django_typer

Release 1.1.2

Brian Kohan

Apr 22, 2024

CONTENTS:

Use Typer to define the CLI for your Django management commands. Provides a TyperCommand class that inherits from BaseCommand and allows typer-style annotated parameter types. All of the BaseCommand functionality is preserved, so that TyperCommand can be a drop in replacement.

django-typer makes it easy to:

- Define your command CLI interface in a clear, DRY and safe way using type hints
- Create subcommand and group command hierarchies.
- Use the full power of Typer's parameter types to validate and parse command line inputs.
- Create beautiful and information dense help outputs.
- Configure the rendering of exception stack traces using rich.
- *Install shell tab-completion support* for TyperCommands and normal Django commands for bash, zsh, fish and powershell.
- *Create custom and portable shell tab-completions for your CLI parameters.*
- Refactor existing management commands into TyperCommands because TyperCommand is interface compatible with BaseCommand.

Installation

1. Clone django-typer from GitHub or install a release off PyPI :

   ```
   pip install django-typer
   ```

   rich is a powerful library for rich text and beautiful formatting in the terminal. It is not required, but highly recommended for the best experience:

   ```
   pip install "django-typer[rich]"
   ```

2. Add `django_typer` to your `INSTALLED_APPS` setting:

   ```
   INSTALLED_APPS = [
       ...
       'django_typer',
   ]
   ```

   You only need to install django_typer as an app if you want to use the shellcompletion command to enable tab-completion or if you would like django-typer to install rich traceback rendering for you - which it does by default if rich is also installed.

Basic Example

For example TyperCommands can be a very simple drop in replacement for BaseCommands. All of the documented features of BaseCommand work!

Listing 1: A Basic Command

```
from django_typer import TyperCommand


class Command(TyperCommand):
    def handle(self, arg1: str, arg2: str, arg3: float = 0.5, arg4: int = 1):
        """
        A basic command that uses Typer
        """
```

Multiple Subcommands Example

Or commands with multiple subcommands can be defined:

Listing 2: A Command w/Subcommands

```python
import typing as t

from django.utils.translation import gettext_lazy as _
from typer import Argument

from django_typer import TyperCommand, command


class Command(TyperCommand):
    """
    A command that defines subcommands.
    """

    @command()
    def create(
        self,
        name: t.Annotated[str, Argument(help=_("The name of the object to create."))],
    ):
        """
        Create an object.
        """

    @command()
    def delete(
        self, id: t.Annotated[int, Argument(help=_("The id of the object to delete.
↪"))]
    ):
        """
        Delete an object.
        """
```

Grouping and Hierarchies Example

Or more complex groups and subcommand hierarchies can be defined. For example this command defines a group of commands called math, with subcommands divide and multiply. The group has a common initializer that optionally sets a float precision value. We would invoke this command like so:

```
./manage.py hierarchy math --precision 5 divide 10 2.1
4.76190
./manage.py hierarchy math multiply 10 2
20.00
```

Any number of groups and subcommands and subgroups of other groups can be defined allowing for arbitrarily complex command hierarchies.

Listing 3: A Command w/Grouping Hierarchy

```python
import typing as t
from functools import reduce

from django.utils.translation import gettext_lazy as _
from typer import Argument, Option

from django_typer import TyperCommand, group


class Command(TyperCommand):

    help = _("A more complex command that defines a hierarchy of subcommands.")

    precision = 2

    @group(help=_("Do some math at the given precision."))
    def math(
        self,
        precision: t.Annotated[
            int, Option(help=_("The number of decimal places to output."))
        ] = precision,
    ):
        self.precision = precision

    @math.command(help=_("Multiply the given numbers."))
    def multiply(
        self,
        numbers: t.Annotated[
            t.List[float], Argument(help=_("The numbers to multiply"))
        ],
    ):
        return f"{reduce(lambda x, y: x * y, [1, *numbers]):.{self.precision}f}"

    @math.command()
    def divide(
        self,
        numerator: t.Annotated[float, Argument(help=_("The numerator"))],
        denominator: t.Annotated[float, Argument(help=_("The denominator"))],
        floor: t.Annotated[bool, Option(help=_("Use floor division"))] = False,
    ):
        """
        Divide the given numbers.
        """
        if floor:
            return str(numerator // denominator)
        return f"{numerator / denominator:.{self.precision}f}"
```

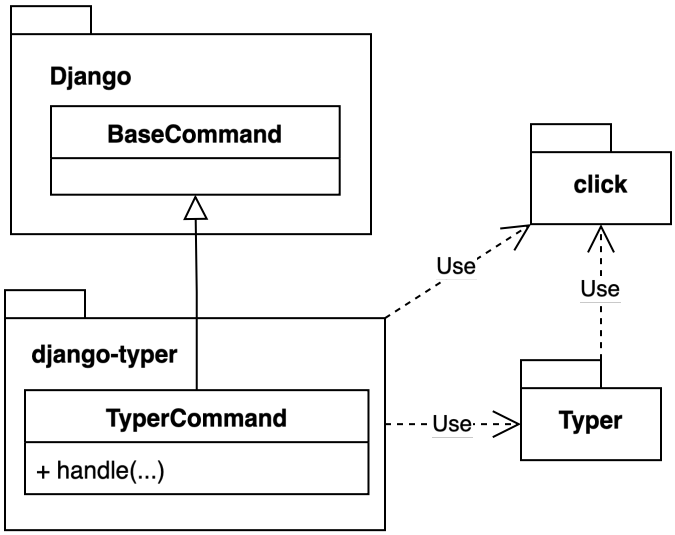
CHAPTER

ONE

TUTORIAL

Using the *TyperCommand* class is very similar to using the BaseCommand class. The main difference is that we use Typer's decorators, classes and type annotations to define the command's command line interface instead of argparse as BaseCommand expects.

1.1 Upstream Libraries



django-typer merges the Django BaseCommand interface with the Typer interface and Typer itself is built on top of click. This means when using django-typer you will encounter interfaces and concepts from *all three* of these upstream libraries:

- BaseCommand

  Django has a good tutorial for understanding how commands are organized and built in Django. If you are unfamiliar with using BaseCommand please first work through the polls Tutorial in the Django documentation.

- Typer

  This tutorial can be completed without working through the Typer tutorials, but familiarizing yourself with Typer will make this easier and will also be helpful when you want to define CLIs outside of Django! We use the Typer interface to define Arguments and Options so please refer to the Typer documentation for any questions about how to define these.

- click

Click interfaces and concepts are relatively hidden by Typer, but occasionally you may need to refer to the click documentation when you want to implement more complex behaviors like passing context parameters. It is not necessary to familiarize yourself with click to use django-typer, but you should know that it exists and is the engine behind much of this functionality.

1.2 Install django-typer

1. Install the latest release off PyPI :

```
pip install "django-typer[rich]"
```

rich is a powerful library for rich text and beautiful formatting in the terminal. It is not required, but highly recommended for the best experience:

Note: If you install rich, traceback rendering will be enabled by default. Refer to the *how-to* if you would like to disable it.

---

2. Add `django_typer` to your `INSTALLED_APPS` setting:

```
INSTALLED_APPS = [
    ...
    'django_typer',
]
```

Note: Adding django_typer to INSTALLED_APPS is not strictly necessary if you do not wish to use shell tab completions or configure rich traceback rendering.

1.3 Convert the closepoll command to a `TyperCommand`

Recall our closepoll command from the polls Tutorial in the Django documentation looks like this:

```
from django.core.management.base import BaseCommand, CommandError
from polls.models import Question as Poll


class Command(BaseCommand):
    help = "Closes the specified poll for voting"

    def add_arguments(self, parser):
        parser.add_argument("poll_ids", nargs="+", type=int)

        # Named (optional) arguments
        parser.add_argument(
            "--delete",
            action="store_true",
            help="Delete poll instead of closing it",
        )

    def handle(self, *args, **options):
```

(continues on next page)

(continued from previous page)

```
        for poll_id in options["poll_ids"]:
            try:
                poll = Poll.objects.get(pk=poll_id)
            except Poll.DoesNotExist:
                raise CommandError(f'Poll "{poll_id}" does not exist')

            poll.opened = False
            poll.save()

            self.stdout.write(
                self.style.SUCCESS(f'Successfully closed poll "{poll.id}"')
            )

            if options["delete"]:
                poll.delete()
```

1.3.1 Inherit from `TyperCommand`

We first need to change the inheritance to *TyperCommand* and then move the argument and option definitions from add_arguments into the method signature of handle. A minimal conversion may look like this:

```
import typing as t

from django_typer import TyperCommand
from django.core.management.base import CommandError
from polls.models import Question as Poll

class Command(TyperCommand):
    help = "Closes the specified poll for voting"

    def handle(
        self,
        poll_ids: t.List[int],
        delete: bool = False,
    ):
        for poll_id in poll_ids:
            try:
                poll = Poll.objects.get(pk=poll_id)
            except Poll.DoesNotExist:
                raise CommandError(f'Poll "{poll_id}" does not exist')

            poll.opened = False
            poll.save()

            self.stdout.write(
                self.style.SUCCESS(f'Successfully closed poll "{poll.id}"')
            )

            if delete:
                poll.delete()
```

You'll note that we've removed add_arguments entirely and specified the arguments and options as parameters to the handle method. django-typer will interpret the parameters on the handle() method as the command line interface for the command. If we have rich installed the help for our new closepoll command will look like this:

Note: *TyperCommand* adds the standard set of default options to the command line interface, with the exception of verbosity.

1.3.2 Add Helps with Type annotations

Typer allows us to use Annotated types to add additional controls to how the command line interface behaves. The most common use case for this is to add help text to the command line interface. We will annotate our parameter type hints with one of two Typer parameter types, either Argument or Option. Arguments are positional parameters and Options are named parameters (i.e. *–delete*). In our polls example, the poll_ids are arguments and the delete flag is an option. Here is what that would look like:

```
import typing as t

from django.core.management.base import CommandError
from django.utils.translation import gettext_lazy as _
from typer import Argument, Option

from django_typer import TyperCommand
from polls.models import Question as Poll


class Command(TyperCommand):
    help = "Closes the specified poll for voting"

    def handle(
        self,
        poll_ids: t.Annotated[
            t.List[int], Argument(help=_("The database IDs of the poll(s) to close."))
        ],
        delete: t.Annotated[
            bool, Option(help=_("Delete poll instead of closing it."))
        ] = False,
    ):
        # ...
```

See that our help text now shows up in the command line interface. Also note, that lazy translations work for the help strings. Typer also allows us to specify our help text in the docstrings of the command function or class, in this case either Command or handle() - but docstrings are not available to the translation system. If translation is not necessary and your help text is extensive or contains markup the docstring may be the more appropriate place to put it.

Note: On Python <=3.8 you will need to import Annotated from typing_extensions instead of the standard library.

1.3.3 Defining custom and reusable parameter types

We may have other commands that need to operate on Poll objects from given poll ids. We could duplicate our for loop that loads Poll objects from ids, but that wouldn't be very DRY. Instead, Typer allows us to define custom parsers for arbitrary parameter types. Lets see what that would look like if we used the Poll class as our type hint:

```
import typing as t

# ...

def get_poll_from_id(poll: t.Union[str, Poll]) -> Poll:
    # our parser may be passed a Poll object depending on how
    # users might call our command from code - so we must check
    # to be sure we have something to parse at all!
    if isinstance(poll, Poll):
        return poll
    try:
        return Poll.objects.get(pk=int(poll))
    except Poll.DoesNotExist:
        raise CommandError(f'Poll "{poll_id}" does not exist')


class Command(TyperCommand):

    def handle(
        self,
        polls: t.Annotated[
            t.List[Poll],  # change our type hint to a list of Polls!
            Argument(
                parser=get_poll_from_id,  # pass our parser to the Argument!
                help=_("The database IDs of the poll(s) to close."),
            ),
        ],
        delete: t.Annotated[
            bool,
            Option(
                "--delete",  # we can also get rid of that unnecessary --no-delete␣
↪flag
                help=_("Delete poll instead of closing it."),
            ),
        ] = False,
    ):
        """
        Closes the specified poll for voting.

        As mentioned in the last section, helps can also
        be set in the docstring
        """
        for poll in polls:
            poll.opened = False
            poll.save()
            self.stdout.write(
                self.style.SUCCESS(f'Successfully closed poll "{poll.id}"')
            )
            if delete:
                poll.delete()
```

django-typer offers some built-in *parsers* that can be used for common Django types. For example, the *ModelObjectParser* can be used to fetch a model object from a given field. By default it will use the primary key, so we could rewrite the relevant lines above like so:

```
from django_typer.parsers import ModelObjectParser

# ...

t.Annotated[
    t.List[Poll],
    Argument(
        parser=ModelObjectParser(Poll),
        help=_("The database IDs of the poll(s) to close.")
    )
]

# ...
```

1.3.4 Add shell tab-completion suggestions for polls

It's very annoying to have to know the database primary key of the poll to close it. django-typer makes it easy to add tab completion suggestions! You can always implement your own completer functions, but as with, *parsers*, there are some out-of-the-box *completers* that make this easy. Let's see what the relevant updates to our closepoll command would look like:

```
from django_typer.parsers import ModelObjectParser
from django_typer.completers import ModelObjectCompleter

# ...

t.Annotated[
    t.List[Poll],
    Argument(
        parser=ModelObjectParser(Poll),
        shell_complete=ModelObjectCompleter(Poll, help_field='question_text'),
        help=_("The database IDs of the poll(s) to close.")
    )
]

# ...
```

Note: For tab-completions to work you will need to *install the shell completion scripts for your shell*.

1.3.5 Putting it all together

When we're using a *ModelObjectParser* and *ModelObjectCompleter* we can use the *model_parser_completer()* convenience function to reduce the amount of boiler plate. Let's put everything together and see what our full-featured refactored closepoll command looks like:

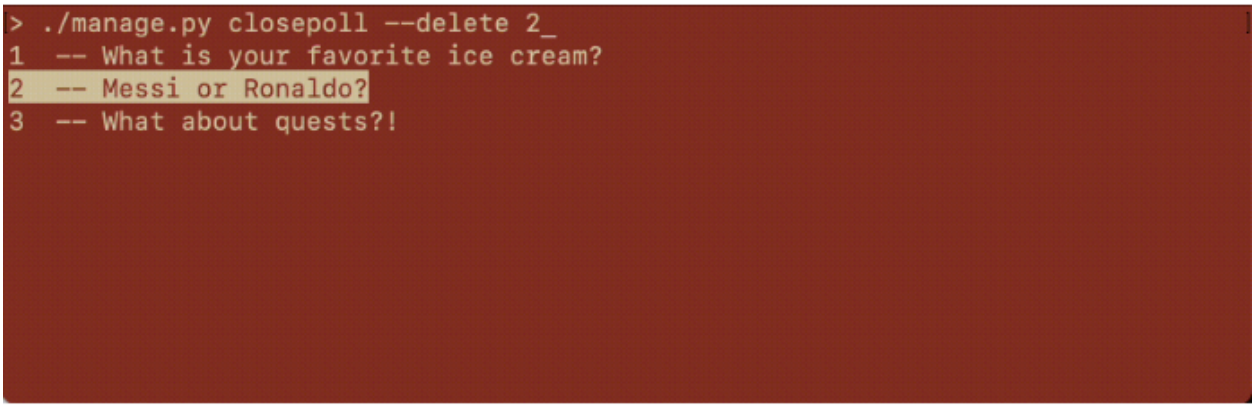
```python
import typing as t

from django.utils.translation import gettext_lazy as _
from typer import Argument, Option

from django_typer import TyperCommand, model_parser_completer
from polls.models import Question as Poll


class Command(TyperCommand):
    help = _("Closes the specified poll for voting.")

    def handle(
        self,
        polls: Annotated[
            t.List[Poll],
            Argument(
                **model_parser_completer(Poll, help_field="question_text"),
                help=_("The database IDs of the poll(s) to close."),
            ),
        ],
        delete: Annotated[
            bool, Option(help=_("Delete poll instead of closing it.")),
        ] = False,
    ):
        for poll in polls:
            poll.opened = False
            poll.save()
            self.stdout.write(
                self.style.SUCCESS(f'Successfully closed poll "{poll.id}"')
            )
            if delete:
                poll.delete()
```



CHAPTER

TWO

HOW-TO

2.1 Define an Argument

Positional arguments on a command are treated as positional command line arguments by Typer. For example to define an integer positional argument we could simply do:

```
def handle(self, int_arg: int):
    ...
```

You will likely want to add additional meta information to your arguments for Typer to render things like helps and usage strings. You can do this by annotating the type hint with the *typer.Argument* class:

```
import typing as t
from typer import Argument

# ...

def handle(self, int_arg: t.Annotated[int, Argument(help="An integer argument")]):
    ...
```

Tip: Refer to the Typer docs on arguments for more details.

2.2 Define an Option

Options are like arguments but are not position dependent and instead provided with a preceding identifier string (e.g. *–name*).

When a default value is provided for a parameter, Typer will treat it as an option. For example:

```
def handle(self, flag: bool = False):
    ...
```

Would be called like this:

```
$ mycommand --flag
```

If the type hint on the option is something other than a boolean it will accept a value:

```
def handle(self, name: str = "world"):
    ...
```

Would be called like this:

```
$ mycommand --name=world
$ mycommand --name world  # this also works
```

To add meta information, we annotate with the *typer.Option* class:

```
import typing as t
from typer import Option

# ...

def handle(self, name: t.Annotated[str, Option(help="The name of the thing")]):
    ...
```

Tip: Refer to the Typer docs on options for more details.

2.3 Define Multiple Subcommands

Commands with a single executable function should simply implement handle(), but if you would like have multiple subcommands you can define any number of functions decorated with *command()*:

```
from django_Typer import TyperCommand, command

class Command(TyperCommand):

    @command()
    def subcommand1(self):
        ...

    @command()
    def subcommand2(self):
        ...
```

Note: When no handle() method is defined, you cannot invoke a command instance as a callable. instead you should invoke subcommands directly:

```
from django_typer import get_command

command = get_command("mycommand")
command.subcommand1()
command.subcommand2()

command() # this will raise an error
```

2.4 Define Multiple Subcommands w/ a Default

We can also implement a default subcommand by defining a handle() method, and we can rename it to whatever we want the command to be. For example to define three subcommands but have one as the default we can do this:

```
from django_typer import TyperCommand, command

class Command(TyperCommand):

    @command(name='subcommand1')
    def handle(self):
        ...

    @command()
    def subcommand2(self):
        ...

    @command()
    def subcommand3(self):
        ...
```

```
from django_typer import get_command

command = get_command("mycommand")
command.subcommand2()
command.subcommand3()

command() # this will invoke handle (i.e. subcommand1)

# note - we *cannot* do this:
command.handle()

# or this:
command.subcommand1()
```

Lets look at the help output:

```
django_typer

Usage: django_typer [OPTIONS] COMMAND [ARGS]...

╭─ Options ──────────────────────────────────────────────────────────────────────╮
│ --help          Show this message and exit.                                    │
╰────────────────────────────────────────────────────────────────────────────────╯
╭─ Django ───────────────────────────────────────────────────────────────────────╮
│ --version                    Show program's version number and exit.           │
│ --settings          TEXT     The Python path to a settings module, e.g.        │
│                              "myproject.settings.main". If this isn't          │
│                              provided, the DJANGO_SETTINGS_MODULE environment  │
│                              variable will be used.                            │
│ --pythonpath        PATH     A directory to add to the Python path, e.g.       │
│                              "/home/djangoprojects/myproject".                 │
│                              [default: None]                                   │
│ --traceback                  Raise on CommandError exceptions                  │
│ --no-color                   Don't colorize the command output.                │
│ --force -color               Force colorization of the command output.         │
│ --skip -checks               Skip system checks.                               │
╰────────────────────────────────────────────────────────────────────────────────╯
╭─ Commands ─────────────────────────────────────────────────────────────────────╮
│ subcommand1                                                                    │
│ subcommand2                                                                    │
│ subcommand3                                                                    │
╰────────────────────────────────────────────────────────────────────────────────╯
```

2.5 Define Groups of Commands

Any depth of command tree can be defined. Use the *group()* decorator to define a group of subcommands:

```
from django_Typer import TyperCommand, command, group

class Command(TyperCommand):

    @group()
    def group1(self, common_option: bool = False):
        # you can define common options that will be available to all subcommands of
        # the group, and implement common initialization logic here.

    @group()
    def group2(self):
        ...

    # attach subcommands to groups by using the command decorator on the group␣
↪function
    @group1.command()
    def grp1_subcommand1(self):
        ...

    @group1.command()
    def grp1_subcommand1(self):
        ...
```

(continues on next page)

(continued from previous page)

```
    # groups can have subgroups!
    @group1.group()
    def subgroup1(self):
        ...

    @subgroup1.command()
    def subgrp_command(self):
        ...
```

2.6 Define an Initialization Callback

You can define an initializer function that takes arguments and options that will be invoked before your handle() command or subcommands using the *initialize()* decorator. This is like defining a group at the command root and is an extension of the typer callback mechanism.

```
from django_Typer import TyperCommand, initialize, command

class Command(TyperCommand):

    @initialize()
    def init(self, common_option: bool = False):
        # you can define common options that will be available to all subcommands of
        # the command, and implement common initialization logic here. This will be
        # invoked before the chosen command
        ...

    @command()
    def subcommand1(self):
        ...

    @command()
    def subcommand2(self):
        ...
```

2.7 Call TyperCommands from Code

There are two options for invoking a *TyperCommand* from code without spawning off a subprocess. The first is to use Django's builtin call_command function. This function will work exactly as it does for normal BaseCommand derived commands. django-typer however adds another mechanism that can be more efficient, especially if your options and arguments are already of the correct type and require no parsing:

Say we have this command, called `mycommand`:

```
from django_typer import TyperCommand, command

class Command(TyperCommand):

    def handle(self, count: int=5):
        return count
```

```
from django.core.management import call_command
from django_typer import get_command

# we can use use call_command like with any Django command
call_command("mycommand", count=10)
call_command("mycommand", '--count=10')  # this will work too

# or we can use the get_command function to get the command instance and invoke it␣
↪directly
mycommand = get_command("mycommand")
mycommand(count=10)
mycommand(10) # this will work too

# return values are also available
assert mycommand(10) == 10
```

The rule of them is this:

- Use call_command if your options and arguments need parsing.
- Use *get_command()* and invoke the command functions directly if your options and arguments are already of the correct type.

Tip: Also refer to the *get_command()* docs and *here* and *here* for the nuances of calling commands when handle() is and is not implemented.

2.8 Change Default Django Options

TyperCommand classes preserve all of the functionality of BaseCommand derivatives. This means that you can still use class members like suppressed_base_arguments to suppress default options.

By default *TyperCommand* suppresses `--verbosity`. You can add it back by setting `suppressed_base_arguments` to an empty list. If you want to use verbosity you can simply redefine it or use one of django-typer's *provided type hints* for the default BaseCommand options:

```
from django_typer import TyperCommand
from django_typer.types import Verbosity

class Command(TyperCommand):

    suppressed_base_arguments = ['--settings']  # remove the --settings option

    def handle(self, verbosity: Verbosity=1):
        ...
```

2.9 Configure the Typer Application

Typer apps can be configured using a number of parameters. These parameters are usually passed to the Typer class constructor when the application is created. django-typer provides a way to pass these options upstream to Typer by supplying them as keyword arguments to the *TyperCommand* class inheritance:

```
from django_typer import TyperCommand

class Command(TyperCommand, chain=True):
    # here we pass chain=True to typer telling it to allow invocation of
    # multiple subcommands
    ...
```

Tip: See *TyperCommandMeta* for a list of available parameters. Also refer to the Typer docs for more details. Note that not all of these parameters make sense in the context of a Django management command, so behavior for some is undefined.

2.10 Define Shell Tab Completions for Parameters

See the section on *defining shell completions.*

2.11 Debug Shell Tab Completers

See the section on *debugging shell completers.*

2.12 Extend/Override TyperCommands

You can extend typer commands simply by subclassing them. All of the normal inheritance rules apply. You can either subclass an existing command from an upstream app and leave its module the same name to extend and override the command or you can subclass and rename the module to provide an adapted version of the upstream command with a different name.

2.13 Configure rich Stack Traces

When rich is installed it may be configured to display rendered stack traces for unhandled exceptions. These stack traces are information dense and can be very helpful for debugging. By default, if rich is installed django-typer will configure it to render stack traces. You can disable this behavior by setting the `DT_RICH_TRACEBACK_CONFIG` config to `False`. You may also set `DT_RICH_TRACEBACK_CONFIG` to a dictionary holding the parameters to pass to *rich.traceback.install*.

This provides a common hook for configuring rich that you can control on a per-deployment basis:

Listing 1: settings.py

```
# refer to the rich docs for details
DT_RICH_TRACEBACK_CONFIG = {
    "console": rich.console.Console(), # create a custom Console object for rendering
    "width": 100,                      # default is 100
    "extra_lines": 3,                  # default is 3
    "theme": None,                     # predefined themes
    "word_wrap": False,                # default is False
    "show_locals": True,               # rich default is False, but we turn this on
    "locals_max_length":               # default is 10
    "locals_max_string":               # default is 80
    "locals_hide_dunder": True,        # default is True
    "locals_hide_sunder": False,       # default is None
    "indent_guides": True,             # default is True
    "suppress": [],                    # suppress frames from these module import␣
↪paths
    "max_frames": 100                  # default is 100
}

# or turn off rich traceback rendering
DT_RICH_TRACEBACK_CONFIG = False
```

Tip: There are traceback configuration options that can be supplied as configuration parameters to the Typer application. It is best to not set these and allow users to configure tracebacks via the `DT_RICH_TRACEBACK_CONFIG` setting.

2.14 Add Help Text to Commands

There are multiple places to add help text to your commands. There is however a precedence order, and while lazy translation is supported in help texts, if you use docstrings as the helps they will not be translated.

The precedence order, for a simple command is as follows:

```
from django_typer import TyperCommand, command
from django.utils.translation import gettext_lazy as _

class Command(TyperCommand, help=_('2')):

    help = _("3")

    @command(help=_("1"))
    def handle(self):
        """
        Docstring is last priority and is not subject to translation.
        """
```

2.15 Document Commands w/Sphinx

Checkout this Sphinx extension that can be used to render your rich helps to Sphinx docs.

For example, to document a *TyperCommand* with sphinxcontrib-typer, you would do something like this:

```
.. typer:: django_typer.management.commands.shellcompletion.Command:typer_app
    :prog: ./manage.py shellcompletion
    :show-nested:
    :width: 80
```

The Typer application object is a property of the command class and is named *typer_app*. The typer directive simply needs to be given the fully qualified import path of the application object.

Or we could render the helps for individual subcommands as well:

```
.. typer:: django_typer.management.commands.shellcompletion.Command:typer_app:install
    :prog: ./manage.py shellcompletion
    :width: 80
```

You'll also need to make sure that Django is bootstrapped in your conf.py file:

Listing 2: conf.py

```
import django

os.environ.setdefault('DJANGO_SETTINGS_MODULE', 'path.to.your.settings')
django.setup()
```

CHAPTER

THREE

SHELL TAB-COMPLETIONS

Shell completions are helpful suggestions that are displayed when you press the <TAB> key while typing a command in a shell. They are especially useful when you are not sure about the exact name of a command, its options, its arguments or the potential values of either.

Django has some support for bash completions, but it is not enabled by default and left to the user to install.

django-typer augments the upstream functionality of Typer and Click to provide both an easy way to define shell completions for your custom CLI options and arguments as well as a way to install them in your shell.

Note: django-typer supports shell completion installation for bash, zsh, fish and powershell.

3.1 Installation

Each shell has its own mechanism for enabling completions and this is further complicated by how different shells are installed and configured on different platforms. All shells have the same basic process. Completion logic needs to be registered with the shell that will be invoked when tabs are pressed for a specific command or script. To install tab completions for django commands we need to register our completion logic for Django manage script with the shell. This process has two phases:

1. Ensure that your shell is configured to support completions.

2. Use the *shellcompletion* command to install the completion hook for your Django manage script. This usually entails adding a specifically named script to a certain directory or adding lines to an existing script. The *shellcompletion* command will handle this for you.

The goal of this guide is not to be an exhaustive list of how to enable completions for each supported shell on all possible platforms, but rather to provide general guidance on how to enable completions for the most common platforms and environments. If you encounter issues or have solutions, please report them on our issues page

3.1.1 Windows

powershell is now bundled with most modern versions of Windows. There should be no additional installation steps necessary, but please refer to the Windows documentation if powershell is not present on your system.

3.1.2 Linux

bash is the default shell on most Linux distributions. Completions should be enabled by default.

3.1.3 OSX

zsh is currently the default shell on OSX. Unfortunately completions are not supported out of the box. We recommend using homebrew to install the zsh-completions package.

After installing the package you will need to add some configuration to your `.zshrc` file. We have had luck with the following:

```
zstyle ':completion:*' menu select

if type brew &>/dev/null; then
    FPATH=~/.zfunc:$(brew --prefix)/share/zsh-completions:$FPATH

    autoload -Uz compinit
    compinit
fi

fpath+=~/.zfunc
```

3.1.4 Install the Completion Hook

django-typer comes with a management command called *shellcompletion*. To install completions for your Django project simply run the install command:

```
./manage.py shellcompletion install
```

Note: The manage script may be named differently in your project - this is fine. The only requirement is that you invoke the shellcompletion command in the same way you would invoke any commands you would like tab completions to work for.

The installation script should be able to automatically detect your shell and install the appropriate scripts. If it is unable to do so you may force it to install for a specific shell by passing the shell name as an argument. Refer to the *shellcompletion* for details.

After installation you will need to restart your shell or source the appropriate rc file.

> **Warning:** In production environments it is recommended that your management script be installed as a command on your system path. This will produce the most reliable installation.

3.1.5 Enabling Completions in Development Environments

Most shells work best when the manage script is installed as an executable on the system path. This is not always the case, especially in development environments. In these scenarios completion installation *should still work*, but you may need to always invoke the script from the same path. Fish may not work at all in this mode.

3.1.6 Integrating with Other CLI Completion Libraries

When tab completion is requested for a command that is not a TyperCommand, django-typer will delegate that request to Django's autocomplete function as a fallback. This means that using django-typer to install completion scripts will enable completions for Django BaseCommands in all supported shells.

However, if you are using a separate package to define custom tab completions for your commands you may use the –fallback parameter to supply a separate fallback hook that will invoke the appropriate completion function for your commands. If there are other popular completion libraries please consider letting us know or submitting a PR to support these libraries as a fallback out of the box.

The long-term solution here should be that Django itself manages completion installation and provides hooks for implementing libraries to provide completions for their own commands.

3.2 Defining Custom Completions

To define custom completion logic for your arguments and options pass the `shell_completion` parameter in your type hint annotations. django-typer comes with a *few provided completers* for common Django types. One of the provided completers completes Django app labels and names. We might build a similar completer that only works for Django app labels like this:

```
import typing as t
import typer
from click import Context, Parameter
from click.shell_completion import CompletionItem
from django.apps import apps

from django_typer import TyperCommand


# the completer function signature must match this exactly
def complete_app_label(
    ctx: Context,
    param: Parameter,
    incomplete: str
) -> t.List[CompletionItem]:

    # don't offer apps that are already present as completion suggestions
    present = [app.label for app in (ctx.params.get(param.name or "") or [])]
    return [
```

(continues on next page)

(continued from previous page)

```
        CompletionItem(app.label)
        for app in apps.get_app_configs()
        if app.label.startswith(incomplete) and app.label not in present
    ]


class MyCommand(TyperCommand):

    @command()
    def handle(
        self,
        apps: t.Annotated[
            t.List[str],
            typer.Argument(
                help="The app label",
                shell_complete=complete_app_label  # pass the completer function here
            )
        ]
    ):
        pass
```

Tip: See the *ModelObjectCompleter* for a completer that works for many Django model field types.

3.3 Debugging Tab Completers

Debugging tab completion code can be tricky because when invoked in situ in the shell the completer code is run as a subprocess and it's output is captured by the shell. This means you can't set a breakpoint and enter into the debugger easily.

To help with this django-typer provides a debug mode that will enter into the tab-completion logic flow. Use the *shell-completion complete()* command, to pass the command line string that you would like to debug. For example:

```
./manage.py shellcompletion complete "mycommand --"
```

CHAPTER

FOUR

REFERENCE

4.1 django_typer

```
    ____ _                                  ______
   /    (_) __ _ _ __    __ _  ____    /__    \_    _ _ __    ____ _ __
  / /\ / |/ _` | '_ \ / _` |/ _ \     / /\/ | | | '_ \ / _ \ '__|
 / /_//| | (_| | | | | (_| | (_) |   / /   | |_| | |_) |  __/ |
/___,'_/ |\__,_|_| |_|\__, |\___/    \/     \__, | .__/ \___|_|
     |__/               |___/                 |___/|_|
```

django-typer provides an extension class, *TyperCommand*, to the BaseCommand class that melds the Typer/click infrastructure with the Django infrastructure. The result is all the ease of specifying commands, groups and options and arguments using Typer and click in a way that feels like and is interface compatible with Django's BaseCommand This should enable a smooth transition for existing Django commands and an intuitive feel for implementing new commands.

django-typer also supports shell completion for bash, zsh, fish and powershell and extends that support to native Django management commands as well.

The goal of django-typer is to provide full Typer style functionality while maintaining compatibility with the Django management command system. This means that the BaseCommand interface is preserved and the Typer interface is added on top of it. This means that this code base is more robust to changes in the Django management command system - because most of the base class functionality is preserved but many Typer and click internals are used directly to achieve this. We rely on robust CI to catch breaking changes upstream.

django_typer.**command**(*name: str | None = None, *, cls: ~typing.Type[~django_typer.TyperCommandWrapper] = <class 'django_typer.TyperCommandWrapper'>, context_settings: ~typing.Dict[~typing.Any, ~typing.Any] | None = None, help: str | None = None, epilog: str | None = None, short_help: str | None = None, options_metavar: str = '[OPTIONS]', add_help_option: bool = True, no_args_is_help: bool = False, hidden: bool = False, deprecated: bool = False, rich_help_panel: str | None = <typer.models.DefaultPlaceholder object>, **kwargs: ~typing.Dict[str, ~typing.Any]*) → Callable[[Callable[[P], R]], Callable[[P], R]]

A function decorator that creates a new command and attaches it to the root command group. This is a passthrough to Typer.command() and the options are the same, except we swap the default command class for our wrapper.

We do not need to decorate handle() functions with this decorator, but if we want to pass options upstream to typer we can:

```
class Command(TyperCommand):

    @command(epilog="This is the epilog for the command.")
    def handle(self):
        ...
```

We can also use the command decorator to define multiple subcommands:

```
class Command(TyperCommand):

    @command()
    def command1(self):
        # execute command1 logic here

    @command(name='command2')
    def other_command(self):
        # arguments passed to the decorator are passed to typer and control
        # various aspects of the command, for instance here we've changed the
        # name of the command to 'command2' from 'other_command'
```

The decorated function is the command function. It may also be invoked directly as a method from an instance of the *TyperCommand* class, see *get_command()*.

Parameters

- **name** – the name of the command (defaults to the name of the decorated function)
- **cls** – the command class to use
- **context_settings** – the context settings to use - see click docs.
- **help** – the help string to use, defaults to the function docstring, if you need the help to be translated you should use the help kwarg instead because docstrings will not be translated.
- **epilog** – the epilog to use in the help output
- **short_help** – the short help to use in the help output
- **options_metavar** – the metavar to use for options in the help output
- **add_help_option** – whether to add the help option to the command
- **no_args_is_help** – whether to show the help if no arguments are provided
- **hidden** – whether to hide the command from help output
- **deprecated** – show a deprecation warning
- **rich_help_panel** – the rich help panel to use - if rich is installed this can be used to group commands into panels in the help output.

django_typer.**get_command**(*command_name: str*, **subcommand: str*, *stdout: IO[str] | None = None*, *stderr: IO[str] | None = None*, *no_color: bool = False*, *force_color: bool = False*) → BaseCommand | MethodType

Get a Django command by its name and instantiate it with the provided options. This will work for subclasses of BaseCommand as well as for *TyperCommand* subclasses. If subcommands are listed for a *TyperCommand*, the method that corresponds to the command name will be returned. This method may then be invoked directly. If no subcommands are listed the command instance will be returned.

Using `get_command` to fetch a command instance and then invoking the instance as a callable is the preferred way to execute *TyperCommand* commands from code. The arguments and options passed to the __call__ method of the command should be fully resolved to their expected parameter types before being passed to the command. The call_command interface also works, but arguments must be unparsed strings and options may be either strings or resolved parameter types. The following is more efficient than call_command.

```
basic = get_command('basic')
result = basic(
    arg1,
```

(continues on next page)

(continued from previous page)

```
    arg2,
    arg3=0.5,
    arg4=1
)
```

Subcommands may be retrieved by passing the subcommand names as additional arguments:

```
divide = get_command('hierarchy', 'math', 'divide')
result = divide(10, 2)
```

Parameters

- **command_name** – the name of the command to get
- **subcommand** – the subcommand to get if any
- **stdout** – the stdout stream to use
- **stderr** – the stderr stream to use
- **no_color** – whether to disable color
- **force_color** – whether to force color

Raises

- **ModuleNotFoundError** – if the command is not found
- **LookupError** – if the subcommand is not found

django_typer.**group**(*name: str | None = <typer.models.DefaultPlaceholder object>, cls: ~typing.Type[~django_typer.TyperGroupWrapper] = <class 'django_typer.TyperGroupWrapper'>, invoke_without_command: bool = <typer.models.DefaultPlaceholder object>, no_args_is_help: bool = <typer.models.DefaultPlaceholder object>, subcommand_metavar: str | None = <typer.models.DefaultPlaceholder object>, chain: bool = <typer.models.DefaultPlaceholder object>, result_callback: ~typing.Callable[[...], ~typing.Any] | None = <typer.models.DefaultPlaceholder object>, context_settings: ~typing.Dict[~typing.Any, ~typing.Any] | None = <typer.models.DefaultPlaceholder object>, help: str | None = <typer.models.DefaultPlaceholder object>, epilog: str | None = <typer.models.DefaultPlaceholder object>, short_help: str | None = <typer.models.DefaultPlaceholder object>, options_metavar: str = <typer.models.DefaultPlaceholder object>, add_help_option: bool = <typer.models.DefaultPlaceholder object>, hidden: bool = <typer.models.DefaultPlaceholder object>, deprecated: bool = <typer.models.DefaultPlaceholder object>, rich_help_panel: str | None = <typer.models.DefaultPlaceholder object>, **kwargs: ~typing.Dict[str, ~typing.Any]*) → Callable[[Callable[[...], Any]], *GroupFunction*]

A function decorator that creates a new subgroup and attaches it to the root command group. This is like creating a new Typer app and adding it to a parent Typer app. The kwargs are passed through to the Typer() constructor. The group() functions work like *initialize()* functions for their command groups.

Listing 1: management/commands/example.py

```
from django_typer import TyperCommand, group

class Command(TyperCommand):
```

(continues on next page)

(continued from previous page)

```
    @group()
    def group1(self, flag: bool = False):
        # do group init stuff here

    # to attach a command to the group, use the command() decorator
    # on the group function
    @group1.command()
    def command1(self):
        ...

    # you can also attach subgroups to groups!
    @group1.group()
    def subgroup(self):
        # do subgroup init stuff here

    @subgroup.command()
    def subcommand(self):
        ...
```

These groups and subcommands can be invoked from the command line like so:

```
$ ./manage.py example group1 --flag command1
$ ./manage.py example group1 --flag subgroup subcommand
```

Parameters

- **name** – the name of the group (defaults to the name of the decorated function)
- **cls** – the group class to use
- **invoke_without_command** – whether to invoke the group callback if no command was specified.
- **no_args_is_help** – whether to show the help if no arguments are provided
- **subcommand_metavar** – the metavar to use for subcommands in the help output
- **chain** – whether to chain commands, this allows multiple commands from the group to be specified and run in order sequentially in one call from the command line.
- **result_callback** – a callback to invoke with the result of the command
- **context_settings** – the click context settings to use - see click docs.
- **help** – the help string to use, defaults to the function docstring, if you need to translate the help you should use the help kwarg instead because docstrings will not be translated.
- **epilog** – the epilog to use in the help output
- **short_help** – the short help to use in the help output
- **options_metavar** – the metavar to use for options in the help output
- **add_help_option** – whether to add the help option to the command
- **hidden** – whether to hide this group from the help output
- **deprecated** – show a deprecation warning
- **rich_help_panel** – the rich help panel to use - if rich is installed this can be used to group commands into panels in the help output.

django_typer.**initialize**(*name: str | None = <typer.models.DefaultPlaceholder object>, *, cls: ~typing.Type[~django_typer.TyperGroupWrapper] = <class 'django_typer.TyperGroupWrapper'>, invoke_without_command: bool = <typer.models.DefaultPlaceholder object>, no_args_is_help: bool = <typer.models.DefaultPlaceholder object>, subcommand_metavar: str | None = <typer.models.DefaultPlaceholder object>, chain: bool = <typer.models.DefaultPlaceholder object>, result_callback: ~typing.Callable[[...], ~typing.Any] | None = <typer.models.DefaultPlaceholder object>, context_settings: ~typing.Dict[~typing.Any, ~typing.Any] | None = <typer.models.DefaultPlaceholder object>, help: str | None = <typer.models.DefaultPlaceholder object>, epilog: str | None = <typer.models.DefaultPlaceholder object>, short_help: str | None = <typer.models.DefaultPlaceholder object>, options_metavar: str = <typer.models.DefaultPlaceholder object>, add_help_option: bool = <typer.models.DefaultPlaceholder object>, hidden: bool = <typer.models.DefaultPlaceholder object>, deprecated: bool = <typer.models.DefaultPlaceholder object>, rich_help_panel: str | None = <typer.models.DefaultPlaceholder object>, **kwargs: ~typing.Dict[str, ~typing.Any]*) → Callable[[Callable[[P], R]], Callable[[P], R]]

A function decorator that creates a Typer callback. This decorator wraps the Typer.callback() functionality. We've renamed it to `initialize()` because `callback()` is to general and not intuitive. Callbacks in Typer are functions that are invoked before a command is invoked and that can accept their own arguments. When an `initialize()` function is supplied to a django *`TyperCommand`* the default Django options will be added as parameters. You can specify these parameters (see *`django_typer.types`*) as arguments on the wrapped function if you wish to receive them - otherwise they will be intercepted by the base class infrastructure and used to their purpose.

The parameters are passed through to Typer.callback()

For example the below command defines two subcommands that both have a common initializer that accepts a –precision parameter option:

Listing 2: management/commands/math.py

```
import typing as t
from typer import Argument, Option
from django_typer import TyperCommand, initialize, command


class Command(TyperCommand):

    precision = 2

    @initialize(help="Do some math at the given precision.")
    def init(
        self,
        precision: t.Annotated[
            int, Option(help="The number of decimal places to output.")
        ] = precision,
    ):
        self.precision = precision

    @command(help="Multiply the given numbers.")
    def multiply(
        self,
        numbers: t.Annotated[
            t.List[float], Argument(help="The numbers to multiply")
```

(continues on next page)

(continued from previous page)

```
        ],
    ):
        ...

    @command()
    def divide(
        self,
        numerator: t.Annotated[float, Argument(help="The numerator")],
        denominator: t.Annotated[float, Argument(help="The denominator")]
    ):
        ...
```

When we run, the command we should provide the –precision option before the subcommand:

```
$ ./manage.py math --precision 5 multiply 2 2.333
4.66600
```

Parameters

- **name** – the name of the callback (defaults to the name of the decorated function)
- **cls** – the command class to use - (the initialize() function is technically the root command group)
- **invoke_without_command** – whether to invoke the callback if no command was specified.
- **no_args_is_help** – whether to show the help if no arguments are provided
- **subcommand_metavar** – the metavar to use for subcommands in the help output
- **chain** – whether to chain commands, this allows multiple commands from the group to be specified and run in order sequentially in one call from the command line.
- **result_callback** – a callback to invoke with the result of the command
- **context_settings** – the click context settings to use - see click docs.
- **help** – the help string to use, defaults to the function docstring, if you need to translate the help you should use the help kwarg instead because docstrings will not be translated.
- **epilog** – the epilog to use in the help output
- **short_help** – the short help to use in the help output
- **options_metavar** – the metavar to use for options in the help output
- **add_help_option** – whether to add the help option to the command
- **hidden** – whether to hide this group from the help output
- **deprecated** – show a deprecation warning
- **rich_help_panel** – the rich help panel to use - if rich is installed this can be used to group commands into panels in the help output.

django_typer.**model_parser_completer**(*model_cls: Type[Model]*, *lookup_field: str | None = None*, *case_insensitive: bool = False*, *help_field: str | None = None*, *query: Callable[[ModelObjectCompleter, Context, Parameter, str], Q] | None = None*, *limit: int | None = 50*, *distinct: bool = True*, *on_error: Callable[[Type[Model], str, Exception], None] | None = None*) → Dict[str, Any]

A factory function that returns a dictionary that can be used to specify a parser and completer for a typer.Option or typer.Argument. This is a convenience function that can be used to specify the parser and completer for a model object in one go.

```
def handle(
    self,
    obj: t.Annotated[
        ModelClass,
        typer.Argument(
            **model_parser_completer(ModelClass, 'field_name'),
            help=_("Fetch objects by their field_names.")
        ),
    ]
):
    ...
```

Parameters

- **model_cls** – the model class to use for lookup
- **lookup_field** – the field to use for lookup, by default the primary key
- **case_insensitive** – whether to perform case insensitive lookups and completions, default: False
- **help_field** – the field to use for help output in completion suggestions, by default no help will be provided
- **query** – a callable that will be used to build the query for completions, by default the query will be reasonably determined by the field type
- **limit** – the maximum number of completions to return, default: 50
- **distinct** – whether to filter out already provided parameters in the completion suggestions, True by default
- **on_error** – a callable that will be called if the parser lookup fails to produce a matching object - by default a CommandError will be raised

class django_typer.**TyperCommand**(*stdout: TextIO | None = None*, *stderr: TextIO | None = None*, *no_color: bool = False*, *force_color: bool = False*, ***kwargs: Dict[str, Any]*)

An extension of BaseCommand that uses the Typer library to parse arguments and options. This class adapts BaseCommand using a light touch that relies on most of the original BaseCommand implementation to handle default arguments and behaviors.

All of the documented BaseCommand functionality works as expected. call_command also works as expected. TyperCommands however add a few extra features:

- We define arguments and options using concise and optionally annotated type hints.
- Simple TyperCommands implemented only using handle() can be called directly by invoking the command as a callable.
- We can define arbitrarily complex subcommand group hierarchies using the *group()* and *command()* decorators.
- Commands and subcommands can be fetched and invoked directly as functions using *get_command()*
- We can define common initialization logic for groups of commands using *initialize()*
- TyperCommands may safely return non-string values from handle()

Defining a typer command is a lot like defining a BaseCommand except that we do not have an add_arguments() method. Instead we define the parameters using type hints directly on handle():

```
import typing as t
from django_typer import TyperCommand

class Command(TyperCommand):

    def handle(
        self,
        arg: str,
        option: t.Optional[str] = None
    ):
        # do command logic here
```

TyperCommands can be extremely simple like above, or we can create really complex command group hierarchies with subcommands and subgroups (see *group()* and *command()*).

Typer apps can be configured with a number of parameters to control behavior such as exception behavior, help output, help markup interpretation, result processing and execution flow. These parameters can be passed to typer as keyword arguments in your Command class inheritance:

Listing 3: management/commands/chain.py

```
import typing as t
from django_typer import TyperCommand, command


class Command(TyperCommand, rich_markup_mode='markdown', chain=True):

    suppressed_base_arguments = [
        '--verbosity', '--traceback', '--no-color', '--force-color',
        '--skip_checks', '--settings', '--pythonpath', '--version'
    ]

    @command()
    def command1(self, option: t.Optional[str] = None):
        """This is a *markdown* help string"""
        print('command1')
        return option

    @command()
    def command2(self, option: t.Optional[str] = None):
        """This is a *markdown* help string"""
        print('command2')
        return option
```

We're doing a number of things here:

- Using the *command()* decorator to define multiple subcommands.
- Using the suppressed_base_arguments attribute to suppress the default options Django adds to the command interface.
- Using the rich_markup_mode parameter to enable markdown rendering in help output.
- Using the chain parameter to enable command chaining.

We can see that our help renders like so:

And we can see the chain behavior by calling our command(s) like so:

```
$ ./manage.py chain command1 --option one command2 --option two
command1
command2
['one', 'two']
```

See *TyperCommandMeta* for the list of accepted parameters. Also refer to the Typer docs for more information on the behaviors expected for those parameters - they are passed through to the Typer class constructor. Not all parameters may make sense in the context of a django command.

Parameters

- **stdout** – the stdout stream to use
- **stderr** – the stderr stream to use
- **no_color** – whether to disable color output
- **force_color** – whether to force color output even if the stream is not a tty

class django_typer.**TyperCommandMeta**(*name, bases, attrs, cls: ~typing.Type[~typer.core.TyperGroup] | None = <class 'django_typer.TyperGroupWrapper'>, invoke_without_command: bool = <typer.models.DefaultPlaceholder object>, no_args_is_help: bool = <typer.models.DefaultPlaceholder object>, subcommand_metavar: str | None = <typer.models.DefaultPlaceholder object>, chain: bool = <typer.models.DefaultPlaceholder object>, result_callback: ~typing.Callable[[...], ~typing.Any] | None = <typer.models.DefaultPlaceholder object>, context_settings: ~typing.Dict[~typing.Any, ~typing.Any] | None = <typer.models.DefaultPlaceholder object>, callback: ~typing.Callable[[...], ~typing.Any] | None = <typer.models.DefaultPlaceholder object>, help: str | None = <typer.models.DefaultPlaceholder object>, epilog: str | None = <typer.models.DefaultPlaceholder object>, short_help: str | None = <typer.models.DefaultPlaceholder object>, options_metavar: str = <typer.models.DefaultPlaceholder object>, add_help_option: bool = <typer.models.DefaultPlaceholder object>, hidden: bool = <typer.models.DefaultPlaceholder object>, deprecated: bool = <typer.models.DefaultPlaceholder object>, rich_markup_mode: ~typing.Literal['markdown', 'rich', None] = None, rich_help_panel: str | None = <typer.models.DefaultPlaceholder object>, pretty_exceptions_enable: ~typer.models.DefaultPlaceholder | bool = <typer.models.DefaultPlaceholder object>, pretty_exceptions_show_locals: ~typer.models.DefaultPlaceholder | bool = <typer.models.DefaultPlaceholder object>, pretty_exceptions_short: ~typer.models.DefaultPlaceholder | bool = <typer.models.DefaultPlaceholder object>*)

The metaclass used to build the TyperCommand class. This metaclass is responsible for building Typer app using the arguments supplied to the TyperCommand constructor. It also discovers if handle() was supplied as the single command implementation.

Warning: This metaclass is private because it may change substantially in the future to support changes in the upstream libraries. The TyperCommand interface should be considered the stable interface.

Parameters

- **name** – the name of the class being created
- **bases** – the base classes of the class being created
- **attrs** – the attributes of the class being created
- **cls** – The class to use as the core typer group wrapper
- **invoke_without_command** – whether to invoke the group callback if no command was specified.
- **no_args_is_help** – whether to show the help if no arguments are provided
- **subcommand_metavar** – the metavar to use for subcommands in the help output
- **chain** – whether to chain commands, this allows multiple commands from the group to be specified and run in order sequentially in one call from the command line.
- **result_callback** – a callback to invoke with the result of the command
- **context_settings** – the click context settings to use - see click docs.
- **help** – the help string to use, defaults to the function docstring, if you need to translate the help you should use the help kwarg instead because docstrings will not be translated.
- **epilog** – the epilog to use in the help output
- **short_help** – the short help to use in the help output
- **options_metavar** – the metavar to use for options in the help output
- **add_help_option** – whether to add the help option to the command
- **hidden** – whether to hide this group from the help output
- **deprecated** – show a deprecation warning
- **rich_markup_mode** – the rich markup mode to use - if rich is installed this can be used to enable rich markup or Markdown in the help output. Can be “markdown”, “rich” or None to disable markup rendering.
- **rich_help_panel** – the rich help panel to use - if rich is installed this can be used to group commands into panels in the help output.
- **pretty_exceptions_enable** – whether to enable pretty exceptions - if rich is installed this can be used to enable pretty exception rendering. This will default to on if the traceback configuration settings installs the rich traceback handler. This allows tracebacks to be configured by the user on a per deployment basis in the settings file. We therefore do not advise hardcoding this value.
- **pretty_exceptions_show_locals** – whether to show local variables in pretty exceptions - if rich is installed. This will default to the ‘show_locals’ setting in the traceback configuration setting (on by default). This allows tracebacks to be configured by the user on a per deployment basis in the settings file. We therefore do not advise hardcoding this value.
- **pretty_exceptions_short** – whether to show short tracebacks in pretty exceptions - if rich is installed. This will default to the ‘short’ setting in the traceback configuration setting

(off by default). This allows tracebacks to be configured by the user on a per deployment basis in the settings file. We therefore do not advise hardcoding this value.

class django_typer.**GroupFunction**(**args*, ***kwargs*)

Typer adds additional groups of commands by adding Typer apps to parent Typer apps. This class extends the `typer.Typer` class so that we can add the additional information necessary to attach this app to the root app and other groups specified on the django command.

command(*name: str | None = None, *, cls: ~typing.Type[~typer.core.TyperCommand] = <class 'django_typer.TyperCommandWrapper'>, context_settings: ~typing.Dict[~typing.Any, ~typing.Any] | None = None, help: str | None = None, epilog: str | None = None, short_help: str | None = None, options_metavar: str = '[OPTIONS]', add_help_option: bool = True, no_args_is_help: bool = False, hidden: bool = False, deprecated: bool = False, rich_help_panel: str | None = <typer.models.DefaultPlaceholder object>, **kwargs: ~typing.Dict[str, ~typing.Any]*) → Callable[[Callable[[P], R]], Callable[[P], R]]

A function decorator that creates a new command and attaches it to this group. This is a passthrough to Typer.command() and the options are the same, except we swap the default command class for our wrapper.

The decorated function is the command function. It may also be invoked directly as a method from an instance of the django command class.

```
class Command(TyperCommand):

    @group()
    def group1(self):
        pass

    @group1.command()
    def command1(self):
        # do stuff here
```

Note: If you need to use a different command class you will need to either inherit from django-typer or make sure yours is interface compatible with our extensions. You shouldn't need to do this though - if the library does not do something you need it to please submit an issue.

Parameters

- **name** – the name of the command (defaults to the name of the decorated function)
- **cls** – the command class to use
- **context_settings** – the context settings to use - see click docs.
- **help** – the help string to use, defaults to the function docstring, if you need the help to be translated you should use the help kwarg instead because docstrings will not be translated.
- **epilog** – the epilog to use in the help output
- **short_help** – the short help to use in the help output
- **options_metavar** – the metavar to use for options in the help output
- **add_help_option** – whether to add the help option to the command
- **no_args_is_help** – whether to show the help if no arguments are provided
- **hidden** – whether to hide the command from help output
- **deprecated** – show a deprecation warning

- **rich_help_panel** – the rich help panel to use - if rich is installed this can be used to group commands into panels in the help output.

group(*name: str | None = <typer.models.DefaultPlaceholder object>, cls: ~typing.Type[~typer.core.TyperGroup] = <class 'django_typer.TyperGroupWrapper'>, invoke_without_command: bool = <typer.models.DefaultPlaceholder object>, no_args_is_help: bool = <typer.models.DefaultPlaceholder object>, subcommand_metavar: str | None = <typer.models.DefaultPlaceholder object>, chain: bool = <typer.models.DefaultPlaceholder object>, result_callback: ~typing.Callable[[...], ~typing.Any] | None = <typer.models.DefaultPlaceholder object>, context_settings: ~typing.Dict[~typing.Any, ~typing.Any] | None = <typer.models.DefaultPlaceholder object>, help: str | None = <typer.models.DefaultPlaceholder object>, epilog: str | None = <typer.models.DefaultPlaceholder object>, short_help: str | None = <typer.models.DefaultPlaceholder object>, options_metavar: str = <typer.models.DefaultPlaceholder object>, add_help_option: bool = <typer.models.DefaultPlaceholder object>, hidden: bool = <typer.models.DefaultPlaceholder object>, deprecated: bool = <typer.models.DefaultPlaceholder object>, rich_help_panel: str | None = <typer.models.DefaultPlaceholder object>, **kwargs: ~typing.Dict[str, ~typing.Any]*) → Callable[[Callable[[...], Any]], *GroupFunction*]

Create a new subgroup and attach it to this group. This is like creating a new Typer app and adding it to a parent Typer app. The kwargs are passed through to the Typer() constructor.

```
class Command(TyperCommand):

    @group()
    def group1(self):
        pass

    @group1.group()
    def subgroup(self):
        # do common group init stuff here

    @subgroup.command(help=_('My command does good stuff!'))
    def subcommand(self):
        # do command stuff here
```

Parameters

- **name** – the name of the group
- **cls** – the group class to use
- **invoke_without_command** – whether to invoke the group callback if no command was specified.
- **no_args_is_help** – whether to show the help if no arguments are provided
- **subcommand_metavar** – the metavar to use for subcommands in the help output
- **chain** – whether to chain commands, this allows multiple commands from the group to be specified and run in order sequentially in one call from the command line.
- **result_callback** – a callback to invoke with the result of the command
- **context_settings** – the click context settings to use - see click docs.
- **help** – the help string to use, defaults to the function docstring, if you need to translate the help you should use the help kwarg instead because docstrings will not be translated.
- **epilog** – the epilog to use in the help output
- **short_help** – the short help to use in the help output

- **options_metavar** – the metavar to use for options in the help output
- **add_help_option** – whether to add the help option to the command
- **hidden** – whether to hide this group from the help output
- **deprecated** – show a deprecation warning
- **rich_help_panel** – the rich help panel to use - if rich is installed this can be used to group commands into panels in the help output.

4.2 types

Common types for command line argument specification.

`django_typer.types.`**`ForceColor`**

The type hint for the Django –force-color option.

The –force-color option is included by default and behaves the same as on BaseCommand use it to force colorization of the command. You can check the supplied value of –force-color by checking the force_color attribute of the command instance.

alias of `Annotated`[`bool`]

`django_typer.types.`**`NoColor`**

The type hint for the Django –no-color option.

The –no-color option is included by default and behaves the same as on BaseCommand use it to force disable colorization of the command. You can check the supplied value of –no-color by checking the no_color attribute of the command instance.

alias of `Annotated`[`bool`]

`django_typer.types.`**`PythonPath`**

The type hint for the Django –pythonpath option.

The –pythonpath option is included by default and behaves the same as on BaseCommand use it to specify a directory to add to the Python sys path.

alias of `Annotated`[`Path` | `None`]

`django_typer.types.`**`Settings`**

The type hint for the Django –settings option.

The –settings option is included by default and behaves the same as on BaseCommand use it to specify or override the settings module to use.

alias of `Annotated`[`str`]

`django_typer.types.`**`SkipChecks`**

The type hint for the Django –skip-checks option.

The –skip-checks option is included by default and behaves the same as on BaseCommand use it to skip system checks.

alias of `Annotated`[`bool`]

django_typer.types.**Traceback**

The type hint for the Django –traceback option.

The –traceback option is included by default and behaves the same as on BaseCommand use it to allow CommandError exceptions to propagate out of the command and produce a stack trace.

alias of Annotated[bool]

django_typer.types.**Verbosity**

The type hint for the Django –verbosity option. *TyperCommand* does not include the verbosity option by default, but it can be added to the command like so if needed.

```
from django_typer.types import Verbosity

def handle(self, verbosity: Verbosity = 1):
    ...
```

alias of Annotated[int]

django_typer.types.**Version**

The type hint for the Django –version option.

The –version option is included by default and behaves the same as on BaseCommand.

alias of Annotated[bool]

django_typer.types.**print_version**(*context*, *_*, *value*)

A callback to run the get_version() routine of the command when –version is specified.

django_typer.types.**set_force_color**(*context*, *param*, *value*)

If the value was provided set it on the command.

django_typer.types.**set_no_color**(*context*, *param*, *value*)

If the value was provided set it on the command.

4.3 parsers

Typer supports custom parsers for options and arguments. If you would like to type a parameter with a type that isn't supported by Typer you can implement your own parser , or ParamType in click parlance.

This module contains a collection of parsers that turn strings into useful Django types. Pass these parsers to the *parser* argument of typer.Option and typer.Argument. Parsers are provided for:

- **Model Objects**: Turn a string into a model object instance using *ModelObjectParser*.
- **App Labels**: Turn a string into an AppConfig instance using *parse_app_label()*.

Warning:

If you implement a custom parser, please take care to ensure that it:

- Handles the case where the value is already the expected type.
- Returns None if the value is None (already implemented if subclassing ParamType).
- Raises a CommandError if the value is invalid.
- Handles the case where the param and context are None.

class django_typer.parsers.**ModelObjectParser**(*model_cls: Type[Model]*, *lookup_field: str | None = None*, *case_insensitive: bool = False*, *on_error: Callable[[Type[Model], str, Exception], None] | None = None*)

A parser that will turn strings into model object instances based on the configured lookup field and model class.

```
from django_typer.parsers import ModelObjectParser

class Command(TyperCommand):
    def handle(
        self,
        django_apps: Annotated[
            t.List[MyModel],
            typer.Argument(
                parser=ModelObjectParser(MyModel, lookup_field="name"),
                help=_("One or more application labels."),
            ),
        ],
    ):
```

Note: Typer does not respect the shell_complete functions on ParamTypes passed as parsers. To add shell_completion see *ModelObjectCompleter* or the *model_parser_completer()* convenience function.

Parameters

- **model_cls** – The model class to use for lookup.
- **lookup_field** – The field to use for lookup. Defaults to 'pk'.
- **on_error** – A callable that will be called if the lookup fails. The callable should accept three arguments: the model class, the value that failed to lookup, and the exception that was raised. If not provided, a CommandError will be raised.

convert(*value: Any*, *param: Parameter | None*, *ctx: Context | None*)

Invoke the parsing action on the given string. If the value is already a model instance of the expected type the value will be returned. Otherwise the value will be treated as a value to query against the lookup_field. If no model object is found the error handler is invoked if one was provided.

Parameters

- **value** – The value to parse.
- **param** – The parameter that the value is associated with.
- **ctx** – The context of the command.

Raises

CommandError – If the lookup fails and no error handler is provided.

django_typer.parsers.**parse_app_label**(*label: str | AppConfig*)

A parser for app labels. If the label is already an AppConfig instance, the instance is returned. The label will be tried first, if that fails the label will be treated as the app name.

```
import typing as t
import typer
from django_typer import TyperCommand
```

(continues on next page)

(continued from previous page)

```
from django_typer.parsers import parse_app_label

class Command(TyperCommand):

    def handle(
        self,
        django_apps: t.Annotated[
            t.List[AppConfig],
            typer.Argument(
                parser=parse_app_label,
                help=_("One or more application labels.")
            )
        ]
    ):
        ...
```

Parameters
label – The label to map to an AppConfig instance.

Raises
CommandError – If no matching app can be found.

4.4 completers

Typer and click provide tab-completion hooks for individual parameters. As with *parsers* custom completion logic can be implemented for custom parameter types and added to the annotation of the parameter. Previous versions of Typer supporting click 7 used the autocompletion argument to provide completion logic, Typer still supports this, but passing `shell_complete` to the annotation is the preferred way to do this.

This module provides some completer functions and classes that work with common Django types:

- **Model Objects**: Complete model object field strings using *ModelObjectCompleter*.
- **App Labels**: Complete app labels or names using *complete_app_label()*.

class django_typer.completers.**ModelObjectCompleter**(*model_cls: Type[Model]*, *lookup_field: str | None = None*, *help_field: str | None = None*, *query: Callable[[ModelObjectCompleter, Context, Parameter, str], Q] | None = None*, *limit: int | None = 50*, *case_insensitive: bool = False*, *distinct: bool = True*)

A completer for generic Django model objects. This completer will work for most Django core model field types where completion makes sense.

This completer currently supports the following field types and their subclasses:

- **IntegerField**
  - AutoField
  - BigAutoField
  - BigIntegerField
  - SmallIntegerField

  - PositiveIntegerField
  - PositiveSmallIntegerField
  - SmallAutoField
- **CharField**
  - SlugField
  - URLField
  - EmailField
- TextField
- UUIDField
- FloatField
- DecimalField

The completer query logic is pluggable, but the defaults cover most use cases. The limit field is important. It defaults to 50 meaning if more than 50 potential completions are found only the first 50 will be returned and there will be no indication to the user that there are more. This is to prevent the shell from becoming unresponsive when offering completion for large tables.

To use this completer, pass an instance of this class to the *shell_complete* argument of a typer.Option or typer.Argument:

```
from django_typer.completers import ModelObjectCompleter

class Command(TyperCommand):

    def handle(
        self,
        model_obj: Annotated[
            MyModel,
            typer.Argument(
                shell_complete=ModelObjectCompleter(MyModel, lookup_field="name"),
                help=_("The model object to use.")
            )
        ]
    ):
        ...
```

Note: See also *model_parser_completer()* for a convenience function that returns a configured parser and completer for a model object and helps reduce boilerplate.

Parameters

- **model_cls** – The Django model class to query.
- **lookup_field** – The name of the model field to use for lookup.
- **help_field** – The name of the model field to use for help text or None if no help text should be provided.
- **query** – A callable that accepts the completer object instance, the click context, the click parameter, and the incomplete string and returns a Q object to use for filtering the queryset.

The default query will use the relevant class methods depending on the lookup field class. See the query methods for details.

- **limit** – The maximum number of completion items to return. If None, all matching items will be returned. When offering completion for large tables you'll want to set this to a reasonable limit. Default: 50
- **case_insensitive** – Whether or not to perform case insensitive matching when completing text-based fields. Defaults to False.
- **distinct** – Whether or not to filter out duplicate values. Defaults to True. This is not the same as calling distinct() on the queryset - which will happen regardless - but rather whether or not to filter out values that are already given for the parameter on the command line.

float_query(*context: Context*, *parameter: Parameter*, *incomplete: str*) → Q

The default completion query builder for float fields. This method will return a Q object that will match any value that starts with the incomplete string. For example, if the incomplete string is "1.1", the query will match 1.1 <= float(incomplete) < 1.2

Parameters

- **context** – The click context.
- **parameter** – The click parameter.
- **incomplete** – The incomplete string.

Returns
A Q object to use for filtering the queryset.

Raises

- **ValueError** – If the incomplete string is not a valid float.
- **TypeError** – If the incomplete string is not a valid float.

int_query(*context: Context*, *parameter: Parameter*, *incomplete: str*) → Q

The default completion query builder for integer fields. This method will return a Q object that will match any value that starts with the incomplete string. For example, if the incomplete string is "1", the query will match 1, 10-19, 100-199, 1000-1999, etc.

Parameters

- **context** – The click context.
- **parameter** – The click parameter.
- **incomplete** – The incomplete string.

Returns
A Q object to use for filtering the queryset.

Raises

- **ValueError** – If the incomplete string is not a valid integer.
- **TypeError** – If the incomplete string is not a valid integer.

text_query(*context: Context*, *parameter: Parameter*, *incomplete: str*) → Q

The default completion query builder for text-based fields. This method will return a Q object that will match any value that starts with the incomplete string. Case sensitivity is determined by the case_insensitive constructor parameter.

Parameters

- **context** – The click context.
- **parameter** – The click parameter.
- **incomplete** – The incomplete string.

Returns
A Q object to use for filtering the queryset.

uuid_query(*context: Context*, *parameter: Parameter*, *incomplete: str*) → Q

The default completion query builder for UUID fields. This method will return a Q object that will match any value that starts with the incomplete string. The incomplete string will be stripped of all non-alphanumeric characters and padded with zeros to 32 characters. For example, if the incomplete string is "a", the query will match a0000000-0000-0000-0000-000000000000 to affffffff-ffff-ffff-ffff-ffffffffff.

Parameters

- **context** – The click context.
- **parameter** – The click parameter.
- **incomplete** – The incomplete string.

Returns
A Q object to use for filtering the queryset.

Raises
ValueError – If the incomplete string is too long or contains invalid UUID characters. Anything other than (0-9a-fA-F).

django_typer.completers.**complete_app_label**(*ctx: Context*, *param: Parameter*, *incomplete: str*) → List[CompletionItem]

A case-sensitive completer for Django app labels or names. The completer prefers labels but names will also work.

```
import typing as t
import typer
from django_typer import TyperCommand
from django_typer.parsers import parse_app_label
from django_typer.completers import complete_app_label

class Command(TyperCommand):

    def handle(
        self,
        django_apps: t.Annotated[
            t.List[AppConfig],
            typer.Argument(
                parser=parse_app_label,
                shell_complete=complete_app_label,
                help=_("One or more application labels.")
            )
        ]
    ):
        ...
```

Parameters

- **ctx** – The click context.
- **param** – The click parameter.
- **incomplete** – The incomplete string.

Returns
A list of matching app labels or names. Labels already present for the parameter on the command line will be filtered out.

4.5 utils

A collection of useful utilities.

django_typer.utils.**get_current_command**() → *TyperCommand* | None

Returns the current typer command. This can be used as a way to access the current command object from anywhere if we are executing inside of one from higher on the stack. We primarily need this because certain monkey patches are required in typer code - namely for enabling/disabling color based on configured parameters.

This function is thread safe.

This is analogous to click's get_current_context but for command execution.

Returns
The current typer command or None if there is no active command.

django_typer.utils.**traceback_config**() → bool | Dict[str, Any]

Fetch the rich traceback installation parameters from our settings. By default rich tracebacks are on with show_locals = True. If the config is set to False or None rich tracebacks will not be installed even if the library is present.

This allows us to have a common traceback configuration for all commands. If rich tracebacks are managed separately this setting can also be switched off.

4.6 shellcompletion

The shellcompletion command is a Django management command that installs and removes shellcompletion scripts for supported shells (bash, fish, zsh, powershell). This command is also the entry point for running the completion logic and can be used to debug completer code.

install() invokes typer's shell completion installation logic, but does have to patch the installed scripts. This is because there is only one installation for all Django management commands, not each individual command. The completion logic here will failover to Django's builtin autocomplete if the command in question is not a *TyperCommand*. To promote compatibility with other management command libraries or custom completion logic, a fallback completion function can also be specified.

class django_typer.management.commands.shellcompletion.**Command**(*stdout: TextIO | None = None*, *stderr: TextIO | None = None*, *no_color: bool = False*, *force_color: bool = False*, ***kwargs: Dict[str, Any]*)

This command installs autocompletion for the current shell. This command uses the typer/click autocompletion scripts to generate the autocompletion items, but monkey patches the scripts to invoke our bundled shell complete script which fails over to the django autocomplete function when the command being completed is not a *TyperCommand*. When the django autocomplete function is used we also wrap it so that it works for any supported click/typer shell, not just bash.

We also provide a remove command to easily remove the installed script.

Great pains are taken to use the upstream dependency's shell completion logic. This is so advances and additional shell support implemented upstream should just work. However, it would be possible to add support for new shells here using the pluggable logic that click provides. It is probably a better idea however to add support for new shells at the typer level.

Shell autocompletion can be brittle with every shell having its own quirks and nuances. We make a good faith effort here to support all the shells that typer/click support, but there can easily be system specific configuration issues that prevent this from working. In those cases users should refer to the online documentation for their specific shell to troubleshoot.

complete (*cmd_str: str | None = None*, *shell: Shells | None = None*, *fallback: str | None = None*)

We implement the shell complete generation script as a Django command because the Django environment needs to be bootstrapped for it to work. This also allows us to test completion logic in a platform agnostic way.

Tip: This command is super useful for debugging shell_complete logic. For example to enter into the debugger, we could set a breakpoint in our `shell_complete` function for the option parameter and then run the following command:

```
$ ./manage.py shellcompletion complete "./manage.py your_command --option "
```

---

django_fallback ()

Run django's builtin bash autocomplete function. We wrap the click completion class to make it work for all supported shells, not just bash.

install (*shell: Shells | None = 'sh'*, *manage_script: str | None = None*, *fallback: str | None = None*)

Install autocompletion for the given shell. If the shell is not specified, it will try to detect the shell. If the shell is not detected, it will fail.

We run the upstream typer installation routines, with some augmentation.

property manage_script: str | Path

Returns the name of the manage command as a string if it is available as a command on the user path. If it is a script that is not available as a command on the path it will return an absolute Path object to the script.

property manage_script_name: str

Get the name of the manage script as a command available from the shell's path.

noop ()

This is a no-op command that is used to bootstrap click Completion classes. It has no use other than to avoid any potential attribute access errors when we spoof completion logic

property noop_command

This is a no-op command that is used to bootstrap click Completion classes. It has no use other than to avoid any potential attribute errors when we emulate upstream completion logic

patch_script (*shell: Shells | None = None*, *fallback: str | None = None*) → None

We have to monkey patch the typer completion scripts to point to our custom shell complete script. This is potentially brittle but thats why we have robust CI!

Parameters

- **shell** – The shell to patch the completion script for.
- **fallback** – The python import path to a fallback autocomplete function to use when the completion command is not a TyperCommand. Defaults to None, which means the bundled

complete script will fallback to the django autocomplete function, but wrap it so it works for all supported shells other than just bash.

remove (*shell: Shells | None = 'sh'*, *manage_script: str | None = None*)

Remove the autocompletion for the given shell. If the shell is not specified, it will try to detect the shell. If the shell is not detected, it will fail.

Since the installation routine is upstream we first run install to determine where the completion script is installed and then we remove it.

property shell: Shells

Get the active shell. If not explicitly set, it first tries to find the shell in the environment variable shell complete scripts set and failing that it will try to autodetect the shell.

CHAPTER

FIVE

CHANGE LOG

5.1 v1.1.2

- Fixed Overridden common Django arguments fail to pass through when passed through call_command

5.2 v1.1.1

- Implemented Fix pyright type checking and add to CI
- Implemented Convert CONTRIBUTING.rst to markdown

5.3 v1.1.0

- Implemented Convert readme to markdown.
- Fixed typer 0.12.0 breaks django_typer 1.0.9

5.4 v1.0.9 (yanked)

- Fixed Support typer 0.12.0

5.5 v1.0.8

- Fixed Support typer 0.10 and 0.11

5.6 v1.0.7

- Fixed Helps throw an exception when invoked from an absolute path that is not relative to the getcwd()

5.7 v1.0.6

- Fixed prompt options on groups still prompt when given as named parameters on call_command

5.8 v1.0.5

- Fixed Options with prompt=True are prompted twice

5.9 v1.0.4

- Fixed Help sometimes shows full script path in Usage: when it shouldnt.
- Fixed METAVAR when ModelObjectParser supplied should default to model name

5.10 v1.0.3

- Fixed Incomplete typing info for @command decorator

5.11 v1.0.2

- Fixed name property on TyperCommand is too generic and should be private.
- Fixed When usage errors are thrown the help output should be that of the subcommand invoked not the parent group.
- Fixed typer installs its own system exception hook when commands are run and this may step on the installed rich hook
- Fixed Add py.typed stub
- Fixed Run type checking with django-stubs installed.
- Fixed Add pyright to linting and resolve any pyright errors.
- Fixed Missing subcommand produces stack trace without –traceback.
- Fixed Allow handle() to be an initializer.

5.12 v1.0.1

- Fixed shell_completion broken for click < 8.1

5.13 v1.0.0

- Initial production/stable release.

5.14 v0.6.1b

- Incremental beta release - this is also the second release candidate for version 1.
- Peg typer version to 0.9.x

5.15 v0.6.0b

- Incremental beta release - this is also the first release candidate for version 1.

5.16 v0.5.0b

- Incremental Beta Release

5.17 v0.4.0b

- Incremental Beta Release

5.18 v0.3.0b

- Incremental Beta Release

5.19 v0.2.0b

- Incremental Beta Release

5.20 v0.1.0b

- Initial Release (Beta)

PYTHON MODULE INDEX

d

INDEX

R

S

T

U

V