click documentation

1.0

Armin Ronacher

2017 12 02

Contents

Click is a Python package for creating beautiful command line interfaces in a composable way with as little code as necessary. It's the "Command Line Interface Creation Kit". It's highly configurable but comes with sensible defaults out of the box.

It aims to make the process of writing command line tools quick and fun while also preventing any frustration caused by the inability to implement an intended CLI API.

Click in three points:

- arbitrary nesting of commands
- automatic help page generation
- supports lazy loading of subcommands at runtime

What does it look like? Here is an example of a simple Click program:

```
import click

@click.command()
@click.option('--count', default=1, help='Number of greetings.')
@click.option('--name', prompt='Your name',
              help='The person to greet.')
def hello(count, name):
    """Simple program that greets NAME for a total of COUNT times."""
    for x in range(count):
        click.echo('Hello %s!' % name)

if __name__ == '__main__':
    hello()
```

And what it looks like when run:

```
$ python hello.py --count=3
Your name: John
Hello John!
Hello John!
Hello John!
```

It automatically generates nicely formatted help pages:

```
$ python hello.py --help
Usage: hello.py [OPTIONS]

  Simple program that greets NAME for a total of COUNT times.

Options:
  --count INTEGER  Number of greetings.
  --name TEXT      The person to greet.
  --help           Show this message and exit.
```

You can get the library directly from PyPI:

```
pip install click
```

CHAPTER 1

Documentation Contents

This part of the documentation guides you through all of the library's usage patterns.

1.1 Why Click?

There are so many libraries out there for writing command line utilities; why does Click exist?

This question is easy to answer: because there is not a single command line utility for Python out there which ticks the following boxes:

- is lazily composable without restrictions
- fully follows the Unix command line conventions
- supports loading values from environment variables out of the box
- supports for prompting of custom values
- is fully nestable and composable
- works the same in Python 2 and 3
- supports file handling out of the box
- comes with useful common helpers (getting terminal dimensions, ANSI colors, fetching direct keyboard input, screen clearing, finding config paths, launching apps and editors, etc.)

There are many alternatives to Click and you can have a look at them if you enjoy them better. The obvious ones are `optparse` and `argparse` from the standard library.

Click is actually implemented as a wrapper around a mild fork of `optparse` and does not implement any parsing itself. The reason it's not based on `argparse` is that `argparse` does not allow proper nesting of commands by design and has some deficiencies when it comes to POSIX compliant argument handling.

Click is designed to be fun to work with and at the same time not stand in your way. It's not overly flexible either. Currently, for instance, it does not allow you to customize the help pages too much. This is intentional because Click

is designed to allow you to nest command line utilities. The idea is that you can have a system that works together with another system by tacking two Click instances together and they will continue working as they should.

Too much customizability would break this promise.

Click was written to support the Flask microframework ecosystem because no tool could provide it with the functionality it needed.

To get an understanding of what Click is all about, I strongly recommend looking at the *Complex Applications* chapter to see what it's useful for.

1.1.1 Why not Argparse?

Click is internally based on optparse instead of argparse. This however is an implementation detail that a user does not have to be concerned with. The reason however Click is not using argparse is that it has some problematic behaviors that make handling arbitrary command line interfaces hard:

- argparse has built-in magic behavior to guess if something is an argument or an option. This becomes a problem when dealing with incomplete command lines as it's not possible to know without having a full understanding of the command line how the parser is going to behave. This goes against Click's ambitions of dispatching to subparsers.
- argparse currently does not support disabling of interspersed arguments. Without this feature it's not possible to safely implement Click's nested parsing nature.

1.1.2 Why not Docopt etc.?

Docopt and many tools like it are cool in how they work, but very few of these tools deal with nesting of commands and composability in a way like Click. To the best of the developer's knowledge, Click is the first Python library that aims to create a level of composability of applications that goes beyond what the system itself supports.

Docopt, for instance, acts by parsing your help pages and then parsing according to those rules. The side effect of this is that docopt is quite rigid in how it handles the command line interface. The upside of docopt is that it gives you strong control over your help page; the downside is that due to this it cannot rewrap your output for the current terminal width and it makes translations hard. On top of that docopt is restricted to basic parsing. It does not handle argument dispatching and callback invocation or types. This means there is a lot of code that needs to be written in addition to the basic help page to handle the parsing results.

Most of all, however, it makes composability hard. While docopt does support dispatching to subcommands, it for instance does not directly support any kind of automatic subcommand enumeration based on what's available or it does not enforce subcommands to work in a consistent way.

This is fine, but it's different from how Click wants to work. Click aims to support fully composable command line user interfaces by doing the following:

- Click does not just parse, it also dispatches to the appropriate code.
- Click has a strong concept of an invocation context that allows subcommands to respond to data from the parent command.
- Click has strong information available for all parameters and commands so that it can generate unified help pages for the full CLI and to assist the user in converting the input data as necessary.
- Click has a strong understanding of what types are and can give the user consistent error messages if something goes wrong. A subcommand written by a different developer will not suddenly die with a different error messsage because it's manually handled.

- Click has enough meta information available for its whole program that it can evolve over time to improve the user experience without forcing developers to adjust their programs. For instance, if Click decides to change how help pages are formatted, all Click programs will automatically benefit from this.

The aim of Click is to make composable systems, whereas the aim of docopt is to build the most beautiful and hand-crafted command line interfaces. These two goals conflict with one another in subtle ways. Click actively prevents people from implementing certain patterns in order to achieve unified command line interfaces. You have very little input on reformatting your help pages for instance.

1.1.3 Why Hardcoded Behaviors?

The other question is why Click goes away from optparse and hardcodes certain behaviors instead of staying configurable. There are multiple reasons for this. The biggest one is that too much configurability makes it hard to achieve a consistent command line experience.

The best example for this is optparse's `callback` functionality for accepting arbitrary number of arguments. Due to syntactical ambiguities on the command line, there is no way to implement fully variadic arguments. There are always tradeoffs that need to be made and in case of `argparse` these tradeoffs have been critical enough, that a system like Click cannot even be implemented on top of it.

In this particular case, Click attempts to stay with a handful of accepted paradigms for building command line interfaces that can be well documented and tested.

1.1.4 Why No Auto Correction?

The question came up why Click does not auto correct parameters given that even optparse and argparse support automatic expansion of long arguments. The reason for this is that it's a liability for backwards compatibility. If people start relying on automatically modified parameters and someone adds a new parameter in the future, the script might stop working. These kinds of problems are hard to find so Click does not attempt to be magical about this.

This sort of behavior however can be implemented on a higher level to support things such as explicit aliases. For more information see *Command Aliases*.

1.2 Quickstart

You can get the library directly from PyPI:

```
pip install click
```

The installation into a *virtualenv* is heavily recommended.

1.2.1 virtualenv

Virtualenv is probably what you want to use for developing Click applications.

What problem does virtualenv solve? Chances are that you want to use it for other projects besides your Click script. But the more projects you have, the more likely it is that you will be working with different versions of Python itself, or at least different versions of Python libraries. Let's face it: quite often libraries break backwards compatibility, and it's unlikely that any serious application will have zero dependencies. So what do you do if two or more of your projects have conflicting dependencies?

Virtualenv to the rescue! Virtualenv enables multiple side-by-side installations of Python, one for each project. It doesn't actually install separate copies of Python, but it does provide a clever way to keep different project environments isolated. Let's see how virtualenv works.

If you are on Mac OS X or Linux, chances are that one of the following two commands will work for you:

```
$ sudo easy_install virtualenv
```

or even better:

```
$ sudo pip install virtualenv
```

One of these will probably install virtualenv on your system. Maybe it's even in your package manager. If you use Ubuntu, try:

```
$ sudo apt-get install python-virtualenv
```

If you are on Windows (or none of the above methods worked) you must install `pip` first. For more information about this, see installing pip. Once you have it installed, run the `pip` command from above, but without the *sudo* prefix.

Once you have virtualenv installed, just fire up a shell and create your own environment. I usually create a project folder and a *venv* folder within:

```
$ mkdir myproject
$ cd myproject
$ virtualenv venv
New python executable in venv/bin/python
Installing setuptools, pip.............done.
```

Now, whenever you want to work on a project, you only have to activate the corresponding environment. On OS X and Linux, do the following:

```
$ . venv/bin/activate
```

If you are a Windows user, the following command is for you:

```
$ venv\scripts\activate
```

Either way, you should now be using your virtualenv (notice how the prompt of your shell has changed to show the active environment).

And if you want to go back to the real world, use the following command:

```
$ deactivate
```

After doing this, the prompt of your shell should be as familar as before.

Now, let's move on. Enter the following command to get Click activated in your virtualenv:

```
$ pip install Click
```

A few seconds later and you are good to go.

1.2.2 Screencast and Examples

There is a screencast available which shows the basic API of Click and how to build simple applications with it. It also explores how to build commands with subcommands.

- Building Command Line Applications with Click

Examples of Click applications can be found in the documentation as well as in the GitHub repository together with readme files:

- `inout`: File input and output
- `naval`: Port of docopt naval example
- `aliases`: Command alias example
- `repo`: Git-/Mercurial-like command line interface
- `complex`: Complex example with plugin loading
- `validation`: Custom parameter validation example
- `colors`: Colorama ANSI color support
- `termui`: Terminal UI functions demo
- `imagepipe`: Multi command chaining demo

1.2.3 Basic Concepts

Click is based on declaring commands through decorators. Internally, there is a non-decorator interface for advanced use cases, but it's discouraged for high-level usage.

A function becomes a Click command line tool by decorating it through *`click.command()`*. At its simplest, just decorating a function with this decorator will make it into a callable script:

```
import click

@click.command()
def hello():
    click.echo('Hello World!')
```

What's happening is that the decorator converts the function into a *`Command`* which then can be invoked:

```
if __name__ == '__main__':
    hello()
```

And what it looks like:

```
$ python hello.py
Hello World!
```

And the corresponding help page:

```
$ python hello.py --help
Usage: hello.py [OPTIONS]

Options:
  --help  Show this message and exit.
```

1.2.4 Echoing

Why does this example use *`echo()`* instead of the regular `print()` function? The answer to this question is that Click attempts to support both Python 2 and Python 3 the same way and to be very robust even when the environment is misconfigured. Click wants to be functional at least on a basic level even if everything is completely broken.

What this means is that the *echo()* function applies some error correction in case the terminal is misconfigured instead of dying with an UnicodeError.

As an added benefit, starting with Click 2.0, the echo function also has good support for ANSI colors. It will automatically strip ANSI codes if the output stream is a file and if colorama is supported, ANSI colors will also work on Windows. See *ANSI Colors* for more information.

If you don't need this, you can also use the *print()* construct / function.

1.2.5 Nesting Commands

Commands can be attached to other commands of type *Group*. This allows arbitrary nesting of scripts. As an example here is a script that implements two commands for managing databases:

```
@click.group()
def cli():
    pass

@click.command()
def initdb():
    click.echo('Initialized the database')

@click.command()
def dropdb():
    click.echo('Dropped the database')

cli.add_command(initdb)
cli.add_command(dropdb)
```

As you can see, the *group()* decorator works like the *command()* decorator, but creates a *Group* object instead which can be given multiple subcommands that can be attached with *Group.add_command()*.

For simple scripts, it's also possible to automatically attach and create a command by using the *Group.command()* decorator instead. The above script can instead be written like this:

```
@click.group()
def cli():
    pass

@cli.command()
def initdb():
    click.echo('Initialized the database')

@cli.command()
def dropdb():
    click.echo('Dropped the database')
```

1.2.6 Adding Parameters

To add parameters, use the *option()* and *argument()* decorators:

```
@click.command()
@click.option('--count', default=1, help='number of greetings')
@click.argument('name')
def hello(count, name):
```

```
    for x in range(count):
        click.echo('Hello %s!' % name)
```

What it looks like:

```
$ python hello.py --help
Usage: hello.py [OPTIONS] NAME

Options:
  --count INTEGER  number of greetings
  --help           Show this message and exit.
```

1.2.7 Switching to Setuptools

In the code you wrote so far there is a block at the end of the file which looks like this: `if __name__ == '__main__':`. This is traditionally how a standalone Python file looks like. With Click you can continue doing that, but there are better ways through setuptools.

There are two main (and many more) reasons for this:

The first one is that setuptools automatically generates executable wrappers for Windows so your command line utilities work on Windows too.

The second reason is that setuptools scripts work with virtualenv on Unix without the virtualenv having to be activated. This is a very useful concept which allows you to bundle your scripts with all requirements into a virtualenv.

Click is perfectly equipped to work with that and in fact the rest of the documentation will assume that you are writing applications through setuptools.

I strongly recommend to have a look at the *Setuptools Integration* chapter before reading the rest as the examples assume that you will be using setuptools.

1.3 Setuptools Integration

When writing command line utilities, it's recommended to write them as modules that are distributed with setuptools instead of using Unix shebangs.

Why would you want to do that? There are a bunch of reasons:

1. One of the problems with the traditional approach is that the first module the Python interpreter loads has an incorrect name. This might sound like a small issue but it has quite significant implications.

   The first module is not called by its actual name, but the interpreter renames it to `__main__`. While that is a perfectly valid name it means that if another piece of code wants to import from that module it will trigger the import a second time under its real name and all of a sudden your code is imported twice.

2. Not on all platforms are things that easy to execute. On Linux and OS X you can add a comment to the beginning of the file (`#!/usr/bin/env python`) and your script works like an executable (assuming it has the executable bit set). This however does not work on Windows. While on Windows you can associate interpreters with file extensions (like having everything ending in `.py` execute through the Python interpreter) you will then run into issues if you want to use the script in a virtualenv.

   In fact running a script in a virtualenv is an issue with OS X and Linux as well. With the traditional approach you need to have the whole virtualenv activated so that the correct Python interpreter is used. Not very user friendly.

3. The main trick only works if the script is a Python module. If your application grows too large and you want to start using a package you will run into issues.

1.3.1 Introduction

To bundle your script with setuptools, all you need is the script in a Python package and a `setup.py` file.

Imagine this directory structure:

```
yourscript.py
setup.py
```

Contents of `yourscript.py`:

```
import click

@click.command()
def cli():
    """Example script."""
    click.echo('Hello World!')
```

Contents of `setup.py`:

```
from setuptools import setup

setup(
    name='yourscript',
    version='0.1',
    py_modules=['yourscript'],
    install_requires=[
        'Click',
    ],
    entry_points='''
        [console_scripts]
        yourscript=yourscript:cli
    ''',
)
```

The magic is in the `entry_points` parameter. Below `console_scripts`, each line identifies one console script. The first part before the equals sign (=) is the name of the script that should be generated, the second part is the import path followed by a colon (`:`) with the Click command.

That's it.

1.3.2 Testing The Script

To test the script, you can make a new virtualenv and then install your package:

```
$ virtualenv venv
$ . venv/bin/activate
$ pip install --editable .
```

Afterwards, your command should be available:

```
$ yourscript
Hello World!
```

1.3.3 Scripts in Packages

If your script is growing and you want to switch over to your script being contained in a Python package the changes necessary are minimal. Let's assume your directory structure changed to this:

```
yourpackage/
    __init__.py
    main.py
    utils.py
    scripts/
        __init__.py
        yourscript.py
```

In this case instead of using `py_modules` in your `setup.py` file you can use `packages` and the automatic package finding support of setuptools. In addition to that it's also recommended to include other package data.

These would be the modified contents of `setup.py`:

```
from setuptools import setup, find_packages

setup(
    name='yourpackage',
    version='0.1',
    packages=find_packages(),
    include_package_data=True,
    install_requires=[
        'Click',
    ],
    entry_points='''
        [console_scripts]
        yourscript=yourpackage.scripts.yourscript:cli
    ''',
)
```

1.4 Parameters

Click supports two types of parameters for scripts: options and arguments. There is generally some confusion among authors of command line scripts of when to use which, so here is a quick overview of the differences. As its name indicates, an option is optional. While arguments can be optional within reason, they are much more restricted in how optional they can be.

To help you decide between options and arguments, the recommendation is to use arguments exclusively for things like going to subcommands or input filenames / URLs, and have everything else be an option instead.

1.4.1 Differences

Arguments can do less than options. The following features are only available for options:

- automatic prompting for missing input
- act as flags (boolean or otherwise)
- option values can be pulled from environment variables, arguments can not
- options are fully documented in the help page, arguments are not (this is intentional as arguments might be too specific to be automatically documented)

On the other hand arguments, unlike options, can accept an arbitrary number of arguments. Options can strictly ever only accept a fixed number of arguments (defaults to 1).

1.4.2 Parameter Types

Parameters can be of different types. Types can be implemented with different behavior and some are supported out of the box:

str / *click.STRING*: The default parameter type which indicates unicode strings.

int / *click.INT*: A parameter that only accepts integers.

float / *click.FLOAT*: A parameter that only accepts floating point values.

bool / *click.BOOL*: A parameter that accepts boolean values. This is automatically used for boolean flags. If used with string values `1`, `yes`, `y` and `true` convert to *True* and `0`, `no`, `n` and `false` convert to *False*.

***click.UUID*:** A parameter that accepts UUID values. This is not automatically guessed but represented as `uuid.UUID`.

class click.**File**(*mode='r'*, *encoding=None*, *errors='strict'*, *lazy=None*, *atomic=False*)
Declares a parameter to be a file for reading or writing. The file is automatically closed once the context tears down (after the command finished working).

Files can be opened for reading or writing. The special value – indicates stdin or stdout depending on the mode.

By default, the file is opened for reading text data, but it can also be opened in binary mode or for writing. The encoding parameter can be used to force a specific encoding.

The *lazy* flag controls if the file should be opened immediately or upon first IO. The default is to be non lazy for standard input and output streams as well as files opened for reading, lazy otherwise.

Starting with Click 2.0, files can also be opened atomically in which case all writes go into a separate file in the same folder and upon completion the file will be moved over to the original location. This is useful if a file regularly read by other users is modified.

See *File Arguments* for more information.

class click.**Path**(*exists=False*, *file_okay=True*, *dir_okay=True*, *writable=False*, *readable=True*, *resolve_path=False*, *allow_dash=False*, *path_type=None*)
The path type is similar to the *File* type but it performs different checks. First of all, instead of returning an open file handle it returns just the filename. Secondly, it can perform various basic checks about what the file or directory should be.

6.0 : *allow_dash* was added.

- **exists** – if set to true, the file or directory needs to exist for this value to be valid. If this is not required and a file does indeed not exist, then all further checks are silently skipped.
- **file_okay** – controls if a file is a possible value.
- **dir_okay** – controls if a directory is a possible value.
- **writable** – if true, a writable check is performed.
- **readable** – if true, a readable check is performed.
- **resolve_path** – if this is true, then the path is fully resolved before the value is passed onwards. This means that it's absolute and symlinks are resolved.
- **allow_dash** – If this is set to *True*, a single dash to indicate standard streams is permitted.

- **type** – optionally a string type that should be used to represent the path. The default is *None* which means the return value will be either bytes or unicode depending on what makes most sense given the input data Click deals with.

class click.**Choice**(*choices*)
The choice type allows a value to be checked against a fixed set of supported values. All of these values have to be strings.

See *Choice Options* for an example.

class click.**IntRange**(*min=None*, *max=None*, *clamp=False*)
A parameter that works similar to *click.INT* but restricts the value to fit into a range. The default behavior is to fail if the value falls outside the range, but it can also be silently clamped between the two edges.

See *Range Options* for an example.

Custom parameter types can be implemented by subclassing *click.ParamType*. For simple cases, passing a Python function that fails with a *ValueError* is also supported, though discouraged.

1.4.3 Parameter Names

Parameters (both options and arguments) accept a number of positional arguments which are the parameter declarations. Each string with a single dash is added as short argument; each string starting with a double dash as long one. If a string is added without any dashes, it becomes the internal parameter name which is also used as variable name.

If a parameter is not given a name without dashes, a name is generated automatically by taking the longest argument and converting all dashes to underscores. For an option with `('-f', '--foo-bar')`, the parameter name is *foo_bar*. For an option with `('-x',)`, the parameter is *x*. For an option with `('-f', '--filename', 'dest')`, the parameter is called *dest*.

1.4.4 Implementing Custom Types

To implement a custom type, you need to subclass the *ParamType* class. Types can be invoked with or without context and parameter object, which is why they need to be able to deal with this.

The following code implements an integer type that accepts hex and octal numbers in addition to normal integers, and converts them into regular integers:

```
import click

class BasedIntParamType(click.ParamType):
    name = 'integer'

    def convert(self, value, param, ctx):
        try:
            if value[:2].lower() == '0x':
                return int(value[2:], 16)
            elif value[:1] == '0':
                return int(value, 8)
            return int(value, 10)
        except ValueError:
            self.fail('%s is not a valid integer' % value, param, ctx)

BASED_INT = BasedIntParamType()
```

As you can see, a subclass needs to implement the *ParamType.convert()* method and optionally provide the *ParamType.name* attribute. The latter can be used for documentation purposes.

1.5 Options

Adding options to commands can be accomplished by the *option()* decorator. Since options can come in various different versions, there are a ton of parameters to configure their behavior. Options in click are distinct from *positional arguments*.

1.5.1 Basic Value Options

The most basic option is a value option. These options accept one argument which is a value. If no type is provided, the type of the default value is used. If no default value is provided, the type is assumed to be *STRING*. By default, the name of the parameter is the first long option defined; otherwise the first short one is used.

```
@click.command()
@click.option('--n', default=1)
def dots(n):
    click.echo('.' * n)
```

And on the command line:

```
$ dots --n=2
..
```

In this case the option is of type *INT* because the default value is an integer.

1.5.2 Multi Value Options

Sometimes, you have options that take more than one argument. For options, only a fixed number of arguments is supported. This can be configured by the `nargs` parameter. The values are then stored as a tuple.

```
@click.command()
@click.option('--pos', nargs=2, type=float)
def findme(pos):
    click.echo('%s / %s' % pos)
```

And on the command line:

```
$ findme --pos 2.0 3.0
2.0 / 3.0
```

1.5.3 Tuples as Multi Value Options

4.0 .

As you can see that by using *nargs* set to a specific number each item in the resulting tuple is of the same type. This might not be what you want. Commonly you might want to use different types for different indexes in the tuple. For this you can directly specify a tuple as type:

```
@click.command()
@click.option('--item', type=(unicode, int))
def putitem(item):
    click.echo('name=%s id=%d' % item)
```

And on the command line:

```
$ putitem --item peter 1338
name=peter id=1338
```

By using a tuple literal as type, *nargs* gets automatically set to the length of the tuple and the *click.Tuple* type is automatically used. The above example is thus equivalent to this:

```
@click.command()
@click.option('--item', nargs=2, type=click.Tuple([unicode, int]))
def putitem(item):
    click.echo('name=%s id=%d' % item)
```

1.5.4 Multiple Options

Similarly to `nargs`, there is also the case of wanting to support a parameter being provided multiple times to and have all values recorded – not just the last one. For instance, `git commit -m foo -m bar` would record two lines for the commit message: `foo` and `bar`. This can be accomplished with the `multiple` flag:

Example:

```
@click.command()
@click.option('--message', '-m', multiple=True)
def commit(message):
    click.echo('\n'.join(message))
```

And on the command line:

```
$ commit -m foo -m bar
foo
bar
```

1.5.5 Counting

In some very rare circumstances, it is interesting to use the repetition of options to count an integer up. This can be used for verbosity flags, for instance:

```
@click.command()
@click.option('-v', '--verbose', count=True)
def log(verbose):
    click.echo('Verbosity: %s' % verbose)
```

And on the command line:

```
$ log -vvv
Verbosity: 3
```

1.5.6 Boolean Flags

Boolean flags are options that can be enabled or disabled. This can be accomplished by defining two flags in one go separated by a slash (`/`) for enabling or disabling the option. (If a slash is in an option string, Click automatically knows that it's a boolean flag and will pass `is_flag=True` implicitly.) Click always wants you to provide an enable and disable flag so that you can change the default later.

Example:

```
import sys

@click.command()
@click.option('--shout/--no-shout', default=False)
def info(shout):
    rv = sys.platform
    if shout:
        rv = rv.upper() + '!!!!111'
    click.echo(rv)
```

And on the command line:

```
$ info --shout
LINUX2!!!!111
$ info --no-shout
linux2
```

If you really don't want an off-switch, you can just define one and manually inform Click that something is a flag:

```
import sys

@click.command()
@click.option('--shout', is_flag=True)
def info(shout):
    rv = sys.platform
    if shout:
        rv = rv.upper() + '!!!!111'
    click.echo(rv)
```

And on the command line:

```
$ info --shout
LINUX2!!!!111
```

Note that if a slash is contained in your option already (for instance, if you use Windows-style parameters where `/` is the prefix character), you can alternatively split the parameters through `;` instead:

```
@click.command()
@click.option('/debug;/no-debug')
def log(debug):
    click.echo('debug=%s' % debug)

if __name__ == '__main__':
    log()
```

6.0 .

If you want to define an alias for the second option only, then you will need to use leading whitespace to disambiguate the format string:

Example:

```
import sys

@click.command()
@click.option('--shout/--no-shout', ' /-S', default=False)
def info(shout):
    rv = sys.platform
```

```
    if shout:
        rv = rv.upper() + '!!!!111'
    click.echo(rv)
```

```
$ info --help
Usage: info [OPTIONS]

Options:
  --shout / -S, --no-shout
  --help                    Show this message and exit.
```

1.5.7 Feature Switches

In addition to boolean flags, there are also feature switches. These are implemented by setting multiple options to the same parameter name and defining a flag value. Note that by providing the `flag_value` parameter, Click will implicitly set `is_flag=True`.

To set a default flag, assign a value of *True* to the flag that should be the default.

```
import sys

@click.command()
@click.option('--upper', 'transformation', flag_value='upper',
              default=True)
@click.option('--lower', 'transformation', flag_value='lower')
def info(transformation):
    click.echo(getattr(sys.platform, transformation)())
```

And on the command line:

```
$ info --upper
LINUX2
$ info --lower
linux2
$ info
LINUX2
```

1.5.8 Choice Options

Sometimes, you want to have a parameter be a choice of a list of values. In that case you can use *Choice* type. It can be instantiated with a list of valid values.

Example:

```
@click.command()
@click.option('--hash-type', type=click.Choice(['md5', 'sha1']))
def digest(hash_type):
    click.echo(hash_type)
```

What it looks like:

```
$ digest --hash-type=md5
md5
```

```
$ digest --hash-type=foo
Usage: digest [OPTIONS]

Error: Invalid value for "--hash-type": invalid choice: foo. (choose from md5, sha1)

$ digest --help
Usage: digest [OPTIONS]

Options:
  --hash-type [md5|sha1]
  --help                  Show this message and exit.
```

1.5.9 Prompting

In some cases, you want parameters that can be provided from the command line, but if not provided, ask for user input instead. This can be implemented with Click by defining a prompt string.

Example:

```
@click.command()
@click.option('--name', prompt=True)
def hello(name):
    click.echo('Hello %s!' % name)
```

And what it looks like:

```
$ hello --name=John
Hello John!
$ hello
Name: John
Hello John!
```

If you are not happy with the default prompt string, you can ask for a different one:

```
@click.command()
@click.option('--name', prompt='Your name please')
def hello(name):
    click.echo('Hello %s!' % name)
```

What it looks like:

```
$ hello
Your name please: John
Hello John!
```

1.5.10 Password Prompts

Click also supports hidden prompts and asking for confirmation. This is useful for password input:

```
@click.command()
@click.option('--password', prompt=True, hide_input=True,
              confirmation_prompt=True)
def encrypt(password):
    click.echo('Encrypting password to %s' % password.encode('rot13'))
```

What it looks like:

```
$ encrypt
Password:
Repeat for confirmation:
Encrypting password to frperg
```

Because this combination of parameters is quite common, this can also be replaced with the *password_option()* decorator:

```
@click.command()
@click.password_option()
def encrypt(password):
    click.echo('Encrypting password to %s' % password.encode('rot13'))
```

1.5.11 Dynamic Defaults for Prompts

The `auto_envvar_prefix` and `default_map` options for the context allow the program to read option values from the environment or a configuration file. However, this overrides the prompting mechanism, so that the user does not get the option to change the value interactively.

If you want to let the user configure the default value, but still be prompted if the option isn't specified on the command line, you can do so by supplying a callable as the default value. For example, to get a default from the environment:

```
@click.command()
@click.option('--username', prompt=True,
              default=lambda: os.environ.get('USER', ''))
def hello(username):
    print("Hello,", username)
```

1.5.12 Callbacks and Eager Options

Sometimes, you want a parameter to completely change the execution flow. For instance, this is the case when you want to have a `--version` parameter that prints out the version and then exits the application.

Note: an actual implementation of a `--version` parameter that is reusable is available in Click as *click.version_option()*. The code here is merely an example of how to implement such a flag.

In such cases, you need two concepts: eager parameters and a callback. An eager parameter is a parameter that is handled before others, and a callback is what executes after the parameter is handled. The eagerness is necessary so that an earlier required parameter does not produce an error message. For instance, if `--version` was not eager and a parameter `--foo` was required and defined before, you would need to specify it for `--version` to work. For more information, see *Callback Evaluation Order*.

A callback is a function that is invoked with two parameters: the current *Context* and the value. The context provides some useful features such as quitting the application and gives access to other already processed parameters.

Here an example for a `--version` flag:

```
def print_version(ctx, param, value):
    if not value or ctx.resilient_parsing:
        return
    click.echo('Version 1.0')
    ctx.exit()

@click.command()
```

```
@click.option('--version', is_flag=True, callback=print_version,
              expose_value=False, is_eager=True)
def hello():
    click.echo('Hello World!')
```

The *expose_value* parameter prevents the pretty pointless `version` parameter from being passed to the callback. If that was not specified, a boolean would be passed to the *hello* script. The *resilient_parsing* flag is applied to the context if Click wants to parse the command line without any destructive behavior that would change the execution flow. In this case, because we would exit the program, we instead do nothing.

What it looks like:

```
$ hello
Hello World!
$ hello --version
Version 1.0
```

Callback Signature Changes

In Click 2.0 the signature for callbacks changed. For more information about these changes see *Upgrading to 2.0*.

1.5.13 Yes Parameters

For dangerous operations, it's very useful to be able to ask a user for confirmation. This can be done by adding a boolean `--yes` flag and asking for confirmation if the user did not provide it and to fail in a callback:

```
def abort_if_false(ctx, param, value):
    if not value:
        ctx.abort()

@click.command()
@click.option('--yes', is_flag=True, callback=abort_if_false,
              expose_value=False,
              prompt='Are you sure you want to drop the db?')
def dropdb():
    click.echo('Dropped all tables!')
```

And what it looks like on the command line:

```
$ dropdb
Are you sure you want to drop the db? [y/N]: n
Aborted!
$ dropdb --yes
Dropped all tables!
```

Because this combination of parameters is quite common, this can also be replaced with the *confirmation_option()* decorator:

```
@click.command()
@click.confirmation_option(prompt='Are you sure you want to drop the db?')
def dropdb():
    click.echo('Dropped all tables!')
```

Callback Signature Changes

In Click 2.0 the signature for callbacks changed. For more information about these changes see *Upgrading to 2.0*.

1.5.14 Values from Environment Variables

A very useful feature of Click is the ability to accept parameters from environment variables in addition to regular parameters. This allows tools to be automated much easier. For instance, you might want to pass a configuration file with a `--config` parameter but also support exporting a `TOOL_CONFIG=hello.cfg` key-value pair for a nicer development experience.

This is supported by Click in two ways. One is to automatically build environment variables which is supported for options only. To enable this feature, the `auto_envvar_prefix` parameter needs to be passed to the script that is invoked. Each command and parameter is then added as an uppercase underscore-separated variable. If you have a subcommand called `foo` taking an option called `bar` and the prefix is `MY_TOOL`, then the variable is `MY_TOOL_FOO_BAR`.

Example usage:

```
@click.command()
@click.option('--username')
def greet(username):
    click.echo('Hello %s!' % username)

if __name__ == '__main__':
    greet(auto_envvar_prefix='GREETER')
```

And from the command line:

```
$ export GREETER_USERNAME=john
$ greet
Hello john!
```

The second option is to manually pull values in from specific environment variables by defining the name of the environment variable on the option.

Example usage:

```
@click.command()
@click.option('--username', envvar='USERNAME')
def greet(username):
    click.echo('Hello %s!' % username)

if __name__ == '__main__':
    greet()
```

And from the command line:

```
$ export USERNAME=john
$ greet
Hello john!
```

In that case it can also be a list of different environment variables where the first one is picked.

1.5.15 Multiple Values from Environment Values

As options can accept multiple values, pulling in such values from environment variables (which are strings) is a bit more complex. The way Click solves this is by leaving it up to the type to customize this behavior. For both

`multiple` and `nargs` with values other than `1`, Click will invoke the *ParamType.split_envvar_value()* method to perform the splitting.

The default implementation for all types is to split on whitespace. The exceptions to this rule are the *File* and *Path* types which both split according to the operating system's path splitting rules. On Unix systems like Linux and OS X, the splitting happens for those on every colon (`:`), and for Windows, on every semicolon (`;`).

Example usage:

```
@click.command()
@click.option('paths', '--path', envvar='PATHS', multiple=True,
              type=click.Path())
def perform(paths):
    for path in paths:
        click.echo(path)

if __name__ == '__main__':
    perform()
```

And from the command line:

```
$ export PATHS=./foo/bar:./test
$ perform
./foo/bar
./test
```

1.5.16 Other Prefix Characters

Click can deal with alternative prefix characters other than `-` for options. This is for instance useful if you want to handle slashes as parameters `/` or something similar. Note that this is strongly discouraged in general because Click wants developers to stay close to POSIX semantics. However in certain situations this can be useful:

```
@click.command()
@click.option('+w/-w')
def chmod(w):
    click.echo('writable=%s' % w)

if __name__ == '__main__':
    chmod()
```

And from the command line:

```
$ chmod +w
writable=True
$ chmod -w
writable=False
```

Note that if you are using `/` as prefix character and you want to use a boolean flag you need to separate it with `;` instead of `/`:

```
@click.command()
@click.option('/debug;/no-debug')
def log(debug):
    click.echo('debug=%s' % debug)

if __name__ == '__main__':
    log()
```

1.5.17 Range Options

A special mention should go to the *IntRange* type, which works very similarly to the *INT* type, but restricts the value to fall into a specific range (inclusive on both edges). It has two modes:

- the default mode (non-clamping mode) where a value that falls outside of the range will cause an error.
- an optional clamping mode where a value that falls outside of the range will be clamped. This means that a range of `0-5` would return `5` for the value `10` or `0` for the value `-1` (for example).

Example:

```
@click.command()
@click.option('--count', type=click.IntRange(0, 20, clamp=True))
@click.option('--digit', type=click.IntRange(0, 10))
def repeat(count, digit):
    click.echo(str(digit) * count)

if __name__ == '__main__':
    repeat()
```

And from the command line:

```
$ repeat --count=1000 --digit=5
5555555555555555555
$ repeat --count=1000 --digit=12
Usage: repeat [OPTIONS]

Error: Invalid value for "--digit": 12 is not in the valid range of 0 to 10.
```

If you pass `None` for any of the edges, it means that the range is open at that side.

1.5.18 Callbacks for Validation

2.0 .

If you want to apply custom validation logic, you can do this in the parameter callbacks. These callbacks can both modify values as well as raise errors if the validation does not work.

In Click 1.0, you can only raise the *UsageError* but starting with Click 2.0, you can also raise the *BadParameter* error, which has the added advantage that it will automatically format the error message to also contain the parameter name.

Example:

```
def validate_rolls(ctx, param, value):
    try:
        rolls, dice = map(int, value.split('d', 2))
        return (dice, rolls)
    except ValueError:
        raise click.BadParameter('rolls need to be in format NdM')

@click.command()
@click.option('--rolls', callback=validate_rolls, default='1d6')
def roll(rolls):
    click.echo('Rolling a %d-sided dice %d time(s)' % rolls)

if __name__ == '__main__':
    roll()
```

And what it looks like:

```
$ roll --rolls=42
Usage: roll [OPTIONS]

Error: Invalid value for "--rolls": rolls need to be in format NdM

$ roll --rolls=2d12
Rolling a 12-sided dice 2 time(s)
```

1.6 Arguments

Arguments work similarly to *options* but are positional. They also only support a subset of the features of options due to their syntactical nature. Click will also not attempt to document arguments for you and wants you to document them manually in order to avoid ugly help pages.

1.6.1 Basic Arguments

The most basic option is a simple string argument of one value. If no type is provided, the type of the default value is used, and if no default value is provided, the type is assumed to be *STRING*.

Example:

```
@click.command()
@click.argument('filename')
def touch(filename):
    click.echo(filename)
```

And what it looks like:

```
$ touch foo.txt
foo.txt
```

1.6.2 Variadic Arguments

The second most common version is variadic arguments where a specific (or unlimited) number of arguments is accepted. This can be controlled with the `nargs` parameter. If it is set to `-1`, then an unlimited number of arguments is accepted.

The value is then passed as a tuple. Note that only one argument can be set to `nargs=-1`, as it will eat up all arguments.

Example:

```
@click.command()
@click.argument('src', nargs=-1)
@click.argument('dst', nargs=1)
def copy(src, dst):
    for fn in src:
        click.echo('move %s to folder %s' % (fn, dst))
```

And what it looks like:

```
$ copy foo.txt bar.txt my_folder
move foo.txt to folder my_folder
move bar.txt to folder my_folder
```

Note that this is not how you would write this application. The reason for this is that in this particular example the arguments are defined as strings. Filenames, however, are not strings! They might be on certain operating systems, but not necessarily on all. For better ways to write this, see the next sections.

Note on Non-Empty Variadic Arguments

If you come from `argparse`, you might be missing support for setting `nargs` to `+` to indicate that at least one argument is required.

This is supported by setting `required=True`. However, this should not be used if you can avoid it as we believe scripts should gracefully degrade into becoming noops if a variadic argument is empty. The reason for this is that very often, scripts are invoked with wildcard inputs from the command line and they should not error out if the wildcard is empty.

1.6.3 File Arguments

Since all the examples have already worked with filenames, it makes sense to explain how to deal with files properly. Command line tools are more fun if they work with files the Unix way, which is to accept `-` as a special file that refers to stdin/stdout.

Click supports this through the *`click.File`* type which intelligently handles files for you. It also deals with Unicode and bytes correctly for all versions of Python so your script stays very portable.

Example:

```
@click.command()
@click.argument('input', type=click.File('rb'))
@click.argument('output', type=click.File('wb'))
def inout(input, output):
    while True:
        chunk = input.read(1024)
        if not chunk:
            break
        output.write(chunk)
```

And what it does:

```
$ inout - hello.txt
hello
^D
$ inout hello.txt -
hello
```

1.6.4 File Path Arguments

In the previous example, the files were opened immediately. But what if we just want the filename? The naïve way is to use the default string argument type. However, remember that Click is Unicode-based, so the string will always be a Unicode value. Unfortunately, filenames can be Unicode or bytes depending on which operating system is being used. As such, the type is insufficient.

Instead, you should be using the *Path* type, which automatically handles this ambiguity. Not only will it return either bytes or Unicode depending on what makes more sense, but it will also be able to do some basic checks for you such as existence checks.

Example:

```
@click.command()
@click.argument('f', type=click.Path(exists=True))
def touch(f):
    click.echo(click.format_filename(f))
```

And what it does:

```
$ touch hello.txt
hello.txt

$ touch missing.txt
Usage: touch [OPTIONS] F

Error: Invalid value for "f": Path "missing.txt" does not exist.
```

1.6.5 File Opening Safety

The `FileType` type has one problem it needs to deal with, and that is to decide when to open a file. The default behavior is to be "intelligent" about it. What this means is that it will open stdin/stdout and files opened for reading immediately. This will give the user direct feedback when a file cannot be opened, but it will only open files for writing the first time an IO operation is performed by automatically wrapping the file in a special wrapper.

This behavior can be forced by passing `lazy=True` or `lazy=False` to the constructor. If the file is opened lazily, it will fail its first IO operation by raising an *FileError*.

Since files opened for writing will typically immediately empty the file, the lazy mode should only be disabled if the developer is absolutely sure that this is intended behavior.

Forcing lazy mode is also very useful to avoid resource handling confusion. If a file is opened in lazy mode, it will receive a `close_intelligently` method that can help figure out if the file needs closing or not. This is not needed for parameters, but is necessary for manually prompting with the *prompt()* function as you do not know if a stream like stdout was opened (which was already open before) or a real file that needs closing.

Starting with Click 2.0, it is also possible to open files in atomic mode by passing `atomic=True`. In atomic mode, all writes go into a separate file in the same folder, and upon completion, the file will be moved over to the original location. This is useful if a file regularly read by other users is modified.

1.6.6 Environment Variables

Like options, arguments can also grab values from an environment variable. Unlike options, however, this is only supported for explicitly named environment variables.

Example usage:

```
@click.command()
@click.argument('src', envvar='SRC', type=click.File('r'))
def echo(src):
    click.echo(src.read())
```

And from the command line:

```
$ export SRC=hello.txt
$ echo
Hello World!
```

In that case, it can also be a list of different environment variables where the first one is picked.

Generally, this feature is not recommended because it can cause the user a lot of confusion.

1.6.7 Option-Like Arguments

Sometimes, you want to process arguments that look like options. For instance, imagine you have a file named `-foo.txt`. If you pass this as an argument in this manner, Click will treat it as an option.

To solve this, Click does what any POSIX style command line script does, and that is to accept the string `--` as a separator for options and arguments. After the `--` marker, all further parameters are accepted as arguments.

Example usage:

```
@click.command()
@click.argument('files', nargs=-1, type=click.Path())
def touch(files):
    for filename in files:
        click.echo(filename)
```

And from the command line:

```
$ touch -- -foo.txt bar.txt
-foo.txt
bar.txt
```

1.7 Commands and Groups

The most important feature of Click is the concept of arbitrarily nesting command line utilities. This is implemented through the *Command* and *Group* (actually *MultiCommand*).

1.7.1 Callback Invocation

For a regular command, the callback is executed whenever the command runs. If the script is the only command, it will always fire (unless a parameter callback prevents it. This for instance happens if someone passes `--help` to the script).

For groups and multi commands, the situation looks different. In this case, the callback fires whenever a subcommand fires (unless this behavior is changed). What this means in practice is that an outer command runs when an inner command runs:

```
@click.group()
@click.option('--debug/--no-debug', default=False)
def cli(debug):
    click.echo('Debug mode is %s' % ('on' if debug else 'off'))

@cli.command()
def sync():
    click.echo('Synching')
```

Here is what this looks like:

```
$ tool.py
Usage: tool.py [OPTIONS] COMMAND [ARGS]...

Options:
  --debug / --no-debug
  --help                Show this message and exit.

Commands:
  sync

$ tool.py --debug sync
Debug mode is on
Synching
```

1.7.2 Passing Parameters

Click strictly separates parameters between commands and subcommands. What this means is that options and arguments for a specific command have to be specified *after* the command name itself, but *before* any other command names.

This behavior is already observable with the predefined `--help` option. Suppose we have a program called `tool.py`, containing a subcommand called `sub`.

- `tool.py --help` will return the help for the whole program (listing subcommands).
- `tool.py sub --help` will return the help for the `sub` subcommand.
- But `tool.py --help sub` will treat `--help` as an argument for the main program. Click then invokes the callback for `--help`, which prints the help and aborts the program before click can process the subcommand.

1.7.3 Nested Handling and Contexts

As you can see from the earlier example, the basic command group accepts a debug argument which is passed to its callback, but not to the sync command itself. The sync command only accepts its own arguments.

This allows tools to act completely independent of each other, but how does one command talk to a nested one? The answer to this is the *Context*.

Each time a command is invoked, a new context is created and linked with the parent context. Normally, you can't see these contexts, but they are there. Contexts are passed to parameter callbacks together with the value automatically. Commands can also ask for the context to be passed by marking themselves with the *pass_context()* decorator. In that case, the context is passed as first argument.

The context can also carry a program specified object that can be used for the program's purposes. What this means is that you can build a script like this:

```
@click.group()
@click.option('--debug/--no-debug', default=False)
@click.pass_context
def cli(ctx, debug):
    ctx.obj['DEBUG'] = debug

@cli.command()
@click.pass_context
def sync(ctx):
```

```
    click.echo('Debug is %s' % (ctx.obj['DEBUG'] and 'on' or 'off'))

if __name__ == '__main__':
    cli(obj={})
```

If the object is provided, each context will pass the object onwards to its children, but at any level a context's object can be overridden. To reach to a parent, `context.parent` can be used.

In addition to that, instead of passing an object down, nothing stops the application from modifying global state. For instance, you could just flip a global `DEBUG` variable and be done with it.

1.7.4 Decorating Commands

As you have seen in the earlier example, a decorator can change how a command is invoked. What actually happens behind the scenes is that callbacks are always invoked through the *Context.invoke()* method which automatically invokes a command correctly (by either passing the context or not).

This is very useful when you want to write custom decorators. For instance, a common pattern would be to configure an object representing state and then storing it on the context and then to use a custom decorator to find the most recent object of this sort and pass it as first argument.

For instance, the *pass_obj()* decorator can be implemented like this:

```
from functools import update_wrapper

def pass_obj(f):
    @click.pass_context
    def new_func(ctx, *args, **kwargs):
        return ctx.invoke(f, ctx.obj, *args, **kwargs)
    return update_wrapper(new_func, f)
```

The *Context.invoke()* command will automatically invoke the function in the correct way, so the function will either be called with `f(ctx, obj)` or `f(obj)` depending on whether or not it itself is decorated with *pass_context()*.

This is a very powerful concept that can be used to build very complex nested applications; see *Complex Applications* for more information.

1.7.5 Group Invocation Without Command

By default, a group or multi command is not invoked unless a subcommand is passed. In fact, not providing a command automatically passes `--help` by default. This behavior can be changed by passing `invoke_without_command=True` to a group. In that case, the callback is always invoked instead of showing the help page. The context object also includes information about whether or not the invocation would go to a subcommand.

Example:

```
@click.group(invoke_without_command=True)
@click.pass_context
def cli(ctx):
    if ctx.invoked_subcommand is None:
        click.echo('I was invoked without subcommand')
    else:
        click.echo('I am about to invoke %s' % ctx.invoked_subcommand)
```

```
@cli.command()
def sync():
    click.echo('The subcommand')
```

And how it works in practice:

```
$ tool
I was invoked without subcommand
$ tool sync
I am about to invoke sync
The subcommand
```

1.7.6 Custom Multi Commands

In addition to using *click.group()*, you can also build your own custom multi commands. This is useful when you want to support commands being loaded lazily from plugins.

A custom multi command just needs to implement a list and load method:

```
import click
import os

plugin_folder = os.path.join(os.path.dirname(__file__), 'commands')

class MyCLI(click.MultiCommand):

    def list_commands(self, ctx):
        rv = []
        for filename in os.listdir(plugin_folder):
            if filename.endswith('.py'):
                rv.append(filename[:-3])
        rv.sort()
        return rv

    def get_command(self, ctx, name):
        ns = {}
        fn = os.path.join(plugin_folder, name + '.py')
        with open(fn) as f:
            code = compile(f.read(), fn, 'exec')
            eval(code, ns, ns)
        return ns['cli']

cli = MyCLI(help='This tool\'s subcommands are loaded from a '
            'plugin folder dynamically.')

if __name__ == '__main__':
    cli()
```

These custom classes can also be used with decorators:

```
@click.command(cls=MyCLI)
def cli():
    pass
```

1.7.7 Merging Multi Commands

In addition to implementing custom multi commands, it can also be interesting to merge multiple together into one script. While this is generally not as recommended as it nests one below the other, the merging approach can be useful in some circumstances for a nicer shell experience.

The default implementation for such a merging system is the *CommandCollection* class. It accepts a list of other multi commands and makes the commands available on the same level.

Example usage:

```
import click

@click.group()
def cli1():
    pass

@cli1.command()
def cmd1():
    """Command on cli1"""

@click.group()
def cli2():
    pass

@cli2.command()
def cmd2():
    """Command on cli2"""

cli = click.CommandCollection(sources=[cli1, cli2])

if __name__ == '__main__':
    cli()
```

And what it looks like:

```
$ cli --help
Usage: cli [OPTIONS] COMMAND [ARGS]...

Options:
  --help  Show this message and exit.

Commands:
  cmd1  Command on cli1
  cmd2  Command on cli2
```

In case a command exists in more than one source, the first source wins.

1.7.8 Multi Command Chaining

3.0 .

Sometimes it is useful to be allowed to invoke more than one subcommand in one go. For instance if you have installed a setuptools package before you might be familiar with the `setup.py sdist bdist_wheel upload` command chain which invokes `dist` before `bdist_wheel` before `upload`. Starting with Click 3.0 this is very simple to implement. All you have to do is to pass `chain=True` to your multicommand:

```
@click.group(chain=True)
def cli():
    pass


@cli.command('sdist')
def sdist():
    click.echo('sdist called')


@cli.command('bdist_wheel')
def bdist_wheel():
    click.echo('bdist_wheel called')
```

Now you can invoke it like this:

```
$ setup.py sdist bdist_wheel
sdist called
bdist_wheel called
```

When using multi command chaining you can only have one command (the last) use `nargs=-1` on an argument. It is also not possible to nest multi commands below chained multicommands. Other than that there are no restrictions on how they work. They can accept options and arguments as normal.

Another note: the *Context.invoked_subcommand* attribute is a bit useless for multi commands as it will give `'*'` as value if more than one command is invoked. This is necessary because the handling of subcommands happens one after another so the exact subcommands that will be handled are not yet available when the callback fires.

: It is currently not possible for chain commands to be nested. This will be fixed in future versions of Click.

1.7.9 Multi Command Pipelines

3.0 .

A very common usecase of multi command chaining is to have one command process the result of the previous command. There are various ways in which this can be facilitated. The most obvious way is to store a value on the context object and process it from function to function. This works by decorating a function with *pass_context()* after which the context object is provided and a subcommand can store its data there.

Another way to accomplish this is to setup pipelines by returning processing functions. Think of it like this: when a subcommand gets invoked it processes all of its parameters and comes up with a plan of how to do its processing. At that point it then returns a processing function and returns.

Where do the returned functions go? The chained multicommand can register a callback with *MultiCommand.resultcallback()* that goes over all these functions and then invoke them.

To make this a bit more concrete consider this example:

```
@click.group(chain=True, invoke_without_command=True)
@click.option('-i', '--input', type=click.File('r'))
def cli(input):
    pass

@cli.resultcallback()
def process_pipeline(processors, input):
    iterator = (x.rstrip('\r\n') for x in input)
```

```
    for processor in processors:
        iterator = processor(iterator)
    for item in iterator:
        click.echo(item)

@cli.command('uppercase')
def make_uppercase():
    def processor(iterator):
        for line in iterator:
            yield line.upper()
    return processor

@cli.command('lowercase')
def make_lowercase():
    def processor(iterator):
        for line in iterator:
            yield line.lower()
    return processor

@cli.command('strip')
def make_strip():
    def processor(iterator):
        for line in iterator:
            yield line.strip()
    return processor
```

That's a lot in one go, so let's go through it step by step.

1. The first thing is to make a *group()* that is chainable. In addition to that we also instruct Click to invoke even if no subcommand is defined. If this would not be done, then invoking an empty pipeline would produce the help page instead of running the result callbacks.
2. The next thing we do is to register a result callback on our group. This callback will be invoked with an argument which is the list of all return values of all subcommands and then the same keyword parameters as our group itself. This means we can access the input file easily there without having to use the context object.
3. In this result callback we create an iterator of all the lines in the input file and then pass this iterator through all the returned callbacks from all subcommands and finally we print all lines to stdout.

After that point we can register as many subcommands as we want and each subcommand can return a processor function to modify the stream of lines.

One important thing of note is that Click shuts down the context after each callback has been run. This means that for instance file types cannot be accessed in the *processor* functions as the files will already be closed there. This limitation is unlikely to change because it would make resource handling much more complicated. For such it's recommended to not use the file type and manually open the file through *open_file()*.

For a more complex example that also improves upon handling of the pipelines have a look at the imagepipe multi command chaining demo in the Click repository. It implements a pipeline based image editing tool that has a nice internal structure for the pipelines.

1.7.10 Overriding Defaults

By default, the default value for a parameter is pulled from the `default` flag that is provided when it's defined, but that's not the only place defaults can be loaded from. The other place is the `Context.default_map` (a dictionary) on the context. This allows defaults to be loaded from a configuration file to override the regular defaults.

This is useful if you plug in some commands from another package but you're not satisfied with the defaults.

The default map can be nested arbitrarily for each subcommand and provided when the script is invoked. Alternatively, it can also be overriden at any point by commands. For instance, a top-level command could load the defaults from a configuration file.

Example usage:

```
import click

@click.group()
def cli():
    pass

@cli.command()
@click.option('--port', default=8000)
def runserver(port):
    click.echo('Serving on http://127.0.0.1:%d/' % port)

if __name__ == '__main__':
    cli(default_map={
        'runserver': {
            'port': 5000
        }
    })
```

And in action:

```
$ cli runserver
Serving on http://127.0.0.1:5000/
```

1.7.11 Context Defaults

2.0 .

Starting with Click 2.0 you can override defaults for contexts not just when calling your script, but also in the decorator that declares a command. For instance given the previous example which defines a custom `default_map` this can also be accomplished in the decorator now.

This example does the same as the previous example:

```
import click

CONTEXT_SETTINGS = dict(
    default_map={'runserver': {'port': 5000}}
)

@click.group(context_settings=CONTEXT_SETTINGS)
def cli():
    pass

@cli.command()
@click.option('--port', default=8000)
def runserver(port):
    click.echo('Serving on http://127.0.0.1:%d/' % port)

if __name__ == '__main__':
    cli()
```

And again the example in action:

```
$ cli runserver
Serving on http://127.0.0.1:5000/
```

1.7.12 Command Return Values

3.0 .

One of the new introductions in Click 3.0 is the full support for return values from command callbacks. This enables a whole range of features that were previously hard to implement.

In essence any command callback can now return a value. This return value is bubbled to certain receivers. One usecase for this has already been show in the example of *Multi Command Chaining* where it has been demonstrated that chained multi commands can have callbacks that process all return values.

When working with command return values in Click, this is what you need to know:

- The return value of a command callback is generally returned from the `BaseCommand.invoke()` method. The exception to this rule has to do with `Group`s:
  - In a group the return value is generally the return value of the subcommand invoked. The only exception to this rule is that the return value is the return value of the group callback if it's invoked without arguments and *invoke_without_command* is enabled.
  - If a group is set up for chaining then the return value is a list of all subcommands' results.
  - Return values of groups can be processed through a `MultiCommand.result_callback`. This is invoked with the list of all return values in chain mode, or the single return value in case of non chained commands.
- The return value is bubbled through from the `Context.invoke()` and `Context.forward()` methods. This is useful in situations where you internally want to call into another command.
- Click does not have any hard requirements for the return values and does not use them itself. This allows return values to be used for custom decorators or workflows (like in the multi command chaining example).
- When a Click script is invoked as command line application (through `BaseCommand.main()`) the return value is ignored unless the *standalone_mode* is disabled in which case it's bubbled through.

1.8 User Input Prompts

Click supports prompts in two different places. The first is automated prompts when the parameter handling happens, and the second is to ask for prompts at a later point independently.

This can be accomplished with the `prompt()` function, which asks for valid input according to a type, or the `confirm()` function, which asks for confirmation (yes/no).

1.8.1 Option Prompts

Option prompts are integrated into the option interface. See *Prompting* for more information. Internally, it automatically calls either `prompt()` or `confirm()` as necessary.

1.8.2 Input Prompts

To manually ask for user input, you can use the *prompt()* function. By default, it accepts any Unicode string, but you can ask for any other type. For instance, you can ask for a valid integer:

```
value = click.prompt('Please enter a valid integer', type=int)
```

Additionally, the type will be determined automatically if a default value is provided. For instance, the following will only accept floats:

```
value = click.prompt('Please enter a number', default=42.0)
```

1.8.3 Confirmation Prompts

To ask if a user wants to continue with an action, the *confirm()* function comes in handy. By default, it returns the result of the prompt as a boolean value:

```
if click.confirm('Do you want to continue?'):
    click.echo('Well done!')
```

There is also the option to make the function automatically abort the execution of the program if it does not return `True`:

```
click.confirm('Do you want to continue?', abort=True)
```

1.9 Documenting Scripts

Click makes it very easy to document your command line tools. First of all, it automatically generates help pages for you. While these are currently not customizable in terms of their layout, all of the text can be changed.

1.9.1 Help Texts

Commands and options accept help arguments. In the case of commands, the docstring of the function is automatically used if provided.

Simple example:

```
@click.command()
@click.option('--count', default=1, help='number of greetings')
@click.argument('name')
def hello(count, name):
    """This script prints hello NAME COUNT times."""
    for x in range(count):
        click.echo('Hello %s!' % name)
```

And what it looks like:

```
$ hello --help
Usage: hello [OPTIONS] NAME

  This script prints hello NAME COUNT times.
```

```
Options:
  --count INTEGER  number of greetings
  --help           Show this message and exit.
```

Arguments cannot be documented this way. This is to follow the general convention of Unix tools of using arguments for only the most necessary things and to document them in the introduction text by referring to them by name.

1.9.2 Preventing Rewrapping

The default behavior of Click is to rewrap text based on the width of the terminal. In some circumstances, this can become a problem. The main issue is when showing code examples, where newlines are significant.

Rewrapping can be disabled on a per-paragraph basis by adding a line with solely the `\b` escape marker in it. This line will be removed from the help text and rewrapping will be disabled.

Example:

```
@click.command()
def cli():
    """First paragraph.

    This is a very long second paragraph and as you
    can see wrapped very early in the source text
    but will be rewrapped to the terminal width in
    the final output.

    \b
    This is
    a paragraph
    without rewrapping.

    And this is a paragraph
    that will be rewrapped again.
    """
```

And what it looks like:

```
$ cli --help
Usage: cli [OPTIONS]

  First paragraph.

  This is a very long second paragraph and as you can see wrapped very early
  in the source text but will be rewrapped to the terminal width in the
  final output.

  This is
  a paragraph
  without rewrapping.

  And this is a paragraph that will be rewrapped again.

Options:
  --help  Show this message and exit.
```

1.9.3 Meta Variables

Options and parameters accept a `metavar` argument that can change the meta variable in the help page. The default version is the parameter name in uppercase with underscores, but can be annotated differently if desired. This can be customized at all levels:

```
@click.command(options_metavar='<options>')
@click.option('--count', default=1, help='number of greetings',
              metavar='<int>')
@click.argument('name', metavar='<name>')
def hello(count, name):
    """This script prints hello <name> <int> times."""
    for x in range(count):
        click.echo('Hello %s!' % name)
```

Example:

```
$ hello --help
Usage: hello <options> <name>

  This script prints hello <name> <int> times.

Options:
  --count <int>  number of greetings
  --help         Show this message and exit.
```

1.9.4 Command Short Help

For commands, a short help snippet is generated. By default, it's the first sentence of the help message of the command, unless it's too long. This can also be overridden:

```
@click.group()
def cli():
    """A simple command line tool."""

@cli.command('init', short_help='init the repo')
def init():
    """Initializes the repository."""

@cli.command('delete', short_help='delete the repo')
def delete():
    """Deletes the repository."""
```

And what it looks like:

```
$ repo.py
Usage: repo.py [OPTIONS] COMMAND [ARGS]...

  A simple command line tool.

Options:
  --help  Show this message and exit.

Commands:
  delete  delete the repo
  init    init the repo
```

1.9.5 Help Parameter Customization

2.0 .

The help parameter is implemented in Click in a very special manner. Unlike regular parameters it's automatically added by Click for any command and it performs automatic conflict resolution. By default it's called `--help`, but this can be changed. If a command itself implements a parameter with the same name, the default help parameter stops accepting it. There is a context setting that can be used to override the names of the help parameters called *help_option_names*.

This example changes the default parameters to `-h` and `--help` instead of just `--help`:

```
CONTEXT_SETTINGS = dict(help_option_names=['-h', '--help'])

@click.command(context_settings=CONTEXT_SETTINGS)
def cli():
    pass
```

And what it looks like:

```
$ cli -h
Usage: cli [OPTIONS]

Options:
  -h, --help  Show this message and exit.
```

1.10 Complex Applications

Click is designed to assist with the creation of complex and simple CLI tools alike. However, the power of its design is the ability to arbitrarily nest systems together. For instance, if you have ever used Django, you will have realized that it provides a command line utility, but so does Celery. When using Celery with Django, there are two tools that need to interact with each other and be cross-configured.

In a theoretical world of two separate Click command line utilities, they could solve this problem by nesting one inside the other. For instance, the web framework could also load the commands for the message queue framework.

1.10.1 Basic Concepts

To understand how this works, you need to understand two concepts: contexts and the calling convention.

Contexts

Whenever a Click command is executed, a *Context* object is created which holds state for this particular invocation. It remembers parsed parameters, what command created it, which resources need to be cleaned up at the end of the function, and so forth. It can also optionally hold an application-defined object.

Context objects build a linked list until they hit the top one. Each context is linked to a parent context. This allows a command to work below another command and store its own information there without having to be afraid of altering up the state of the parent command.

Because the parent data is available, however, it is possible to navigate to it if needed.

Most of the time, you do not see the context object, but when writing more complex applications it comes in handy. This brings us to the next point.

Calling Convention

When a Click command callback is executed, it's passed all the non-hidden parameters as keyword arguments. Notably absent is the context. However, a callback can opt into being passed to the context object by marking itself with *pass_context()*.

So how do you invoke a command callback if you don't know if it should receive the context or not? The answer is that the context itself provides a helper function (*Context.invoke()*) which can do this for you. It accepts the callback as first argument and then invokes the function correctly.

1.10.2 Building a Git Clone

In this example, we want to build a command line tool that resembles a version control system. Systems like Git usually provide one over-arching command that already accepts some parameters and configuration, and then have extra subcommands that do other things.

The Root Command

At the top level, we need a group that can hold all our commands. In this case, we use the basic *click.group()* which allows us to register other Click commands below it.

For this command, we also want to accept some parameters that configure the state of our tool:

```
import os
import click


class Repo(object):
    def __init__(self, home=None, debug=False):
        self.home = os.path.abspath(home or '.')
        self.debug = debug


@click.group()
@click.option('--repo-home', envvar='REPO_HOME', default='.repo')
@click.option('--debug/--no-debug', default=False,
              envvar='REPO_DEBUG')
@click.pass_context
def cli(ctx, repo_home, debug):
    ctx.obj = Repo(repo_home, debug)
```

Let's understand what this does. We create a group command which can have subcommands. When it is invoked, it will create an instance of a `Repo` class. This holds the state for our command line tool. In this case, it just remembers some parameters, but at this point it could also start loading configuration files and so on.

This state object is then remembered by the context as *obj*. This is a special attribute where commands are supposed to remember what they need to pass on to their children.

In order for this to work, we need to mark our function with *pass_context()*, because otherwise, the context object would be entirely hidden from us.

The First Child Command

Let's add our first child command to it, the clone command:

```
@cli.command()
@click.argument('src')
@click.argument('dest', required=False)
def clone(src, dest):
    pass
```

So now we have a clone command, but how do we get access to the repo? As you can imagine, one way is to use the *pass_context()* function which again will make our callback also get the context passed on which we memorized the repo. However, there is a second version of this decorator called *pass_obj()* which will just pass the stored object, (in our case the repo):

```
@cli.command()
@click.argument('src')
@click.argument('dest', required=False)
@click.pass_obj
def clone(repo, src, dest):
    pass
```

Interleaved Commands

While not relevant for the particular program we want to build, there is also quite good support for interleaving systems. Imagine for instance that there was a super cool plugin for our version control system that needed a lot of configuration and wanted to store its own configuration as *obj*. If we would then attach another command below that, we would all of a sudden get the plugin configuration instead of our repo object.

One obvious way to remedy this is to store a reference to the repo in the plugin, but then a command needs to be aware that it's attached below such a plugin.

There is a much better system that can built by taking advantage of the linked nature of contexts. We know that the plugin context is linked to the context that created our repo. Because of that, we can start a search for the last level where the object stored by the context was a repo.

Built-in support for this is provided by the *make_pass_decorator()* factory, which will create decorators for us that find objects (it internally calls into *Context.find_object()*). In our case, we know that we want to find the closest `Repo` object, so let's make a decorator for this:

```
pass_repo = click.make_pass_decorator(Repo)
```

If we now use `pass_repo` instead of `pass_obj`, we will always get a repo instead of something else:

```
@cli.command()
@click.argument('src')
@click.argument('dest', required=False)
@pass_repo
def clone(repo, src, dest):
    pass
```

Ensuring Object Creation

The above example only works if there was an outer command that created a `Repo` object and stored it in the context. For some more advanced use cases, this might become a problem. The default behavior of *make_pass_decorator()* is to call *Context.find_object()* which will find the object. If it can't find the object, it will raise an error. The alternative behavior is to use *Context.ensure_object()* which will find the object, and if it cannot find it, will create one and store it in the innermost context. This behavior can also be enabled for *make_pass_decorator()* by passing `ensure=True`:

```
pass_repo = click.make_pass_decorator(Repo, ensure=True)
```

In this case, the innermost context gets an object created if it is missing. This might replace objects being placed there earlier. In this case, the command stays executable, even if the outer command does not run. For this to work, the object type needs to have a constructor that accepts no arguments.

As such it runs standalone:

```
@click.command()
@pass_repo
def cp(repo):
    click.echo(repo)
```

As you can see:

```
$ cp
<Repo object at 0x7f196c7c1150>
```

1.11 Advanced Patterns

In addition to common functionality that is implemented in the library itself, there are countless patterns that can be implemented by extending Click. This page should give some insight into what can be accomplished.

1.11.1 Command Aliases

Many tools support aliases for commands. For instance, you can configure `git` to accept `git ci` as alias for `git commit`. Other tools also support auto-discovery for aliases by automatically shortening them.

Click does not support this out of the box, but it's very easy to customize the *Group* or any other *MultiCommand* to provide this functionality.

As explained in *Custom Multi Commands*, a multi command can provide two methods: *list_commands()* and *get_command()*. In this particular case, you only need to override the latter as you generally don't want to enumerate the aliases on the help page in order to avoid confusion.

This following example implements a subclass of *Group* that accepts a prefix for a command. If there were a command called `push`, it would accept `pus` as an alias (so long as it was unique):

```
class AliasedGroup(click.Group):

    def get_command(self, ctx, cmd_name):
        rv = click.Group.get_command(self, ctx, cmd_name)
        if rv is not None:
            return rv
        matches = [x for x in self.list_commands(ctx)
                   if x.startswith(cmd_name)]
        if not matches:
            return None
        elif len(matches) == 1:
            return click.Group.get_command(self, ctx, matches[0])
        ctx.fail('Too many matches: %s' % ', '.join(sorted(matches)))
```

And it can then be used like this:

```
@click.command(cls=AliasedGroup)
def cli():
    pass

@cli.command()
def push():
    pass

@cli.command()
def pop():
    pass
```

1.11.2 Parameter Modifications

Parameters (options and arguments) are forwarded to the command callbacks as you have seen. One common way to prevent a parameter from being passed to the callback is the *expose_value* argument to a parameter which hides the parameter entirely. The way this works is that the *Context* object has a *params* attribute which is a dictionary of all parameters. Whatever is in that dictionary is being passed to the callbacks.

This can be used to make up addition parameters. Generally this pattern is not recommended but in some cases it can be useful. At the very least it's good to know that the system works this way.

```
import urllib

def open_url(ctx, param, value):
    if value is not None:
        ctx.params['fp'] = urllib.urlopen(value)
        return value

@click.command()
@click.option('--url', callback=open_url)
def cli(url, fp=None):
    if fp is not None:
        click.echo('%s: %s' % (url, fp.code))
```

In this case the callback returns the URL unchanged but also passes a second `fp` value to the callback. What's more recommended is to pass the information in a wrapper however:

```
import urllib

class URL(object):

    def __init__(self, url, fp):
        self.url = url
        self.fp = fp

def open_url(ctx, param, value):
    if value is not None:
        return URL(value, urllib.urlopen(value))

@click.command()
@click.option('--url', callback=open_url)
def cli(url):
    if url is not None:
        click.echo('%s: %s' % (url.url, url.fp.code))
```

1.11.3 Token Normalization

2.0 .

Starting with Click 2.0, it's possible to provide a function that is used for normalizing tokens. Tokens are option names, choice values, or command values. This can be used to implement case insensitive options, for instance.

In order to use this feature, the context needs to be passed a function that performs the normalization of the token. For instance, you could have a function that converts the token to lowercase:

```
CONTEXT_SETTINGS = dict(token_normalize_func=lambda x: x.lower())

@click.command(context_settings=CONTEXT_SETTINGS)
@click.option('--name', default='Pete')
def cli(name):
    click.echo('Name: %s' % name)
```

And how it works on the command line:

```
$ cli --NAME=Pete
Name: Pete
```

1.11.4 Invoking Other Commands

Sometimes, it might be interesting to invoke one command from another command. This is a pattern that is generally discouraged with Click, but possible nonetheless. For this, you can use the *Context.invoke()* or *Context.forward()* methods.

They work similarly, but the difference is that *Context.invoke()* merely invokes another command with the arguments you provide as a caller, whereas *Context.forward()* fills in the arguments from the current command. Both accept the command as the first argument and everything else is passed onwards as you would expect.

Example:

```
cli = click.Group()

@cli.command()
@click.option('--count', default=1)
def test(count):
    click.echo('Count: %d' % count)

@cli.command()
@click.option('--count', default=1)
@click.pass_context
def dist(ctx, count):
    ctx.forward(test)
    ctx.invoke(test, count=42)
```

And what it looks like:

```
$ cli dist
Count: 1
Count: 42
```

1.11.5 Callback Evaluation Order

Click works a bit differently than some other command line parsers in that it attempts to reconcile the order of arguments as defined by the programmer with the order of arguments as defined by the user before invoking any callbacks.

This is an important concept to understand when porting complex patterns to Click from optparse or other systems. A parameter callback invocation in optparse happens as part of the parsing step, whereas a callback invocation in Click happens after the parsing.

The main difference is that in optparse, callbacks are invoked with the raw value as it happens, whereas a callback in Click is invoked after the value has been fully converted.

Generally, the order of invocation is driven by the order in which the user provides the arguments to the script; if there is an option called `--foo` and an option called `--bar` and the user calls it as `--bar --foo`, then the callback for `bar` will fire before the one for `foo`.

There are three exceptions to this rule which are important to know:

Eagerness: An option can be set to be "eager". All eager parameters are evaluated before all non-eager parameters, but again in the order as they were provided on the command line by the user.

This is important for parameters that execute and exit like `--help` and `--version`. Both are eager parameters, but whatever parameter comes first on the command line will win and exit the program.

Repeated parameters: If an option or argument is split up on the command line into multiple places because it is repeated – for instance, `--exclude foo --include baz --exclude bar` – the callback will fire based on the position of the first option. In this case, the callback will fire for `exclude` and it will be passed both options (`foo` and `bar`), then the callback for `include` will fire with `baz` only.

Note that even if a parameter does not allow multiple versions, Click will still accept the position of the first, but it will ignore every value except the last. The reason for this is to allow composability through shell aliases that set defaults.

Missing parameters: If a parameter is not defined on the command line, the callback will still fire. This is different from how it works in optparse where undefined values do not fire the callback. Missing parameters fire their callbacks at the very end which makes it possible for them to default to values from a parameter that came before.

Most of the time you do not need to be concerned about any of this, but it is important to know how it works for some advanced cases.

1.11.6 Forwarding Unknown Options

In some situations it is interesting to be able to accept all unknown options for further manual processing. Click can generally do that as of Click 4.0, but it has some limitations that lie in the nature of the problem. The support for this is provided through a parser flag called `ignore_unknown_options` which will instruct the parser to collect all unknown options and to put them to the leftover argument instead of triggering a parsing error.

This can generally be activated in two different ways:

1. It can be enabled on custom *Command* subclasses by changing the *ignore_unknown_options* attribute.
2. It can be enabled by changing the attribute of the same name on the context class (*Context.ignore_unknown_options*). This is best changed through the `context_settings` dictionary on the command.

For most situations the easiest solution is the second. Once the behavior is changed something needs to pick up those leftover options (which at this point are considered arguments). For this again you have two options:

1. You can use *pass_context()* to get the context passed. This will only work if in addition to *ignore_unknown_options* you also set *allow_extra_args* as otherwise the command will abort with an error that there are leftover arguments. If you go with this solution, the extra arguments will be collected in *Context.args*.
2. You can attach a *argument()* with `nargs` set to *-1* which will eat up all leftover arguments. In this case it's recommeded to set the *type* to *UNPROCESSED* to avoid any string processing on those arguments as otherwise they are forced into unicode strings automatically which is often not what you want.

In the end you end up with something like this:

```
import sys
from subprocess import call

@click.command(context_settings=dict(
    ignore_unknown_options=True,
))
@click.option('-v', '--verbose', is_flag=True, help='Enables verbose mode')
@click.argument('timeit_args', nargs=-1, type=click.UNPROCESSED)
def cli(verbose, timeit_args):
    """A wrapper around Python's timeit."""
    cmdline = ['python', '-mtimeit'] + list(timeit_args)
    if verbose:
        click.echo('Invoking: %s' % ' '.join(cmdline))
    call(cmdline)
```

And what it looks like:

```
$ cli --help
Usage: cli [OPTIONS] [TIMEIT_ARGS]...

  A wrapper around Python's timeit.

Options:
  -v, --verbose  Enables verbose mode
  --help         Show this message and exit.

$ cli -n 100 "a = 1; b = 2; a * b"
100 loops, best of 3: 0.0787 usec per loop

$ cli -v "a = 1; b = 2; a * b"
Invoking: python -mtimeit a = 1; b = 2; a * b
10000000 loops, best of 3: 0.0774 usec per loop
```

As you can see the verbosity flag is handled by Click, everything else ends up in the *timeit_args* variable for further processing which then for instance, allows invoking a subprocess. There are a few things that are important to know about how this ignoring of unhandled flag happens:

- Unknown long options are generally ignored and not processed at all. So for instance if `--foo=bar` or `--foo bar` are passed they generally end up like that. Note that because the parser cannot know if an option will accept an argument or not, the `bar` part might be handled as an argument.
- Unknown short options might be partially handled and reassmebled if necessary. For instance in the above example there is an option called `-v` which enables verbose mode. If the command would be ignored with `-va` then the `-v` part would be handled by Click (as it is known) and `-a` would end up in the leftover parameters for further processing.
- Depending on what you plan on doing you might have some success by disabling interspersed arguments (*allow_interspersed_args*) which instructs the parser to not allow arguments and options to be mixed.

Depending on your situation this might improve your results.

Generally though the combinated handling of options and arguments from your own commands and commands from another application are discouraged and if you can avoid it, you should. It's a much better idea to have everything below a subcommand be forwarded to another application than to handle some arguments yourself.

1.11.7 Global Context Access

5.0 .

Starting with Click 5.0 it is possible to access the current context from anywhere within the same through through the use of the *get_current_context()* function which returns it. This is primarily useful for accessing the context bound object as well as some flags that are stored on it to customize the runtime behavior. For instance the *echo()* function does this to infer the default value of the *color* flag.

Example usage:

```
def get_current_command_name():
    return click.get_current_context().info_name
```

It should be noted that this only works within the current thread. If you spawn additional threads then those threads will not have the ability to refer to the current context. If you want to give another thread the ability to refer to this context you need to use the context within the thread as a context manager:

```
def spawn_thread(ctx, func):
    def wrapper():
        with ctx:
            func()
    t = threading.Thread(target=wrapper)
    t.start()
    return t
```

Now the thread function can access the context like the main thread would do. However if you do use this for threading you need to be very careful as the vast majority of the context is not thread safe! You are only allowed to read from the context, but not to perform any modifications on it.

1.12 Testing Click Applications

For basic testing, Click provides the `click.testing` module which provides test functionality that helps you invoke command line applications and check their behavior.

These tools should really only be used for testing as they change the entire interpreter state for simplicity and are not in any way thread-safe!

1.12.1 Basic Testing

The basic functionality for testing Click applications is the *CliRunner* which can invoke commands as command line scripts. The *CliRunner.invoke()* method runs the command line script in isolation and captures the output as both bytes and binary data.

The return value is a *Result* object, which has the captured output data, exit code, and optional exception attached.

Example:

```
import click
from click.testing import CliRunner

def test_hello_world():
    @click.command()
    @click.argument('name')
    def hello(name):
        click.echo('Hello %s!' % name)

    runner = CliRunner()
    result = runner.invoke(hello, ['Peter'])
    assert result.exit_code == 0
    assert result.output == 'Hello Peter!\n'
```

For subcommand testing, a subcommand name must be specified in the *args* parameter of *CliRunner.invoke()* method.

Example:

```
import click
from click.testing import CliRunner

def test_sync():
    @click.group()
    @click.option('--debug/--no-debug', default=False)
    def cli(debug):
        click.echo('Debug mode is %s' % ('on' if debug else 'off'))

    @cli.command()
    def sync():
        click.echo('Syncing')

    runner = CliRunner()
    result = runner.invoke(cli, ['--debug', 'sync'])
    assert result.exit_code == 0
    assert 'Debug mode is on' in result.output
    assert 'Syncing' in result.output
```

1.12.2 File System Isolation

For basic command line tools that want to operate with the file system, the *CliRunner.isolated_filesystem()* method comes in useful which sets up an empty folder and changes the current working directory to.

Example:

```
import click
from click.testing import CliRunner

def test_cat():
    @click.command()
    @click.argument('f', type=click.File())
    def cat(f):
        click.echo(f.read())

    runner = CliRunner()
    with runner.isolated_filesystem():
```

```
        with open('hello.txt', 'w') as f:
            f.write('Hello World!')

        result = runner.invoke(cat, ['hello.txt'])
        assert result.exit_code == 0
        assert result.output == 'Hello World!\n'
```

1.12.3 Input Streams

The test wrapper can also be used to provide input data for the input stream (stdin). This is very useful for testing prompts, for instance:

```
import click
from click.testing import CliRunner

def test_prompts():
    @click.command()
    @click.option('--foo', prompt=True)
    def test(foo):
        click.echo('foo=%s' % foo)

    runner = CliRunner()
    result = runner.invoke(test, input='wau wau\n')
    assert not result.exception
    assert result.output == 'Foo: wau wau\nfoo=wau wau\n'
```

Note that prompts will be emulated so that they write the input data to the output stream as well. If hidden input is expected then this obviously does not happen.

1.13 Utilities

Besides the functionality that Click provides to interface with argument parsing and handling, it also provides a bunch of addon functionality that is useful for writing command line utilities.

1.13.1 Printing to Stdout

The most obvious helper is the *echo()* function, which in many ways works like the Python `print` statement or function. The main difference is that it works the same in Python 2 and 3, it intelligently detects misconfigured output streams, and it will never fail (except in Python 3; for more information see *Python 3 Limitations*).

Example:

```
import click

click.echo('Hello World!')
```

Most importantly, it can print both Unicode and binary data, unlike the built-in `print` function in Python 3, which cannot output any bytes. It will, however, emit a trailing newline by default, which needs to be suppressed by passing `nl=False`:

```
click.echo(b'\xe2\x98\x83', nl=False)
```

Last but not least *echo()* uses click's intelligent internal output streams to stdout and stderr which support unicode output on the Windows console. This means for as long as you are using *click.echo* you can output unicode character (there are some limitations on the default font with regards to which characters can be displayed). This functionality is new in Click 6.0.

6.0 .

Click now emulates output streams on Windows to support unicode to the Windows console through separate APIs. For more information see **'wincmd'_**.

3.0 .

Starting with Click 3.0 you can also easily print to standard error by passing `err=True`:

```
click.echo('Hello World!', err=True)
```

1.13.2 ANSI Colors

2.0 .

Starting with Click 2.0, the *echo()* function gained extra functionality to deal with ANSI colors and styles. Note that on Windows, this functionality is only available if colorama is installed. If it is installed, then ANSI codes are intelligently handled.

Primarily this means that:

- Click's *echo()* function will automatically strip ANSI color codes if the stream is not connected to a terminal.
- the *echo()* function will transparently connect to the terminal on Windows and translate ANSI codes to terminal API calls. This means that colors will work on Windows the same way they do on other operating systems.

Note for *colorama* support: Click will automatically detect when *colorama* is available and use it. Do *not* call `colorama.init()`!

To install *colorama*, run this command:

```
$ pip install colorama
```

For styling a string, the *style()* function can be used:

```
import click

click.echo(click.style('Hello World!', fg='green'))
click.echo(click.style('Some more text', bg='blue', fg='white'))
click.echo(click.style('ATTENTION', blink=True, bold=True))
```

The combination of *echo()* and *style()* is also available in a single function called *secho()*:

```
click.secho('Hello World!', fg='green')
click.secho('Some more text', bg='blue', fg='white')
click.secho('ATTENTION', blink=True, bold=True)
```

1.13.3 Pager Support

In some situations, you might want to show long texts on the terminal and let a user scroll through it. This can be achieved by using the *echo_via_pager()* function which works similarly to the *echo()* function, but always writes to stdout and, if possible, through a pager.

Example:

```
@click.command()
def less():
    click.echo_via_pager('\n'.join('Line %d' % idx
                                   for idx in range(200)))
```

1.13.4 Screen Clearing

2.0 .

To clear the terminal screen, you can use the *clear()* function that is provided starting with Click 2.0. It does what the name suggests: it clears the entire visible screen in a platform-agnostic way:

```
import click
click.clear()
```

1.13.5 Getting Characters from Terminal

2.0 .

Normally, when reading input from the terminal, you would read from standard input. However, this is buffered input and will not show up until the line has been terminated. In certain circumstances, you might not want to do that and instead read individual characters as they are being written.

For this, Click provides the *getchar()* function which reads a single character from the terminal buffer and returns it as a Unicode character.

Note that this function will always read from the terminal, even if stdin is instead a pipe.

Example:

```
import click

click.echo('Continue? [yn] ', nl=False)
c = click.getchar()
click.echo()
if c == 'y':
    click.echo('We will go on')
elif c == 'n':
    click.echo('Abort!')
else:
    click.echo('Invalid input :(')
```

Note that this reads raw input, which means that things like arrow keys will show up in the platform's native escape format. The only characters translated are `^C` and `^D` which are converted into keyboard interrupts and end of file exceptions respectively. This is done because otherwise, it's too easy to forget about that and to create scripts that cannot be properly exited.

1.13.6 Waiting for Key Press

2.0 .

Sometimes, it's useful to pause until the user presses any key on the keyboard. This is especially useful on Windows where `cmd.exe` will close the window at the end of the command execution by default, instead of waiting.

In click, this can be accomplished with the *pause()* function. This function will print a quick message to the terminal (which can be customized) and wait for the user to press a key. In addition to that, it will also become a NOP (no operation instruction) if the script is not run interactively.

Example:

```
import click
click.pause()
```

1.13.7 Launching Editors

2.0 .

Click supports launching editors automatically through *edit()*. This is very useful for asking users for multi-line input. It will automatically open the user's defined editor or fall back to a sensible default. If the user closes the editor without saving, the return value will be *None* otherwise the entered text.

Example usage:

```
import click

def get_commit_message():
    MARKER = '# Everything below is ignored\n'
    message = click.edit('\n\n' + MARKER)
    if message is not None:
        return message.split(MARKER, 1)[0].rstrip('\n')
```

Alternatively, the function can also be used to launch editors for files by a specific filename. In this case, the return value is always *None*.

Example usage:

```
import click
click.edit(filename='/etc/passwd')
```

1.13.8 Launching Applications

2.0 .

Click supports launching applications through *launch()*. This can be used to open the default application assocated with a URL or filetype. This can be used to launch web browsers or picture viewers, for instance. In addition to this, it can also launch the file manager and automatically select the provided file.

Example usage:

```
click.launch('http://click.pocoo.org/')
click.launch('/my/downloaded/file.txt', locate=True)
```

1.13.9 Printing Filenames

Because filenames might not be Unicode, formatting them can be a bit tricky. Generally, this is easier in Python 2 than on 3, as you can just write the bytes to stdout with the `print` function, but in Python 3, you will always need to operate in Unicode.

The way this works with click is through the *format_filename()* function. It does a best-effort conversion of the filename to Unicode and will never fail. This makes it possible to use these filenames in the context of a full Unicode string.

Example:

```
click.echo('Path: %s' % click.format_filename(b'foo.txt'))
```

1.13.10 Standard Streams

For command line utilities, it's very important to get access to input and output streams reliably. Python generally provides access to these streams through `sys.stdout` and friends, but unfortunately, there are API differences between 2.x and 3.x, especially with regards to how these streams respond to Unicode and binary data.

Because of this, click provides the *get_binary_stream()* and *get_text_stream()* functions, which produce consistent results with different Python versions and for a wide variety pf terminal configurations.

The end result is that these functions will always return a functional stream object (except in very odd cases in Python 3; see *Python 3 Limitations*).

Example:

```
import click

stdin_text = click.get_text_stream('stdin')
stdout_binary = click.get_binary_stream('stdout')
```

6.0 .

Click now emulates output streams on Windows to support unicode to the Windows console through separate APIs. For more information see **'wincmd'_**.

1.13.11 Intelligent File Opening

3.0 .

Starting with Click 3.0 the logic for opening files from the *File* type is exposed through the *open_file()* function. It can intelligently open stdin/stdout as well as any other file.

Example:

```
import click

stdout = click.open_file('-', 'w')
test_file = click.open_file('test.txt', 'w')
```

If stdin or stdout are returned, the return value is wrapped in a special file where the context manager will prevent the closing of the file. This makes the handling of standard streams transparent and you can always use it like this:

```
with click.open_file(filename, 'w') as f:
    f.write('Hello World!\n')
```

1.13.12 Finding Application Folders

2.0 .

Very often, you want to open a configuration file that belongs to your application. However, different operating systems store these configuration files in different locations depending on their standards. Click provides a *get_app_dir()* function which returns the most appropriate location for per-user config files for your application depending on the OS.

Example usage:

```
import os
import click
import ConfigParser

APP_NAME = 'My Application'

def read_config():
    cfg = os.path.join(click.get_app_dir(APP_NAME), 'config.ini')
    parser = ConfigParser.RawConfigParser()
    parser.read([cfg])
    rv = {}
    for section in parser.sections():
        for key, value in parser.items(section):
            rv['%s.%s' % (section, key)] = value
    return rv
```

1.13.13 Showing Progress Bars

2.0 .

Sometimes, you have command line scripts that need to process a lot of data, but you want to quickly show the user some progress about how long that will take. Click supports simple progress bar rendering for that through the *progressbar()* function.

The basic usage is very simple: the idea is that you have an iterable that you want to operate on. For each item in the iterable it might take some time to do processing. So say you have a loop like this:

```
for user in all_the_users_to_process:
    modify_the_user(user)
```

To hook this up with an automatically updating progress bar, all you need to do is to change the code to this:

```
import click

with click.progressbar(all_the_users_to_process) as bar:
    for user in bar:
        modify_the_user(user)
```

Click will then automatically print a progress bar to the terminal and calculate the remaining time for you. The calculation of remaining time requires that the iterable has a length. If it does not have a length but you know the length, you can explicitly provide it:

```
with click.progressbar(all_the_users_to_process,
                       length=number_of_users) as bar:
    for user in bar:
        modify_the_user(user)
```

Another useful feature is to associate a label with the progress bar which will be shown preceding the progress bar:

```
with click.progressbar(all_the_users_to_process,
                       label='Modifying user accounts',
                       length=number_of_users) as bar:
    for user in bar:
        modify_the_user(user)
```

Sometimes, one may need to iterate over an external iterator, and advance the progress bar irregularly. To do so, you need to specify the length (and no iterable), and use the update method on the context return value instead of iterating directly over it:

```
with click.progressbar(length=total_size,
                       label='Unzipping archive') as bar:
    for archive in zip_file:
        archive.extract()
        bar.update(archive.size)
```

1.14 Bash Complete

2.0 .

As of Click 2.0, there is built-in support for Bash completion for any Click script. There are certain restrictions on when this completion is available, but for the most part it should just work.

1.14.1 Limitations

Bash completion is only available if a script has been installed properly, and not executed through the `python` command. For information about how to do that, see *Setuptools Integration*. Also, Click currently only supports completion for Bash.

Currently, Bash completion is an internal feature that is not customizable. This might be relaxed in future versions.

1.14.2 What it Completes

Generally, the Bash completion support will complete subcommands and parameters. Subcommands are always listed whereas parameters only if at least a dash has been provided. Example:

```
$ repo <TAB><TAB>
clone    commit   copy     delete   setuser
$ repo clone -<TAB><TAB>
--deep     --help     --rev      --shallow  -r
```

1.14.3 Activation

In order to activate Bash completion, you need to inform Bash that completion is available for your script, and how. Any Click application automatically provides support for that. The general way this works is through a magic environment variable called `_<PROG_NAME>_COMPLETE`, where `<PROG_NAME>` is your application executable name in uppercase with dashes replaced by underscores.

If your tool is called `foo-bar`, then the magic variable is called `_FOO_BAR_COMPLETE`. By exporting it with the `source` value it will spit out the activation script which can be trivally activated.

For instance, to enable Bash completion for your `foo-bar` script, this is what you would need to put into your `.bashrc`:

```
eval "$(_FOO_BAR_COMPLETE=source foo-bar)"
```

From this point onwards, your script will have Bash completion enabled.

1.14.4 Activation Script

The above activation example will always invoke your application on startup. This might be slowing down the shell activation time significantly if you have many applications. Alternatively, you could also ship a file with the contents of that, which is what Git and other systems are doing.

This can be easily accomplished:

```
_FOO_BAR_COMPLETE=source foo-bar > foo-bar-complete.sh
```

And then you would put this into your bashrc instead:

```
. /path/to/foo-bar-complete.sh
```

1.15 Exception Handling

Click internally uses exceptions to signal various error conditions that the user of the application might have caused. Primarily this is things like incorrect usage.

1.15.1 Where are Errors Handled?

Click's main error handling is happening in *`BaseCommand.main()`*. In there it handles all subclasses of *`ClickException`* as well as the standard `EOFError` and `KeyboardInterrupt` exceptions. The latter are internally translated into a *`Abort`*.

The logic applied is the following:

1. If an `EOFError` or `KeyboardInterrupt` happens, reraise it as *`Abort`*.
2. If an *`ClickException`* is raised, invoke the `ClickException.show()` method on it to display it and then exit the program with `ClickException.exit_code`.
3. If an *`Abort`* exception is raised print the string `Aborted!` to standard error and exit the program with exit code `1`.
4. if it goes through well, exit the program with exit code `0`.

1.15.2 What if I don't want that?

Generally you always have the option to invoke the `invoke()` method yourself. For instance if you have a *`Command`* you can invoke it manually like this:

```
ctx = command.make_context('command-name', ['args', 'go', 'here'])
with ctx:
    result = command.invoke(ctx)
```

In this case exceptions will not be handled at all and bubbled up as you would expect.

Starting with Click 3.0 you can also use the `Command.main()` method but disable the standalone mode which will do two things: disable exception handling and disable the implicit `sys.exit()` at the end.

So you can do something like this:

```
command.main(['command-name', 'args', 'go', 'here'],
             standalone_mode=False)
```

1.15.3 Which Exceptions Exist?

Click has two exception bases: *ClickException* which is raised for all exceptions that Click wants to signal to the user and *Abort* which is used to instruct Click to abort the execution.

A *ClickException* has a `show()` method which can render an error message to stderr or the given file object. If you want to use the exception yourself for doing something check the API docs about what else they provide.

The following common subclasses exist:

- *UsageError* to inform the user that something went wrong.
- *BadParameter* to inform the user that something went wrong with a specific parameter. These are often handled internally in Click and augmented with extra information if possible. For instance if those are raised from a callback Click will automatically augment it with the parameter name if possible.
- *FileError* this is an error that is raised by the `FileType` if Click encounters issues opening the file.

1.16 Python 3 Support

Click supports Python 3, but like all other command line utility libraries, it suffers from the Unicode text model in Python 3. All examples in the documentation were written so that they could run on both Python 2.x and Python 3.3 or higher.

At the moment, it is strongly recommended is to use Python 2 for Click utilities unless Python 3 is a hard requirement.

1.16.1 Python 3 Limitations

At the moment, Click suffers from a few problems with Python 3:

- The command line in Unix traditionally is in bytes, not Unicode. While there are encoding hints for all of this, there are generally some situations where this can break. The most common one is SSH connections to machines with different locales.

  Misconfigured environments can currently cause a wide range of Unicode problems in Python 3 due to the lack of support for roundtripping surrogate escapes. This will not be fixed in Click itself!

  For more information see *Python 3 Surrogate Handling*.

- Standard input and output in Python 3 is opened in Unicode mode by default. Click has to reopen the stream in binary mode in certain situations. Because there is no standardized way to do this, this might not always work. Primarily this can become a problem when testing command-line applications.

  This is not supported:

  ```
  sys.stdin = io.StringIO('Input here')
  sys.stdout = io.StringIO()
  ```

Instead you need to do this:

```
input = 'Input here'
in_stream = io.BytesIO(input.encode('utf-8'))
sys.stdin = io.TextIOWrapper(in_stream, encoding='utf-8')
out_stream = io.BytesIO()
sys.stdout = io.TextIOWrapper(out_stream, encoding='utf-8')
```

Remember that in that case, you need to use `out_stream.getvalue()` and not `sys.stdout.getvalue()` if you want to access the buffer contents as the wrapper will not forward that method.

1.16.2 Python 2 and 3 Differences

Click attempts to minimize the differences between Python 2 and Python 3 by following best practices for both languages.

In Python 2, the following is true:

- `sys.stdin`, `sys.stdout`, and `sys.stderr` are opened in binary mode, but under some circumstances they support Unicode output. Click attempts to not subvert this but provides support for forcing streams to be Unicode-based.
- `sys.argv` is always byte-based. Click will pass bytes to all input types and convert as necessary. The *`STRING`* type automatically will decode properly the input value into a string by trying the most appropriate encodings.
- When dealing with files, Click will never go through the Unicode APIs and will instead use the operating system's byte APIs to open the files.

In Python 3, the following is true:

- `sys.stdin`, `sys.stdout` and `sys.stderr` are by default text-based. When Click needs a binary stream, it attempts to discover the underlying binary stream. See *Python 3 Limitations* for how this works.
- `sys.argv` is always Unicode-based. This also means that the native type for input values to the types in Click is Unicode, and not bytes.

  This causes problems if the terminal is incorrectly set and Python does not figure out the encoding. In that case, the Unicode string will contain error bytes encoded as surrogate escapes.
- When dealing with files, Click will always use the Unicode file system API calls by using the operating system's reported or guessed filesystem encoding. Surrogates are supported for filenames, so it should be possible to open files through the *`File`* type even if the environment is misconfigured.

1.16.3 Python 3 Surrogate Handling

Click in Python 3 does all the Unicode handling in the standard library and is subject to its behavior. In Python 2, Click does all the Unicode handling itself, which means there are differences in error behavior.

The most glaring difference is that in Python 2, Unicode will "just work", while in Python 3, it requires extra care. The reason for this is that in Python 3, the encoding detection is done in the interpreter, and on Linux and certain other operating systems, its encoding handling is problematic.

The biggest source of frustration is that Click scripts invoked by init systems (sysvinit, upstart, systemd, etc.), deployment tools (salt, puppet), or cron jobs (cron) will refuse to work unless a Unicode locale is exported.

If Click encounters such an environment it will prevent further execution to force you to set a locale. This is done because Click cannot know about the state of the system once it's invoked and restore the values before Python's Unicode handling kicked in.

If you see something like this error in Python 3:

```
Traceback (most recent call last):
  ...
RuntimeError: Click will abort further execution because Python 3 was
  configured to use ASCII as encoding for the environment. Either switch
  to Python 2 or consult http://click.pocoo.org/python3/ for
  mitigation steps.
```

You are dealing with an environment where Python 3 thinks you are restricted to ASCII data. The solution to these problems is different depending on which locale your computer is running in.

For instance, if you have a German Linux machine, you can fix the problem by exporting the locale to `de_DE.utf-8`:

```
export LC_ALL=de_DE.utf-8
export LANG=de_DE.utf-8
```

If you are on a US machine, `en_US.utf-8` is the encoding of choice. On some newer Linux systems, you could also try `C.UTF-8` as the locale:

```
export LC_ALL=C.UTF-8
export LANG=C.UTF-8
```

On some systems it was reported that *UTF-8* has to be written as *UTF8* and vice versa. To see which locales are supported you can invoke `locale -a`:

```
locale -a
```

You need to do this before you invoke your Python script. If you are curious about the reasons for this, you can join the discussions in the Python 3 bug tracker:

- ASCII is a bad filesystem default encoding
- Use surrogateescape as default error handler
- Python 3 raises Unicode errors in the C locale
- LC_CTYPE=C: pydoc leaves terminal in an unusable state (this is relevant to Click because the pager support is provided by the stdlib pydoc module)

1.16.4 Unicode Literals

Starting with Click 5.0 there will be a warning for the use of the `unicode_literals` future import in Python 2. This has been done due to the negative consequences of this import with regards to unintentionally causing bugs due to introducing Unicode data to APIs that are incapable of handling them. For some examples of this issue, see the discussion on this github issue: python-future#22.

If you use `unicode_literals` in any file that defines a Click command or that invokes a click command you will be given a warning. You are strongly encouraged to not use `unicode_literals` and instead use explicit `u` prefixes for your Unicode strings.

If you do want to ignore the warning and continue using `unicode_literals` on your own peril, you can disable the warning as follows:

```
import click
click.disable_unicode_literals_warning = True
```

1.17 Windows Console Notes

6.0 .

Until Click 6.0 there are various bugs and limitations with using Click on a Windows console. Most notably the decoding of command line arguments was performed with the wrong encoding on Python 2 and on all versions of Python output of unicode characters was impossible. Starting with Click 6.0 we now emulate output streams on Windows to support unicode to the Windows console through separate APIs and we perform different decoding of parameters.

Here is a brief overview of how this works and what it means to you.

1.17.1 Unicode Arguments

Click internally is generally based on the concept that any argument can come in as either byte string or unicode string and conversion is performed to the type expected value as late as possible. This has some advantages as it allows us to accept the data in the most appropriate form for the operating system and Python version.

For instance paths are left as bytes on Python 2 unless you explicitly tell it otherwise.

This caused some problems on Windows where initially the wrong encoding was used and garbage ended up in your input data. We not only fixed the encoding part, but we also now extract unicode parameters from *sys.argv*.

This means that on Python 2 under Windows, the arguments processed will *most likely* be of unicode nature and not bytes. This was something that previously did not really happen unless you explicitly passed in unicode parameters so your custom types need to be aware of this.

There is also another limitation with this: if *sys.argv* was modified prior to invoking a click handler, we have to fall back to the regular byte input in which case not all unicode values are available but only a subset of the codepage used for parameters.

1.17.2 Unicode Output and Input

Unicode output and input on Windows is implemented through the concept of a dispatching text stream. What this means is that when click first needs a text output (or input) stream on windows it goes through a few checks to figure out of a windows console is connected or not. If no Windows console is present then the text output stream is returned as such and the encoding for that stream is set to `utf-8` like on all platforms.

However if a console is connected the stream will instead be emulated and use the cmd.exe unicode APIs to output text information. In this case the stream will also use `utf-16-le` as internal encoding. However there is some hackery going on that the underlying raw IO buffer is still bypassing the unicode APIs and byte output through an indirection is still possible.

This hackery is used on both Python 2 and Python 3 as neither version of Python has native support for cmd.exe with unicode characters. There are some limitations you need to be aware of:

- this unicode support is limited to `click.echo`, `click.prompt` as well as `click.get_text_stream`.
- depending on if unicode values or byte strings are passed the control flow goes completely different places internally which can have some odd artifacts if data partially ends up being buffered. Click attempts to protect against that by manually always flushing but if you are mixing and matching different string types to `stdout` or `stderr` you will need to manually flush.

Another important thing to note is that the Windows console's default fonts do not support a lot of characters which means that you are mostly limited to international letters but no emojis or special characters.

CHAPTER 2

API Reference

If you are looking for information on a specific function, class, or method, this part of the documentation is for you.

2.1 API

This part of the documentation lists the full API reference of all public classes and functions.

2.1.1 Decorators

`click.`**`command`**(*name=None*, *cls=None*, ***attrs*)

Creates a new *Command* and uses the decorated function as callback. This will also automatically attach all decorated *option()*s and *argument()*s as parameters to the command.

The name of the command defaults to the name of the function. If you want to change that, you can pass the intended name as the first argument.

All keyword arguments are forwarded to the underlying command class.

Once decorated the function turns into a *Command* instance that can be invoked as a command line utility or be attached to a command *Group*.

- **`name`** – the name of the command. This defaults to the function name.
- **`cls`** – the command class to instantiate. This defaults to *Command*.

`click.`**`group`**(*name=None*, ***attrs*)

Creates a new *Group* with a function as callback. This works otherwise the same as *command()* just that the *cls* parameter is set to *Group*.

`click.`**`argument`**(**param_decls*, ***attrs*)

Attaches an argument to the command. All positional arguments are passed as parameter declarations to *Argument*; all keyword arguments are forwarded unchanged (except `cls`). This is equivalent to creating an *Argument* instance manually and attaching it to the *Command.params* list.

cls – the argument class to instantiate. This defaults to *Argument*.

click.**option**(**param_decls*, ***attrs*)

Attaches an option to the command. All positional arguments are passed as parameter declarations to *Option*; all keyword arguments are forwarded unchanged (except `cls`). This is equivalent to creating an *Option* instance manually and attaching it to the *Command.params* list.

cls – the option class to instantiate. This defaults to *Option*.

click.**password_option**(**param_decls*, ***attrs*)

Shortcut for password prompts.

This is equivalent to decorating a function with *option()* with the following parameters:

```
@click.command()
@click.option('--password', prompt=True, confirmation_prompt=True,
              hide_input=True)
def changeadmin(password):
    pass
```

click.**confirmation_option**(**param_decls*, ***attrs*)

Shortcut for confirmation prompts that can be ignored by passing `--yes` as parameter.

This is equivalent to decorating a function with *option()* with the following parameters:

```
def callback(ctx, param, value):
    if not value:
        ctx.abort()

@click.command()
@click.option('--yes', is_flag=True, callback=callback,
              expose_value=False, prompt='Do you want to continue?')
def dropdb():
    pass
```

click.**version_option**(*version=None*, **param_decls*, ***attrs*)

Adds a `--version` option which immediately ends the program printing out the version number. This is implemented as an eager option that prints the version and exits the program in the callback.

- **version** – the version number to show. If not provided Click attempts an auto discovery via setuptools.
- **prog_name** – the name of the program (defaults to autodetection)
- **message** – custom message to show instead of the default (`'%(prog)s, version %(version)s'`)
- **others** – everything else is forwarded to *option()*.

click.**help_option**(**param_decls*, ***attrs*)

Adds a `--help` option which immediately ends the program printing out the help page. This is usually unnecessary to add as this is added by default to all commands unless suppressed.

Like *version_option()*, this is implemented as eager option that prints in the callback and exits.

All arguments are forwarded to *option()*.

click.**pass_context**(*f*)

Marks a callback as wanting to receive the current context object as first argument.

click.**pass_obj**(*f*)

Similar to *pass_context()*, but only pass the object on the context onwards (*Context.obj*). This is useful if that object represents the state of a nested system.

click.**make_pass_decorator**(*object_type*, *ensure=False*)

Given an object type this creates a decorator that will work similar to *pass_obj()* but instead of passing the object of the current context, it will find the innermost context of type `object_type()`.

This generates a decorator that works roughly like this:

```
from functools import update_wrapper

def decorator(f):
    @pass_context
    def new_func(ctx, *args, **kwargs):
        obj = ctx.find_object(object_type)
        return ctx.invoke(f, obj, *args, **kwargs)
    return update_wrapper(new_func, f)
return decorator
```

- **object_type** – the type of the object to pass.
- **ensure** – if set to *True*, a new object will be created and remembered on the context if it's not there yet.

2.1.2 Utilities

click.**echo**(*message=None*, *file=None*, *nl=True*, *err=False*, *color=None*)

Prints a message plus a newline to the given file or stdout. On first sight, this looks like the print function, but it has improved support for handling Unicode and binary data that does not fail no matter how badly configured the system is.

Primarily it means that you can print binary data as well as Unicode data on both 2.x and 3.x to the given file in the most appropriate way possible. This is a very carefree function as in that it will try its best to not fail. As of Click 6.0 this includes support for unicode output on the Windows console.

In addition to that, if colorama is installed, the echo function will also support clever handling of ANSI codes. Essentially it will then do the following:

- add transparent handling of ANSI color codes on Windows.
- hide ANSI codes automatically if the destination file is not a terminal.

6.0 : As of Click 6.0 the echo function will properly support unicode output on the windows console. Not that click does not modify the interpreter in any way which means that *sys.stdout* or the print statement or function will still not provide unicode support.

2.0 : Starting with version 2.0 of Click, the echo function will work with colorama if it's installed.

3.0 : The *err* parameter was added.

4.0 : Added the *color* flag.

- **message** – the message to print
- **file** – the file to write to (defaults to `stdout`)

- **err** – if set to true the file defaults to `stderr` instead of `stdout`. This is faster and easier than calling `get_text_stderr()` yourself.
- **nl** – if set to *True* (the default) a newline is printed afterwards.
- **color** – controls if the terminal supports ANSI colors or not. The default is autodetection.

`click.`**echo_via_pager**(*text*, *color=None*)

This function takes a text and shows it via an environment specific pager on stdout.

3.0 : Added the *color* flag.

- **text** – the text to page.
- **color** – controls if the pager supports ANSI colors or not. The default is autodetection.

`click.`**prompt**(*text*, *default=None*, *hide_input=False*, *confirmation_prompt=False*, *type=None*, *value_proc=None*, *prompt_suffix=': '*, *show_default=True*, *err=False*)

Prompts a user for input. This is a convenience function that can be used to prompt a user for input later.

If the user aborts the input by sending a interrupt signal, this function will catch it and raise a *Abort* exception.

6.0 : Added unicode support for cmd.exe on Windows.

4.0 : Added the *err* parameter.

- **text** – the text to show for the prompt.
- **default** – the default value to use if no input happens. If this is not given it will prompt until it's aborted.
- **hide_input** – if this is set to true then the input value will be hidden.
- **confirmation_prompt** – asks for confirmation for the value.
- **type** – the type to use to check the value against.
- **value_proc** – if this parameter is provided it's a function that is invoked instead of the type conversion to convert a value.
- **prompt_suffix** – a suffix that should be added to the prompt.
- **show_default** – shows or hides the default value in the prompt.
- **err** – if set to true the file defaults to `stderr` instead of `stdout`, the same as with echo.

`click.`**confirm**(*text*, *default=False*, *abort=False*, *prompt_suffix=': '*, *show_default=True*, *err=False*)

Prompts for confirmation (yes/no question).

If the user aborts the input by sending a interrupt signal this function will catch it and raise a *Abort* exception.

4.0 : Added the *err* parameter.

- **text** – the question to ask.
- **default** – the default for the prompt.
- **abort** – if this is set to *True* a negative answer aborts the exception by raising *Abort*.
- **prompt_suffix** – a suffix that should be added to the prompt.
- **show_default** – shows or hides the default value in the prompt.
- **err** – if set to true the file defaults to `stderr` instead of `stdout`, the same as with echo.

click.**progressbar**(*iterable=None*, *length=None*, *label=None*, *show_eta=True*, *show_percent=None*, *show_pos=False*, *item_show_func=None*, *fill_char='#'*, *empty_char='-'*, *bar_template='%(label)s [%(bar)s] %(info)s'*, *info_sep=' '*, *width=36*, *file=None*, *color=None*)

This function creates an iterable context manager that can be used to iterate over something while showing a progress bar. It will either iterate over the *iterable* or *length* items (that are counted up). While iteration happens, this function will print a rendered progress bar to the given *file* (defaults to stdout) and will attempt to calculate remaining time and more. By default, this progress bar will not be rendered if the file is not a terminal.

The context manager creates the progress bar. When the context manager is entered the progress bar is already displayed. With every iteration over the progress bar, the iterable passed to the bar is advanced and the bar is updated. When the context manager exits, a newline is printed and the progress bar is finalized on screen.

No printing must happen or the progress bar will be unintentionally destroyed.

Example usage:

```
with progressbar(items) as bar:
    for item in bar:
        do_something_with(item)
```

Alternatively, if no iterable is specified, one can manually update the progress bar through the *update()* method instead of directly iterating over the progress bar. The update method accepts the number of steps to increment the bar with:

```
with progressbar(length=chunks.total_bytes) as bar:
    for chunk in chunks:
        process_chunk(chunk)
        bar.update(chunks.bytes)
```

2.0 .

4.0 : Added the *color* parameter. Added a *update* method to the progressbar object.

- **iterable** – an iterable to iterate over. If not provided the length is required.
- **length** – the number of items to iterate over. By default the progressbar will attempt to ask the iterator about its length, which might or might not work. If an iterable is also provided this parameter can be used to override the length. If an iterable is not provided the progress bar will iterate over a range of that length.
- **label** – the label to show next to the progress bar.
- **show_eta** – enables or disables the estimated time display. This is automatically disabled if the length cannot be determined.
- **show_percent** – enables or disables the percentage display. The default is *True* if the iterable has a length or *False* if not.
- **show_pos** – enables or disables the absolute position display. The default is *False*.
- **item_show_func** – a function called with the current item which can return a string to show the current item next to the progress bar. Note that the current item can be *None*!
- **fill_char** – the character to use to show the filled part of the progress bar.
- **empty_char** – the character to use to show the non-filled part of the progress bar.
- **bar_template** – the format string to use as template for the bar. The parameters in it are `label` for the label, `bar` for the progress bar and `info` for the info section.

- **info_sep** – the separator between multiple info items (eta etc.)
- **width** – the width of the progress bar in characters, 0 means full terminal width
- **file** – the file to write to. If this is not a terminal then only the label is printed.
- **color** – controls if the terminal supports ANSI colors or not. The default is autodetection. This is only needed if ANSI codes are included anywhere in the progress bar output which is not the case by default.

click.**clear**()

Clears the terminal screen. This will have the effect of clearing the whole visible space of the terminal and moving the cursor to the top left. This does not do anything if not connected to a terminal.

2.0 .

click.**style**(*text*, *fg=None*, *bg=None*, *bold=None*, *dim=None*, *underline=None*, *blink=None*, *reverse=None*, *reset=True*)

Styles a text with ANSI styles and returns the new string. By default the styling is self contained which means that at the end of the string a reset code is issued. This can be prevented by passing `reset=False`.

Examples:

```
click.echo(click.style('Hello World!', fg='green'))
click.echo(click.style('ATTENTION!', blink=True))
click.echo(click.style('Some things', reverse=True, fg='cyan'))
```

Supported color names:

- `black` (might be a gray)
- `red`
- `green`
- `yellow` (might be an orange)
- `blue`
- `magenta`
- `cyan`
- `white` (might be light gray)
- `reset` (reset the color code only)

2.0 .

- **text** – the string to style with ansi codes.
- **fg** – if provided this will become the foreground color.
- **bg** – if provided this will become the background color.
- **bold** – if provided this will enable or disable bold mode.
- **dim** – if provided this will enable or disable dim mode. This is badly supported.
- **underline** – if provided this will enable or disable underline.
- **blink** – if provided this will enable or disable blinking.
- **reverse** – if provided this will enable or disable inverse rendering (foreground becomes background and the other way round).

- **reset** – by default a reset-all code is added at the end of the string which means that styles do not carry over. This can be disabled to compose styles.

click.**unstyle**(*text*)
Removes ANSI styling information from a string. Usually it's not necessary to use this function as Click's echo function will automatically remove styling if necessary.

2.0 .

text – the text to remove style information from.

click.**secho**(*text*, *file=None*, *nl=True*, *err=False*, *color=None*, ***styles*)
This function combines *echo()* and *style()* into one call. As such the following two calls are the same:

```
click.secho('Hello World!', fg='green')
click.echo(click.style('Hello World!', fg='green'))
```

All keyword arguments are forwarded to the underlying functions depending on which one they go with.

2.0 .

click.**edit**(*text=None*, *editor=None*, *env=None*, *require_save=True*, *extension='.txt'*, *filename=None*)
Edits the given text in the defined editor. If an editor is given (should be the full path to the executable but the regular operating system search path is used for finding the executable) it overrides the detected editor. Optionally, some environment variables can be used. If the editor is closed without changes, *None* is returned. In case a file is edited directly the return value is always *None* and *require_save* and *extension* are ignored.

If the editor cannot be opened a *UsageError* is raised.

Note for Windows: to simplify cross-platform usage, the newlines are automatically converted from POSIX to Windows and vice versa. As such, the message here will have `\n` as newline markers.

- **text** – the text to edit.
- **editor** – optionally the editor to use. Defaults to automatic detection.
- **env** – environment variables to forward to the editor.
- **require_save** – if this is true, then not saving in the editor will make the return value become *None*.
- **extension** – the extension to tell the editor about. This defaults to *.txt* but changing this might change syntax highlighting.
- **filename** – if provided it will edit this file instead of the provided text contents. It will not use a temporary file as an indirection in that case.

click.**launch**(*url*, *wait=False*, *locate=False*)
This function launches the given URL (or filename) in the default viewer application for this file type. If this is an executable, it might launch the executable in a new session. The return value is the exit code of the launched application. Usually, `0` indicates success.

Examples:

```
click.launch('http://click.pocoo.org/')
click.launch('/my/downloaded/file', locate=True)
```

2.0 .

- **url** – URL or filename of the thing to launch.

- **wait** – waits for the program to stop.
- **locate** – if this is set to *True* then instead of launching the application associated with the URL it will attempt to launch a file manager with the file located. This might have weird effects if the URL does not point to the filesystem.

click.**getchar**(*echo=False*)

Fetches a single character from the terminal and returns it. This will always return a unicode character and under certain rare circumstances this might return more than one character. The situations which more than one character is returned is when for whatever reason multiple characters end up in the terminal buffer or standard input was not actually a terminal.

Note that this will always read from the terminal, even if something is piped into the standard input.

2.0 .

echo – if set to *True*, the character read will also show up on the terminal. The default is to not show it.

click.**pause**(*info='Press any key to continue ...'*, *err=False*)

This command stops execution and waits for the user to press any key to continue. This is similar to the Windows batch "pause" command. If the program is not run through a terminal, this command will instead do nothing.

2.0 .

4.0 : Added the *err* parameter.

- **info** – the info string to print before pausing.
- **err** – if set to message goes to `stderr` instead of `stdout`, the same as with echo.

click.**get_terminal_size**()

Returns the current size of the terminal as tuple in the form `(width, height)` in columns and rows.

click.**get_binary_stream**(*name*)

Returns a system stream for byte processing. This essentially returns the stream from the sys module with the given name but it solves some compatibility issues between different Python versions. Primarily this function is necessary for getting binary streams on Python 3.

name – the name of the stream to open. Valid names are `'stdin'`, `'stdout'` and `'stderr'`

click.**get_text_stream**(*name*, *encoding=None*, *errors='strict'*)

Returns a system stream for text processing. This usually returns a wrapped stream around a binary stream returned from *get_binary_stream()* but it also can take shortcuts on Python 3 for already correctly configured streams.

- **name** – the name of the stream to open. Valid names are `'stdin'`, `'stdout'` and `'stderr'`
- **encoding** – overrides the detected default encoding.
- **errors** – overrides the default error mode.

click.**open_file**(*filename*, *mode='r'*, *encoding=None*, *errors='strict'*, *lazy=False*, *atomic=False*)

This is similar to how the *File* works but for manual usage. Files are opened non lazy by default. This can open regular files as well as stdin/stdout if `'-'` is passed.

If stdin/stdout is returned the stream is wrapped so that the context manager will not close the stream accidentally. This makes it possible to always use the function like this without having to worry to accidentally close a standard stream:

```
with open_file(filename) as f:
    ...
```

3.0 .

- **filename** – the name of the file to open (or `'-'` for stdin/stdout).
- **mode** – the mode in which to open the file.
- **encoding** – the encoding to use.
- **errors** – the error handling for this file.
- **lazy** – can be flipped to true to open the file lazily.
- **atomic** – in atomic mode writes go into a temporary file and it's moved on close.

`click.`**get_app_dir**(*app_name*, *roaming=True*, *force_posix=False*)

Returns the config folder for the application. The default behavior is to return whatever is most appropriate for the operating system.

To give you an idea, for an app called `"Foo Bar"`, something like the following folders could be returned:

Mac OS X: `~/Library/Application Support/Foo Bar`

Mac OS X (POSIX): `~/.foo-bar`

Unix: `~/.config/foo-bar`

Unix (POSIX): `~/.foo-bar`

Win XP (roaming): `C:\Documents and Settings\<user>\Local Settings\Application Data\Foo Bar`

Win XP (not roaming): `C:\Documents and Settings\<user>\Application Data\Foo Bar`

Win 7 (roaming): `C:\Users\<user>\AppData\Roaming\Foo Bar`

Win 7 (not roaming): `C:\Users\<user>\AppData\Local\Foo Bar`

2.0 .

- **app_name** – the application name. This should be properly capitalized and can contain whitespace.
- **roaming** – controls if the folder should be roaming or not on Windows. Has no affect otherwise.
- **force_posix** – if this is set to *True* then on any POSIX system the folder will be stored in the home folder with a leading dot instead of the XDG config home or darwin's application support folder.

`click.`**format_filename**(*filename*, *shorten=False*)

Formats a filename for user display. The main purpose of this function is to ensure that the filename can be displayed at all. This will decode the filename to unicode if necessary in a way that it will not fail. Optionally, it can shorten the filename to not include the full path to the filename.

- **filename** – formats a filename for UI display. This will also convert the filename into unicode without failing.

- **shorten** – this optionally shortens the filename to strip of the path that leads up to it.

2.1.3 Commands

class click.**BaseCommand**(*name*, *context_settings=None*)

The base command implements the minimal API contract of commands. Most code will never use this as it does not implement a lot of useful functionality but it can act as the direct subclass of alternative parsing methods that do not depend on the Click parser.

For instance, this can be used to bridge Click and other systems like argparse or docopt.

Because base commands do not implement a lot of the API that other parts of Click take for granted, they are not supported for all operations. For instance, they cannot be used with the decorators usually and they have no built-in callback system.

2.0 : Added the *context_settings* parameter.

- **name** – the name of the command to use unless a group overrides it.
- **context_settings** – an optional dictionary with defaults that are passed to the context object.

allow_extra_args = False

the default for the *Context.allow_extra_args* flag.

allow_interspersed_args = True

the default for the *Context.allow_interspersed_args* flag.

context_settings = None

an optional dictionary with defaults passed to the context.

ignore_unknown_options = False

the default for the *Context.ignore_unknown_options* flag.

invoke(*ctx*)

Given a context, this invokes the command. The default implementation is raising a not implemented error.

main(*args=None*, *prog_name=None*, *complete_var=None*, *standalone_mode=True*, ***extra*)

This is the way to invoke a script with all the bells and whistles as a command line application. This will always terminate the application after a call. If this is not wanted, `SystemExit` needs to be caught.

This method is also available by directly calling the instance of a *Command*.

3.0 : Added the *standalone_mode* flag to control the standalone mode.

- **args** – the arguments that should be used for parsing. If not provided, `sys.argv[1:]` is used.
- **prog_name** – the program name that should be used. By default the program name is constructed by taking the file name from `sys.argv[0]`.
- **complete_var** – the environment variable that controls the bash completion support. The default is `"_<prog_name>_COMPLETE"` with prog name in uppercase.
- **standalone_mode** – the default behavior is to invoke the script in standalone mode. Click will then handle exceptions and convert them into error messages and the function will never return but shut down the interpreter. If this is set to *False* they will be propagated to the caller and the return value of this function is the return value of *invoke()*.

- **extra** – extra keyword arguments are forwarded to the context constructor. See *Context* for more information.

make_context(*info_name*, *args*, *parent=None*, ***extra*)
This function when given an info name and arguments will kick off the parsing and create a new *Context*. It does not invoke the actual command callback though.

- **info_name** – the info name for this invokation. Generally this is the most descriptive name for the script or command. For the toplevel script it's usually the name of the script, for commands below it it's the name of the script.
- **args** – the arguments to parse as list of strings.
- **parent** – the parent context if available.
- **extra** – extra keyword arguments forwarded to the context constructor.

name = None
the name the command thinks it has. Upon registering a command on a *Group* the group will default the command name with this information. You should instead use the *Context*'s *info_name* attribute.

parse_args(*ctx*, *args*)
Given a context and a list of arguments this creates the parser and parses the arguments, then modifies the context as necessary. This is automatically invoked by *make_context()*.

class click.**Command**(*name*, *context_settings=None*, *callback=None*, *params=None*, *help=None*, *epilog=None*, *short_help=None*, *options_metavar='[OPTIONS]'*, *add_help_option=True*)
Commands are the basic building block of command line interfaces in Click. A basic command handles command line parsing and might dispatch more parsing to commands nested below it.

2.0 : Added the *context_settings* parameter.

- **name** – the name of the command to use unless a group overrides it.
- **context_settings** – an optional dictionary with defaults that are passed to the context object.
- **callback** – the callback to invoke. This is optional.
- **params** – the parameters to register with this command. This can be either *Option* or *Argument* objects.
- **help** – the help string to use for this command.
- **epilog** – like the help string but it's printed at the end of the help page after everything else.
- **short_help** – the short help to use for this command. This is shown on the command listing of the parent command.
- **add_help_option** – by default each command registers a `--help` option. This can be disabled by this parameter.

callback = None
the callback to execute when the command fires. This might be *None* in which case nothing happens.

collect_usage_pieces(*ctx*)
Returns all the pieces that go into the usage line and returns it as a list of strings.

format_epilog(*ctx*, *formatter*)
Writes the epilog into the formatter if it exists.

format_help(*ctx*, *formatter*)
Writes the help into the formatter if it exists.

This calls into the following methods:

- *format_usage()*
- *format_help_text()*
- *format_options()*
- *format_epilog()*

format_help_text(*ctx*, *formatter*)
Writes the help text to the formatter if it exists.

format_options(*ctx*, *formatter*)
Writes all the options into the formatter if they exist.

format_usage(*ctx*, *formatter*)
Writes the usage line into the formatter.

get_help(*ctx*)
Formats the help into a string and returns it. This creates a formatter and will call into the following formatting methods:

get_help_option(*ctx*)
Returns the help option object.

get_help_option_names(*ctx*)
Returns the names for the help option.

invoke(*ctx*)
Given a context, this invokes the attached callback (if it exists) in the right way.

make_parser(*ctx*)
Creates the underlying option parser for this command.

params = None
the list of parameters for this command in the order they should show up in the help page and execute. Eager parameters will automatically be handled before non eager ones.

class click.**MultiCommand**(*name=None*, *invoke_without_command=False*, *no_args_is_help=None*, *subcommand_metavar=None*, *chain=False*, *result_callback=None*, ***attrs*)
A multi command is the basic implementation of a command that dispatches to subcommands. The most common version is the *Group*.

- **invoke_without_command** – this controls how the multi command itself is invoked. By default it's only invoked if a subcommand is provided.
- **no_args_is_help** – this controls what happens if no arguments are provided. This option is enabled by default if *invoke_without_command* is disabled or disabled if it's enabled. If enabled this will add `--help` as argument if no arguments are passed.
- **subcommand_metavar** – the string that is used in the documentation to indicate the subcommand place.

- **chain** – if this is set to *True* chaining of multiple subcommands is enabled. This restricts the form of commands in that they cannot have optional arguments but it allows multiple commands to be chained together.
- **result_callback** – the result callback to attach to this multi command.

format_commands(*ctx*, *formatter*)
Extra format methods for multi methods that adds all the commands after the options.

get_command(*ctx*, *cmd_name*)
Given a context and a command name, this returns a *Command* object if it exists or returns *None*.

list_commands(*ctx*)
Returns a list of subcommand names in the order they should appear.

result_callback = None
The result callback that is stored. This can be set or overridden with the *resultcallback()* decorator.

resultcallback(*replace=False*)
Adds a result callback to the chain command. By default if a result callback is already registered this will chain them but this can be disabled with the *replace* parameter. The result callback is invoked with the return value of the subcommand (or the list of return values from all subcommands if chaining is enabled) as well as the parameters as they would be passed to the main callback.

Example:

```
@click.group()
@click.option('-i', '--input', default=23)
def cli(input):
    return 42

@cli.resultcallback()
def process_result(result, input):
    return result + input
```

3.0 .

replace – if set to *True* an already existing result callback will be removed.

class click.**Group**(*name=None*, *commands=None*, ***attrs*)
A group allows a command to have subcommands attached. This is the most common way to implement nesting in Click.

commands – a dictionary of commands.

add_command(*cmd*, *name=None*)
Registers another *Command* with this group. If the name is not provided, the name of the command is used.

command(**args*, ***kwargs*)
A shortcut decorator for declaring and attaching a command to the group. This takes the same arguments as *command()* but immediately registers the created command with this instance by calling into *add_command()*.

commands = None
the registered subcommands by their exported names.

group(**args*, ***kwargs*)
A shortcut decorator for declaring and attaching a group to the group. This takes the same arguments as *group()* but immediately registers the created command with this instance by calling into *add_command()*.

class click.**CommandCollection**(*name=None*, *sources=None*, ***attrs*)

A command collection is a multi command that merges multiple multi commands together into one. This is a straightforward implementation that accepts a list of different multi commands as sources and provides all the commands for each of them.

add_source(*multi_cmd*)

Adds a new multi command to the chain dispatcher.

sources = None

The list of registered multi commands.

2.1.4 Parameters

class click.**Parameter**(*param_decls=None*, *type=None*, *required=False*, *default=None*, *callback=None*, *nargs=None*, *metavar=None*, *expose_value=True*, *is_eager=False*, *envvar=None*)

A parameter to a command comes in two versions: they are either *Option*s or *Argument*s. Other subclasses are currently not supported by design as some of the internals for parsing are intentionally not finalized.

Some settings are supported by both options and arguments.

2.0 : Changed signature for parameter callback to also be passed the parameter. In Click 2.0, the old callback format will still work, but it will raise a warning to give you change to migrate the code easier.

- **param_decls** – the parameter declarations for this option or argument. This is a list of flags or argument names.
- **type** – the type that should be used. Either a *ParamType* or a Python type. The later is converted into the former automatically if supported.
- **required** – controls if this is optional or not.
- **default** – the default value if omitted. This can also be a callable, in which case it's invoked when the default is needed without any arguments.
- **callback** – a callback that should be executed after the parameter was matched. This is called as `fn(ctx, param, value)` and needs to return the value. Before Click 2.0, the signature was `(ctx, value)`.
- **nargs** – the number of arguments to match. If not `1` the return value is a tuple instead of single value. The default for nargs is `1` (except if the type is a tuple, then it's the arity of the tuple).
- **metavar** – how the value is represented in the help page.
- **expose_value** – if this is *True* then the value is passed onwards to the command callback and stored on the context, otherwise it's skipped.
- **is_eager** – eager values are processed before non eager ones. This should not be set for arguments or it will inverse the order of processing.
- **envvar** – a string or list of strings that are environment variables that should be checked.

get_default(*ctx*)

Given a context variable this calculates the default value.

human_readable_name

Returns the human readable name of this parameter. This is the same as the name for options, but the metavar for arguments.

process_value(*ctx*, *value*)
Given a value and context this runs the logic to convert the value as necessary.

type_cast_value(*ctx*, *value*)
Given a value this runs it properly through the type system. This automatically handles things like *nargs* and *multiple* as well as composite types.

class click.**Option**(*param_decls=None*, *show_default=False*, *prompt=False*, *confirmation_prompt=False*, *hide_input=False*, *is_flag=None*, *flag_value=None*, *multiple=False*, *count=False*, *allow_from_autoenv=True*, *type=None*, *help=None*, ***attrs*)
Options are usually optional values on the command line and have some extra features that arguments don't have.

All other parameters are passed onwards to the parameter constructor.

- **show_default** – controls if the default value should be shown on the help page. Normally, defaults are not shown.
- **prompt** – if set to *True* or a non empty string then the user will be prompted for input if not set. If set to *True* the prompt will be the option name capitalized.
- **confirmation_prompt** – if set then the value will need to be confirmed if it was prompted for.
- **hide_input** – if this is *True* then the input on the prompt will be hidden from the user. This is useful for password input.
- **is_flag** – forces this option to act as a flag. The default is auto detection.
- **flag_value** – which value should be used for this flag if it's enabled. This is set to a boolean automatically if the option string contains a slash to mark two options.
- **multiple** – if this is set to *True* then the argument is accepted multiple times and recorded. This is similar to `nargs` in how it works but supports arbitrary number of arguments.
- **count** – this flag makes an option increment an integer.
- **allow_from_autoenv** – if this is enabled then the value of this parameter will be pulled from an environment variable in case a prefix is defined on the context.
- **help** – the help string.

class click.**Argument**(*param_decls*, *required=None*, ***attrs*)
Arguments are positional parameters to a command. They generally provide fewer features than options but can have infinite `nargs` and are required by default.

All parameters are passed onwards to the parameter constructor.

2.1.5 Context

class click.**Context**(*command*, *parent=None*, *info_name=None*, *obj=None*, *auto_envvar_prefix=None*, *default_map=None*, *terminal_width=None*, *max_content_width=None*, *resilient_parsing=False*, *allow_extra_args=None*, *allow_interspersed_args=None*, *ignore_unknown_options=None*, *help_option_names=None*, *token_normalize_func=None*, *color=None*)
The context is a special internal object that holds state relevant for the script execution at every single level. It's normally invisible to commands unless they opt-in to getting access to it.

The context is useful as it can pass internal objects around and can control special execution features such as reading data from environment variables.

A context can be used as context manager in which case it will call `close()` on teardown.

2.0 : Added the *resilient_parsing*, *help_option_names*, *token_normalize_func* parameters.

3.0 : Added the *allow_extra_args* and *allow_interspersed_args* parameters.

4.0 : Added the *color*, *ignore_unknown_options*, and *max_content_width* parameters.

- **command** – the command class for this context.
- **parent** – the parent context.
- **info_name** – the info name for this invocation. Generally this is the most descriptive name for the script or command. For the toplevel script it is usually the name of the script, for commands below it it's the name of the script.
- **obj** – an arbitrary object of user data.
- **auto_envvar_prefix** – the prefix to use for automatic environment variables. If this is *None* then reading from environment variables is disabled. This does not affect manually set environment variables which are always read.
- **default_map** – a dictionary (like object) with default values for parameters.
- **terminal_width** – the width of the terminal. The default is inherit from parent context. If no context defines the terminal width then auto detection will be applied.
- **max_content_width** – the maximum width for content rendered by Click (this currently only affects help pages). This defaults to 80 characters if not overridden. In other words: even if the terminal is larger than that, Click will not format things wider than 80 characters by default. In addition to that, formatters might add some safety mapping on the right.
- **resilient_parsing** – if this flag is enabled then Click will parse without any interactivity or callback invocation. This is useful for implementing things such as completion support.
- **allow_extra_args** – if this is set to *True* then extra arguments at the end will not raise an error and will be kept on the context. The default is to inherit from the command.
- **allow_interspersed_args** – if this is set to *False* then options and arguments cannot be mixed. The default is to inherit from the command.
- **ignore_unknown_options** – instructs click to ignore options it does not know and keeps them for later processing.
- **help_option_names** – optionally a list of strings that define how the default help parameter is named. The default is `['--help']`.
- **token_normalize_func** – an optional function that is used to normalize tokens (options, choices, etc.). This for instance can be used to implement case insensitive behavior.
- **color** – controls if the terminal supports ANSI colors or not. The default is autodetection. This is only needed if ANSI codes are used in texts that Click prints which is by default not the case. This for instance would affect help output.

abort()

Aborts the script.

allow_extra_args = None
Indicates if the context allows extra args or if it should fail on parsing.

3.0 .

allow_interspersed_args = None
Indicates if the context allows mixing of arguments and options or not.

3.0 .

args = None
the leftover arguments.

call_on_close(*f*)
This decorator remembers a function as callback that should be executed when the context tears down. This is most useful to bind resource handling to the script execution. For instance, file objects opened by the *File* type will register their close callbacks here.

f – the function to execute on teardown.

close()
Invokes all close callbacks.

color = None
Controls if styling output is wanted or not.

command = None
the *Command* for this context.

command_path
The computed command path. This is used for the `usage` information on the help page. It's automatically created by combining the info names of the chain of contexts to the root.

ensure_object(*object_type*)
Like *find_object()* but sets the innermost object to a new instance of *object_type* if it does not exist.

exit(*code=0*)
Exits the application with a given exit code.

fail(*message*)
Aborts the execution of the program with a specific error message.

message – the error message to fail with.

find_object(*object_type*)
Finds the closest object of a given type.

find_root()
Finds the outermost context.

forward(**args*, ***kwargs*)
Similar to *invoke()* but fills in default keyword arguments from the current context if the other command expects it. This cannot invoke callbacks directly, only other commands.

get_help()
Helper method to get formatted help page for the current context and command.

get_usage()
Helper method to get formatted usage string for the current context and command.

help_option_names = None
The names for the help options.

ignore_unknown_options = None
Instructs click to ignore options that a command does not understand and will store it on the context for later processing. This is primarily useful for situations where you want to call into external programs. Generally this pattern is strongly discouraged because it's not possibly to losslessly forward all arguments.

4.0 .

info_name = None
the descriptive information name

invoke(**args*, ***kwargs*)
Invokes a command callback in exactly the way it expects. There are two ways to invoke this method:

1. the first argument can be a callback and all other arguments and keyword arguments are forwarded directly to the function.
2. the first argument is a click command object. In that case all arguments are forwarded as well but proper click parameters (options and click arguments) must be keyword arguments and Click will fill in defaults.

Note that before Click 3.2 keyword arguments were not properly filled in against the intention of this code and no context was created. For more information about this change and why it was done in a bugfix release see *Upgrading to 3.2*.

invoked_subcommand = None
This flag indicates if a subcommand is going to be executed. A group callback can use this information to figure out if it's being executed directly or because the execution flow passes onwards to a subcommand. By default it's None, but it can be the name of the subcommand to execute.

If chaining is enabled this will be set to `'*'` in case any commands are executed. It is however not possible to figure out which ones. If you require this knowledge you should use a `resultcallback()`.

lookup_default(*name*)
Looks up the default for a parameter name. This by default looks into the `default_map` if available.

make_formatter()
Creates the formatter for the help and usage output.

max_content_width = None
The maximum width of formatted content (None implies a sensible default which is 80 for most things).

meta
This is a dictionary which is shared with all the contexts that are nested. It exists so that click utiltiies can store some state here if they need to. It is however the responsibility of that code to manage this dictionary well.

The keys are supposed to be unique dotted strings. For instance module paths are a good choice for it. What is stored in there is irrelevant for the operation of click. However what is important is that code that places data here adheres to the general semantics of the system.

Example usage:

```
LANG_KEY = __name__ + '.lang'

def set_language(value):
    ctx = get_current_context()
    ctx.meta[LANG_KEY] = value

def get_language():
    return get_current_context().meta.get(LANG_KEY, 'en_US')
```

5.0 .

obj = None
the user object stored.

params = None
the parsed parameters except if the value is hidden in which case it's not remembered.

parent = None
the parent context or *None* if none exists.

protected_args = None
protected arguments. These are arguments that are prepended to *args* when certain parsing scenarios are encountered but must be never propagated to another arguments. This is used to implement nested parsing.

resilient_parsing = None
Indicates if resilient parsing is enabled. In that case Click will do its best to not cause any failures.

scope(**args*, ***kwds*)
This helper method can be used with the context object to promote it to the current thread local (see *get_current_context()*). The default behavior of this is to invoke the cleanup functions which can be disabled by setting *cleanup* to *False*. The cleanup functions are typically used for things such as closing file handles.

If the cleanup is intended the context object can also be directly used as a context manager.

Example usage:

```
with ctx.scope():
    assert get_current_context() is ctx
```

This is equivalent:

```
with ctx:
    assert get_current_context() is ctx
```

5.0 .

cleanup – controls if the cleanup functions should be run or not. The default is to run these functions. In some situations the context only wants to be temporarily pushed in which case this can be disabled. Nested pushes automatically defer the cleanup.

terminal_width = None
The width of the terminal (None is autodetection).

token_normalize_func = None
An optional normalization function for tokens. This is options, choices, commands etc.

click.**get_current_context**(*silent=False*)
Returns the current click context. This can be used as a way to access the current context object from anywhere. This is a more implicit alternative to the *pass_context()* decorator. This function is primarily useful for helpers such as *echo()* which might be interested in changing it's behavior based on the current context.

To push the current context, *Context.scope()* can be used.

5.0 .

silent – is set to *True* the return value is *None* if no context is available. The default behavior is to raise a RuntimeError.

2.1.6 Types

click.**STRING = STRING**
A unicode string parameter type which is the implicit default. This can also be selected by using `str` as type.

click.**INT = INT**
An integer parameter. This can also be selected by using `int` as type.

click.**FLOAT = FLOAT**
A floating point value parameter. This can also be selected by using `float` as type.

click.**BOOL = BOOL**
A boolean parameter. This is the default for boolean flags. This can also be selected by using `bool` as a type.

click.**UUID = UUID**
A UUID parameter.

click.**UNPROCESSED = UNPROCESSED**
A dummy parameter type that just does nothing. From a user's perspective this appears to just be the same as *STRING* but internally no string conversion takes place. This is necessary to achieve the same bytes/unicode behavior on Python 2/3 in situations where you want to not convert argument types. This is usually useful when working with file paths as they can appear in bytes and unicode.

For path related uses the *Path* type is a better choice but there are situations where an unprocessed type is useful which is why it is is provided.

4.0 .

class click.**File**(*mode='r'*, *encoding=None*, *errors='strict'*, *lazy=None*, *atomic=False*)
Declares a parameter to be a file for reading or writing. The file is automatically closed once the context tears down (after the command finished working).

Files can be opened for reading or writing. The special value – indicates stdin or stdout depending on the mode.

By default, the file is opened for reading text data, but it can also be opened in binary mode or for writing. The encoding parameter can be used to force a specific encoding.

The *lazy* flag controls if the file should be opened immediately or upon first IO. The default is to be non lazy for standard input and output streams as well as files opened for reading, lazy otherwise.

Starting with Click 2.0, files can also be opened atomically in which case all writes go into a separate file in the same folder and upon completion the file will be moved over to the original location. This is useful if a file regularly read by other users is modified.

See *File Arguments* for more information.

class click.**Path**(*exists=False*, *file_okay=True*, *dir_okay=True*, *writable=False*, *readable=True*, *resolve_path=False*, *allow_dash=False*, *path_type=None*)
The path type is similar to the *File* type but it performs different checks. First of all, instead of returning an open file handle it returns just the filename. Secondly, it can perform various basic checks about what the file or directory should be.

6.0 : *allow_dash* was added.

- **exists** – if set to true, the file or directory needs to exist for this value to be valid. If this is not required and a file does indeed not exist, then all further checks are silently skipped.
- **file_okay** – controls if a file is a possible value.
- **dir_okay** – controls if a directory is a possible value.
- **writable** – if true, a writable check is performed.

- **readable** – if true, a readable check is performed.
- **resolve_path** – if this is true, then the path is fully resolved before the value is passed onwards. This means that it's absolute and symlinks are resolved.
- **allow_dash** – If this is set to *True*, a single dash to indicate standard streams is permitted.
- **type** – optionally a string type that should be used to represent the path. The default is *None* which means the return value will be either bytes or unicode depending on what makes most sense given the input data Click deals with.

class click.**Choice**(*choices*)

The choice type allows a value to be checked against a fixed set of supported values. All of these values have to be strings.

See *Choice Options* for an example.

class click.**IntRange**(*min=None*, *max=None*, *clamp=False*)

A parameter that works similar to *click.INT* but restricts the value to fit into a range. The default behavior is to fail if the value falls outside the range, but it can also be silently clamped between the two edges.

See *Range Options* for an example.

class click.**Tuple**(*types*)

The default behavior of Click is to apply a type on a value directly. This works well in most cases, except for when *nargs* is set to a fixed count and different types should be used for different items. In this case the *Tuple* type can be used. This type can only be used if *nargs* is set to a fixed number.

For more information see *Tuples as Multi Value Options*.

This can be selected by using a Python tuple literal as a type.

types – a list of types that should be used for the tuple items.

class click.**ParamType**

Helper for converting values through types. The following is necessary for a valid type:

- it needs a name
- it needs to pass through None unchanged
- it needs to convert from a string
- it needs to convert its result type through unchanged (eg: needs to be idempotent)
- it needs to be able to deal with param and context being *None*. This can be the case when the object is used with prompt inputs.

convert(*value*, *param*, *ctx*)

Converts the value. This is not invoked for values that are *None* (the missing value).

envvar_list_splitter = None

if a list of this type is expected and the value is pulled from a string environment variable, this is what splits it up. *None* means any whitespace. For all parameters the general rule is that whitespace splits them up. The exception are paths and files which are split by `os.path.pathsep` by default (":" on Unix and ";" on Windows).

fail(*message*, *param=None*, *ctx=None*)

Helper method to fail with an invalid value message.

get_metavar(*param*)

Returns the metavar default for this param if it provides one.

get_missing_message(*param*)
Optionally might return extra information about a missing parameter.

2.0 .

name = None
the descriptive name of this type

split_envvar_value(*rv*)
Given a value from an environment variable this splits it up into small chunks depending on the defined envvar list splitter.

If the splitter is set to *None*, which means that whitespace splits, then leading and trailing whitespace is ignored. Otherwise, leading and trailing splitters usually lead to empty items being included.

2.1.7 Exceptions

exception click.**ClickException**(*message*)
An exception that Click can handle and show to the user.

exception click.**Abort**
An internal signalling exception that signals Click to abort.

exception click.**UsageError**(*message*, *ctx=None*)
An internal exception that signals a usage error. This typically aborts any further handling.

- **message** – the error message to display.
- **ctx** – optionally the context that caused this error. Click will fill in the context automatically in some situations.

exception click.**BadParameter**(*message*, *ctx=None*, *param=None*, *param_hint=None*)
An exception that formats out a standardized error message for a bad parameter. This is useful when thrown from a callback or type as Click will attach contextual information to it (for instance, which parameter it is).

2.0 .

- **param** – the parameter object that caused this error. This can be left out, and Click will attach this info itself if possible.
- **param_hint** – a string that shows up as parameter name. This can be used as alternative to *param* in cases where custom validation should happen. If it is a string it's used as such, if it's a list then each item is quoted and separated.

exception click.**FileError**(*filename*, *hint=None*)
Raised if a file cannot be opened.

exception click.**NoSuchOption**(*option_name*, *message=None*, *possibilities=None*, *ctx=None*)
Raised if click attempted to handle an option that does not exist.

4.0 .

exception click.**BadOptionUsage**(*message*, *ctx=None*)
Raised if an option is generally supplied but the use of the option was incorrect. This is for instance raised if the number of arguments for an option is not correct.

4.0 .

exception click.**BadArgumentUsage**(*message*, *ctx=None*)
Raised if an argument is generally supplied but the use of the argument was incorrect. This is for instance raised if the number of values for an argument is not correct.

6.0 .

2.1.8 Formatting

class click.**HelpFormatter**(*indent_increment=2*, *width=None*, *max_width=None*)
This class helps with formatting text-based help pages. It's usually just needed for very special internal cases, but it's also exposed so that developers can write their own fancy outputs.

At present, it always writes into memory.

- **indent_increment** – the additional increment for each level.
- **width** – the width for the text. This defaults to the terminal width clamped to a maximum of 78.

dedent()
Decreases the indentation.

getvalue()
Returns the buffer contents.

indent()
Increases the indentation.

indentation(**args*, ***kwds*)
A context manager that increases the indentation.

section(**args*, ***kwds*)
Helpful context manager that writes a paragraph, a heading, and the indents.

name – the section name that is written as heading.

write(*string*)
Writes a unicode string into the internal buffer.

write_dl(*rows*, *col_max=30*, *col_spacing=2*)
Writes a definition list into the buffer. This is how options and commands are usually formatted.

- **rows** – a list of two item tuples for the terms and values.
- **col_max** – the maximum width of the first column.
- **col_spacing** – the number of spaces between the first and second column.

write_heading(*heading*)
Writes a heading into the buffer.

write_paragraph()
Writes a paragraph into the buffer.

write_text(*text*)
Writes re-indented text into the buffer. This rewraps and preserves paragraphs.

write_usage(*prog*, *args=''*, *prefix='Usage: '*)
Writes a usage line into the buffer.

- **prog** – the program name.
- **args** – whitespace separated list of arguments.
- **prefix** – the prefix for the first line.

click.**wrap_text**(*text*, *width=78*, *initial_indent=''*, *subsequent_indent=''*, *preserve_paragraphs=False*)

A helper function that intelligently wraps text. By default, it assumes that it operates on a single paragraph of text but if the *preserve_paragraphs* parameter is provided it will intelligently handle paragraphs (defined by two empty lines).

If paragraphs are handled, a paragraph can be prefixed with an empty line containing the `\b` character (`\x08`) to indicate that no rewrapping should happen in that block.

- **text** – the text that should be rewrapped.
- **width** – the maximum width for the text.
- **initial_indent** – the initial indent that should be placed on the first line as a string.
- **subsequent_indent** – the indent string that should be placed on each consecutive line.
- **preserve_paragraphs** – if this flag is set then the wrapping will intelligently handle paragraphs.

2.1.9 Parsing

class click.**OptionParser**(*ctx=None*)

The option parser is an internal class that is ultimately used to parse options and arguments. It's modelled after optparse and brings a similar but vastly simplified API. It should generally not be used directly as the high level Click classes wrap it for you.

It's not nearly as extensible as optparse or argparse as it does not implement features that are implemented on a higher level (such as types or defaults).

ctx – optionally the *Context* where this parser should go with.

add_argument(*dest*, *nargs=1*, *obj=None*)

Adds a positional argument named *dest* to the parser.

The *obj* can be used to identify the option in the order list that is returned from the parser.

add_option(*opts*, *dest*, *action=None*, *nargs=1*, *const=None*, *obj=None*)

Adds a new option named *dest* to the parser. The destination is not inferred (unlike with optparse) and needs to be explicitly provided. Action can be any of `store`, `store_const`, `append`, `appnd_const` or `count`.

The *obj* can be used to identify the option in the order list that is returned from the parser.

allow_interspersed_args = None

This controls how the parser deals with interspersed arguments. If this is set to *False*, the parser will stop on the first non-option. Click uses this to implement nested subcommands safely.

ctx = None

The *Context* for this parser. This might be *None* for some advanced use cases.

ignore_unknown_options = None

This tells the parser how to deal with unknown options. By default it will error out (which is sensible),

but there is a second mode where it will ignore it and continue processing after shifting all the unknown options into the resulting args.

parse_args(*args*)
Parses positional arguments and returns `(values, args, order)` for the parsed options and arguments as well as the leftover arguments if there are any. The order is a list of objects as they appear on the command line. If arguments appear multiple times they will be memorized multiple times as well.

2.1.10 Testing

class click.testing.**CliRunner**(*charset=None*, *env=None*, *echo_stdin=False*)
The CLI runner provides functionality to invoke a Click command line script for unittesting purposes in a isolated environment. This only works in single-threaded systems without any concurrency as it changes the global interpreter state.

- **charset** – the character set for the input and output data. This is UTF-8 by default and should not be changed currently as the reporting to Click only works in Python 2 properly.
- **env** – a dictionary with environment variables for overriding.
- **echo_stdin** – if this is set to *True*, then reading from stdin writes to stdout. This is useful for showing examples in some circumstances. Note that regular prompts will automatically echo the input.

get_default_prog_name(*cli*)
Given a command object it will return the default program name for it. The default is the *name* attribute or `"root"` if not set.

invoke(*cli*, *args=None*, *input=None*, *env=None*, *catch_exceptions=True*, *color=False*, ***extra*)
Invokes a command in an isolated environment. The arguments are forwarded directly to the command line script, the *extra* keyword arguments are passed to the `main()` function of the command.

This returns a *Result* object.

3.0 : The `catch_exceptions` parameter was added.

3.0 : The result object now has an *exc_info* attribute with the traceback if available.

4.0 : The `color` parameter was added.

- **cli** – the command to invoke
- **args** – the arguments to invoke
- **input** – the input data for *sys.stdin*.
- **env** – the environment overrides.
- **catch_exceptions** – Whether to catch any other exceptions than `SystemExit`.
- **extra** – the keyword arguments to pass to `main()`.
- **color** – whether the output should contain color codes. The application can still override this explicitly.

isolated_filesystem(**args*, ***kwds*)
A context manager that creates a temporary folder and changes the current working directory to it for isolated filesystem tests.

isolation(**args*, ***kwds*)
A context manager that sets up the isolation for invoking of a command line tool. This sets up stdin with the given input data and *os.environ* with the overrides from the given dictionary. This also rebinds some internals in Click to be mocked (like the prompt functionality).

This is automatically done in the *invoke()* method.

4.0 : The `color` parameter was added.

- **input** – the input stream to put into sys.stdin.
- **env** – the environment overrides as dictionary.
- **color** – whether the output should contain color codes. The application can still override this explicitly.

make_env(*overrides=None*)
Returns the environment overrides for invoking a script.

class `click.testing.`**Result**(*runner*, *output_bytes*, *exit_code*, *exception*, *exc_info=None*)
Holds the captured result of an invoked CLI script.

exc_info = None
The traceback

exception = None
The exception that happend if one did.

exit_code = None
The exit code as integer.

output
The output as unicode string.

output_bytes = None
The output as bytes.

runner = None
The runner that created the result

CHAPTER 3

Miscellaneous Pages

3.1 click-contrib

As the userbase of Click grows, more and more major feature requests pop up in Click's bugtracker. As reasonable as it may be for those features to be bundled with Click instead of being a standalone project, many of those requested features are either highly experimental or have unproven practical use, while potentially being a burden to maintain.

This is why click-contrib exists. The GitHub organization is a collection of possibly experimental third-party packages whose featureset does not belong into Click, but also a playground for major features that may be added to Click in the future. It is also meant to coordinate and concentrate effort on writing third-party extensions for Click, and to ease the effort of searching for such extensions. In that sense it could be described as a low-maintenance alternative to extension repositories of other frameworks.

Please note that the quality and stability of those packages may be different than what you expect from Click itself. While published under a common organization, they are still projects separate from Click.

3.2 Upgrading To Newer Releases

Click attempts the highest level of backwards compatibility but sometimes this is not entirely possible. In case we need to break backwards compatibility this document gives you information about how to upgrade or handle backwards compatibility properly.

3.2.1 Upgrading to 3.2

Click 3.2 had to perform two changes to multi commands which were triggered by a change between Click 2 and Click 3 that had bigger consequences than anticipated.

Context Invokes

Click 3.2 contains a fix for the `Context.invoke()` function when used with other commands. The original intention of this function was to invoke the other command as as if it came from the command line when it was passed a context object instead of a function. This use was only documented in a single place in the documentation before and there was no proper explanation for the method in the API documentation.

The core issue is that before 3.2 this call worked against intentions:

```
ctx.invoke(other_command, 'arg1', 'arg2')
```

This was never intended to work as it does not allow Click to operate on the parameters. Given that this pattern was never documented and ill intended the decision was made to change this behavior in a bugfix release before it spreads by accident and developers depend on it.

The correct invocation for the above command is the following:

```
ctx.invoke(other_command, name_of_arg1='arg1', name_of_arg2='arg2')
```

This also allowed us to fix the issue that defaults were not handled properly by this function.

Multicommand Chaining API

Click 3 introduced multicommand chaning. This required a change in how Click internally dispatches. Unfortunately this change was not correctly implemented and it appeared that it was possible to provide an API that can inform the super command about all the subcommands that will be invoked.

This assumption however does not work with one of the API guarantees that have been given in the past. As such this functionality has been removed in 3.2 as it was already broken. Instead the accidentally broken functionality of the `Context.invoked_subcommand` attribute was restored.

If you do require the know which exact commands will be invoked there are different ways to cope with this. The first one is to let the subcommands all return functions and then to invoke the functions in a `Context.resultcallback()`.

3.2.2 Upgrading to 2.0

Click 2.0 has one breaking change which is the signature for parameter callbacks. Before 2.0, the callback was invoked with `(ctx, value)` whereas now it's `(ctx, param, value)`. This change was necessary as it otherwise made reusing callbacks too complicated.

To ease the transition Click will still accept old callbacks. Starting with Click 3.0 it will start to issue a warning to stderr to encourage you to upgrade.

In case you want to support both Click 1.0 and Click 2.0, you can make a simple decorator that adjusts the signatures:

```
import click
from functools import update_wrapper

def compatcallback(f):
    # Click 1.0 does not have a version string stored, so we need to
    # use getattr here to be safe.
    if getattr(click, '__version__', '0.0') >= '2.0':
        return f
    return update_wrapper(lambda ctx, value: f(ctx, None, value), f)
```

With that helper you can then write something like this:

```
@compatcallback
def callback(ctx, param, value):
    return value.upper()
```

Note that because Click 1.0 did not pass a parameter, the *param* argument here would be *None*, so a compatibility callback could not use that argument.

3.3 License

Click is licensed under a three-clause BSD License. It basically means: do whatever you want with it as long as the copyright in Click sticks around, the conditions are not modified and the disclaimer is present. Furthermore, you must not use the names of the authors to promote derivatives of the software without written consent.

3.3.1 License Text

C

L

M

N

O

P

R

S

T

U

W