

Quickstart

You can get the library directly from PyPI:

```
pip install click
```

The installation into a [virtualenv](#) is heavily recommended.

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.

Create your project folder, then a virtualenv within it:

```
$ mkdir myproject
$ cd myproject
$ python3 -m venv .venv
```

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
(venv) $
```

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

```
> .venv\scripts\activate
(venv) >
```

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

 v: 8.1.x ▼

And if you want to stop using the virtualenv, use the following command:

```
$ deactivate
```

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

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: [Color support demo](#)
- termui: [Terminal UI functions demo](#)
- imagepipe: [Multi command chaining demo](#)

Basic Concepts - Creating a Command

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!')
```

 v: 8.1.x ▼

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.
```

Echoing

Why does this example use `echo()` instead of the regular `print()` function? The answer to this question is that Click attempts to support different environments consistently 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 a `UnicodeError`.

The echo function also supports color and other styles in output. It will automatically remove styles if the output stream is a file. On Windows, colorama is automatically installed and used. See [ANSI Colors](#).

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

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
```

 v: 8.1.x ▼

```

@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')

```

You would then invoke the `Group` in your setuptools entry points or other invocations:

```

if __name__ == '__main__':
    cli()

```

Registering Commands Later

Instead of using the `@group.command()` decorator, commands can be decorated with the plain `@click.command()` decorator and registered with a group later with `group.add_command()`. This could be used to split commands into multiple Python modules.

```

@click.command()
def greet():
    click.echo("Hello, World!")

```

 v: 8.1.x ▾

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

group.add_command(greet)
```

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(f"Hello {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.
```


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 document  [v: 8.1.x](#) ▼ 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.