16, 2015)

Updated `SayReturn` logic, which was broken, in order to support an upgrade of `show`

**1.3.3** (August 16, 2015)

Added `sep` method for separators.

Some code cleanups and a few additional tests.å

Officially switched to YAML-format Change Log (`CHANGES.yml`)

**1.3.2** (August 12, 2015)

Code cleanups.

**1.3.1** (August 11, 2015)

Doc, config, and testing updates. Removed `joiner` module and tests. May import that funcationality from `quoter` module in future.

Python 2.6 currently unsupported due to issues with underlying `stuf` module. Support may return, depending on compatibility upgrades for future `stuf` releases.

**1.3** (July 22, 2015)

Added `Template` class. A deferred-rendering version of `Text`

**1.2.6** (July 22, 2015)

Configuration, testing matrix, and doc tweaks.

**1.2.5** (December 29, 2014)

Fixed problem that was occuring with use of Unicode characters when rendered inside the Komodo IDE, which set the `sys.stdout` encoding to `US-ASCII` not `UTF-8`. In those cases, now inserts a codec-based writer object to do the encoding.

**1.2.4** (June 4, 2014)

Now testing for Python 3.3 and 3.4. One slight problem with them when encoding to base64 or similar bytes-oriented output that did not appear in earlier Python 3 builds. Examining.

Added gittip link as an experiment.

**1.2.1** (October 16, 2013)

Fixed bug with quoting of style names/definitions.

Tweaked documentation of style definitions.

**1.2.0** (September 30, 2013)

Added style definitions and convenient access to ANSI colors.

**1.1.0** (September 24, 2013)

Line numbering now an optional way to format output.

Line wrapping is now much more precise. The `wrap` parameter now specifies the line length desired, including however many characters are consumed by prefix, suffix, and indentation.

Vertical spacing is regularized and much better tested. The `vsep` option, previously available only on a few methods, is now available everywhere. `vsep=N` gives N blank lines before and after the given output statement. `vsep=(M,N)` gives M blank lines before, and N blank lines after. A new `Vertical` class describes vertical spacing behind the scenes.

`Say` no longer attempts to handle file encoding itself, but passes this responsibility off to file objects, such as those returned by `io.open`. This is cleaner, though it does remove the whimsical possibility of automagical base64 and rot13 encodings. The `encoding` option is withdrawn as a result.

You can now set the files you'd like to output to in the same way you'd set any other option (e.g. `say.set(files=[...])` or `say.clone(files=[...])`). "Magic" parameter handling is enabled so that if any of the items listed are strings, then a file of that name is opened for writing. Beware, however, that if you manage the files option explicitly (e.g. `say.options.files.append(...)`), you had better provide proper open files. No magical interpretation is done then. The previously-necessary `say.setfiles()` API remains, but is now deprecated.

`fmt()` is now handled by `Fmt`, a proper subclass of `Say`, rather than just through instance settings.

`say()` no longer returns the value it outputs. `retvalue` and `encoded` options have therefore been withdrawn.

**1.0.4** (September 16, 2013)

Had to back out part of the common `__version__` grabbing. Not compatible with Sphinx / readthedocs build process.

**1.0.3** (September 16, 2013)

Added `FmtException` class

Tightened imports for namespace cleanliness.

Doc tweaks.

Added `__version__` metadata common to module, `setup.py`, and docs.

**1.0.2** (September 14, 2013)

Added `prefix` and `suffix` options to `say` and `fmt`, along with docs and tests.

**1.0.1** (September 13, 2013)

Moved main documentation to Sphinx format in `./docs`, and hosted the long-form documentation on readthedocs.org. `README.rst` now an abridged version/teaser for the module.

**1.0** (September 14, 2013)

Cleaned up source for better PEP8 conformance

Bumped version number to 1.0 as part of move to semantic versioning, or at least enough of it so as to not screw up Python installation procedures (which don't seem to understand 0.401 is a lesser version that 0.5, because 401 > 5).

# Index