File 1 - _compat.py:

```
(0)             import codecs
(0)             import io
(0)             import os
(0)             import re
(0)             import sys
(0)             import typing as t
(0)             from weakref import WeakKeyDictionary
(0)             CYGWIN = sys.platform.startswith("cygwin")
(0)             WIN = sys.platform.startswith("win")
(0)             auto_wrap_for_ansi: t.Optional[t.Callable[[t.TextIO], t.TextIO]] = None
(0)             _ansi_re = re.compile(r"\033\[[;?0-9]*[a-zA-Z]")
(0)             def _make_text_stream(
(4)                 stream: t.BinaryIO,
(4)                 encoding: t.Optional[str],
(4)                 errors: t.Optional[str],
(4)                 force_readable: bool = False,
(4)                 force_writable: bool = False,
(0)             ) -> t.TextIO:
(4)                 if encoding is None:
(8)                     encoding = get_best_encoding(stream)
(4)                 if errors is None:
(8)                     errors = "replace"
(4)                 return _NonClosingTextIOWrapper(
(8)                     stream,
(8)                     encoding,
(8)                     errors,
(8)                     line_buffering=True,
(8)                     force_readable=force_readable,
(8)                     force_writable=force_writable,
(4)                 )
(0)             def is_ascii_encoding(encoding: str) -> bool:
(4)                 """Checks if a given encoding is ascii."""
(4)                 try:
(8)                     return codecs.lookup(encoding).name == "ascii"
(4)                 except LookupError:
(8)                     return False
(0)             def get_best_encoding(stream: t.IO[t.Any]) -> str:
(4)                 """Returns the default stream encoding if not found."""
(4)                 rv = getattr(stream, "encoding", None) or sys.getdefaultencoding()
(4)                 if is_ascii_encoding(rv):
(8)                     return "utf-8"
(4)                 return rv
(0)             class _NonClosingTextIOWrapper(io.TextIOWrapper):
(4)                 def __init__(
(8)                     self,
(8)                     stream: t.BinaryIO,
(8)                     encoding: t.Optional[str],
(8)                     errors: t.Optional[str],
(8)                     force_readable: bool = False,
(8)                     force_writable: bool = False,
(8)                     **extra: t.Any,
(4)                 ) -> None:
(8)                     self._stream = stream = t.cast(
(12)                        t.BinaryIO, _FixupStream(stream, force_readable, force_writable)
(8)                     )
(8)                     super().__init__(stream, encoding, errors, **extra)
(4)                 def __del__(self) -> None:
(8)                     try:
(12)                        self.detach()
(8)                     except Exception:
(12)                        pass
(4)                 def isatty(self) -> bool:
(8)                     return self._stream.isatty()
(0)             class _FixupStream:
(4)                 """The new io interface needs more from streams than streams
(4)                 traditionally implement.  As such, this fix-up code is necessary in
(4)                 some circumstances.
```

```
(4)         The forcing of readable and writable flags are there because some tools
(4)         put badly patched objects on sys (one such offender are certain version
(4)         of jupyter notebook).
(4)         """
(4)         def __init__(
(8)             self,
(8)             stream: t.BinaryIO,
(8)             force_readable: bool = False,
(8)             force_writable: bool = False,
(4)         ):
(8)             self._stream = stream
(8)             self._force_readable = force_readable
(8)             self._force_writable = force_writable
(4)         def __getattr__(self, name: str) -> t.Any:
(8)             return getattr(self._stream, name)
(4)         def read1(self, size: int) -> bytes:
(8)             f = getattr(self._stream, "read1", None)
(8)             if f is not None:
(12)                return t.cast(bytes, f(size))
(8)             return self._stream.read(size)
(4)         def readable(self) -> bool:
(8)             if self._force_readable:
(12)                return True
(8)             x = getattr(self._stream, "readable", None)
(8)             if x is not None:
(12)                return t.cast(bool, x())
(8)             try:
(12)                self._stream.read(0)
(8)             except Exception:
(12)                return False
(8)             return True
(4)         def writable(self) -> bool:
(8)             if self._force_writable:
(12)                return True
(8)             x = getattr(self._stream, "writable", None)
(8)             if x is not None:
(12)                return t.cast(bool, x())
(8)             try:
(12)                self._stream.write("")  # type: ignore
(8)             except Exception:
(12)                try:
(16)                    self._stream.write(b"")
(12)                except Exception:
(16)                    return False
(8)             return True
(4)         def seekable(self) -> bool:
(8)             x = getattr(self._stream, "seekable", None)
(8)             if x is not None:
(12)                return t.cast(bool, x())
(8)             try:
(12)                self._stream.seek(self._stream.tell())
(8)             except Exception:
(12)                return False
(8)             return True
(0)     def _is_binary_reader(stream: t.IO[t.Any], default: bool = False) -> bool:
(4)         try:
(8)             return isinstance(stream.read(0), bytes)
(4)         except Exception:
(8)             return default
(0)     def _is_binary_writer(stream: t.IO[t.Any], default: bool = False) -> bool:
(4)         try:
(8)             stream.write(b"")
(4)         except Exception:
(8)             try:
(12)                stream.write("")
(12)                return False
(8)             except Exception:
(12)                pass
(8)             return default
```

```
(4)                return True
(0)            def _find_binary_reader(stream: t.IO[t.Any]) -> t.Optional[t.BinaryIO]:
(4)                if _is_binary_reader(stream, False):
(8)                    return t.cast(t.BinaryIO, stream)
(4)                buf = getattr(stream, "buffer", None)
(4)                if buf is not None and _is_binary_reader(buf, True):
(8)                    return t.cast(t.BinaryIO, buf)
(4)                return None
(0)            def _find_binary_writer(stream: t.IO[t.Any]) -> t.Optional[t.BinaryIO]:
(4)                if _is_binary_writer(stream, False):
(8)                    return t.cast(t.BinaryIO, stream)
(4)                buf = getattr(stream, "buffer", None)
(4)                if buf is not None and _is_binary_writer(buf, True):
(8)                    return t.cast(t.BinaryIO, buf)
(4)                return None
(0)            def _stream_is_misconfigured(stream: t.TextIO) -> bool:
(4)                """A stream is misconfigured if its encoding is ASCII."""
(4)                return is_ascii_encoding(getattr(stream, "encoding", None) or "ascii")
(0)            def _is_compat_stream_attr(stream: t.TextIO, attr: str, value: t.Optional[str]) -> bool:
(4)                """A stream attribute is compatible if it is equal to the
(4)                desired value or the desired value is unset and the attribute
(4)                has a value.
(4)                """
(4)                stream_value = getattr(stream, attr, None)
(4)                return stream_value == value or (value is None and stream_value is not None)
(0)            def _is_compatible_text_stream(
(4)                stream: t.TextIO, encoding: t.Optional[str], errors: t.Optional[str]
(0)            ) -> bool:
(4)                """Check if a stream's encoding and errors attributes are
(4)                compatible with the desired values.
(4)                """
(4)                return _is_compat_stream_attr(
(8)                    stream, "encoding", encoding
(4)                ) and _is_compat_stream_attr(stream, "errors", errors)
(0)            def _force_correct_text_stream(
(4)                text_stream: t.IO[t.Any],
(4)                encoding: t.Optional[str],
(4)                errors: t.Optional[str],
(4)                is_binary: t.Callable[[t.IO[t.Any], bool], bool],
(4)                find_binary: t.Callable[[t.IO[t.Any]], t.Optional[t.BinaryIO]],
(4)                force_readable: bool = False,
(4)                force_writable: bool = False,
(0)            ) -> t.TextIO:
(4)                if is_binary(text_stream, False):
(8)                    binary_reader = t.cast(t.BinaryIO, text_stream)
(4)                else:
(8)                    text_stream = t.cast(t.TextIO, text_stream)
(8)                    if _is_compatible_text_stream(text_stream, encoding, errors) and not (
(12)                       encoding is None and _stream_is_misconfigured(text_stream)
(8)                    ):
(12)                       return text_stream
(8)                    possible_binary_reader = find_binary(text_stream)
(8)                    if possible_binary_reader is None:
(12)                       return text_stream
(8)                    binary_reader = possible_binary_reader
(4)                if errors is None:
(8)                    errors = "replace"
(4)                return _make_text_stream(
(8)                    binary_reader,
(8)                    encoding,
(8)                    errors,
(8)                    force_readable=force_readable,
(8)                    force_writable=force_writable,
(4)                )
(0)            def _force_correct_text_reader(
(4)                text_reader: t.IO[t.Any],
(4)                encoding: t.Optional[str],
204: (4)                    errors: t.Optional[str],
205: (4)                    force_readable: bool = False,
206: (0)                ) -> t.TextIO:
207: (4)                    return _force_correct_text_stream(
208: (8)                        text_reader,
209: (8)                        encoding,
210: (8)                        errors,
211: (8)                        _is_binary_reader,
212: (8)                        _find_binary_reader,
213: (8)                        force_readable=force_readable,
214: (4)                    )
215: (0)                def _force_correct_text_writer(
216: (4)                    text_writer: t.IO[t.Any],
217: (4)                    encoding: t.Optional[str],
218: (4)                    errors: t.Optional[str],
219: (4)                    force_writable: bool = False,
220: (0)                ) -> t.TextIO:
221: (4)                    return _force_correct_text_stream(
222: (8)                        text_writer,
223: (8)                        encoding,
224: (8)                        errors,
225: (8)                        _is_binary_writer,
226: (8)                        _find_binary_writer,
227: (8)                        force_writable=force_writable,
228: (4)                    )
229: (0)                def get_binary_stdin() -> t.BinaryIO:
230: (4)                    reader = _find_binary_reader(sys.stdin)
231: (4)                    if reader is None:
232: (8)                        raise RuntimeError("Was not able to determine binary stream for sys.stdin.")
233: (4)                    return reader
234: (0)                def get_binary_stdout() -> t.BinaryIO:
235: (4)                    writer = _find_binary_writer(sys.stdout)
236: (4)                    if writer is None:
237: (8)                        raise RuntimeError("Was not able to determine binary stream for sys.stdout.")
238: (4)                    return writer
239: (0)                def get_binary_stderr() -> t.BinaryIO:
240: (4)                    writer = _find_binary_writer(sys.stderr)
241: (4)                    if writer is None:
242: (8)                        raise RuntimeError("Was not able to determine binary stream for sys.stderr.")
243: (4)                    return writer
244: (0)                def get_text_stdin(
245: (4)                    encoding: t.Optional[str] = None, errors: t.Optional[str] = None
246: (0)                ) -> t.TextIO:
247: (4)                    rv = _get_windows_console_stream(sys.stdin, encoding, errors)
248: (4)                    if rv is not None:
249: (8)                        return rv
250: (4)                    return _force_correct_text_reader(sys.stdin, encoding, errors, force_readable=True)
251: (0)                def get_text_stdout(
252: (4)                    encoding: t.Optional[str] = None, errors: t.Optional[str] = None
253: (0)                ) -> t.TextIO:
254: (4)                    rv = _get_windows_console_stream(sys.stdout, encoding, errors)
255: (4)                    if rv is not None:
256: (8)                        return rv
257: (4)                    return _force_correct_text_writer(sys.stdout, encoding, errors, force_writable=True)
258: (0)                def get_text_stderr(
259: (4)                    encoding: t.Optional[str] = None, errors: t.Optional[str] = None
260: (0)                ) -> t.TextIO:
261: (4)                    rv = _get_windows_console_stream(sys.stderr, encoding, errors)
262: (4)                    if rv is not None:
263: (8)                        return rv
264: (4)                    return _force_correct_text_writer(sys.stderr, encoding, errors, force_writable=True)
265: (0)                def _wrap_io_open(
266: (4)                    file: t.Union[str, "os.PathLike[str]", int],
(4)                 mode: str,
(4)                 encoding: t.Optional[str],
(4)                 errors: t.Optional[str],
(0)             ) -> t.IO[t.Any]:
(4)                 """Handles not passing ``encoding`` and ``errors`` in binary mode."""
(4)                 if "b" in mode:
(8)                     return open(file, mode)
(4)                 return open(file, mode, encoding=encoding, errors=errors)
(0)             def open_stream(
(4)                 filename: "t.Union[str, os.PathLike[str]]",
(4)                 mode: str = "r",
(4)                 encoding: t.Optional[str] = None,
(4)                 errors: t.Optional[str] = "strict",
(4)                 atomic: bool = False,
(0)             ) -> t.Tuple[t.IO[t.Any], bool]:
(4)                 binary = "b" in mode
(4)                 filename = os.fspath(filename)
(4)                 if os.fsdecode(filename) == "-":
(8)                     if any(m in mode for m in ["w", "a", "x"]):
(12)                        if binary:
(16)                            return get_binary_stdout(), False
(12)                        return get_text_stdout(encoding=encoding, errors=errors), False
(8)                     if binary:
(12)                        return get_binary_stdin(), False
(8)                     return get_text_stdin(encoding=encoding, errors=errors), False
(4)                 if not atomic:
(8)                     return _wrap_io_open(filename, mode, encoding, errors), True
(4)                 if "a" in mode:
(8)                     raise ValueError(
(12)                        "Appending to an existing file is not supported, because that"
(12)                        " would involve an expensive `copy`-operation to a temporary"
(12)                        " file. Open the file in normal `w`-mode and copy explicitly"
(12)                        " if that's what you're after."
(8)                     )
(4)                 if "x" in mode:
(8)                     raise ValueError("Use the `overwrite`-parameter instead.")
(4)                 if "w" not in mode:
(8)                     raise ValueError("Atomic writes only make sense with `w`-mode.")
(4)                 import errno
(4)                 import random
(4)                 try:
(8)                     perm: t.Optional[int] = os.stat(filename).st_mode
(4)                 except OSError:
(8)                     perm = None
(4)                 flags = os.O_RDWR | os.O_CREAT | os.O_EXCL
(4)                 if binary:
(8)                     flags |= getattr(os, "O_BINARY", 0)
(4)                 while True:
(8)                     tmp_filename = os.path.join(
(12)                        os.path.dirname(filename),
(12)                        f".__atomic-write{random.randrange(1 << 32):08x}",
(8)                     )
(8)                     try:
(12)                        fd = os.open(tmp_filename, flags, 0o666 if perm is None else perm)
(12)                        break
(8)                     except OSError as e:
(12)                        if e.errno == errno.EEXIST or (
(16)                            os.name == "nt"
(16)                            and e.errno == errno.EACCES
(16)                            and os.path.isdir(e.filename)
(16)                            and os.access(e.filename, os.W_OK)
(12)                        ):
(16)                            continue
(12)                        raise
(4)                 if perm is not None:
(8)                     os.chmod(tmp_filename, perm)  # in case perm includes bits in umask
(4)                 f = _wrap_io_open(fd, mode, encoding, errors)
(4)                 af = _AtomicFile(f, tmp_filename, os.path.realpath(filename))
(4)                 return t.cast(t.IO[t.Any], af), True
```

```
(0)             class _AtomicFile:
(4)                 def __init__(self, f: t.IO[t.Any], tmp_filename: str, real_filename: str)
-> None:
(8)                     self._f = f
(8)                     self._tmp_filename = tmp_filename
(8)                     self._real_filename = real_filename
(8)                     self.closed = False
(4)                 @property
(4)                 def name(self) -> str:
(8)                     return self._real_filename
(4)                 def close(self, delete: bool = False) -> None:
(8)                     if self.closed:
(12)                        return
(8)                     self._f.close()
(8)                     os.replace(self._tmp_filename, self._real_filename)
(8)                     self.closed = True
(4)                 def __getattr__(self, name: str) -> t.Any:
(8)                     return getattr(self._f, name)
(4)                 def __enter__(self) -> "_AtomicFile":
(8)                     return self
(4)                 def __exit__(self, exc_type: t.Optional[t.Type[BaseException]], *_: t.Any)
-> None:
(8)                     self.close(delete=exc_type is not None)
(4)                 def __repr__(self) -> str:
(8)                     return repr(self._f)
(0)             def strip_ansi(value: str) -> str:
(4)                 return _ansi_re.sub("", value)
(0)             def _is_jupyter_kernel_output(stream: t.IO[t.Any]) -> bool:
(4)                 while isinstance(stream, (_FixupStream, _NonClosingTextIOWrapper)):
(8)                     stream = stream._stream
(4)                 return stream.__class__.__module__.startswith("ipykernel.")
(0)             def should_strip_ansi(
(4)                 stream: t.Optional[t.IO[t.Any]] = None, color: t.Optional[bool] = None
(0)             ) -> bool:
(4)                 if color is None:
(8)                     if stream is None:
(12)                        stream = sys.stdin
(8)                     return not isatty(stream) and not _is_jupyter_kernel_output(stream)
(4)                 return not color
(0)             if sys.platform.startswith("win") and WIN:
(4)                 from ._winconsole import _get_windows_console_stream
(4)                 def _get_argv_encoding() -> str:
(8)                     import locale
(8)                     return locale.getpreferredencoding()
(4)                 _ansi_stream_wrappers: t.MutableMapping[t.TextIO, t.TextIO] =
WeakKeyDictionary()
(4)                 def auto_wrap_for_ansi(  # noqa: F811
(8)                     stream: t.TextIO, color: t.Optional[bool] = None
(4)                 ) -> t.TextIO:
(8)                     """Support ANSI color and style codes on Windows by wrapping a
(8)                     stream with colorama.
(8)                     """
(8)                     try:
(12)                        cached = _ansi_stream_wrappers.get(stream)
(8)                     except Exception:
(12)                        cached = None
(8)                     if cached is not None:
(12)                        return cached
(8)                     import colorama
(8)                     strip = should_strip_ansi(stream, color)
(8)                     ansi_wrapper = colorama.AnsiToWin32(stream, strip=strip)
(8)                     rv = t.cast(t.TextIO, ansi_wrapper.stream)
(8)                     _write = rv.write
(8)                     def _safe_write(s):
(12)                        try:
(16)                            return _write(s)
(12)                        except BaseException:
(16)                            ansi_wrapper.reset_all()
(16)                            raise
```

```
        rv.write = _safe_write
        try:
            _ansi_stream_wrappers[stream] = rv
        except Exception:
            pass
        return rv
else:
    def _get_argv_encoding() -> str:
        return getattr(sys.stdin, "encoding", None) or sys.getfilesystemencoding()
    def _get_windows_console_stream(
        f: t.TextIO, encoding: t.Optional[str], errors: t.Optional[str]
    ) -> t.Optional[t.TextIO]:
        return None
def term_len(x: str) -> int:
    return len(strip_ansi(x))
def isatty(stream: t.IO[t.Any]) -> bool:
    try:
        return stream.isatty()
    except Exception:
        return False
def _make_cached_stream_func(
    src_func: t.Callable[[], t.Optional[t.TextIO]],
    wrapper_func: t.Callable[[], t.TextIO],
) -> t.Callable[[], t.Optional[t.TextIO]]:
    cache: t.MutableMapping[t.TextIO, t.TextIO] = WeakKeyDictionary()
    def func() -> t.Optional[t.TextIO]:
        stream = src_func()
        if stream is None:
            return None
        try:
            rv = cache.get(stream)
        except Exception:
            rv = None
        if rv is not None:
            return rv
        rv = wrapper_func()
        try:
            cache[stream] = rv
        except Exception:
            pass
        return rv
    return func
_default_text_stdin = _make_cached_stream_func(lambda: sys.stdin, get_text_stdin)
_default_text_stdout = _make_cached_stream_func(lambda: sys.stdout, get_text_stdout)
_default_text_stderr = _make_cached_stream_func(lambda: sys.stderr, get_text_stderr)
binary_streams: t.Mapping[str, t.Callable[[], t.BinaryIO]] = {
    "stdin": get_binary_stdin,
    "stdout": get_binary_stdout,
    "stderr": get_binary_stderr,
}
text_streams: t.Mapping[
    str, t.Callable[[t.Optional[str], t.Optional[str]], t.TextIO]
] = {
    "stdin": get_text_stdin,
    "stdout": get_text_stdout,
    "stderr": get_text_stderr,
}
```

File 2 - __init__.py:

```
"""
Click is a simple Python module inspired by the stdlib optparse to make
writing command line scripts fun. Unlike other modules, it's based
```

```
around a simple API that does not come with too much magic and is
composable.
"""
from .core import Argument as Argument
from .core import BaseCommand as BaseCommand
from .core import Command as Command
from .core import CommandCollection as CommandCollection
from .core import Context as Context
from .core import Group as Group
from .core import MultiCommand as MultiCommand
from .core import Option as Option
from .core import Parameter as Parameter
from .decorators import argument as argument
from .decorators import command as command
from .decorators import confirmation_option as confirmation_option
from .decorators import group as group
from .decorators import help_option as help_option
from .decorators import make_pass_decorator as make_pass_decorator
from .decorators import option as option
from .decorators import pass_context as pass_context
from .decorators import pass_obj as pass_obj
from .decorators import password_option as password_option
from .decorators import version_option as version_option
from .exceptions import Abort as Abort
from .exceptions import BadArgumentUsage as BadArgumentUsage
from .exceptions import BadOptionUsage as BadOptionUsage
from .exceptions import BadParameter as BadParameter
from .exceptions import ClickException as ClickException
from .exceptions import FileError as FileError
from .exceptions import MissingParameter as MissingParameter
from .exceptions import NoSuchOption as NoSuchOption
from .exceptions import UsageError as UsageError
from .formatting import HelpFormatter as HelpFormatter
from .formatting import wrap_text as wrap_text
from .globals import get_current_context as get_current_context
from .parser import OptionParser as OptionParser
from .termui import clear as clear
from .termui import confirm as confirm
from .termui import echo_via_pager as echo_via_pager
from .termui import edit as edit
from .termui import getchar as getchar
from .termui import launch as launch
from .termui import pause as pause
from .termui import progressbar as progressbar
from .termui import prompt as prompt
from .termui import secho as secho
from .termui import style as style
from .termui import unstyle as unstyle
from .types import BOOL as BOOL
from .types import Choice as Choice
from .types import DateTime as DateTime
from .types import File as File
from .types import FLOAT as FLOAT
from .types import FloatRange as FloatRange
from .types import INT as INT
from .types import IntRange as IntRange
from .types import ParamType as ParamType
from .types import Path as Path
from .types import STRING as STRING
from .types import Tuple as Tuple
from .types import UNPROCESSED as UNPROCESSED
from .types import UUID as UUID
from .utils import echo as echo
from .utils import format_filename as format_filename
from .utils import get_app_dir as get_app_dir
from .utils import get_binary_stream as get_binary_stream
from .utils import get_text_stream as get_text_stream
from .utils import open_file as open_file
__version__ = "8.1.7"
```

File 3 - _textwrap.py:

```python
import textwrap
import typing as t
from contextlib import contextmanager
class TextWrapper(textwrap.TextWrapper):
    def _handle_long_word(
        self,
        reversed_chunks: t.List[str],
        cur_line: t.List[str],
        cur_len: int,
        width: int,
    ) -> None:
        space_left = max(width - cur_len, 1)
        if self.break_long_words:
            last = reversed_chunks[-1]
            cut = last[:space_left]
            res = last[space_left:]
            cur_line.append(cut)
            reversed_chunks[-1] = res
        elif not cur_line:
            cur_line.append(reversed_chunks.pop())
    @contextmanager
    def extra_indent(self, indent: str) -> t.Iterator[None]:
        old_initial_indent = self.initial_indent
        old_subsequent_indent = self.subsequent_indent
        self.initial_indent += indent
        self.subsequent_indent += indent
        try:
            yield
        finally:
            self.initial_indent = old_initial_indent
            self.subsequent_indent = old_subsequent_indent
    def indent_only(self, text: str) -> str:
        rv = []
        for idx, line in enumerate(text.splitlines()):
            indent = self.initial_indent
            if idx > 0:
                indent = self.subsequent_indent
            rv.append(f"{indent}{line}")
        return "\n".join(rv)
```

File 4 - _termui_impl.py:

```python
"""
This module contains implementations for the termui module. To keep the
import time of Click down, some infrequently used functionality is
placed in this module and only imported as needed.
"""
import contextlib
import math
import os
import sys
import time
import typing as t
from gettext import gettext as _
from io import StringIO
from types import TracebackType
from ._compat import _default_text_stdout
from ._compat import CYGWIN
from ._compat import get_best_encoding
from ._compat import isatty
from ._compat import open_stream
from ._compat import strip_ansi
```

```
(0)             from ._compat import term_len
(0)             from ._compat import WIN
(0)             from .exceptions import ClickException
(0)             from .utils import echo
(0)             V = t.TypeVar("V")
(0)             if os.name == "nt":
(4)                 BEFORE_BAR = "\r"
(4)                 AFTER_BAR = "\n"
(0)             else:
(4)                 BEFORE_BAR = "\r\033[?25l"
(4)                 AFTER_BAR = "\033[?25h\n"
(0)             class ProgressBar(t.Generic[V]):
(4)                 def __init__(
(8)                     self,
(8)                     iterable: t.Optional[t.Iterable[V]],
(8)                     length: t.Optional[int] = None,
(8)                     fill_char: str = "#",
(8)                     empty_char: str = " ",
(8)                     bar_template: str = "%(bar)s",
(8)                     info_sep: str = "  ",
(8)                     show_eta: bool = True,
(8)                     show_percent: t.Optional[bool] = None,
(8)                     show_pos: bool = False,
(8)                     item_show_func: t.Optional[t.Callable[[t.Optional[V]], t.Optional[str]]] = None,
(8)                     label: t.Optional[str] = None,
(8)                     file: t.Optional[t.TextIO] = None,
(8)                     color: t.Optional[bool] = None,
(8)                     update_min_steps: int = 1,
(8)                     width: int = 30,
(4)                 ) -> None:
(8)                     self.fill_char = fill_char
(8)                     self.empty_char = empty_char
(8)                     self.bar_template = bar_template
(8)                     self.info_sep = info_sep
(8)                     self.show_eta = show_eta
(8)                     self.show_percent = show_percent
(8)                     self.show_pos = show_pos
(8)                     self.item_show_func = item_show_func
(8)                     self.label: str = label or ""
(8)                     if file is None:
(12)                        file = _default_text_stdout()
(12)                        if file is None:
(16)                            file = StringIO()
(8)                     self.file = file
(8)                     self.color = color
(8)                     self.update_min_steps = update_min_steps
(8)                     self._completed_intervals = 0
(8)                     self.width: int = width
(8)                     self.autowidth: bool = width == 0
(8)                     if length is None:
(12)                        from operator import length_hint
(12)                        length = length_hint(iterable, -1)
(12)                        if length == -1:
(16)                            length = None
(8)                     if iterable is None:
(12)                        if length is None:
(16)                            raise TypeError("iterable or length is required")
(12)                        iterable = t.cast(t.Iterable[V], range(length))
(8)                     self.iter: t.Iterable[V] = iter(iterable)
(8)                     self.length = length
(8)                     self.pos = 0
(8)                     self.avg: t.List[float] = []
(8)                     self.last_eta: float
(8)                     self.start: float
(8)                     self.start = self.last_eta = time.time()
(8)                     self.eta_known: bool = False
(8)                     self.finished: bool = False
(8)                     self.max_width: t.Optional[int] = None
```

```
(8)             self.entered: bool = False
(8)             self.current_item: t.Optional[V] = None
(8)             self.is_hidden: bool = not isatty(self.file)
(8)             self._last_line: t.Optional[str] = None
(4)         def __enter__(self) -> "ProgressBar[V]":
(8)             self.entered = True
(8)             self.render_progress()
(8)             return self
(4)         def __exit__(
(8)             self,
(8)             exc_type: t.Optional[t.Type[BaseException]],
(8)             exc_value: t.Optional[BaseException],
(8)             tb: t.Optional[TracebackType],
(4)         ) -> None:
(8)             self.render_finish()
(4)         def __iter__(self) -> t.Iterator[V]:
(8)             if not self.entered:
(12)                raise RuntimeError("You need to use progress bars in a with
block.")
(8)             self.render_progress()
(8)             return self.generator()
(4)         def __next__(self) -> V:
(8)             return next(iter(self))
(4)         def render_finish(self) -> None:
(8)             if self.is_hidden:
(12)                return
(8)             self.file.write(AFTER_BAR)
(8)             self.file.flush()
(4)         @property
(4)         def pct(self) -> float:
(8)             if self.finished:
(12)                return 1.0
(8)             return min(self.pos / (float(self.length or 1) or 1), 1.0)
(4)         @property
(4)         def time_per_iteration(self) -> float:
(8)             if not self.avg:
(12)                return 0.0
(8)             return sum(self.avg) / float(len(self.avg))
(4)         @property
(4)         def eta(self) -> float:
(8)             if self.length is not None and not self.finished:
(12)                return self.time_per_iteration * (self.length - self.pos)
(8)             return 0.0
(4)         def format_eta(self) -> str:
(8)             if self.eta_known:
(12)                t = int(self.eta)
(12)                seconds = t % 60
(12)                t //= 60
(12)                minutes = t % 60
(12)                t //= 60
(12)                hours = t % 24
(12)                t //= 24
(12)                if t > 0:
(16)                    return f"{t}d {hours:02}:{minutes:02}:{seconds:02}"
(12)                else:
(16)                    return f"{hours:02}:{minutes:02}:{seconds:02}"
(8)             return ""
(4)         def format_pos(self) -> str:
(8)             pos = str(self.pos)
(8)             if self.length is not None:
(12)                pos += f"/{self.length}"
(8)             return pos
(4)         def format_pct(self) -> str:
(8)             return f"{int(self.pct * 100): 4}%"[1:]
(4)         def format_bar(self) -> str:
(8)             if self.length is not None:
(12)                bar_length = int(self.pct * self.width)
(12)                bar = self.fill_char * bar_length
(12)                bar += self.empty_char * (self.width - bar_length)
```

```
(8)          elif self.finished:
(12)             bar = self.fill_char * self.width
(8)          else:
(12)             chars = list(self.empty_char * (self.width or 1))
(12)             if self.time_per_iteration != 0:
(16)                 chars[
(20)                     int(
(24)                         (math.cos(self.pos * self.time_per_iteration) / 2.0 + 0.5)
(24)                         * self.width
(20)                     )
(16)                 ] = self.fill_char
(12)             bar = "".join(chars)
(8)          return bar
(4)      def format_progress_line(self) -> str:
(8)          show_percent = self.show_percent
(8)          info_bits = []
(8)          if self.length is not None and show_percent is None:
(12)             show_percent = not self.show_pos
(8)          if self.show_pos:
(12)             info_bits.append(self.format_pos())
(8)          if show_percent:
(12)             info_bits.append(self.format_pct())
(8)          if self.show_eta and self.eta_known and not self.finished:
(12)             info_bits.append(self.format_eta())
(8)          if self.item_show_func is not None:
(12)             item_info = self.item_show_func(self.current_item)
(12)             if item_info is not None:
(16)                 info_bits.append(item_info)
(8)          return (
(12)             self.bar_template
(12)             % {
(16)                 "label": self.label,
(16)                 "bar": self.format_bar(),
(16)                 "info": self.info_sep.join(info_bits),
(12)             }
(8)          ).rstrip()
(4)      def render_progress(self) -> None:
(8)          import shutil
(8)          if self.is_hidden:
(12)             if self._last_line != self.label:
(16)                 self._last_line = self.label
(16)                 echo(self.label, file=self.file, color=self.color)
(12)             return
(8)          buf = []
(8)          if self.autowidth:
(12)             old_width = self.width
(12)             self.width = 0
(12)             clutter_length = term_len(self.format_progress_line())
(12)             new_width = max(0, shutil.get_terminal_size().columns - clutter_length)
(12)             if new_width < old_width:
(16)                 buf.append(BEFORE_BAR)
(16)                 buf.append(" " * self.max_width)  # type: ignore
(16)                 self.max_width = new_width
(12)             self.width = new_width
(8)          clear_width = self.width
(8)          if self.max_width is not None:
(12)             clear_width = self.max_width
(8)          buf.append(BEFORE_BAR)
(8)          line = self.format_progress_line()
(8)          line_len = term_len(line)
(8)          if self.max_width is None or self.max_width < line_len:
(12)             self.max_width = line_len
(8)          buf.append(line)
(8)          buf.append(" " * (clear_width - line_len))
(8)          line = "".join(buf)
(8)          if line != self._last_line:
(12)             self._last_line = line
```

```
(12)                    echo(line, file=self.file, color=self.color, nl=False)
(12)                    self.file.flush()
(4)             def make_step(self, n_steps: int) -> None:
(8)                 self.pos += n_steps
(8)                 if self.length is not None and self.pos >= self.length:
(12)                    self.finished = True
(8)                 if (time.time() - self.last_eta) < 1.0:
(12)                    return
(8)                 self.last_eta = time.time()
(8)                 if self.pos:
(12)                    step = (time.time() - self.start) / self.pos
(8)                 else:
(12)                    step = time.time() - self.start
(8)                 self.avg = self.avg[-6:] + [step]
(8)                 self.eta_known = self.length is not None
(4)             def update(self, n_steps: int, current_item: t.Optional[V] = None) -> None:
(8)                 """Update the progress bar by advancing a specified number of
(8)                 steps, and optionally set the ``current_item`` for this new
(8)                 position.
(8)                 :param n_steps: Number of steps to advance.
(8)                 :param current_item: Optional item to set as ``current_item``
(12)                    for the updated position.
(8)                 .. versionchanged:: 8.0
(12)                    Added the ``current_item`` optional parameter.
(8)                 .. versionchanged:: 8.0
(12)                    Only render when the number of steps meets the
(12)                    ``update_min_steps`` threshold.
(8)                 """
(8)                 if current_item is not None:
(12)                    self.current_item = current_item
(8)                 self._completed_intervals += n_steps
(8)                 if self._completed_intervals >= self.update_min_steps:
(12)                    self.make_step(self._completed_intervals)
(12)                    self.render_progress()
(12)                    self._completed_intervals = 0
(4)             def finish(self) -> None:
(8)                 self.eta_known = False
(8)                 self.current_item = None
(8)                 self.finished = True
(4)             def generator(self) -> t.Iterator[V]:
(8)                 """Return a generator which yields the items added to the bar
(8)                 during construction, and updates the progress bar *after* the
(8)                 yielded block returns.
(8)                 """
(8)                 if not self.entered:
(12)                    raise RuntimeError("You need to use progress bars in a with block.")
(8)                 if self.is_hidden:
(12)                    yield from self.iter
(8)                 else:
(12)                    for rv in self.iter:
(16)                        self.current_item = rv
(16)                        if self._completed_intervals == 0:
(20)                            self.render_progress()
(16)                        yield rv
(16)                        self.update(1)
(12)                    self.finish()
(12)                    self.render_progress()
(0)         def pager(generator: t.Iterable[str], color: t.Optional[bool] = None) -> None:
(4)             """Decide what method to use for paging through text."""
(4)             stdout = _default_text_stdout()
(4)             if stdout is None:
(8)                 stdout = StringIO()
(4)             if not isatty(sys.stdin) or not isatty(stdout):
(8)                 return _nullpager(stdout, generator, color)
(4)             pager_cmd = (os.environ.get("PAGER", None) or "").strip()
(4)             if pager_cmd:
(8)                 if WIN:
```

```
(12)                return _tempfilepager(generator, pager_cmd, color)
(8)             return _pipepager(generator, pager_cmd, color)
(4)         if os.environ.get("TERM") in ("dumb", "emacs"):
(8)             return _nullpager(stdout, generator, color)
(4)         if WIN or sys.platform.startswith("os2"):
(8)             return _tempfilepager(generator, "more <", color)
(4)         if hasattr(os, "system") and os.system("(less) 2>/dev/null") == 0:
(8)             return _pipepager(generator, "less", color)
(4)         import tempfile
(4)         fd, filename = tempfile.mkstemp()
(4)         os.close(fd)
(4)         try:
(8)             if hasattr(os, "system") and os.system(f'more "{filename}"') == 0:
(12)                return _pipepager(generator, "more", color)
(8)             return _nullpager(stdout, generator, color)
(4)         finally:
(8)             os.unlink(filename)
(0)     def _pipepager(generator: t.Iterable[str], cmd: str, color: t.Optional[bool]) -> None:
(4)         """Page through text by feeding it to another program.  Invoking a
(4)         pager through this might support colors.
(4)         """
(4)         import subprocess
(4)         env = dict(os.environ)
(4)         cmd_detail = cmd.rsplit("/", 1)[-1].split()
(4)         if color is None and cmd_detail[0] == "less":
(8)             less_flags = f"{os.environ.get('LESS', '')}{' '.join(cmd_detail[1:])}"
(8)             if not less_flags:
(12)                env["LESS"] = "-R"
(12)                color = True
(8)             elif "r" in less_flags or "R" in less_flags:
(12)                color = True
(4)         c = subprocess.Popen(cmd, shell=True, stdin=subprocess.PIPE, env=env)
(4)         stdin = t.cast(t.BinaryIO, c.stdin)
(4)         encoding = get_best_encoding(stdin)
(4)         try:
(8)             for text in generator:
(12)                if not color:
(16)                    text = strip_ansi(text)
(12)                stdin.write(text.encode(encoding, "replace"))
(4)         except (OSError, KeyboardInterrupt):
(8)             pass
(4)         else:
(8)             stdin.close()
(4)         while True:
(8)             try:
(12)                c.wait()
(8)             except KeyboardInterrupt:
(12)                pass
(8)             else:
(12)                break
(0)     def _tempfilepager(
(4)         generator: t.Iterable[str], cmd: str, color: t.Optional[bool]
(0)     ) -> None:
(4)         """Page through text by invoking a program on a temporary file."""
(4)         import tempfile
(4)         fd, filename = tempfile.mkstemp()
(4)         text = "".join(generator)
(4)         if not color:
(8)             text = strip_ansi(text)
(4)         encoding = get_best_encoding(sys.stdout)
(4)         with open_stream(filename, "wb")[0] as f:
(8)             f.write(text.encode(encoding))
(4)         try:
(8)             os.system(f'{cmd} "{filename}"')
(4)         finally:
(8)             os.close(fd)
(8)             os.unlink(filename)
(0)     def _nullpager(
(4)         stream: t.TextIO, generator: t.Iterable[str], color: t.Optional[bool]
(0)     ) -> None:
(4)         """Simply print unformatted text.  This is the ultimate fallback."""
(4)         for text in generator:
(8)             if not color:
(12)                text = strip_ansi(text)
(8)             stream.write(text)
(0)     class Editor:
(4)         def __init__(
(8)             self,
(8)             editor: t.Optional[str] = None,
(8)             env: t.Optional[t.Mapping[str, str]] = None,
(8)             require_save: bool = True,
(8)             extension: str = ".txt",
(4)         ) -> None:
(8)             self.editor = editor
(8)             self.env = env
(8)             self.require_save = require_save
(8)             self.extension = extension
(4)         def get_editor(self) -> str:
(8)             if self.editor is not None:
(12)                return self.editor
(8)             for key in "VISUAL", "EDITOR":
(12)                rv = os.environ.get(key)
(12)                if rv:
(16)                    return rv
(8)             if WIN:
(12)                return "notepad"
(8)             for editor in "sensible-editor", "vim", "nano":
(12)                if os.system(f"which {editor} >/dev/null 2>&1") == 0:
(16)                    return editor
(8)             return "vi"
(4)         def edit_file(self, filename: str) -> None:
(8)             import subprocess
(8)             editor = self.get_editor()
(8)             environ: t.Optional[t.Dict[str, str]] = None
(8)             if self.env:
(12)                environ = os.environ.copy()
(12)                environ.update(self.env)
(8)             try:
(12)                c = subprocess.Popen(f'{editor} "{filename}"', env=environ, shell=True)
(12)                exit_code = c.wait()
(12)                if exit_code != 0:
(16)                    raise ClickException(
(20)                        _("{editor}: Editing failed").format(editor=editor)
(16)                    )
(8)             except OSError as e:
(12)                raise ClickException(
(16)                    _("{editor}: Editing failed: {e}").format(editor=editor, e=e)
(12)                ) from e
(4)         def edit(self, text: t.Optional[t.AnyStr]) -> t.Optional[t.AnyStr]:
(8)             import tempfile
(8)             if not text:
(12)                data = b""
(8)             elif isinstance(text, (bytes, bytearray)):
(12)                data = text
(8)             else:
(12)                if text and not text.endswith("\n"):
(16)                    text += "\n"
(12)                if WIN:
(16)                    data = text.replace("\n", "\r\n").encode("utf-8-sig")
(12)                else:
(16)                    data = text.encode("utf-8")
(8)             fd, name = tempfile.mkstemp(prefix="editor-", suffix=self.extension)
(8)             f: t.BinaryIO
(8)             try:
(12)                with os.fdopen(fd, "wb") as f:
(16)                    f.write(data)
```

```
(12)                os.utime(name, (os.path.getatime(name), os.path.getmtime(name) - 2))
(12)                timestamp = os.path.getmtime(name)
(12)                self.edit_file(name)
(12)                if self.require_save and os.path.getmtime(name) == timestamp:
(16)                    return None
(12)                with open(name, "rb") as f:
(16)                    rv = f.read()
(12)                if isinstance(text, (bytes, bytearray)):
(16)                    return rv
(12)                return rv.decode("utf-8-sig").replace("\r\n", "\n")  # type: ignore
(8)             finally:
(12)                os.unlink(name)
(0)     def open_url(url: str, wait: bool = False, locate: bool = False) -> int:
(4)         import subprocess
(4)         def _unquote_file(url: str) -> str:
(8)             from urllib.parse import unquote
(8)             if url.startswith("file://"):
(12)                url = unquote(url[7:])
(8)             return url
(4)         if sys.platform == "darwin":
(8)             args = ["open"]
(8)             if wait:
(12)                args.append("-W")
(8)             if locate:
(12)                args.append("-R")
(8)             args.append(_unquote_file(url))
(8)             null = open("/dev/null", "w")
(8)             try:
(12)                return subprocess.Popen(args, stderr=null).wait()
(8)             finally:
(12)                null.close()
(4)         elif WIN:
(8)             if locate:
(12)                url = _unquote_file(url.replace('"', ""))
(12)                args = f'explorer /select,"{url}"'
(8)             else:
(12)                url = url.replace('"', "")
(12)                wait_str = "/WAIT" if wait else ""
(12)                args = f'start {wait_str} "" "{url}"'
(8)             return os.system(args)
(4)         elif CYGWIN:
(8)             if locate:
(12)                url = os.path.dirname(_unquote_file(url).replace('"', ""))
(12)                args = f'cygstart "{url}"'
(8)             else:
(12)                url = url.replace('"', "")
(12)                wait_str = "-w" if wait else ""
(12)                args = f'cygstart {wait_str} "{url}"'
(8)             return os.system(args)
(4)         try:
(8)             if locate:
(12)                url = os.path.dirname(_unquote_file(url)) or "."
(8)             else:
(12)                url = _unquote_file(url)
(8)             c = subprocess.Popen(["xdg-open", url])
(8)             if wait:
(12)                return c.wait()
(8)             return 0
(4)         except OSError:
(8)             if url.startswith(("http://", "https://")) and not locate and not wait:
(12)                import webbrowser
(12)                webbrowser.open(url)
(12)                return 0
(8)             return 1
(0)     def _translate_ch_to_exc(ch: str) -> t.Optional[BaseException]:
(4)         if ch == "\x03":
```

```
(8)                         raise KeyboardInterrupt()
(4)                     if ch == "\x04" and not WIN:  # Unix-like, Ctrl+D
(8)                         raise EOFError()
(4)                     if ch == "\x1a" and WIN:  # Windows, Ctrl+Z
(8)                         raise EOFError()
(4)                     return None
(0)                 if WIN:
(4)                     import msvcrt
(4)                     @contextlib.contextmanager
(4)                     def raw_terminal() -> t.Iterator[int]:
(8)                         yield -1
(4)                     def getchar(echo: bool) -> str:
(8)                         func: t.Callable[[], str]
(8)                         if echo:
(12)                            func = msvcrt.getwche  # type: ignore
(8)                         else:
(12)                            func = msvcrt.getwch  # type: ignore
(8)                         rv = func()
(8)                         if rv in ("\x00", "\xe0"):
(12)                            rv += func()
(8)                         _translate_ch_to_exc(rv)
(8)                         return rv
(0)                 else:
(4)                     import tty
(4)                     import termios
(4)                     @contextlib.contextmanager
(4)                     def raw_terminal() -> t.Iterator[int]:
(8)                         f: t.Optional[t.TextIO]
(8)                         fd: int
(8)                         if not isatty(sys.stdin):
(12)                            f = open("/dev/tty")
(12)                            fd = f.fileno()
(8)                         else:
(12)                            fd = sys.stdin.fileno()
(12)                            f = None
(8)                         try:
(12)                            old_settings = termios.tcgetattr(fd)
(12)                            try:
(16)                                tty.setraw(fd)
(16)                                yield fd
(12)                            finally:
(16)                                termios.tcsetattr(fd, termios.TCSADRAIN, old_settings)
(16)                                sys.stdout.flush()
(16)                                if f is not None:
(20)                                    f.close()
(8)                         except termios.error:
(12)                            pass
(4)                     def getchar(echo: bool) -> str:
(8)                         with raw_terminal() as fd:
(12)                            ch = os.read(fd, 32).decode(get_best_encoding(sys.stdin),
"replace")
(12)                            if echo and isatty(sys.stdout):
(16)                                sys.stdout.write(ch)
(12)                            _translate_ch_to_exc(ch)
(12)                            return ch
```

File 5 - _winconsole.py:

```
(0)                 import io
(0)                 import sys
(0)                 import time
(0)                 import typing as t
(0)                 from ctypes import byref
(0)                 from ctypes import c_char
(0)                 from ctypes import c_char_p
(0)                 from ctypes import c_int
(0)                 from ctypes import c_ssize_t
```

```
(0)         from ctypes import c_ulong
(0)         from ctypes import c_void_p
(0)         from ctypes import POINTER
(0)         from ctypes import py_object
(0)         from ctypes import Structure
(0)         from ctypes.wintypes import DWORD
(0)         from ctypes.wintypes import HANDLE
(0)         from ctypes.wintypes import LPCWSTR
(0)         from ctypes.wintypes import LPWSTR
(0)         from ._compat import _NonClosingTextIOWrapper
(0)         assert sys.platform == "win32"
(0)         import msvcrt  # noqa: E402
(0)         from ctypes import windll  # noqa: E402
(0)         from ctypes import WINFUNCTYPE  # noqa: E402
(0)         c_ssize_p = POINTER(c_ssize_t)
(0)         kernel32 = windll.kernel32
(0)         GetStdHandle = kernel32.GetStdHandle
(0)         ReadConsoleW = kernel32.ReadConsoleW
(0)         WriteConsoleW = kernel32.WriteConsoleW
(0)         GetConsoleMode = kernel32.GetConsoleMode
(0)         GetLastError = kernel32.GetLastError
(0)         GetCommandLineW = WINFUNCTYPE(LPWSTR)(("GetCommandLineW", windll.kernel32))
(0)         CommandLineToArgvW = WINFUNCTYPE(POINTER(LPWSTR), LPCWSTR, POINTER(c_int))(
(4)             ("CommandLineToArgvW", windll.shell32)
(0)         )
(0)         LocalFree = WINFUNCTYPE(c_void_p, c_void_p)(("LocalFree", windll.kernel32))
(0)         STDIN_HANDLE = GetStdHandle(-10)
(0)         STDOUT_HANDLE = GetStdHandle(-11)
(0)         STDERR_HANDLE = GetStdHandle(-12)
(0)         PyBUF_SIMPLE = 0
(0)         PyBUF_WRITABLE = 1
(0)         ERROR_SUCCESS = 0
(0)         ERROR_NOT_ENOUGH_MEMORY = 8
(0)         ERROR_OPERATION_ABORTED = 995
(0)         STDIN_FILENO = 0
(0)         STDOUT_FILENO = 1
(0)         STDERR_FILENO = 2
(0)         EOF = b"\x1a"
(0)         MAX_BYTES_WRITTEN = 32767
(0)         try:
(4)             from ctypes import pythonapi
(0)         except ImportError:
(4)             get_buffer = None
(0)         else:
(4)             class Py_buffer(Structure):
(8)                 _fields_ = [
(12)                    ("buf", c_void_p),
(12)                    ("obj", py_object),
(12)                    ("len", c_ssize_t),
(12)                    ("itemsize", c_ssize_t),
(12)                    ("readonly", c_int),
(12)                    ("ndim", c_int),
(12)                    ("format", c_char_p),
(12)                    ("shape", c_ssize_p),
(12)                    ("strides", c_ssize_p),
(12)                    ("suboffsets", c_ssize_p),
(12)                    ("internal", c_void_p),
(8)                 ]
(4)             PyObject_GetBuffer = pythonapi.PyObject_GetBuffer
(4)             PyBuffer_Release = pythonapi.PyBuffer_Release
(4)             def get_buffer(obj, writable=False):
(8)                 buf = Py_buffer()
(8)                 flags = PyBUF_WRITABLE if writable else PyBUF_SIMPLE
(8)                 PyObject_GetBuffer(py_object(obj), byref(buf), flags)
(8)                 try:
(12)                    buffer_type = c_char * buf.len
(12)                    return buffer_type.from_address(buf.buf)
(8)                 finally:
(12)                    PyBuffer_Release(byref(buf))
```

```
class _WindowsConsoleRawIOBase(io.RawIOBase):
    def __init__(self, handle):
        self.handle = handle
    def isatty(self):
        super().isatty()
        return True
class _WindowsConsoleReader(_WindowsConsoleRawIOBase):
    def readable(self):
        return True
    def readinto(self, b):
        bytes_to_be_read = len(b)
        if not bytes_to_be_read:
            return 0
        elif bytes_to_be_read % 2:
            raise ValueError(
                "cannot read odd number of bytes from UTF-16-LE encoded console"
            )
        buffer = get_buffer(b, writable=True)
        code_units_to_be_read = bytes_to_be_read // 2
        code_units_read = c_ulong()
        rv = ReadConsoleW(
            HANDLE(self.handle),
            buffer,
            code_units_to_be_read,
            byref(code_units_read),
            None,
        )
        if GetLastError() == ERROR_OPERATION_ABORTED:
            time.sleep(0.1)
        if not rv:
            raise OSError(f"Windows error: {GetLastError()}")
        if buffer[0] == EOF:
            return 0
        return 2 * code_units_read.value
class _WindowsConsoleWriter(_WindowsConsoleRawIOBase):
    def writable(self):
        return True
    @staticmethod
    def _get_error_message(errno):
        if errno == ERROR_SUCCESS:
            return "ERROR_SUCCESS"
        elif errno == ERROR_NOT_ENOUGH_MEMORY:
            return "ERROR_NOT_ENOUGH_MEMORY"
        return f"Windows error {errno}"
    def write(self, b):
        bytes_to_be_written = len(b)
        buf = get_buffer(b)
        code_units_to_be_written = min(bytes_to_be_written, MAX_BYTES_WRITTEN) // 2
        code_units_written = c_ulong()
        WriteConsoleW(
            HANDLE(self.handle),
            buf,
            code_units_to_be_written,
            byref(code_units_written),
            None,
        )
        bytes_written = 2 * code_units_written.value
        if bytes_written == 0 and bytes_to_be_written > 0:
            raise OSError(self._get_error_message(GetLastError()))
        return bytes_written
class ConsoleStream:
    def __init__(self, text_stream: t.TextIO, byte_stream: t.BinaryIO) -> None:
        self._text_stream = text_stream
        self.buffer = byte_stream
    @property
    def name(self) -> str:
```

```
(8)                     return self.buffer.name
(4)                 def write(self, x: t.AnyStr) -> int:
(8)                     if isinstance(x, str):
(12)                        return self._text_stream.write(x)
(8)                     try:
(12)                        self.flush()
(8)                     except Exception:
(12)                        pass
(8)                     return self.buffer.write(x)
(4)                 def writelines(self, lines: t.Iterable[t.AnyStr]) -> None:
(8)                     for line in lines:
(12)                        self.write(line)
(4)                 def __getattr__(self, name: str) -> t.Any:
(8)                     return getattr(self._text_stream, name)
(4)                 def isatty(self) -> bool:
(8)                     return self.buffer.isatty()
(4)                 def __repr__(self):
(8)                     return f"<ConsoleStream name={self.name!r} encoding={self.encoding!r}>"
(0)             def _get_text_stdin(buffer_stream: t.BinaryIO) -> t.TextIO:
(4)                 text_stream = _NonClosingTextIOWrapper(
(8)                     io.BufferedReader(_WindowsConsoleReader(STDIN_HANDLE)),
(8)                     "utf-16-le",
(8)                     "strict",
(8)                     line_buffering=True,
(4)                 )
(4)                 return t.cast(t.TextIO, ConsoleStream(text_stream, buffer_stream))
(0)             def _get_text_stdout(buffer_stream: t.BinaryIO) -> t.TextIO:
(4)                 text_stream = _NonClosingTextIOWrapper(
(8)                     io.BufferedWriter(_WindowsConsoleWriter(STDOUT_HANDLE)),
(8)                     "utf-16-le",
(8)                     "strict",
(8)                     line_buffering=True,
(4)                 )
(4)                 return t.cast(t.TextIO, ConsoleStream(text_stream, buffer_stream))
(0)             def _get_text_stderr(buffer_stream: t.BinaryIO) -> t.TextIO:
(4)                 text_stream = _NonClosingTextIOWrapper(
(8)                     io.BufferedWriter(_WindowsConsoleWriter(STDERR_HANDLE)),
(8)                     "utf-16-le",
(8)                     "strict",
(8)                     line_buffering=True,
(4)                 )
(4)                 return t.cast(t.TextIO, ConsoleStream(text_stream, buffer_stream))
(0)             _stream_factories: t.Mapping[int, t.Callable[[t.BinaryIO], t.TextIO]] = {
(4)                 0: _get_text_stdin,
(4)                 1: _get_text_stdout,
(4)                 2: _get_text_stderr,
(0)             }
(0)             def _is_console(f: t.TextIO) -> bool:
(4)                 if not hasattr(f, "fileno"):
(8)                     return False
(4)                 try:
(8)                     fileno = f.fileno()
(4)                 except (OSError, io.UnsupportedOperation):
(8)                     return False
(4)                 handle = msvcrt.get_osfhandle(fileno)
(4)                 return bool(GetConsoleMode(handle, byref(DWORD())))
(0)             def _get_windows_console_stream(
(4)                 f: t.TextIO, encoding: t.Optional[str], errors: t.Optional[str]
(0)             ) -> t.Optional[t.TextIO]:
(4)                 if (
(8)                     get_buffer is not None
(8)                     and encoding in {"utf-16-le", None}
(8)                     and errors in {"strict", None}
(8)                     and _is_console(f)
(4)                 ):
(8)                     func = _stream_factories.get(f.fileno())
(8)                     if func is not None:
(12)                        b = getattr(f, "buffer", None)
```

```
213: (12)                       if b is None:
214: (16)                           return None
215: (12)                       return func(b)
```

File 6 - core.py:

```
1: (0)              import enum
2: (0)              import errno
3: (0)              import inspect
4: (0)              import os
5: (0)              import sys
6: (0)              import typing as t
7: (0)              from collections import abc
8: (0)              from contextlib import contextmanager
9: (0)              from contextlib import ExitStack
10: (0)             from functools import update_wrapper
11: (0)             from gettext import gettext as _
12: (0)             from gettext import ngettext
13: (0)             from itertools import repeat
14: (0)             from types import TracebackType
15: (0)             from . import types
16: (0)             from .exceptions import Abort
17: (0)             from .exceptions import BadParameter
18: (0)             from .exceptions import ClickException
19: (0)             from .exceptions import Exit
20: (0)             from .exceptions import MissingParameter
21: (0)             from .exceptions import UsageError
22: (0)             from .formatting import HelpFormatter
23: (0)             from .formatting import join_options
24: (0)             from .globals import pop_context
25: (0)             from .globals import push_context
26: (0)             from .parser import _flag_needs_value
27: (0)             from .parser import OptionParser
28: (0)             from .parser import split_opt
29: (0)             from .termui import confirm
30: (0)             from .termui import prompt
31: (0)             from .termui import style
32: (0)             from .utils import _detect_program_name
33: (0)             from .utils import _expand_args
34: (0)             from .utils import echo
35: (0)             from .utils import make_default_short_help
36: (0)             from .utils import make_str
37: (0)             from .utils import PacifyFlushWrapper
38: (0)             if t.TYPE_CHECKING:
39: (4)                 import typing_extensions as te
40: (4)                 from .shell_completion import CompletionItem
41: (0)             F = t.TypeVar("F", bound=t.Callable[..., t.Any])
42: (0)             V = t.TypeVar("V")
43: (0)             def _complete_visible_commands(
44: (4)                 ctx: "Context", incomplete: str
45: (0)             ) -> t.Iterator[t.Tuple[str, "Command"]]:
46: (4)                 """List all the subcommands of a group that start with the
47: (4)                 incomplete value and aren't hidden.
48: (4)                 :param ctx: Invocation context for the group.
49: (4)                 :param incomplete: Value being completed. May be empty.
50: (4)                 """
51: (4)                 multi = t.cast(MultiCommand, ctx.command)
52: (4)                 for name in multi.list_commands(ctx):
53: (8)                     if name.startswith(incomplete):
54: (12)                        command = multi.get_command(ctx, name)
55: (12)                        if command is not None and not command.hidden:
56: (16)                            yield name, command
57: (0)             def _check_multicommand(
58: (4)                 base_command: "MultiCommand", cmd_name: str, cmd: "Command", register:
bool = False
59: (0)             ) -> None:
60: (4)                 if not base_command.chain or not isinstance(cmd, MultiCommand):
```

```
(8)                 return
(4)             if register:
(8)                 hint = (
(12)                    "It is not possible to add multi commands as children to"
(12)                    " another multi command that is in chain mode."
(8)                 )
(4)             else:
(8)                 hint = (
(12)                    "Found a multi command as subcommand to a multi command"
(12)                    " that is in chain mode. This is not supported."
(8)                 )
(4)             raise RuntimeError(
(8)                 f"{hint}. Command {base_command.name!r} is set to chain and"
(8)                 f" {cmd_name!r} was added as a subcommand but it in itself is a"
(8)                 f" multi command. ({cmd_name!r} is a {type(cmd).__name__}"
(8)                 f" within a chained {type(base_command).__name__} named"
(8)                 f" {base_command.name!r})."
(4)             )
(0)         def batch(iterable: t.Iterable[V], batch_size: int) -> t.List[t.Tuple[V, ...]]:
(4)             return list(zip(*repeat(iter(iterable), batch_size)))
(0)         @contextmanager
(0)         def augment_usage_errors(
(4)             ctx: "Context", param: t.Optional["Parameter"] = None
(0)         ) -> t.Iterator[None]:
(4)             """Context manager that attaches extra information to exceptions."""
(4)             try:
(8)                 yield
(4)             except BadParameter as e:
(8)                 if e.ctx is None:
(12)                    e.ctx = ctx
(8)                 if param is not None and e.param is None:
(12)                    e.param = param
(8)                 raise
(4)             except UsageError as e:
(8)                 if e.ctx is None:
(12)                    e.ctx = ctx
(8)                 raise
(0)         def iter_params_for_processing(
(4)             invocation_order: t.Sequence["Parameter"],
(4)             declaration_order: t.Sequence["Parameter"],
(0)         ) -> t.List["Parameter"]:
(4)             """Given a sequence of parameters in the order as should be considered
(4)             for processing and an iterable of parameters that exist, this returns
(4)             a list in the correct order as they should be processed.
(4)             """
(4)             def sort_key(item: "Parameter") -> t.Tuple[bool, float]:
(8)                 try:
(12)                    idx: float = invocation_order.index(item)
(8)                 except ValueError:
(12)                    idx = float("inf")
(8)                 return not item.is_eager, idx
(4)             return sorted(declaration_order, key=sort_key)
(0)         class ParameterSource(enum.Enum):
(4)             """This is an :class:`~enum.Enum` that indicates the source of a
(4)             parameter's value.
(4)             Use :meth:`click.Context.get_parameter_source` to get the
(4)             source for a parameter by name.
(4)             .. versionchanged:: 8.0
(8)                 Use :class:`~enum.Enum` and drop the ``validate`` method.
(4)             .. versionchanged:: 8.0
(8)                 Added the ``PROMPT`` value.
(4)             """
(4)             COMMANDLINE = enum.auto()
(4)             """The value was provided by the command line args."""
(4)             ENVIRONMENT = enum.auto()
(4)             """The value was provided with an environment variable."""
(4)             DEFAULT = enum.auto()
(4)             """Used the default specified by the parameter."""
```

```
(4)       DEFAULT_MAP = enum.auto()
(4)       """Used a default provided by :attr:`Context.default_map`."""
(4)       PROMPT = enum.auto()
(4)       """Used a prompt to confirm a default or provide a value."""
(0)   class Context:
(4)       """The context is a special internal object that holds state relevant
(4)       for the script execution at every single level.  It's normally invisible
(4)       to commands unless they opt-in to getting access to it.
(4)       The context is useful as it can pass internal objects around and can
(4)       control special execution features such as reading data from
(4)       environment variables.
(4)       A context can be used as context manager in which case it will call
(4)       :meth:`close` on teardown.
(4)       :param command: the command class for this context.
(4)       :param parent: the parent context.
(4)       :param info_name: the info name for this invocation.  Generally this
(22)                        is the most descriptive name for the script or
(22)                        command.  For the toplevel script it is usually
(22)                        the name of the script, for commands below it it's
(22)                        the name of the script.
(4)       :param obj: an arbitrary object of user data.
(4)       :param auto_envvar_prefix: the prefix to use for automatic environment
(31)                                 variables.  If this is `None` then reading
(31)                                 from environment variables is disabled.  This
(31)                                 does not affect manually set environment
(31)                                 variables which are always read.
(4)       :param default_map: a dictionary (like object) with default values
(24)                          for parameters.
(4)       :param terminal_width: the width of the terminal.  The default is
(27)                             inherit from parent context.  If no context
(27)                             defines the terminal width then auto
(27)                             detection will be applied.
(4)       :param max_content_width: the maximum width for content rendered by
(30)                                Click (this currently only affects help
(30)                                pages).  This defaults to 80 characters if
(30)                                not overridden.  In other words: even if the
(30)                                terminal is larger than that, Click will not
(30)                                format things wider than 80 characters by
(30)                                default.  In addition to that, formatters might
(30)                                add some safety mapping on the right.
(4)       :param resilient_parsing: if this flag is enabled then Click will
(30)                                parse without any interactivity or callback
(30)                                invocation.  Default values will also be
(30)                                ignored.  This is useful for implementing
(30)                                things such as completion support.
(4)       :param allow_extra_args: if this is set to `True` then extra arguments
(29)                               at the end will not raise an error and will be
(29)                               kept on the context.  The default is to inherit
(29)                               from the command.
(4)       :param allow_interspersed_args: if this is set to `False` then options
(36)                                      and arguments cannot be mixed.  The
(36)                                      default is to inherit from the command.
(4)       :param ignore_unknown_options: instructs click to ignore options it does
(35)                                     not know and keeps them for later
(35)                                     processing.
(4)       :param help_option_names: optionally a list of strings that define how
(30)                                the default help parameter is named.  The
(30)                                default is ``['--help']``.
(4)       :param token_normalize_func: an optional function that is used to
(33)                                   normalize tokens (options, choices,
(33)                                   etc.).  This for instance can be used to
(33)                                   implement case insensitive behavior.
(4)       :param color: controls if the terminal supports ANSI colors or not.  The
(18)                    default is autodetection.  This is only needed if ANSI
(18)                    codes are used in texts that Click prints which is by
(18)                    default not the case.  This for instance would affect
(18)                    help output.
(4)       :param show_default: Show the default value for commands. If this
(8)           value is not set, it defaults to the value from the parent
```

```
(8)         context. ``Command.show_default`` overrides this default for the
(8)         specific command.
(4)     .. versionchanged:: 8.1
(8)         The ``show_default`` parameter is overridden by
(8)         ``Command.show_default``, instead of the other way around.
(4)     .. versionchanged:: 8.0
(8)         The ``show_default`` parameter defaults to the value from the
(8)         parent context.
(4)     .. versionchanged:: 7.1
(7)        Added the ``show_default`` parameter.
(4)     .. versionchanged:: 4.0
(8)         Added the ``color``, ``ignore_unknown_options``, and
(8)         ``max_content_width`` parameters.
(4)     .. versionchanged:: 3.0
(8)         Added the ``allow_extra_args`` and ``allow_interspersed_args``
(8)         parameters.
(4)     .. versionchanged:: 2.0
(8)         Added the ``resilient_parsing``, ``help_option_names``, and
(8)         ``token_normalize_func`` parameters.
(4)     """
(4)     formatter_class: t.Type["HelpFormatter"] = HelpFormatter
(4)     def __init__(
(8)         self,
(8)         command: "Command",
(8)         parent: t.Optional["Context"] = None,
(8)         info_name: t.Optional[str] = None,
(8)         obj: t.Optional[t.Any] = None,
(8)         auto_envvar_prefix: t.Optional[str] = None,
(8)         default_map: t.Optional[t.MutableMapping[str, t.Any]] = None,
(8)         terminal_width: t.Optional[int] = None,
(8)         max_content_width: t.Optional[int] = None,
(8)         resilient_parsing: bool = False,
(8)         allow_extra_args: t.Optional[bool] = None,
(8)         allow_interspersed_args: t.Optional[bool] = None,
(8)         ignore_unknown_options: t.Optional[bool] = None,
(8)         help_option_names: t.Optional[t.List[str]] = None,
(8)         token_normalize_func: t.Optional[t.Callable[[str], str]] = None,
(8)         color: t.Optional[bool] = None,
(8)         show_default: t.Optional[bool] = None,
(4)     ) -> None:
(8)         self.parent = parent
(8)         self.command = command
(8)         self.info_name = info_name
(8)         self.params: t.Dict[str, t.Any] = {}
(8)         self.args: t.List[str] = []
(8)         self.protected_args: t.List[str] = []
(8)         self._opt_prefixes: t.Set[str] = set(parent._opt_prefixes) if parent else set()
(8)         if obj is None and parent is not None:
(12)            obj = parent.obj
(8)         self.obj: t.Any = obj
(8)         self._meta: t.Dict[str, t.Any] = getattr(parent, "meta", {})
(8)         if (
(12)            default_map is None
(12)            and info_name is not None
(12)            and parent is not None
(12)            and parent.default_map is not None
(8)         ):
(12)            default_map = parent.default_map.get(info_name)
(8)         self.default_map: t.Optional[t.MutableMapping[str, t.Any]] = default_map
(8)         self.invoked_subcommand: t.Optional[str] = None
(8)         if terminal_width is None and parent is not None:
(12)            terminal_width = parent.terminal_width
(8)         self.terminal_width: t.Optional[int] = terminal_width
(8)         if max_content_width is None and parent is not None:
(12)            max_content_width = parent.max_content_width
(8)         self.max_content_width: t.Optional[int] = max_content_width
(8)         if allow_extra_args is None:
```

```
            allow_extra_args = command.allow_extra_args
        self.allow_extra_args = allow_extra_args
        if allow_interspersed_args is None:
            allow_interspersed_args = command.allow_interspersed_args
        self.allow_interspersed_args: bool = allow_interspersed_args
        if ignore_unknown_options is None:
            ignore_unknown_options = command.ignore_unknown_options
        self.ignore_unknown_options: bool = ignore_unknown_options
        if help_option_names is None:
            if parent is not None:
                help_option_names = parent.help_option_names
            else:
                help_option_names = ["--help"]
        self.help_option_names: t.List[str] = help_option_names
        if token_normalize_func is None and parent is not None:
            token_normalize_func = parent.token_normalize_func
        self.token_normalize_func: t.Optional[
            t.Callable[[str], str]
        ] = token_normalize_func
        self.resilient_parsing: bool = resilient_parsing
        if auto_envvar_prefix is None:
            if (
                parent is not None
                and parent.auto_envvar_prefix is not None
                and self.info_name is not None
            ):
                auto_envvar_prefix = (
                    f"{parent.auto_envvar_prefix}_{self.info_name.upper()}"
                )
        else:
            auto_envvar_prefix = auto_envvar_prefix.upper()
        if auto_envvar_prefix is not None:
            auto_envvar_prefix = auto_envvar_prefix.replace("-", "_")
        self.auto_envvar_prefix: t.Optional[str] = auto_envvar_prefix
        if color is None and parent is not None:
            color = parent.color
        self.color: t.Optional[bool] = color
        if show_default is None and parent is not None:
            show_default = parent.show_default
        self.show_default: t.Optional[bool] = show_default
        self._close_callbacks: t.List[t.Callable[[], t.Any]] = []
        self._depth = 0
        self._parameter_source: t.Dict[str, ParameterSource] = {}
        self._exit_stack = ExitStack()
    def to_info_dict(self) -> t.Dict[str, t.Any]:
        """Gather information that could be useful for a tool generating
        user-facing documentation. This traverses the entire CLI
        structure.
        .. code-block:: python
            with Context(cli) as ctx:
                info = ctx.to_info_dict()
        .. versionadded:: 8.0
        """
        return {
            "command": self.command.to_info_dict(self),
            "info_name": self.info_name,
            "allow_extra_args": self.allow_extra_args,
            "allow_interspersed_args": self.allow_interspersed_args,
            "ignore_unknown_options": self.ignore_unknown_options,
            "auto_envvar_prefix": self.auto_envvar_prefix,
        }
    def __enter__(self) -> "Context":
        self._depth += 1
        push_context(self)
        return self
    def __exit__(
        self,
        exc_type: t.Optional[t.Type[BaseException]],
        exc_value: t.Optional[BaseException],
        tb: t.Optional[TracebackType],
    ) -> None:
        self._depth -= 1
        if self._depth == 0:
            self.close()
        pop_context()
    @contextmanager
    def scope(self, cleanup: bool = True) -> t.Iterator["Context"]:
        """This helper method can be used with the context object to promote
        it to the current thread local (see :func:`get_current_context`).
        The default behavior of this is to invoke the cleanup functions which
        can be disabled by setting `cleanup` to `False`.  The cleanup
        functions are typically used for things such as closing file handles.
        If the cleanup is intended the context object can also be directly
        used as a context manager.
        Example usage::
            with ctx.scope():
                assert get_current_context() is ctx
        This is equivalent::
            with ctx:
                assert get_current_context() is ctx
        .. versionadded:: 5.0
        :param cleanup: controls if the cleanup functions should be run or
                        not.  The default is to run these functions.  In
                        some situations the context only wants to be
                        temporarily pushed in which case this can be disabled.
                        Nested pushes automatically defer the cleanup.
        """
        if not cleanup:
            self._depth += 1
        try:
            with self as rv:
                yield rv
        finally:
            if not cleanup:
                self._depth -= 1
    @property
    def meta(self) -> t.Dict[str, t.Any]:
        """This is a dictionary which is shared with all the contexts
        that are nested.  It exists so that click utilities can store some
        state here if they need to.  It is however the responsibility of
        that code to manage this dictionary well.
        The keys are supposed to be unique dotted strings.  For instance
        module paths are a good choice for it.  What is stored in there is
        irrelevant for the operation of click.  However what is important is
        that code that places data here adheres to the general semantics of
        the system.
        Example usage::
            LANG_KEY = f'{__name__}.lang'
            def set_language(value):
                ctx = get_current_context()
                ctx.meta[LANG_KEY] = value
            def get_language():
                return get_current_context().meta.get(LANG_KEY, 'en_US')
        .. versionadded:: 5.0
        """
        return self._meta
    def make_formatter(self) -> HelpFormatter:
        """Creates the :class:`~click.HelpFormatter` for the help and
        usage output.
        To quickly customize the formatter class used without overriding
        this method, set the :attr:`formatter_class` attribute.
        .. versionchanged:: 8.0
            Added the :attr:`formatter_class` attribute.
        """
        return self.formatter_class(
            width=self.terminal_width, max_width=self.max_content_width
        )
    def with_resource(self, context_manager: t.ContextManager[V]) -> V:
```

```
403: (8)                """Register a resource as if it were used in a ``with``
404: (8)                statement. The resource will be cleaned up when the context is
405: (8)                popped.
406: (8)                Uses :meth:`contextlib.ExitStack.enter_context`. It calls the
407: (8)                resource's ``__enter__()`` method and returns the result. When
408: (8)                the context is popped, it closes the stack, which calls the
409: (8)                resource's ``__exit__()`` method.
410: (8)                To register a cleanup function for something that isn't a
411: (8)                context manager, use :meth:`call_on_close`. Or use something
412: (8)                from :mod:`contextlib` to turn it into a context manager first.
413: (8)                .. code-block:: python
414: (12)                   @click.group()
415: (12)                   @click.option("--name")
416: (12)                   @click.pass_context
417: (12)                   def cli(ctx):
418: (16)                       ctx.obj = ctx.with_resource(connect_db(name))
419: (8)                :param context_manager: The context manager to enter.
420: (8)                :return: Whatever ``context_manager.__enter__()`` returns.
421: (8)                .. versionadded:: 8.0
422: (8)                """
423: (8)                return self._exit_stack.enter_context(context_manager)
424: (4)            def call_on_close(self, f: t.Callable[..., t.Any]) -> t.Callable[...,
t.Any]:
425: (8)                """Register a function to be called when the context tears down.
426: (8)                This can be used to close resources opened during the script
427: (8)                execution. Resources that support Python's context manager
428: (8)                protocol which would be used in a ``with`` statement should be
429: (8)                registered with :meth:`with_resource` instead.
430: (8)                :param f: The function to execute on teardown.
431: (8)                """
432: (8)                return self._exit_stack.callback(f)
433: (4)            def close(self) -> None:
434: (8)                """Invoke all close callbacks registered with
435: (8)                :meth:`call_on_close`, and exit all context managers entered
436: (8)                with :meth:`with_resource`.
437: (8)                """
438: (8)                self._exit_stack.close()
439: (8)                self._exit_stack = ExitStack()
440: (4)            @property
441: (4)            def command_path(self) -> str:
442: (8)                """The computed command path.  This is used for the ``usage``
443: (8)                information on the help page.  It's automatically created by
444: (8)                combining the info names of the chain of contexts to the root.
445: (8)                """
446: (8)                rv = ""
447: (8)                if self.info_name is not None:
448: (12)                   rv = self.info_name
449: (8)                if self.parent is not None:
450: (12)                   parent_command_path = [self.parent.command_path]
451: (12)                   if isinstance(self.parent.command, Command):
452: (16)                       for param in self.parent.command.get_params(self):
453: (20)                           parent_command_path.extend(param.get_usage_pieces(self))
454: (12)                   rv = f"{' '.join(parent_command_path)} {rv}"
455: (8)                return rv.lstrip()
456: (4)            def find_root(self) -> "Context":
457: (8)                """Finds the outermost context."""
458: (8)                node = self
459: (8)                while node.parent is not None:
460: (12)                   node = node.parent
461: (8)                return node
462: (4)            def find_object(self, object_type: t.Type[V]) -> t.Optional[V]:
463: (8)                """Finds the closest object of a given type."""
464: (8)                node: t.Optional["Context"] = self
465: (8)                while node is not None:
466: (12)                   if isinstance(node.obj, object_type):
467: (16)                       return node.obj
468: (12)                   node = node.parent
469: (8)                return None
470: (4)            def ensure_object(self, object_type: t.Type[V]) -> V:
        """Like :meth:`find_object` but sets the innermost object to a
        new instance of `object_type` if it does not exist.
        """
        rv = self.find_object(object_type)
        if rv is None:
            self.obj = rv = object_type()
        return rv
    @t.overload
    def lookup_default(
        self, name: str, call: "te.Literal[True]" = True
    ) -> t.Optional[t.Any]:
        ...
    @t.overload
    def lookup_default(
        self, name: str, call: "te.Literal[False]" = ...
    ) -> t.Optional[t.Union[t.Any, t.Callable[[], t.Any]]]:
        ...
    def lookup_default(self, name: str, call: bool = True) -> t.Optional[t.Any]:
        """Get the default for a parameter from :attr:`default_map`.
        :param name: Name of the parameter.
        :param call: If the default is a callable, call it. Disable to
            return the callable instead.
        .. versionchanged:: 8.0
            Added the ``call`` parameter.
        """
        if self.default_map is not None:
            value = self.default_map.get(name)
            if call and callable(value):
                return value()
            return value
        return None
    def fail(self, message: str) -> "te.NoReturn":
        """Aborts the execution of the program with a specific error
        message.
        :param message: the error message to fail with.
        """
        raise UsageError(message, self)
    def abort(self) -> "te.NoReturn":
        """Aborts the script."""
        raise Abort()
    def exit(self, code: int = 0) -> "te.NoReturn":
        """Exits the application with a given exit code."""
        raise Exit(code)
    def get_usage(self) -> str:
        """Helper method to get formatted usage string for the current
        context and command.
        """
        return self.command.get_usage(self)
    def get_help(self) -> str:
        """Helper method to get formatted help page for the current
        context and command.
        """
        return self.command.get_help(self)
    def _make_sub_context(self, command: "Command") -> "Context":
        """Create a new context of the same type as this context, but
        for a new command.
        :meta private:
        """
        return type(self)(command, info_name=command.name, parent=self)
    @t.overload
    def invoke(
        __self,  # noqa: B902
        __callback: "t.Callable[..., V]",
        *args: t.Any,
        **kwargs: t.Any,
    ) -> V:
        ...
    @t.overload
```

```
(4)                 def invoke(
(8)                     __self,  # noqa: B902
(8)                     __callback: "Command",
(8)                     *args: t.Any,
(8)                     **kwargs: t.Any,
(4)                 ) -> t.Any:
(8)                     ...
(4)                 def invoke(
(8)                     __self,  # noqa: B902
(8)                     __callback: t.Union["Command", "t.Callable[..., V]"],
(8)                     *args: t.Any,
(8)                     **kwargs: t.Any,
(4)                 ) -> t.Union[t.Any, V]:
(8)                     """Invokes a command callback in exactly the way it expects.  There
(8)                     are two ways to invoke this method:
(8)                     1.  the first argument can be a callback and all other arguments and
(12)                        keyword arguments are forwarded directly to the function.
(8)                     2.  the first argument is a click command object.  In that case all
(12)                        arguments are forwarded as well but proper click parameters
(12)                        (options and click arguments) must be keyword arguments and Click
(12)                        will fill in defaults.
(8)                     Note that before Click 3.2 keyword arguments were not properly filled
(8)                     in against the intention of this code and no context was created.  For
(8)                     more information about this change and why it was done in a bugfix
(8)                     release see :ref:`upgrade-to-3.2`.
(8)                     .. versionchanged:: 8.0
(12)                        All ``kwargs`` are tracked in :attr:`params` so they will be
(12)                        passed if :meth:`forward` is called at multiple levels.
(8)                     """
(8)                     if isinstance(__callback, Command):
(12)                        other_cmd = __callback
(12)                        if other_cmd.callback is None:
(16)                            raise TypeError(
(20)                                "The given command does not have a callback that can be invoked."
(16)                            )
(12)                        else:
(16)                            __callback = t.cast("t.Callable[..., V]", other_cmd.callback)
(12)                        ctx = __self._make_sub_context(other_cmd)
(12)                        for param in other_cmd.params:
(16)                            if param.name not in kwargs and param.expose_value:
(20)                                kwargs[param.name] = param.type_cast_value(  # type: ignore
(24)                                    ctx, param.get_default(ctx)
(20)                                )
(12)                        ctx.params.update(kwargs)
(8)                     else:
(12)                        ctx = __self
(8)                     with augment_usage_errors(__self):
(12)                        with ctx:
(16)                            return __callback(*args, **kwargs)
(4)                 def forward(
(8)                     __self, __cmd: "Command", *args: t.Any, **kwargs: t.Any  # noqa: B902
(4)                 ) -> t.Any:
(8)                     """Similar to :meth:`invoke` but fills in default keyword
(8)                     arguments from the current context if the other command expects
(8)                     it.  This cannot invoke callbacks directly, only other commands.
(8)                     .. versionchanged:: 8.0
(12)                        All ``kwargs`` are tracked in :attr:`params` so they will be
(12)                        passed if ``forward`` is called at multiple levels.
(8)                     """
(8)                     if not isinstance(__cmd, Command):
(12)                        raise TypeError("Callback is not a command.")
(8)                     for param in __self.params:
(12)                        if param not in kwargs:
(16)                            kwargs[param] = __self.params[param]
(8)                     return __self.invoke(__cmd, *args, **kwargs)
(4)                 def set_parameter_source(self, name: str, source: ParameterSource) -> None:
```

```
(8)             """Set the source of a parameter. This indicates the location
(8)             from which the value of the parameter was obtained.
(8)             :param name: The name of the parameter.
(8)             :param source: A member of :class:`~click.core.ParameterSource`.
(8)             """
(8)             self._parameter_source[name] = source
(4)         def get_parameter_source(self, name: str) -> t.Optional[ParameterSource]:
(8)             """Get the source of a parameter. This indicates the location
(8)             from which the value of the parameter was obtained.
(8)             This can be useful for determining when a user specified a value
(8)             on the command line that is the same as the default value. It
(8)             will be :attr:`~click.core.ParameterSource.DEFAULT` only if the
(8)             value was actually taken from the default.
(8)             :param name: The name of the parameter.
(8)             :rtype: ParameterSource
(8)             .. versionchanged:: 8.0
(12)                Returns ``None`` if the parameter was not provided from any
(12)                source.
(8)             """
(8)             return self._parameter_source.get(name)
(0)     class BaseCommand:
(4)         """The base command implements the minimal API contract of commands.
(4)         Most code will never use this as it does not implement a lot of useful
(4)         functionality but it can act as the direct subclass of alternative
(4)         parsing methods that do not depend on the Click parser.
(4)         For instance, this can be used to bridge Click and other systems like
(4)         argparse or docopt.
(4)         Because base commands do not implement a lot of the API that other
(4)         parts of Click take for granted, they are not supported for all
(4)         operations.  For instance, they cannot be used with the decorators
(4)         usually and they have no built-in callback system.
(4)         .. versionchanged:: 2.0
(7)            Added the `context_settings` parameter.
(4)         :param name: the name of the command to use unless a group overrides it.
(4)         :param context_settings: an optional dictionary with defaults that are
(29)                                 passed to the context object.
(4)         """
(4)         context_class: t.Type[Context] = Context
(4)         allow_extra_args = False
(4)         allow_interspersed_args = True
(4)         ignore_unknown_options = False
(4)         def __init__(
(8)             self,
(8)             name: t.Optional[str],
(8)             context_settings: t.Optional[t.MutableMapping[str, t.Any]] = None,
(4)         ) -> None:
(8)             self.name = name
(8)             if context_settings is None:
(12)                context_settings = {}
(8)             self.context_settings: t.MutableMapping[str, t.Any] = context_settings
(4)         def to_info_dict(self, ctx: Context) -> t.Dict[str, t.Any]:
(8)             """Gather information that could be useful for a tool generating
(8)             user-facing documentation. This traverses the entire structure
(8)             below this command.
(8)             Use :meth:`click.Context.to_info_dict` to traverse the entire
(8)             CLI structure.
(8)             :param ctx: A :class:`Context` representing this command.
(8)             .. versionadded:: 8.0
(8)             """
(8)             return {"name": self.name}
(4)         def __repr__(self) -> str:
(8)             return f"<{self.__class__.__name__} {self.name}>"
(4)         def get_usage(self, ctx: Context) -> str:
(8)             raise NotImplementedError("Base commands cannot get usage")
(4)         def get_help(self, ctx: Context) -> str:
(8)             raise NotImplementedError("Base commands cannot get help")
(4)         def make_context(
(8)             self,
(8)             info_name: t.Optional[str],
(8)         args: t.List[str],
(8)         parent: t.Optional[Context] = None,
(8)         **extra: t.Any,
(4)     ) -> Context:
(8)         """This function when given an info name and arguments will kick
(8)         off the parsing and create a new :class:`Context`.  It does not
(8)         invoke the actual command callback though.
(8)         To quickly customize the context class used without overriding
(8)         this method, set the :attr:`context_class` attribute.
(8)         :param info_name: the info name for this invocation.  Generally this
(26)                          is the most descriptive name for the script or
(26)                          command.  For the toplevel script it's usually
(26)                          the name of the script, for commands below it's
(26)                          the name of the command.
(8)         :param args: the arguments to parse as list of strings.
(8)         :param parent: the parent context if available.
(8)         :param extra: extra keyword arguments forwarded to the context
(22)                      constructor.
(8)         .. versionchanged:: 8.0
(12)            Added the :attr:`context_class` attribute.
(8)         """
(8)         for key, value in self.context_settings.items():
(12)            if key not in extra:
(16)                extra[key] = value
(8)         ctx = self.context_class(
(12)            self, info_name=info_name, parent=parent, **extra  # type: ignore
(8)         )
(8)         with ctx.scope(cleanup=False):
(12)            self.parse_args(ctx, args)
(8)         return ctx
(4)     def parse_args(self, ctx: Context, args: t.List[str]) -> t.List[str]:
(8)         """Given a context and a list of arguments this creates the parser
(8)         and parses the arguments, then modifies the context as necessary.
(8)         This is automatically invoked by :meth:`make_context`.
(8)         """
(8)         raise NotImplementedError("Base commands do not know how to parse arguments.")
(4)     def invoke(self, ctx: Context) -> t.Any:
(8)         """Given a context, this invokes the command.  The default
(8)         implementation is raising a not implemented error.
(8)         """
(8)         raise NotImplementedError("Base commands are not invocable by default")
(4)     def shell_complete(self, ctx: Context, incomplete: str) -> t.List["CompletionItem"]:
(8)         """Return a list of completions for the incomplete value. Looks
(8)         at the names of chained multi-commands.
(8)         Any command could be part of a chained multi-command, so sibling
(8)         commands are valid at any point during command completion. Other
(8)         command classes will return more completions.
(8)         :param ctx: Invocation context for this command.
(8)         :param incomplete: Value being completed. May be empty.
(8)         .. versionadded:: 8.0
(8)         """
(8)         from click.shell_completion import CompletionItem
(8)         results: t.List["CompletionItem"] = []
(8)         while ctx.parent is not None:
(12)            ctx = ctx.parent
(12)            if isinstance(ctx.command, MultiCommand) and ctx.command.chain:
(16)                results.extend(
(20)                    CompletionItem(name, help=command.get_short_help_str())
(20)                    for name, command in _complete_visible_commands(ctx, incomplete)
(20)                    if name not in ctx.protected_args
(16)                )
(8)         return results
(4)     @t.overload
(4)     def main(
(8)         self,
(8)          args: t.Optional[t.Sequence[str]] = None,
(8)          prog_name: t.Optional[str] = None,
(8)          complete_var: t.Optional[str] = None,
(8)          standalone_mode: "te.Literal[True]" = True,
(8)          **extra: t.Any,
(4)      ) -> "te.NoReturn":
(8)          ...
(4)      @t.overload
(4)      def main(
(8)          self,
(8)          args: t.Optional[t.Sequence[str]] = None,
(8)          prog_name: t.Optional[str] = None,
(8)          complete_var: t.Optional[str] = None,
(8)          standalone_mode: bool = ...,
(8)          **extra: t.Any,
(4)      ) -> t.Any:
(8)          ...
(4)      def main(
(8)          self,
(8)          args: t.Optional[t.Sequence[str]] = None,
(8)          prog_name: t.Optional[str] = None,
(8)          complete_var: t.Optional[str] = None,
(8)          standalone_mode: bool = True,
(8)          windows_expand_args: bool = True,
(8)          **extra: t.Any,
(4)      ) -> t.Any:
(8)          """This is the way to invoke a script with all the bells and
(8)          whistles as a command line application.  This will always terminate
(8)          the application after a call.  If this is not wanted, ``SystemExit``
(8)          needs to be caught.
(8)          This method is also available by directly calling the instance of
(8)          a :class:`Command`.
(8)          :param args: the arguments that should be used for parsing.  If not
(21)                      provided, ``sys.argv[1:]`` is used.
(8)          :param prog_name: the program name that should be used.  By default
(26)                           the program name is constructed by taking the file
(26)                           name from ``sys.argv[0]``.
(8)          :param complete_var: the environment variable that controls the
(29)                              bash completion support.  The default is
(29)                              ``"_<prog_name>_COMPLETE"`` with prog_name in
(29)                              uppercase.
(8)          :param standalone_mode: the default behavior is to invoke the script
(32)                                 in standalone mode.  Click will then
(32)                                 handle exceptions and convert them into
(32)                                 error messages and the function will never
(32)                                 return but shut down the interpreter.  If
(32)                                 this is set to `False` they will be
(32)                                 propagated to the caller and the return
(32)                                 value of this function is the return value
(32)                                 of :meth:`invoke`.
(8)          :param windows_expand_args: Expand glob patterns, user dir, and
(12)             env vars in command line args on Windows.
(8)          :param extra: extra keyword arguments are forwarded to the context
(22)                       constructor.  See :class:`Context` for more information.
(8)          .. versionchanged:: 8.0.1
(12)             Added the ``windows_expand_args`` parameter to allow
(12)             disabling command line arg expansion on Windows.
(8)          .. versionchanged:: 8.0
(12)             When taking arguments from ``sys.argv`` on Windows, glob
(12)             patterns, user dir, and env vars are expanded.
(8)          .. versionchanged:: 3.0
(11)            Added the ``standalone_mode`` parameter.
(8)          """
(8)          if args is None:
(12)             args = sys.argv[1:]
(12)             if os.name == "nt" and windows_expand_args:
(16)                 args = _expand_args(args)
(8)          else:
(12)             args = list(args)
```

```
        if prog_name is None:
            prog_name = _detect_program_name()
        self._main_shell_completion(extra, prog_name, complete_var)
        try:
            try:
                with self.make_context(prog_name, args, **extra) as ctx:
                    rv = self.invoke(ctx)
                    if not standalone_mode:
                        return rv
                    ctx.exit()
            except (EOFError, KeyboardInterrupt) as e:
                echo(file=sys.stderr)
                raise Abort() from e
            except ClickException as e:
                if not standalone_mode:
                    raise
                e.show()
                sys.exit(e.exit_code)
            except OSError as e:
                if e.errno == errno.EPIPE:
                    sys.stdout = t.cast(t.TextIO, PacifyFlushWrapper(sys.stdout))
                    sys.stderr = t.cast(t.TextIO, PacifyFlushWrapper(sys.stderr))
                    sys.exit(1)
                else:
                    raise
        except Exit as e:
            if standalone_mode:
                sys.exit(e.exit_code)
            else:
                return e.exit_code
        except Abort:
            if not standalone_mode:
                raise
            echo(_("Aborted!"), file=sys.stderr)
            sys.exit(1)
    def _main_shell_completion(
        self,
        ctx_args: t.MutableMapping[str, t.Any],
        prog_name: str,
        complete_var: t.Optional[str] = None,
    ) -> None:
        """Check if the shell is asking for tab completion, process
        that, then exit early. Called from :meth:`main` before the
        program is invoked.
        :param prog_name: Name of the executable in the shell.
        :param complete_var: Name of the environment variable that holds
            the completion instruction. Defaults to
            ``_{PROG_NAME}_COMPLETE``.
        .. versionchanged:: 8.2.0
            Dots (``.``) in ``prog_name`` are replaced with underscores (``_``).
        """
        if complete_var is None:
            complete_name = prog_name.replace("-", "_").replace(".", "_")
            complete_var = f"_{complete_name}_COMPLETE".upper()
        instruction = os.environ.get(complete_var)
        if not instruction:
            return
        from .shell_completion import shell_complete
        rv = shell_complete(self, ctx_args, prog_name, complete_var, instruction)
        sys.exit(rv)
    def __call__(self, *args: t.Any, **kwargs: t.Any) -> t.Any:
        """Alias for :meth:`main`."""
        return self.main(*args, **kwargs)
class Command(BaseCommand):
    """Commands are the basic building block of command line interfaces in
```

```
(4)       Click.  A basic command handles command line parsing and might dispatch
(4)       more parsing to commands nested below it.
(4)       :param name: the name of the command to use unless a group overrides it.
(4)       :param context_settings: an optional dictionary with defaults that are
(29)                               passed to the context object.
(4)       :param callback: the callback to invoke.  This is optional.
(4)       :param params: the parameters to register with this command.  This can
(19)                     be either :class:`Option` or :class:`Argument` objects.
(4)       :param help: the help string to use for this command.
(4)       :param epilog: like the help string but it's printed at the end of the
(19)                     help page after everything else.
(4)       :param short_help: the short help to use for this command.  This is
(23)                         shown on the command listing of the parent command.
(4)       :param add_help_option: by default each command registers a ``--help``
(28)                              option.  This can be disabled by this parameter.
(4)       :param no_args_is_help: this controls what happens if no arguments are
(28)                              provided.  This option is disabled by default.
(28)                              If enabled this will add ``--help`` as argument
(28)                              if no arguments are passed
(4)       :param hidden: hide this command from help outputs.
(4)       :param deprecated: issues a message indicating that
(29)                               the command is deprecated.
(4)       .. versionchanged:: 8.1
(8)           ``help``, ``epilog``, and ``short_help`` are stored unprocessed,
(8)           all formatting is done when outputting help text, not at init,
(8)           and is done even if not using the ``@command`` decorator.
(4)       .. versionchanged:: 8.0
(8)           Added a ``repr`` showing the command name.
(4)       .. versionchanged:: 7.1
(8)           Added the ``no_args_is_help`` parameter.
(4)       .. versionchanged:: 2.0
(8)           Added the ``context_settings`` parameter.
(4)       """
(4)       def __init__(
(8)           self,
(8)           name: t.Optional[str],
(8)           context_settings: t.Optional[t.MutableMapping[str, t.Any]] = None,
(8)           callback: t.Optional[t.Callable[..., t.Any]] = None,
(8)           params: t.Optional[t.List["Parameter"]] = None,
(8)           help: t.Optional[str] = None,
(8)           epilog: t.Optional[str] = None,
(8)           short_help: t.Optional[str] = None,
(8)           options_metavar: t.Optional[str] = "[OPTIONS]",
(8)           add_help_option: bool = True,
(8)           no_args_is_help: bool = False,
(8)           hidden: bool = False,
(8)           deprecated: bool = False,
(4)       ) -> None:
(8)           super().__init__(name, context_settings)
(8)           self.callback = callback
(8)           self.params: t.List["Parameter"] = params or []
(8)           self.help = help
(8)           self.epilog = epilog
(8)           self.options_metavar = options_metavar
(8)           self.short_help = short_help
(8)           self.add_help_option = add_help_option
(8)           self.no_args_is_help = no_args_is_help
(8)           self.hidden = hidden
(8)           self.deprecated = deprecated
(4)       def to_info_dict(self, ctx: Context) -> t.Dict[str, t.Any]:
(8)           info_dict = super().to_info_dict(ctx)
(8)           info_dict.update(
(12)              params=[param.to_info_dict() for param in self.get_params(ctx)],
(12)              help=self.help,
(12)              epilog=self.epilog,
(12)              short_help=self.short_help,
(12)              hidden=self.hidden,
(12)              deprecated=self.deprecated,
(8)           )
```

```
        return info_dict
    def get_usage(self, ctx: Context) -> str:
        """Formats the usage line into a string and returns it.
        Calls :meth:`format_usage` internally.
        """
        formatter = ctx.make_formatter()
        self.format_usage(ctx, formatter)
        return formatter.getvalue().rstrip("\n")
    def get_params(self, ctx: Context) -> t.List["Parameter"]:
        rv = self.params
        help_option = self.get_help_option(ctx)
        if help_option is not None:
            rv = [*rv, help_option]
        return rv
    def format_usage(self, ctx: Context, formatter: HelpFormatter) -> None:
        """Writes the usage line into the formatter.
        This is a low-level method called by :meth:`get_usage`.
        """
        pieces = self.collect_usage_pieces(ctx)
        formatter.write_usage(ctx.command_path, " ".join(pieces))
    def collect_usage_pieces(self, ctx: Context) -> t.List[str]:
        """Returns all the pieces that go into the usage line and returns
        it as a list of strings.
        """
        rv = [self.options_metavar] if self.options_metavar else []
        for param in self.get_params(ctx):
            rv.extend(param.get_usage_pieces(ctx))
        return rv
    def get_help_option_names(self, ctx: Context) -> t.List[str]:
        """Returns the names for the help option."""
        all_names = set(ctx.help_option_names)
        for param in self.params:
            all_names.difference_update(param.opts)
            all_names.difference_update(param.secondary_opts)
        return list(all_names)
    def get_help_option(self, ctx: Context) -> t.Optional["Option"]:
        """Returns the help option object."""
        help_options = self.get_help_option_names(ctx)
        if not help_options or not self.add_help_option:
            return None
        def show_help(ctx: Context, param: "Parameter", value: str) -> None:
            if value and not ctx.resilient_parsing:
                echo(ctx.get_help(), color=ctx.color)
                ctx.exit()
        return Option(
            help_options,
            is_flag=True,
            is_eager=True,
            expose_value=False,
            callback=show_help,
            help=_("Show this message and exit."),
        )
    def make_parser(self, ctx: Context) -> OptionParser:
        """Creates the underlying option parser for this command."""
        parser = OptionParser(ctx)
        for param in self.get_params(ctx):
            param.add_to_parser(parser, ctx)
        return parser
    def get_help(self, ctx: Context) -> str:
        """Formats the help into a string and returns it.
        Calls :meth:`format_help` internally.
        """
        formatter = ctx.make_formatter()
        self.format_help(ctx, formatter)
        return formatter.getvalue().rstrip("\n")
    def get_short_help_str(self, limit: int = 45) -> str:
        """Gets short help for the command or makes it by shortening the
        long help string.
        """
```

```
(8)                 if self.short_help:
(12)                    text = inspect.cleandoc(self.short_help)
(8)                 elif self.help:
(12)                    text = make_default_short_help(self.help, limit)
(8)                 else:
(12)                    text = ""
(8)                 if self.deprecated:
(12)                    text = _("(Deprecated) {text}").format(text=text)
(8)                 return text.strip()
(4)             def format_help(self, ctx: Context, formatter: HelpFormatter) -> None:
(8)                 """Writes the help into the formatter if it exists.
(8)                 This is a low-level method called by :meth:`get_help`.
(8)                 This calls the following methods:
(8)                 -   :meth:`format_usage`
(8)                 -   :meth:`format_help_text`
(8)                 -   :meth:`format_options`
(8)                 -   :meth:`format_epilog`
(8)                 """
(8)                 self.format_usage(ctx, formatter)
(8)                 self.format_help_text(ctx, formatter)
(8)                 self.format_options(ctx, formatter)
(8)                 self.format_epilog(ctx, formatter)
(4)             def format_help_text(self, ctx: Context, formatter: HelpFormatter) -> None:
(8)                 """Writes the help text to the formatter if it exists."""
(8)                 if self.help is not None:
(12)                    text = inspect.cleandoc(self.help).partition("\f")[0]
(8)                 else:
(12)                    text = ""
(8)                 if self.deprecated:
(12)                    text = _("(Deprecated) {text}").format(text=text)
(8)                 if text:
(12)                    formatter.write_paragraph()
(12)                    with formatter.indentation():
(16)                        formatter.write_text(text)
(4)             def format_options(self, ctx: Context, formatter: HelpFormatter) -> None:
(8)                 """Writes all the options into the formatter if they exist."""
(8)                 opts = []
(8)                 for param in self.get_params(ctx):
(12)                    rv = param.get_help_record(ctx)
(12)                    if rv is not None:
(16)                        opts.append(rv)
(8)                 if opts:
(12)                    with formatter.section(_("Options")):
(16)                        formatter.write_dl(opts)
(4)             def format_epilog(self, ctx: Context, formatter: HelpFormatter) -> None:
(8)                 """Writes the epilog into the formatter if it exists."""
(8)                 if self.epilog:
(12)                    epilog = inspect.cleandoc(self.epilog)
(12)                    formatter.write_paragraph()
(12)                    with formatter.indentation():
(16)                        formatter.write_text(epilog)
(4)             def parse_args(self, ctx: Context, args: t.List[str]) -> t.List[str]:
(8)                 if not args and self.no_args_is_help and not ctx.resilient_parsing:
(12)                    echo(ctx.get_help(), color=ctx.color)
(12)                    ctx.exit()
(8)                 parser = self.make_parser(ctx)
(8)                 opts, args, param_order = parser.parse_args(args=args)
(8)                 for param in iter_params_for_processing(param_order, self.get_params(ctx)):
(12)                    value, args = param.handle_parse_result(ctx, opts, args)
(8)                 if args and not ctx.allow_extra_args and not ctx.resilient_parsing:
(12)                    ctx.fail(
(16)                        ngettext(
(20)                            "Got unexpected extra argument ({args})",
(20)                            "Got unexpected extra arguments ({args})",
(20)                            len(args),
(16)                        ).format(args=" ".join(map(str, args)))
(12)                    )
```

```
(8)                         ctx.args = args
(8)                         ctx._opt_prefixes.update(parser._opt_prefixes)
(8)                         return args
(4)                     def invoke(self, ctx: Context) -> t.Any:
(8)                         """Given a context, this invokes the attached callback (if it exists)
(8)                         in the right way.
(8)                         """
(8)                         if self.deprecated:
(12)                            message = _(
(16)                                "DeprecationWarning: The command {name!r} is deprecated."
(12)                            ).format(name=self.name)
(12)                            echo(style(message, fg="red"), err=True)
(8)                         if self.callback is not None:
(12)                            return ctx.invoke(self.callback, **ctx.params)
(4)                     def shell_complete(self, ctx: Context, incomplete: str) ->
t.List["CompletionItem"]:
(8)                         """Return a list of completions for the incomplete value. Looks
(8)                         at the names of options and chained multi-commands.
(8)                         :param ctx: Invocation context for this command.
(8)                         :param incomplete: Value being completed. May be empty.
(8)                         .. versionadded:: 8.0
(8)                         """
(8)                         from click.shell_completion import CompletionItem
(8)                         results: t.List["CompletionItem"] = []
(8)                         if incomplete and not incomplete[0].isalnum():
(12)                            for param in self.get_params(ctx):
(16)                                if (
(20)                                    not isinstance(param, Option)
(20)                                    or param.hidden
(20)                                    or (
(24)                                        not param.multiple
(24)                                        and ctx.get_parameter_source(param.name)  # type:
ignore
(24)                                        is ParameterSource.COMMANDLINE
(20)                                    )
(16)                                ):
(20)                                    continue
(16)                                results.extend(
(20)                                    CompletionItem(name, help=param.help)
(20)                                    for name in [*param.opts, *param.secondary_opts]
(20)                                    if name.startswith(incomplete)
(16)                                )
(8)                         results.extend(super().shell_complete(ctx, incomplete))
(8)                         return results
(0)                 class MultiCommand(Command):
(4)                     """A multi command is the basic implementation of a command that
(4)                     dispatches to subcommands.  The most common version is the
(4)                     :class:`Group`.
(4)                     :param invoke_without_command: this controls how the multi command itself
(35)                                                    is invoked.  By default it's only invoked
(35)                                                    if a subcommand is provided.
(4)                     :param no_args_is_help: this controls what happens if no arguments are
(28)                                            provided.  This option is enabled by default if
(28)                                            `invoke_without_command` is disabled or disabled
(28)                                            if it's enabled.  If enabled this will add
(28)                                            ``--help`` as argument if no arguments are
(28)                                            passed.
(4)                     :param subcommand_metavar: the string that is used in the documentation
(31)                                               to indicate the subcommand place.
(4)                     :param chain: if this is set to `True` chaining of multiple subcommands
(18)                                  is enabled.  This restricts the form of commands in that
(18)                                  they cannot have optional arguments but it allows
(18)                                  multiple commands to be chained together.
(4)                     :param result_callback: The result callback to attach to this multi
(8)                         command. This can be set or changed later with the
(8)                         :meth:`result_callback` decorator.
(4)                     :param attrs: Other command arguments described in :class:`Command`.
(4)                     """
(4)                     allow_extra_args = True
```

```
(4)     allow_interspersed_args = False
(4)     def __init__(
(8)         self,
(8)         name: t.Optional[str] = None,
(8)         invoke_without_command: bool = False,
(8)         no_args_is_help: t.Optional[bool] = None,
(8)         subcommand_metavar: t.Optional[str] = None,
(8)         chain: bool = False,
(8)         result_callback: t.Optional[t.Callable[..., t.Any]] = None,
(8)         **attrs: t.Any,
(4)     ) -> None:
(8)         super().__init__(name, **attrs)
(8)         if no_args_is_help is None:
(12)            no_args_is_help = not invoke_without_command
(8)         self.no_args_is_help = no_args_is_help
(8)         self.invoke_without_command = invoke_without_command
(8)         if subcommand_metavar is None:
(12)            if chain:
(16)                subcommand_metavar = "COMMAND1 [ARGS]... [COMMAND2 [ARGS]...]..."
(12)            else:
(16)                subcommand_metavar = "COMMAND [ARGS]..."
(8)         self.subcommand_metavar = subcommand_metavar
(8)         self.chain = chain
(8)         self._result_callback = result_callback
(8)         if self.chain:
(12)            for param in self.params:
(16)                if isinstance(param, Argument) and not param.required:
(20)                    raise RuntimeError(
(24)                        "Multi commands in chain mode cannot have"
(24)                        " optional arguments."
(20)                    )
(4)     def to_info_dict(self, ctx: Context) -> t.Dict[str, t.Any]:
(8)         info_dict = super().to_info_dict(ctx)
(8)         commands = {}
(8)         for name in self.list_commands(ctx):
(12)            command = self.get_command(ctx, name)
(12)            if command is None:
(16)                continue
(12)            sub_ctx = ctx._make_sub_context(command)
(12)            with sub_ctx.scope(cleanup=False):
(16)                commands[name] = command.to_info_dict(sub_ctx)
(8)         info_dict.update(commands=commands, chain=self.chain)
(8)         return info_dict
(4)     def collect_usage_pieces(self, ctx: Context) -> t.List[str]:
(8)         rv = super().collect_usage_pieces(ctx)
(8)         rv.append(self.subcommand_metavar)
(8)         return rv
(4)     def format_options(self, ctx: Context, formatter: HelpFormatter) -> None:
(8)         super().format_options(ctx, formatter)
(8)         self.format_commands(ctx, formatter)
(4)     def result_callback(self, replace: bool = False) -> t.Callable[[F], F]:
(8)         """Adds a result callback to the command.  By default if a
(8)         result callback is already registered this will chain them but
(8)         this can be disabled with the `replace` parameter.  The result
(8)         callback is invoked with the return value of the subcommand
(8)         (or the list of return values from all subcommands if chaining
(8)         is enabled) as well as the parameters as they would be passed
(8)         to the main callback.
(8)         Example::
(12)            @click.group()
(12)            @click.option('-i', '--input', default=23)
(12)            def cli(input):
(16)                return 42
(12)            @cli.result_callback()
(12)            def process_result(result, input):
(16)                return result + input
(8)         :param replace: if set to `True` an already existing result
(24)                        callback will be removed.
```

```
(8)                     .. versionchanged:: 8.0
(12)                        Renamed from ``resultcallback``.
(8)                     .. versionadded:: 3.0
(8)                     """
(8)                     def decorator(f: F) -> F:
(12)                        old_callback = self._result_callback
(12)                        if old_callback is None or replace:
(16)                            self._result_callback = f
(16)                            return f
(12)                        def function(__value, *args, **kwargs):  # type: ignore
(16)                            inner = old_callback(__value, *args, **kwargs)
(16)                            return f(inner, *args, **kwargs)
(12)                        self._result_callback = rv = update_wrapper(t.cast(F, function), f)
(12)                        return rv
(8)                     return decorator
(4)                 def format_commands(self, ctx: Context, formatter: HelpFormatter) -> None:
(8)                     """Extra format methods for multi methods that adds all the commands
(8)                     after the options.
(8)                     """
(8)                     commands = []
(8)                     for subcommand in self.list_commands(ctx):
(12)                        cmd = self.get_command(ctx, subcommand)
(12)                        if cmd is None:
(16)                            continue
(12)                        if cmd.hidden:
(16)                            continue
(12)                        commands.append((subcommand, cmd))
(8)                     if len(commands):
(12)                        limit = formatter.width - 6 - max(len(cmd[0]) for cmd in commands)
(12)                        rows = []
(12)                        for subcommand, cmd in commands:
(16)                            help = cmd.get_short_help_str(limit)
(16)                            rows.append((subcommand, help))
(12)                        if rows:
(16)                            with formatter.section(_("Commands")):
(20)                                formatter.write_dl(rows)
(4)                 def parse_args(self, ctx: Context, args: t.List[str]) -> t.List[str]:
(8)                     if not args and self.no_args_is_help and not ctx.resilient_parsing:
(12)                        echo(ctx.get_help(), color=ctx.color)
(12)                        ctx.exit()
(8)                     rest = super().parse_args(ctx, args)
(8)                     if self.chain:
(12)                        ctx.protected_args = rest
(12)                        ctx.args = []
(8)                     elif rest:
(12)                        ctx.protected_args, ctx.args = rest[:1], rest[1:]
(8)                     return ctx.args
(4)                 def invoke(self, ctx: Context) -> t.Any:
(8)                     def _process_result(value: t.Any) -> t.Any:
(12)                        if self._result_callback is not None:
(16)                            value = ctx.invoke(self._result_callback, value, **ctx.params)
(12)                        return value
(8)                     if not ctx.protected_args:
(12)                        if self.invoke_without_command:
(16)                            with ctx:
(20)                                rv = super().invoke(ctx)
(20)                                return _process_result([] if self.chain else rv)
(12)                        ctx.fail(_("Missing command."))
(8)                     args = [*ctx.protected_args, *ctx.args]
(8)                     ctx.args = []
(8)                     ctx.protected_args = []
(8)                     if not self.chain:
(12)                        with ctx:
(16)                            cmd_name, cmd, args = self.resolve_command(ctx, args)
(16)                            assert cmd is not None
(16)                            ctx.invoked_subcommand = cmd_name
(16)                            super().invoke(ctx)
(16)                            sub_ctx = cmd.make_context(cmd_name, args, parent=ctx)
```

```
1281: (16)                          with sub_ctx:
1282: (20)                              return _process_result(sub_ctx.command.invoke(sub_ctx))
1283: (8)                   with ctx:
1284: (12)                      ctx.invoked_subcommand = "*" if args else None
1285: (12)                      super().invoke(ctx)
1286: (12)                      contexts = []
1287: (12)                      while args:
1288: (16)                          cmd_name, cmd, args = self.resolve_command(ctx, args)
1289: (16)                          assert cmd is not None
1290: (16)                          sub_ctx = cmd.make_context(
1291: (20)                              cmd_name,
1292: (20)                              args,
1293: (20)                              parent=ctx,
1294: (20)                              allow_extra_args=True,
1295: (20)                              allow_interspersed_args=False,
1296: (16)                          )
1297: (16)                          contexts.append(sub_ctx)
1298: (16)                          args, sub_ctx.args = sub_ctx.args, []
1299: (12)                      rv = []
1300: (12)                      for sub_ctx in contexts:
1301: (16)                          with sub_ctx:
1302: (20)                              rv.append(sub_ctx.command.invoke(sub_ctx))
1303: (12)                      return _process_result(rv)
1304: (4)               def resolve_command(
1305: (8)                   self, ctx: Context, args: t.List[str]
1306: (4)               ) -> t.Tuple[t.Optional[str], t.Optional[Command], t.List[str]]:
1307: (8)                   cmd_name = make_str(args[0])
1308: (8)                   original_cmd_name = cmd_name
1309: (8)                   cmd = self.get_command(ctx, cmd_name)
1310: (8)                   if cmd is None and ctx.token_normalize_func is not None:
1311: (12)                      cmd_name = ctx.token_normalize_func(cmd_name)
1312: (12)                      cmd = self.get_command(ctx, cmd_name)
1313: (8)                   if cmd is None and not ctx.resilient_parsing:
1314: (12)                      if split_opt(cmd_name)[0]:
1315: (16)                          self.parse_args(ctx, ctx.args)
1316: (12)                      ctx.fail(_("No such command
{name!r}.").format(name=original_cmd_name))
1317: (8)                   return cmd_name if cmd else None, cmd, args[1:]
1318: (4)               def get_command(self, ctx: Context, cmd_name: str) -> t.Optional[Command]:
1319: (8)                   """Given a context and a command name, this returns a
1320: (8)                   :class:`Command` object if it exists or returns `None`.
1321: (8)                   """
1322: (8)                   raise NotImplementedError
1323: (4)               def list_commands(self, ctx: Context) -> t.List[str]:
1324: (8)                   """Returns a list of subcommand names in the order they should
1325: (8)                   appear.
1326: (8)                   """
1327: (8)                   return []
1328: (4)               def shell_complete(self, ctx: Context, incomplete: str) ->
t.List["CompletionItem"]:
1329: (8)                   """Return a list of completions for the incomplete value. Looks
1330: (8)                   at the names of options, subcommands, and chained
1331: (8)                   multi-commands.
1332: (8)                   :param ctx: Invocation context for this command.
1333: (8)                   :param incomplete: Value being completed. May be empty.
1334: (8)                   .. versionadded:: 8.0
1335: (8)                   """
1336: (8)                   from click.shell_completion import CompletionItem
1337: (8)                   results = [
1338: (12)                      CompletionItem(name, help=command.get_short_help_str())
1339: (12)                      for name, command in _complete_visible_commands(ctx, incomplete)
1340: (8)                   ]
1341: (8)                   results.extend(super().shell_complete(ctx, incomplete))
1342: (8)                   return results
1343: (0)           class Group(MultiCommand):
1344: (4)               """A group allows a command to have subcommands attached. This is
1345: (4)               the most common way to implement nesting in Click.
1346: (4)               :param name: The name of the group command.
1347: (4)               :param commands: A dict mapping names to :class:`Command` objects.
```

```
(8)          Can also be a list of :class:`Command`, which will use
(8)          :attr:`Command.name` to create the dict.
(4)      :param attrs: Other command arguments described in
(8)          :class:`MultiCommand`, :class:`Command`, and
(8)          :class:`BaseCommand`.
(4)      .. versionchanged:: 8.0
(8)          The ``commands`` argument can be a list of command objects.
(4)      """
(4)      command_class: t.Optional[t.Type[Command]] = None
(4)      group_class: t.Optional[t.Union[t.Type["Group"], t.Type[type]]] = None
(4)      def __init__(
(8)          self,
(8)          name: t.Optional[str] = None,
(8)          commands: t.Optional[
(12)             t.Union[t.MutableMapping[str, Command], t.Sequence[Command]]
(8)          ] = None,
(8)          **attrs: t.Any,
(4)      ) -> None:
(8)          super().__init__(name, **attrs)
(8)          if commands is None:
(12)             commands = {}
(8)          elif isinstance(commands, abc.Sequence):
(12)             commands = {c.name: c for c in commands if c.name is not None}
(8)          self.commands: t.MutableMapping[str, Command] = commands
(4)      def add_command(self, cmd: Command, name: t.Optional[str] = None) -> None:
(8)          """Registers another :class:`Command` with this group.  If the name
(8)          is not provided, the name of the command is used.
(8)          """
(8)          name = name or cmd.name
(8)          if name is None:
(12)             raise TypeError("Command has no name.")
(8)          _check_multicommand(self, name, cmd, register=True)
(8)          self.commands[name] = cmd
(4)      @t.overload
(4)      def command(self, __func: t.Callable[..., t.Any]) -> Command:
(8)          ...
(4)      @t.overload
(4)      def command(
(8)          self, *args: t.Any, **kwargs: t.Any
(4)      ) -> t.Callable[[t.Callable[..., t.Any]], Command]:
(8)          ...
(4)      def command(
(8)          self, *args: t.Any, **kwargs: t.Any
(4)      ) -> t.Union[t.Callable[[t.Callable[..., t.Any]], Command], Command]:
(8)          """A shortcut decorator for declaring and attaching a command to
(8)          the group. This takes the same arguments as :func:`command` and
(8)          immediately registers the created command with this group by
(8)          calling :meth:`add_command`.
(8)          To customize the command class used, set the
(8)          :attr:`command_class` attribute.
(8)          .. versionchanged:: 8.1
(12)             This decorator can be applied without parentheses.
(8)          .. versionchanged:: 8.0
(12)             Added the :attr:`command_class` attribute.
(8)          """
(8)          from .decorators import command
(8)          func: t.Optional[t.Callable[..., t.Any]] = None
(8)          if args and callable(args[0]):
(12)             assert (
(16)                 len(args) == 1 and not kwargs
(12)             ), "Use 'command(**kwargs)(callable)' to provide arguments."
(12)             (func,) = args
(12)             args = ()
(8)          if self.command_class and kwargs.get("cls") is None:
(12)             kwargs["cls"] = self.command_class
(8)          def decorator(f: t.Callable[..., t.Any]) -> Command:
(12)             cmd: Command = command(*args, **kwargs)(f)
(12)             self.add_command(cmd)
(12)             return cmd
```

```
(8)             if func is not None:
(12)                return decorator(func)
(8)             return decorator
(4)         @t.overload
(4)         def group(self, __func: t.Callable[..., t.Any]) -> "Group":
(8)             ...
(4)         @t.overload
(4)         def group(
(8)             self, *args: t.Any, **kwargs: t.Any
(4)         ) -> t.Callable[[t.Callable[..., t.Any]], "Group"]:
(8)             ...
(4)         def group(
(8)             self, *args: t.Any, **kwargs: t.Any
(4)         ) -> t.Union[t.Callable[[t.Callable[..., t.Any]], "Group"], "Group"]:
(8)             """A shortcut decorator for declaring and attaching a group to
(8)             the group. This takes the same arguments as :func:`group` and
(8)             immediately registers the created group with this group by
(8)             calling :meth:`add_command`.
(8)             To customize the group class used, set the :attr:`group_class`
(8)             attribute.
(8)             .. versionchanged:: 8.1
(12)                This decorator can be applied without parentheses.
(8)             .. versionchanged:: 8.0
(12)                Added the :attr:`group_class` attribute.
(8)             """
(8)             from .decorators import group
(8)             func: t.Optional[t.Callable[..., t.Any]] = None
(8)             if args and callable(args[0]):
(12)                assert (
(16)                    len(args) == 1 and not kwargs
(12)                ), "Use 'group(**kwargs)(callable)' to provide arguments."
(12)                (func,) = args
(12)                args = ()
(8)             if self.group_class is not None and kwargs.get("cls") is None:
(12)                if self.group_class is type:
(16)                    kwargs["cls"] = type(self)
(12)                else:
(16)                    kwargs["cls"] = self.group_class
(8)             def decorator(f: t.Callable[..., t.Any]) -> "Group":
(12)                cmd: Group = group(*args, **kwargs)(f)
(12)                self.add_command(cmd)
(12)                return cmd
(8)             if func is not None:
(12)                return decorator(func)
(8)             return decorator
(4)         def get_command(self, ctx: Context, cmd_name: str) -> t.Optional[Command]:
(8)             return self.commands.get(cmd_name)
(4)         def list_commands(self, ctx: Context) -> t.List[str]:
(8)             return sorted(self.commands)
(0)     class CommandCollection(MultiCommand):
(4)         """A command collection is a multi command that merges multiple multi
(4)         commands together into one.  This is a straightforward implementation
(4)         that accepts a list of different multi commands as sources and
(4)         provides all the commands for each of them.
(4)         See :class:`MultiCommand` and :class:`Command` for the description of
(4)         ``name`` and ``attrs``.
(4)         """
(4)         def __init__(
(8)             self,
(8)             name: t.Optional[str] = None,
(8)             sources: t.Optional[t.List[MultiCommand]] = None,
(8)             **attrs: t.Any,
(4)         ) -> None:
(8)             super().__init__(name, **attrs)
(8)             self.sources: t.List[MultiCommand] = sources or []
(4)         def add_source(self, multi_cmd: MultiCommand) -> None:
(8)             """Adds a new multi command to the chain dispatcher."""
(8)             self.sources.append(multi_cmd)
(4)         def get_command(self, ctx: Context, cmd_name: str) -> t.Optional[Command]:
```

```
(8)                 for source in self.sources:
(12)                    rv = source.get_command(ctx, cmd_name)
(12)                    if rv is not None:
(16)                        if self.chain:
(20)                            _check_multicommand(self, cmd_name, rv)
(16)                        return rv
(8)                 return None
(4)             def list_commands(self, ctx: Context) -> t.List[str]:
(8)                 rv: t.Set[str] = set()
(8)                 for source in self.sources:
(12)                    rv.update(source.list_commands(ctx))
(8)                 return sorted(rv)
(0)         def _check_iter(value: t.Any) -> t.Iterator[t.Any]:
(4)             """Check if the value is iterable but not a string. Raises a type
(4)             error, or return an iterator over the value.
(4)             """
(4)             if isinstance(value, str):
(8)                 raise TypeError
(4)             return iter(value)
(0)         class Parameter:
(4)             r"""A parameter to a command comes in two versions: they are either
(4)             :class:`Option`\s or :class:`Argument`\s.  Other subclasses are currently
(4)             not supported by design as some of the internals for parsing are
(4)             intentionally not finalized.
(4)             Some settings are supported by both options and arguments.
(4)             :param param_decls: the parameter declarations for this option or
(24)                                argument.  This is a list of flags or argument
(24)                                names.
(4)             :param type: the type that should be used.  Either a :class:`ParamType`
(17)                         or a Python type.  The latter is converted into the former
(17)                         automatically if supported.
(4)             :param required: controls if this is optional or not.
(4)             :param default: the default value if omitted.  This can also be a
callable,
(20)                            in which case it's invoked when the default is needed
(20)                            without any arguments.
(4)             :param callback: A function to further process or validate the value
(8)                 after type conversion. It is called as ``f(ctx, param, value)``
(8)                 and must return the value. It is called for all sources,
(8)                 including prompts.
(4)             :param nargs: the number of arguments to match.  If not ``1`` the return
(18)                          value is a tuple instead of single value.  The default for
(18)                          nargs is ``1`` (except if the type is a tuple, then it's
(18)                          the arity of the tuple). If ``nargs=-1``, all remaining
(18)                          parameters are collected.
(4)             :param metavar: how the value is represented in the help page.
(4)             :param expose_value: if this is `True` then the value is passed onwards
(25)                                 to the command callback and stored on the context,
(25)                                 otherwise it's skipped.
(4)             :param is_eager: eager values are processed before non eager ones.  This
(21)                             should not be set for arguments or it will inverse the
(21)                             order of processing.
(4)             :param envvar: a string or list of strings that are environment variables
(19)                           that should be checked.
(4)             :param shell_complete: A function that returns custom shell
(8)                 completions. Used instead of the param's type completion if
(8)                 given. Takes ``ctx, param, incomplete`` and must return a list
(8)                 of :class:`~click.shell_completion.CompletionItem` or a list of
(8)                 strings.
(4)             .. versionchanged:: 8.0
(8)                 ``process_value`` validates required parameters and bounded
(8)                 ``nargs``, and invokes the parameter callback before returning
(8)                 the value. This allows the callback to validate prompts.
(8)                 ``full_process_value`` is removed.
(4)             .. versionchanged:: 8.0
(8)                 ``autocompletion`` is renamed to ``shell_complete`` and has new
(8)                 semantics described above. The old name is deprecated and will
(8)                 be removed in 8.1, until then it will be wrapped to match the
(8)                 new requirements.
```

```
    .. versionchanged:: 8.0
        For ``multiple=True, nargs>1``, the default must be a list of
        tuples.
    .. versionchanged:: 8.0
        Setting a default is no longer required for ``nargs>1``, it will
        default to ``None``. ``multiple=True`` or ``nargs=-1`` will
        default to ``()``.
    .. versionchanged:: 7.1
        Empty environment variables are ignored rather than taking the
        empty string value. This makes it possible for scripts to clear
        variables if they can't unset them.
    .. versionchanged:: 2.0
        Changed signature for parameter callback to also be passed the
        parameter. The old callback format will still work, but it will
        raise a warning to give you a chance to migrate the code easier.
    """
    param_type_name = "parameter"
    def __init__(
        self,
        param_decls: t.Optional[t.Sequence[str]] = None,
        type: t.Optional[t.Union[types.ParamType, t.Any]] = None,
        required: bool = False,
        default: t.Optional[t.Union[t.Any, t.Callable[[], t.Any]]] = None,
        callback: t.Optional[t.Callable[[Context, "Parameter", t.Any], t.Any]] = None,
        nargs: t.Optional[int] = None,
        multiple: bool = False,
        metavar: t.Optional[str] = None,
        expose_value: bool = True,
        is_eager: bool = False,
        envvar: t.Optional[t.Union[str, t.Sequence[str]]] = None,
        shell_complete: t.Optional[
            t.Callable[
                [Context, "Parameter", str],
                t.Union[t.List["CompletionItem"], t.List[str]],
            ]
        ] = None,
    ) -> None:
        self.name: t.Optional[str]
        self.opts: t.List[str]
        self.secondary_opts: t.List[str]
        self.name, self.opts, self.secondary_opts = self._parse_decls(
            param_decls or (), expose_value
        )
        self.type: types.ParamType = types.convert_type(type, default)
        if nargs is None:
            if self.type.is_composite:
                nargs = self.type.arity
            else:
                nargs = 1
        self.required = required
        self.callback = callback
        self.nargs = nargs
        self.multiple = multiple
        self.expose_value = expose_value
        self.default = default
        self.is_eager = is_eager
        self.metavar = metavar
        self.envvar = envvar
        self._custom_shell_complete = shell_complete
        if __debug__:
            if self.type.is_composite and nargs != self.type.arity:
                raise ValueError(
                    f"'nargs' must be {self.type.arity} (or None) for"
                    f" type {self.type!r}, but it was {nargs}."
                )
            check_default = default if not callable(default) else None
            if check_default is not None:
                if multiple:
```

```python
                    try:
                        check_default = next(_check_iter(check_default), None)
                    except TypeError:
                        raise ValueError(
                            "'default' must be a list when 'multiple' is true."
                        ) from None
                if nargs != 1 and check_default is not None:
                    try:
                        _check_iter(check_default)
                    except TypeError:
                        if multiple:
                            message = (
                                "'default' must be a list of lists when 'multiple' is"
                                " true and 'nargs' != 1."
                            )
                        else:
                            message = "'default' must be a list when 'nargs' != 1."
                        raise ValueError(message) from None
                    if nargs > 1 and len(check_default) != nargs:
                        subject = "item length" if multiple else "length"
                        raise ValueError(
                            f"'default' {subject} must match nargs={nargs}."
                        )
    def to_info_dict(self) -> t.Dict[str, t.Any]:
        """Gather information that could be useful for a tool generating
        user-facing documentation.
        Use :meth:`click.Context.to_info_dict` to traverse the entire
        CLI structure.
        .. versionadded:: 8.0
        """
        return {
            "name": self.name,
            "param_type_name": self.param_type_name,
            "opts": self.opts,
            "secondary_opts": self.secondary_opts,
            "type": self.type.to_info_dict(),
            "required": self.required,
            "nargs": self.nargs,
            "multiple": self.multiple,
            "default": self.default,
            "envvar": self.envvar,
        }
    def __repr__(self) -> str:
        return f"<{self.__class__.__name__} {self.name}>"
    def _parse_decls(
        self, decls: t.Sequence[str], expose_value: bool
    ) -> t.Tuple[t.Optional[str], t.List[str], t.List[str]]:
        raise NotImplementedError()
    @property
    def human_readable_name(self) -> str:
        """Returns the human readable name of this parameter.  This is the
        same as the name for options, but the metavar for arguments.
        """
        return self.name  # type: ignore
    def make_metavar(self) -> str:
        if self.metavar is not None:
            return self.metavar
        metavar = self.type.get_metavar(self)
        if metavar is None:
            metavar = self.type.name.upper()
        if self.nargs != 1:
            metavar += "..."
        return metavar
    @t.overload
    def get_default(
        self, ctx: Context, call: "te.Literal[True]" = True
    ) -> t.Optional[t.Any]:
        ...
    @t.overload
    def get_default(
        self, ctx: Context, call: bool = ...
    ) -> t.Optional[t.Union[t.Any, t.Callable[[], t.Any]]]:
        ...
    def get_default(
        self, ctx: Context, call: bool = True
    ) -> t.Optional[t.Union[t.Any, t.Callable[[], t.Any]]]:
        """Get the default for the parameter. Tries
        :meth:`Context.lookup_default` first, then the local default.
        :param ctx: Current context.
        :param call: If the default is a callable, call it. Disable to
            return the callable instead.
        .. versionchanged:: 8.0.2
            Type casting is no longer performed when getting a default.
        .. versionchanged:: 8.0.1
            Type casting can fail in resilient parsing mode. Invalid
            defaults will not prevent showing help text.
        .. versionchanged:: 8.0
            Looks at ``ctx.default_map`` first.
        .. versionchanged:: 8.0
            Added the ``call`` parameter.
        """
        value = ctx.lookup_default(self.name, call=False)  # type: ignore
        if value is None:
            value = self.default
        if call and callable(value):
            value = value()
        return value
    def add_to_parser(self, parser: OptionParser, ctx: Context) -> None:
        raise NotImplementedError()
    def consume_value(
        self, ctx: Context, opts: t.Mapping[str, t.Any]
    ) -> t.Tuple[t.Any, ParameterSource]:
        value = opts.get(self.name)  # type: ignore
        source = ParameterSource.COMMANDLINE
        if value is None:
            value = self.value_from_envvar(ctx)
            source = ParameterSource.ENVIRONMENT
        if value is None:
            value = ctx.lookup_default(self.name)  # type: ignore
            source = ParameterSource.DEFAULT_MAP
        if value is None:
            value = self.get_default(ctx)
            source = ParameterSource.DEFAULT
        return value, source
    def type_cast_value(self, ctx: Context, value: t.Any) -> t.Any:
        """Convert and validate a value against the option's
        :attr:`type`, :attr:`multiple`, and :attr:`nargs`.
        """
        if value is None:
            return () if self.multiple or self.nargs == -1 else None
        def check_iter(value: t.Any) -> t.Iterator[t.Any]:
            try:
                return _check_iter(value)
            except TypeError:
                raise BadParameter(
                    _("Value must be an iterable."), ctx=ctx, param=self
                ) from None
        if self.nargs == 1 or self.type.is_composite:
            def convert(value: t.Any) -> t.Any:
                return self.type(value, param=self, ctx=ctx)
        elif self.nargs == -1:
            def convert(value: t.Any) -> t.Any:  # t.Tuple[t.Any, ...]
                return tuple(self.type(x, self, ctx) for x in check_iter(value))
        else:  # nargs > 1
```

```
(12)            def convert(value: t.Any) -> t.Any:  # t.Tuple[t.Any, ...]
(16)                value = tuple(check_iter(value))
(16)                if len(value) != self.nargs:
(20)                    raise BadParameter(
(24)                        ngettext(
(28)                            "Takes {nargs} values but 1 was given.",
(28)                            "Takes {nargs} values but {len} were given.",
(28)                            len(value),
(24)                        ).format(nargs=self.nargs, len=len(value)),
(24)                        ctx=ctx,
(24)                        param=self,
(20)                    )
(16)                return tuple(self.type(x, self, ctx) for x in value)
(8)         if self.multiple:
(12)            return tuple(convert(x) for x in check_iter(value))
(8)         return convert(value)
(4)     def value_is_missing(self, value: t.Any) -> bool:
(8)         if value is None:
(12)            return True
(8)         if (self.nargs != 1 or self.multiple) and value == ():
(12)            return True
(8)         return False
(4)     def process_value(self, ctx: Context, value: t.Any) -> t.Any:
(8)         value = self.type_cast_value(ctx, value)
(8)         if self.required and self.value_is_missing(value):
(12)            raise MissingParameter(ctx=ctx, param=self)
(8)         if self.callback is not None:
(12)            value = self.callback(ctx, self, value)
(8)         return value
(4)     def resolve_envvar_value(self, ctx: Context) -> t.Optional[str]:
(8)         if self.envvar is None:
(12)            return None
(8)         if isinstance(self.envvar, str):
(12)            rv = os.environ.get(self.envvar)
(12)            if rv:
(16)                return rv
(8)         else:
(12)            for envvar in self.envvar:
(16)                rv = os.environ.get(envvar)
(16)                if rv:
(20)                    return rv
(8)         return None
(4)     def value_from_envvar(self, ctx: Context) -> t.Optional[t.Any]:
(8)         rv: t.Optional[t.Any] = self.resolve_envvar_value(ctx)
(8)         if rv is not None and self.nargs != 1:
(12)            rv = self.type.split_envvar_value(rv)
(8)         return rv
(4)     def handle_parse_result(
(8)         self, ctx: Context, opts: t.Mapping[str, t.Any], args: t.List[str]
(4)     ) -> t.Tuple[t.Any, t.List[str]]:
(8)         with augment_usage_errors(ctx, param=self):
(12)            value, source = self.consume_value(ctx, opts)
(12)            ctx.set_parameter_source(self.name, source)  # type: ignore
(12)            try:
(16)                value = self.process_value(ctx, value)
(12)            except Exception:
(16)                if not ctx.resilient_parsing:
(20)                    raise
(16)                value = None
(8)         if self.expose_value:
(12)            ctx.params[self.name] = value  # type: ignore
(8)         return value, args
(4)     def get_help_record(self, ctx: Context) -> t.Optional[t.Tuple[str, str]]:
(8)         pass
(4)     def get_usage_pieces(self, ctx: Context) -> t.List[str]:
(8)         return []
(4)     def get_error_hint(self, ctx: Context) -> str:
(8)         """Get a stringified version of the param for use in error messages to
(8)         indicate which param caused the error.
```

```
        """
        hint_list = self.opts or [self.human_readable_name]
        return " / ".join(f"'{x}'" for x in hint_list)
    def shell_complete(self, ctx: Context, incomplete: str) -> t.List["CompletionItem"]:
        """Return a list of completions for the incomplete value. If a
        ``shell_complete`` function was given during init, it is used.
        Otherwise, the :attr:`type`
        :meth:`~click.types.ParamType.shell_complete` function is used.
        :param ctx: Invocation context for this command.
        :param incomplete: Value being completed. May be empty.
        .. versionadded:: 8.0
        """
        if self._custom_shell_complete is not None:
            results = self._custom_shell_complete(ctx, self, incomplete)
            if results and isinstance(results[0], str):
                from click.shell_completion import CompletionItem
                results = [CompletionItem(c) for c in results]
            return t.cast(t.List["CompletionItem"], results)
        return self.type.shell_complete(ctx, self, incomplete)
class Option(Parameter):
    """Options are usually optional values on the command line and
    have some extra features that arguments don't have.
    All other parameters are passed onwards to the parameter constructor.
    :param show_default: Show the default value for this option in its
        help text. Values are not shown by default, unless
        :attr:`Context.show_default` is ``True``. If this value is a
        string, it shows that string in parentheses instead of the
        actual value. This is particularly useful for dynamic options.
        For single option boolean flags, the default remains hidden if
        its value is ``False``.
    :param show_envvar: Controls if an environment variable should be
        shown on the help page. Normally, environment variables are not
        shown.
    :param prompt: If set to ``True`` or a non empty string then the
        user will be prompted for input. If set to ``True`` the prompt
        will be the option name capitalized.
    :param confirmation_prompt: Prompt a second time to confirm the
        value if it was prompted for. Can be set to a string instead of
        ``True`` to customize the message.
    :param prompt_required: If set to ``False``, the user will be
        prompted for input only when the option was specified as a flag
        without a value.
    :param hide_input: If this is ``True`` then the input on the prompt
        will be hidden from the user. This is useful for password input.
    :param is_flag: forces this option to act as a flag.  The default is
                    auto detection.
    :param flag_value: which value should be used for this flag if it's
                       enabled.  This is set to a boolean automatically if
                       the option string contains a slash to mark two options.
    :param multiple: if this is set to `True` then the argument is accepted
                     multiple times and recorded.  This is similar to ``nargs``
                     in how it works but supports arbitrary number of
                     arguments.
    :param count: this flag makes an option increment an integer.
    :param allow_from_autoenv: if this is enabled then the value of this
                               parameter will be pulled from an environment
                               variable in case a prefix is defined on the
                               context.
    :param help: the help string.
    :param hidden: hide this option from help outputs.
    :param attrs: Other command arguments described in :class:`Parameter`.
    .. versionchanged:: 8.1.0
        Help text indentation is cleaned here instead of only in the
        ``@option`` decorator.
    .. versionchanged:: 8.1.0
        The ``show_default`` parameter overrides
        ``Context.show_default``.
    .. versionchanged:: 8.1.0
        The default of a single option boolean flag is not shown if the
        default value is ``False``.
    .. versionchanged:: 8.0.1
        ``type`` is detected from ``flag_value`` if given.
    """
    param_type_name = "option"
    def __init__(
        self,
        param_decls: t.Optional[t.Sequence[str]] = None,
        show_default: t.Union[bool, str, None] = None,
        prompt: t.Union[bool, str] = False,
        confirmation_prompt: t.Union[bool, str] = False,
        prompt_required: bool = True,
        hide_input: bool = False,
        is_flag: t.Optional[bool] = None,
        flag_value: t.Optional[t.Any] = None,
        multiple: bool = False,
        count: bool = False,
        allow_from_autoenv: bool = True,
        type: t.Optional[t.Union[types.ParamType, t.Any]] = None,
        help: t.Optional[str] = None,
        hidden: bool = False,
        show_choices: bool = True,
        show_envvar: bool = False,
        **attrs: t.Any,
    ) -> None:
        if help:
            help = inspect.cleandoc(help)
        default_is_missing = "default" not in attrs
        super().__init__(param_decls, type=type, multiple=multiple, **attrs)
        if prompt is True:
            if self.name is None:
                raise TypeError("'name' is required with 'prompt=True'.")
            prompt_text: t.Optional[str] = self.name.replace("_", " ").capitalize()
        elif prompt is False:
            prompt_text = None
        else:
            prompt_text = prompt
        self.prompt = prompt_text
        self.confirmation_prompt = confirmation_prompt
        self.prompt_required = prompt_required
        self.hide_input = hide_input
        self.hidden = hidden
        self._flag_needs_value = self.prompt is not None and not self.prompt_required
        if is_flag is None:
            if flag_value is not None:
                is_flag = True
            elif self._flag_needs_value:
                is_flag = False
            else:
                is_flag = bool(self.secondary_opts)
        elif is_flag is False and not self._flag_needs_value:
            self._flag_needs_value = flag_value is not None
        self.default: t.Union[t.Any, t.Callable[[], t.Any]]
        if is_flag and default_is_missing and not self.required:
            if multiple:
                self.default = ()
            else:
                self.default = False
        if flag_value is None:
            flag_value = not self.default
        self.type: types.ParamType
        if is_flag and type is None:
            self.type = types.convert_type(None, flag_value)
        self.is_flag: bool = is_flag
        self.is_bool_flag: bool = is_flag and isinstance(self.type,
types.BoolParamType)
(8)         self.flag_value: t.Any = flag_value
(8)         self.count = count
(8)         if count:
(12)            if type is None:
(16)                self.type = types.IntRange(min=0)
(12)            if default_is_missing:
(16)                self.default = 0
(8)         self.allow_from_autoenv = allow_from_autoenv
(8)         self.help = help
(8)         self.show_default = show_default
(8)         self.show_choices = show_choices
(8)         self.show_envvar = show_envvar
(8)         if __debug__:
(12)            if self.nargs == -1:
(16)                raise TypeError("nargs=-1 is not supported for options.")
(12)            if self.prompt and self.is_flag and not self.is_bool_flag:
(16)                raise TypeError("'prompt' is not valid for non-boolean flag.")
(12)            if not self.is_bool_flag and self.secondary_opts:
(16)                raise TypeError("Secondary flag is not valid for non-boolean
flag.")
(12)            if self.is_bool_flag and self.hide_input and self.prompt is not
None:
(16)                raise TypeError(
(20)                    "'prompt' with 'hide_input' is not valid for boolean
flag."
(16)                )
(12)            if self.count:
(16)                if self.multiple:
(20)                    raise TypeError("'count' is not valid with 'multiple'.")
(16)                if self.is_flag:
(20)                    raise TypeError("'count' is not valid with 'is_flag'.")
(4)     def to_info_dict(self) -> t.Dict[str, t.Any]:
(8)         info_dict = super().to_info_dict()
(8)         info_dict.update(
(12)            help=self.help,
(12)            prompt=self.prompt,
(12)            is_flag=self.is_flag,
(12)            flag_value=self.flag_value,
(12)            count=self.count,
(12)            hidden=self.hidden,
(8)         )
(8)         return info_dict
(4)     def _parse_decls(
(8)         self, decls: t.Sequence[str], expose_value: bool
(4)     ) -> t.Tuple[t.Optional[str], t.List[str], t.List[str]]:
(8)         opts = []
(8)         secondary_opts = []
(8)         name = None
(8)         possible_names = []
(8)         for decl in decls:
(12)            if decl.isidentifier():
(16)                if name is not None:
(20)                    raise TypeError(f"Name '{name}' defined twice")
(16)                name = decl
(12)            else:
(16)                split_char = ";" if decl[:1] == "/" else "/"
(16)                if split_char in decl:
(20)                    first, second = decl.split(split_char, 1)
(20)                    first = first.rstrip()
(20)                    if first:
(24)                        possible_names.append(split_opt(first))
(24)                        opts.append(first)
(20)                    second = second.lstrip()
(20)                    if second:
(24)                        secondary_opts.append(second.lstrip())
(20)                    if first == second:
(24)                        raise ValueError(
(28)                            f"Boolean option {decl!r} cannot use the"
```

```
                            " same flag for true/false."
                        )
                else:
                    possible_names.append(split_opt(decl))
                    opts.append(decl)
        if name is None and possible_names:
            possible_names.sort(key=lambda x: -len(x[0]))  # group long options first
            name = possible_names[0][1].replace("-", "_").lower()
            if not name.isidentifier():
                name = None
        if name is None:
            if not expose_value:
                return None, opts, secondary_opts
            raise TypeError("Could not determine name for option")
        if not opts and not secondary_opts:
            raise TypeError(
                f"No options defined but a name was passed ({name})."
                " Did you mean to declare an argument instead? Did"
                f" you mean to pass '--{name}'?"
            )
        return name, opts, secondary_opts
    def add_to_parser(self, parser: OptionParser, ctx: Context) -> None:
        if self.multiple:
            action = "append"
        elif self.count:
            action = "count"
        else:
            action = "store"
        if self.is_flag:
            action = f"{action}_const"
            if self.is_bool_flag and self.secondary_opts:
                parser.add_option(
                    obj=self, opts=self.opts, dest=self.name, action=action, const=True
                )
                parser.add_option(
                    obj=self,
                    opts=self.secondary_opts,
                    dest=self.name,
                    action=action,
                    const=False,
                )
            else:
                parser.add_option(
                    obj=self,
                    opts=self.opts,
                    dest=self.name,
                    action=action,
                    const=self.flag_value,
                )
        else:
            parser.add_option(
                obj=self,
                opts=self.opts,
                dest=self.name,
                action=action,
                nargs=self.nargs,
            )
    def get_help_record(self, ctx: Context) -> t.Optional[t.Tuple[str, str]]:
        if self.hidden:
            return None
        any_prefix_is_slash = False
        def _write_opts(opts: t.Sequence[str]) -> str:
            nonlocal any_prefix_is_slash
            rv, any_slashes = join_options(opts)
            if any_slashes:
                any_prefix_is_slash = True
            if not self.is_flag and not self.count:
                rv += f" {self.make_metavar()}"
            return rv
        rv = [_write_opts(self.opts)]
        if self.secondary_opts:
            rv.append(_write_opts(self.secondary_opts))
        help = self.help or ""
        extra = []
        if self.show_envvar:
            envvar = self.envvar
            if envvar is None:
                if (
                    self.allow_from_autoenv
                    and ctx.auto_envvar_prefix is not None
                    and self.name is not None
                ):
                    envvar = f"{ctx.auto_envvar_prefix}_{self.name.upper()}"
            if envvar is not None:
                var_str = (
                    envvar
                    if isinstance(envvar, str)
                    else ", ".join(str(d) for d in envvar)
                )
                extra.append(_("env var: {var}").format(var=var_str))
        resilient = ctx.resilient_parsing
        ctx.resilient_parsing = True
        try:
            default_value = self.get_default(ctx, call=False)
        finally:
            ctx.resilient_parsing = resilient
        show_default = False
        show_default_is_str = False
        if self.show_default is not None:
            if isinstance(self.show_default, str):
                show_default_is_str = show_default = True
            else:
                show_default = self.show_default
        elif ctx.show_default is not None:
            show_default = ctx.show_default
        if show_default_is_str or (show_default and (default_value is not None)):
            if show_default_is_str:
                default_string = f"({self.show_default})"
            elif isinstance(default_value, (list, tuple)):
                default_string = ", ".join(str(d) for d in default_value)
            elif inspect.isfunction(default_value):
                default_string = _("(dynamic)")
            elif self.is_bool_flag and self.secondary_opts:
                default_string = split_opt(
                    (self.opts if self.default else self.secondary_opts)[0]
                )[1]
            elif self.is_bool_flag and not self.secondary_opts and not default_value:
                default_string = ""
            else:
                default_string = str(default_value)
            if default_string:
                extra.append(_("default: {default}").format(default=default_string))
        if (
            isinstance(self.type, types._NumberRangeBase)
            and not (self.count and self.type.min == 0 and self.type.max is None)
        ):
            range_str = self.type._describe_range()
            if range_str:
                extra.append(range_str)
        if self.required:
            extra.append(_("required"))
        if extra:
            extra_str = "; ".join(extra)
            help = f"{help}  [{extra_str}]" if help else f"[{extra_str}]"
        return ("; " if any_prefix_is_slash else " / ").join(rv), help
    @t.overload
    def get_default(
        self, ctx: Context, call: "te.Literal[True]" = True
    ) -> t.Optional[t.Any]:
        ...
    @t.overload
    def get_default(
        self, ctx: Context, call: bool = ...
    ) -> t.Optional[t.Union[t.Any, t.Callable[[], t.Any]]]:
        ...
    def get_default(
        self, ctx: Context, call: bool = True
    ) -> t.Optional[t.Union[t.Any, t.Callable[[], t.Any]]]:
        if self.is_flag and not self.is_bool_flag:
            for param in ctx.command.params:
                if param.name == self.name and param.default:
                    return t.cast(Option, param).flag_value
            return None
        return super().get_default(ctx, call=call)
    def prompt_for_value(self, ctx: Context) -> t.Any:
        """This is an alternative flow that can be activated in the full
        value processing if a value does not exist.  It will prompt the
        user until a valid value exists and then returns the processed
        value as result.
        """
        assert self.prompt is not None
        default = self.get_default(ctx)
        if self.is_bool_flag:
            return confirm(self.prompt, default)
        return prompt(
            self.prompt,
            default=default,
            type=self.type,
            hide_input=self.hide_input,
            show_choices=self.show_choices,
            confirmation_prompt=self.confirmation_prompt,
            value_proc=lambda x: self.process_value(ctx, x),
        )
    def resolve_envvar_value(self, ctx: Context) -> t.Optional[str]:
        rv = super().resolve_envvar_value(ctx)
        if rv is not None:
            return rv
        if (
            self.allow_from_autoenv
            and ctx.auto_envvar_prefix is not None
            and self.name is not None
        ):
            envvar = f"{ctx.auto_envvar_prefix}_{self.name.upper()}"
            rv = os.environ.get(envvar)
            if rv:
                return rv
        return None
    def value_from_envvar(self, ctx: Context) -> t.Optional[t.Any]:
        rv: t.Optional[t.Any] = self.resolve_envvar_value(ctx)
        if rv is None:
            return None
        value_depth = (self.nargs != 1) + bool(self.multiple)
        if value_depth > 0:
            rv = self.type.split_envvar_value(rv)
            if self.multiple and self.nargs != 1:
                rv = batch(rv, self.nargs)
        return rv
    def consume_value(
        self, ctx: Context, opts: t.Mapping[str, "Parameter"]
    ) -> t.Tuple[t.Any, ParameterSource]:
        value, source = super().consume_value(ctx, opts)
        if value is _flag_needs_value:
            if self.prompt is not None and not ctx.resilient_parsing:
                value = self.prompt_for_value(ctx)
                source = ParameterSource.PROMPT
            else:
                value = self.flag_value
                source = ParameterSource.COMMANDLINE
        elif (
            self.multiple
            and value is not None
            and any(v is _flag_needs_value for v in value)
        ):
            value = [self.flag_value if v is _flag_needs_value else v for v in value]
            source = ParameterSource.COMMANDLINE
        elif (
            source in {None, ParameterSource.DEFAULT}
            and self.prompt is not None
            and (self.required or self.prompt_required)
            and not ctx.resilient_parsing
        ):
            value = self.prompt_for_value(ctx)
            source = ParameterSource.PROMPT
        return value, source
class Argument(Parameter):
    """Arguments are positional parameters to a command.  They generally
    provide fewer features than options but can have infinite ``nargs``
    and are required by default.
    All parameters are passed onwards to the constructor of :class:`Parameter`.
    """
    param_type_name = "argument"
    def __init__(
        self,
        param_decls: t.Sequence[str],
        required: t.Optional[bool] = None,
        **attrs: t.Any,
    ) -> None:
        if required is None:
            if attrs.get("default") is not None:
                required = False
            else:
                required = attrs.get("nargs", 1) > 0
        if "multiple" in attrs:
            raise TypeError("__init__() got an unexpected keyword argument 'multiple'.")
        super().__init__(param_decls, required=required, **attrs)
        if __debug__:
            if self.default is not None and self.nargs == -1:
                raise TypeError("'default' is not supported for nargs=-1.")
    @property
    def human_readable_name(self) -> str:
        if self.metavar is not None:
            return self.metavar
        return self.name.upper()  # type: ignore
    def make_metavar(self) -> str:
        if self.metavar is not None:
            return self.metavar
        var = self.type.get_metavar(self)
        if not var:
            var = self.name.upper()  # type: ignore
        if not self.required:
            var = f"[{var}]"
        if self.nargs != 1:
            var += "..."
        return var
    def _parse_decls(
        self, decls: t.Sequence[str], expose_value: bool
    ) -> t.Tuple[t.Optional[str], t.List[str], t.List[str]]:
```

```
(8)                 if not decls:
(12)                    if not expose_value:
(16)                        return None, [], []
(12)                    raise TypeError("Could not determine name for argument")
(8)                 if len(decls) == 1:
(12)                    name = arg = decls[0]
(12)                    name = name.replace("-", "_").lower()
(8)                 else:
(12)                    raise TypeError(
(16)                        "Arguments take exactly one parameter declaration, got"
(16)                        f" {len(decls)}."
(12)                    )
(8)                 return name, [arg], []
(4)             def get_usage_pieces(self, ctx: Context) -> t.List[str]:
(8)                 return [self.make_metavar()]
(4)             def get_error_hint(self, ctx: Context) -> str:
(8)                 return f"'{self.make_metavar()}'"
(4)             def add_to_parser(self, parser: OptionParser, ctx: Context) -> None:
(8)                 parser.add_argument(dest=self.name, nargs=self.nargs, obj=self)
```

File 7 - decorators.py:

```
(0)             import inspect
(0)             import types
(0)             import typing as t
(0)             from functools import update_wrapper
(0)             from gettext import gettext as _
(0)             from .core import Argument
(0)             from .core import Command
(0)             from .core import Context
(0)             from .core import Group
(0)             from .core import Option
(0)             from .core import Parameter
(0)             from .globals import get_current_context
(0)             from .utils import echo
(0)             if t.TYPE_CHECKING:
(4)                 import typing_extensions as te
(4)                 P = te.ParamSpec("P")
(0)             R = t.TypeVar("R")
(0)             T = t.TypeVar("T")
(0)             _AnyCallable = t.Callable[..., t.Any]
(0)             FC = t.TypeVar("FC", bound=t.Union[_AnyCallable, Command])
(0)             def pass_context(f: "t.Callable[te.Concatenate[Context, P], R]") -> "t.Callable[P, R]":
(4)                 """Marks a callback as wanting to receive the current context
(4)                 object as first argument.
(4)                 """
(4)                 def new_func(*args: "P.args", **kwargs: "P.kwargs") -> "R":
(8)                     return f(get_current_context(), *args, **kwargs)
(4)                 return update_wrapper(new_func, f)
(0)             def pass_obj(f: "t.Callable[te.Concatenate[t.Any, P], R]") -> "t.Callable[P, R]":
(4)                 """Similar to :func:`pass_context`, but only pass the object on the
(4)                 context onwards (:attr:`Context.obj`).  This is useful if that object
(4)                 represents the state of a nested system.
(4)                 """
(4)                 def new_func(*args: "P.args", **kwargs: "P.kwargs") -> "R":
(8)                     return f(get_current_context().obj, *args, **kwargs)
(4)                 return update_wrapper(new_func, f)
(0)             def make_pass_decorator(
(4)                 object_type: t.Type[T], ensure: bool = False
(0)             ) -> t.Callable[["t.Callable[te.Concatenate[T, P], R]"], "t.Callable[P, R]"]:
(4)                 """Given an object type this creates a decorator that will work
(4)                 similar to :func:`pass_obj` but instead of passing the object of the
(4)                 current context, it will find the innermost context of type
(4)                 :func:`object_type`.
(4)                 This generates a decorator that works roughly like this::
```

```
(8)             from functools import update_wrapper
(8)             def decorator(f):
(12)                @pass_context
(12)                def new_func(ctx, *args, **kwargs):
(16)                    obj = ctx.find_object(object_type)
(16)                    return ctx.invoke(f, obj, *args, **kwargs)
(12)                return update_wrapper(new_func, f)
(8)             return decorator
(4)         :param object_type: the type of the object to pass.
(4)         :param ensure: if set to `True`, a new object will be created and
(19)                       remembered on the context if it's not there yet.
(4)         """
(4)         def decorator(f: "t.Callable[te.Concatenate[T, P], R]") -> "t.Callable[P, R]":
(8)             def new_func(*args: "P.args", **kwargs: "P.kwargs") -> "R":
(12)                ctx = get_current_context()
(12)                obj: t.Optional[T]
(12)                if ensure:
(16)                    obj = ctx.ensure_object(object_type)
(12)                else:
(16)                    obj = ctx.find_object(object_type)
(12)                if obj is None:
(16)                    raise RuntimeError(
(20)                        "Managed to invoke callback without a context"
(20)                        f" object of type {object_type.__name__!r}"
(20)                        " existing."
(16)                    )
(12)                return ctx.invoke(f, obj, *args, **kwargs)
(8)             return update_wrapper(new_func, f)
(4)         return decorator  # type: ignore[return-value]
(0)     def pass_meta_key(
(4)         key: str, *, doc_description: t.Optional[str] = None
(0)     ) -> "t.Callable[[t.Callable[te.Concatenate[t.Any, P], R]], t.Callable[P, R]]":
(4)         """Create a decorator that passes a key from
(4)         :attr:`click.Context.meta` as the first argument to the decorated
(4)         function.
(4)         :param key: Key in ``Context.meta`` to pass.
(4)         :param doc_description: Description of the object being passed,
(8)             inserted into the decorator's docstring. Defaults to "the 'key'
(8)             key from Context.meta".
(4)         .. versionadded:: 8.0
(4)         """
(4)         def decorator(f: "t.Callable[te.Concatenate[t.Any, P], R]") -> "t.Callable[P, R]":
(8)             def new_func(*args: "P.args", **kwargs: "P.kwargs") -> R:
(12)                ctx = get_current_context()
(12)                obj = ctx.meta[key]
(12)                return ctx.invoke(f, obj, *args, **kwargs)
(8)             return update_wrapper(new_func, f)
(4)         if doc_description is None:
(8)             doc_description = f"the {key!r} key from :attr:`click.Context.meta`"
(4)         decorator.__doc__ = (
(8)             f"Decorator that passes {doc_description} as the first argument"
(8)             " to the decorated function."
(4)         )
(4)         return decorator  # type: ignore[return-value]
(0)     CmdType = t.TypeVar("CmdType", bound=Command)
(0)     @t.overload
(0)     def command(name: _AnyCallable) -> Command:
(4)         ...
(0)     @t.overload
(0)     def command(
(4)         name: t.Optional[str],
(4)         cls: t.Type[CmdType],
(4)         **attrs: t.Any,
(0)     ) -> t.Callable[[_AnyCallable], CmdType]:
(4)         ...
(0)     @t.overload
```

```
def command(
    name: None = None,
    *,
    cls: t.Type[CmdType],
    **attrs: t.Any,
) -> t.Callable[[_AnyCallable], CmdType]:
    ...
@t.overload
def command(
    name: t.Optional[str] = ..., cls: None = None, **attrs: t.Any
) -> t.Callable[[_AnyCallable], Command]:
    ...
def command(
    name: t.Union[t.Optional[str], _AnyCallable] = None,
    cls: t.Optional[t.Type[CmdType]] = None,
    **attrs: t.Any,
) -> t.Union[Command, t.Callable[[_AnyCallable], t.Union[Command, CmdType]]]:
    r"""Creates a new :class:`Command` and uses the decorated function as
    callback.  This will also automatically attach all decorated
    :func:`option`\s and :func:`argument`\s as parameters to the command.
    The name of the command defaults to the name of the function with
    underscores replaced by dashes.  If you want to change that, you can
    pass the intended name as the first argument.
    All keyword arguments are forwarded to the underlying command class.
    For the ``params`` argument, any decorated params are appended to
    the end of the list.
    Once decorated the function turns into a :class:`Command` instance
    that can be invoked as a command line utility or be attached to a
    command :class:`Group`.
    :param name: the name of the command.  This defaults to the function
                 name with underscores replaced by dashes.
    :param cls: the command class to instantiate.  This defaults to
                :class:`Command`.
    .. versionchanged:: 8.1
        This decorator can be applied without parentheses.
    .. versionchanged:: 8.1
        The ``params`` argument can be used. Decorated params are
        appended to the end of the list.
    """
    func: t.Optional[t.Callable[[_AnyCallable], t.Any]] = None
    if callable(name):
        func = name
        name = None
        assert cls is None, "Use 'command(cls=cls)(callable)' to specify a class."
        assert not attrs, "Use 'command(**kwargs)(callable)' to provide arguments."
    if cls is None:
        cls = t.cast(t.Type[CmdType], Command)
    def decorator(f: _AnyCallable) -> CmdType:
        if isinstance(f, Command):
            raise TypeError("Attempted to convert a callback into a command twice.")
        attr_params = attrs.pop("params", None)
        params = attr_params if attr_params is not None else []
        try:
            decorator_params = f.__click_params__  # type: ignore
        except AttributeError:
            pass
        else:
            del f.__click_params__  # type: ignore
            params.extend(reversed(decorator_params))
        if attrs.get("help") is None:
            attrs["help"] = f.__doc__
        if t.TYPE_CHECKING:
            assert cls is not None
            assert not callable(name)
        cmd = cls(
            name=name or f.__name__.lower().replace("_", "-"),
(12)                    callback=f,
(12)                    params=params,
(12)                    **attrs,
(8)                 )
(8)                 cmd.__doc__ = f.__doc__
(8)                 return cmd
(4)             if func is not None:
(8)                 return decorator(func)
(4)             return decorator
(0)         GrpType = t.TypeVar("GrpType", bound=Group)
(0)         @t.overload
(0)         def group(name: _AnyCallable) -> Group:
(4)             ...
(0)         @t.overload
(0)         def group(
(4)             name: t.Optional[str],
(4)             cls: t.Type[GrpType],
(4)             **attrs: t.Any,
(0)         ) -> t.Callable[[_AnyCallable], GrpType]:
(4)             ...
(0)         @t.overload
(0)         def group(
(4)             name: None = None,
(4)             *,
(4)             cls: t.Type[GrpType],
(4)             **attrs: t.Any,
(0)         ) -> t.Callable[[_AnyCallable], GrpType]:
(4)             ...
(0)         @t.overload
(0)         def group(
(4)             name: t.Optional[str] = ..., cls: None = None, **attrs: t.Any
(0)         ) -> t.Callable[[_AnyCallable], Group]:
(4)             ...
(0)         def group(
(4)             name: t.Union[str, _AnyCallable, None] = None,
(4)             cls: t.Optional[t.Type[GrpType]] = None,
(4)             **attrs: t.Any,
(0)         ) -> t.Union[Group, t.Callable[[_AnyCallable], t.Union[Group, GrpType]]]:
(4)             """Creates a new :class:`Group` with a function as callback.  This
(4)             works otherwise the same as :func:`command` just that the `cls`
(4)             parameter is set to :class:`Group`.
(4)             .. versionchanged:: 8.1
(8)                 This decorator can be applied without parentheses.
(4)             """
(4)             if cls is None:
(8)                 cls = t.cast(t.Type[GrpType], Group)
(4)             if callable(name):
(8)                 return command(cls=cls, **attrs)(name)
(4)             return command(name, cls, **attrs)
(0)         def _param_memo(f: t.Callable[..., t.Any], param: Parameter) -> None:
(4)             if isinstance(f, Command):
(8)                 f.params.append(param)
(4)             else:
(8)                 if not hasattr(f, "__click_params__"):
(12)                    f.__click_params__ = []  # type: ignore
(8)                 f.__click_params__.append(param)  # type: ignore
(0)         def argument(
(4)             *param_decls: str, cls: t.Optional[t.Type[Argument]] = None, **attrs:
t.Any
(0)         ) -> t.Callable[[FC], FC]:
(4)             """Attaches an argument to the command.  All positional arguments are
(4)             passed as parameter declarations to :class:`Argument`; all keyword
(4)             arguments are forwarded unchanged (except ``cls``).
(4)             This is equivalent to creating an :class:`Argument` instance manually
(4)             and attaching it to the :attr:`Command.params` list.
(4)             For the default argument class, refer to :class:`Argument` and
(4)             :class:`Parameter` for descriptions of parameters.
(4)             :param cls: the argument class to instantiate.  This defaults to
(16)                        :class:`Argument`.
```

```
(4)              :param param_decls: Passed as positional arguments to the constructor of
(8)                  ``cls``.
(4)              :param attrs: Passed as keyword arguments to the constructor of ``cls``.
(4)              """
(4)              if cls is None:
(8)                  cls = Argument
(4)              def decorator(f: FC) -> FC:
(8)                  _param_memo(f, cls(param_decls, **attrs))
(8)                  return f
(4)              return decorator
(0)          def option(
(4)              *param_decls: str, cls: t.Optional[t.Type[Option]] = None, **attrs: t.Any
(0)          ) -> t.Callable[[FC], FC]:
(4)              """Attaches an option to the command.  All positional arguments are
(4)              passed as parameter declarations to :class:`Option`; all keyword
(4)              arguments are forwarded unchanged (except ``cls``).
(4)              This is equivalent to creating an :class:`Option` instance manually
(4)              and attaching it to the :attr:`Command.params` list.
(4)              For the default option class, refer to :class:`Option` and
(4)              :class:`Parameter` for descriptions of parameters.
(4)              :param cls: the option class to instantiate.  This defaults to
(16)                         :class:`Option`.
(4)              :param param_decls: Passed as positional arguments to the constructor of
(8)                  ``cls``.
(4)              :param attrs: Passed as keyword arguments to the constructor of ``cls``.
(4)              """
(4)              if cls is None:
(8)                  cls = Option
(4)              def decorator(f: FC) -> FC:
(8)                  _param_memo(f, cls(param_decls, **attrs))
(8)                  return f
(4)              return decorator
(0)          def confirmation_option(*param_decls: str, **kwargs: t.Any) ->
t.Callable[[FC], FC]:
(4)              """Add a ``--yes`` option which shows a prompt before continuing if
(4)              not passed. If the prompt is declined, the program will exit.
(4)              :param param_decls: One or more option names. Defaults to the single
(8)                  value ``"--yes"``.
(4)              :param kwargs: Extra arguments are passed to :func:`option`.
(4)              """
(4)              def callback(ctx: Context, param: Parameter, value: bool) -> None:
(8)                  if not value:
(12)                     ctx.abort()
(4)              if not param_decls:
(8)                  param_decls = ("--yes",)
(4)              kwargs.setdefault("is_flag", True)
(4)              kwargs.setdefault("callback", callback)
(4)              kwargs.setdefault("expose_value", False)
(4)              kwargs.setdefault("prompt", "Do you want to continue?")
(4)              kwargs.setdefault("help", "Confirm the action without prompting.")
(4)              return option(*param_decls, **kwargs)
(0)          def password_option(*param_decls: str, **kwargs: t.Any) -> t.Callable[[FC],
FC]:
(4)              """Add a ``--password`` option which prompts for a password, hiding
(4)              input and asking to enter the value again for confirmation.
(4)              :param param_decls: One or more option names. Defaults to the single
(8)                  value ``"--password"``.
(4)              :param kwargs: Extra arguments are passed to :func:`option`.
(4)              """
(4)              if not param_decls:
(8)                  param_decls = ("--password",)
(4)              kwargs.setdefault("prompt", True)
(4)              kwargs.setdefault("confirmation_prompt", True)
(4)              kwargs.setdefault("hide_input", True)
(4)              return option(*param_decls, **kwargs)
(0)          def version_option(
(4)              version: t.Optional[str] = None,
(4)              *param_decls: str,
(4)              package_name: t.Optional[str] = None,
(4)      prog_name: t.Optional[str] = None,
(4)      message: t.Optional[str] = None,
(4)      **kwargs: t.Any,
(0)  ) -> t.Callable[[FC], FC]:
(4)      """Add a ``--version`` option which immediately prints the version
(4)      number and exits the program.
(4)      If ``version`` is not provided, Click will try to detect it using
(4)      :func:`importlib.metadata.version` to get the version for the
(4)      ``package_name``. On Python < 3.8, the ``importlib_metadata``
(4)      backport must be installed.
(4)      If ``package_name`` is not provided, Click will try to detect it by
(4)      inspecting the stack frames. This will be used to detect the
(4)      version, so it must match the name of the installed package.
(4)      :param version: The version number to show. If not provided, Click
(8)          will try to detect it.
(4)      :param param_decls: One or more option names. Defaults to the single
(8)          value ``"--version"``.
(4)      :param package_name: The package name to detect the version from. If
(8)          not provided, Click will try to detect it.
(4)      :param prog_name: The name of the CLI to show in the message. If not
(8)          provided, it will be detected from the command.
(4)      :param message: The message to show. The values ``%(prog)s``,
(8)          ``%(package)s``, and ``%(version)s`` are available. Defaults to
(8)          ``"%(prog)s, version %(version)s"``.
(4)      :param kwargs: Extra arguments are passed to :func:`option`.
(4)      :raise RuntimeError: ``version`` could not be detected.
(4)      .. versionchanged:: 8.0
(8)          Add the ``package_name`` parameter, and the ``%(package)s``
(8)          value for messages.
(4)      .. versionchanged:: 8.0
(8)          Use :mod:`importlib.metadata` instead of ``pkg_resources``. The
(8)          version is detected based on the package name, not the entry
(8)          point name. The Python package name must match the installed
(8)          package name, or be passed with ``package_name=``.
(4)      """
(4)      if message is None:
(8)          message = _("%(prog)s, version %(version)s")
(4)      if version is None and package_name is None:
(8)          frame = inspect.currentframe()
(8)          f_back = frame.f_back if frame is not None else None
(8)          f_globals = f_back.f_globals if f_back is not None else None
(8)          del frame
(8)          if f_globals is not None:
(12)             package_name = f_globals.get("__name__")
(12)             if package_name == "__main__":
(16)                 package_name = f_globals.get("__package__")
(12)             if package_name:
(16)                 package_name = package_name.partition(".")[0]
(4)      def callback(ctx: Context, param: Parameter, value: bool) -> None:
(8)          if not value or ctx.resilient_parsing:
(12)             return
(8)          nonlocal prog_name
(8)          nonlocal version
(8)          if prog_name is None:
(12)             prog_name = ctx.find_root().info_name
(8)          if version is None and package_name is not None:
(12)             metadata: t.Optional[types.ModuleType]
(12)             try:
(16)                 from importlib import metadata  # type: ignore
(12)             except ImportError:
(16)                 import importlib_metadata as metadata  # type: ignore
(12)             try:
(16)                 version = metadata.version(package_name)  # type: ignore
(12)             except metadata.PackageNotFoundError:  # type: ignore
(16)                 raise RuntimeError(
(20)                     f"{package_name!r} is not installed. Try passing"
(20)                     " 'package_name' instead."
(16)                 ) from None
(8)          if version is None:
```

```
380: (12)                       raise RuntimeError(
381: (16)                           f"Could not determine the version for {package_name!r}
automatically."
382: (12)                       )
383: (8)                    echo(
384: (12)                       message % {"prog": prog_name, "package": package_name, "version":
version},
385: (12)                       color=ctx.color,
386: (8)                    )
387: (8)                    ctx.exit()
388: (4)                if not param_decls:
389: (8)                    param_decls = ("--version",)
390: (4)                kwargs.setdefault("is_flag", True)
391: (4)                kwargs.setdefault("expose_value", False)
392: (4)                kwargs.setdefault("is_eager", True)
393: (4)                kwargs.setdefault("help", _("Show the version and exit."))
394: (4)                kwargs["callback"] = callback
395: (4)                return option(*param_decls, **kwargs)
396: (0)            def help_option(*param_decls: str, **kwargs: t.Any) -> t.Callable[[FC], FC]:
397: (4)                """Add a ``--help`` option which immediately prints the help page
398: (4)                and exits the program.
399: (4)                This is usually unnecessary, as the ``--help`` option is added to
400: (4)                each command automatically unless ``add_help_option=False`` is
401: (4)                passed.
402: (4)                :param param_decls: One or more option names. Defaults to the single
403: (8)                    value ``"--help"``.
404: (4)                :param kwargs: Extra arguments are passed to :func:`option`.
405: (4)                """
406: (4)                def callback(ctx: Context, param: Parameter, value: bool) -> None:
407: (8)                    if not value or ctx.resilient_parsing:
408: (12)                       return
409: (8)                    echo(ctx.get_help(), color=ctx.color)
410: (8)                    ctx.exit()
411: (4)                if not param_decls:
412: (8)                    param_decls = ("--help",)
413: (4)                kwargs.setdefault("is_flag", True)
414: (4)                kwargs.setdefault("expose_value", False)
415: (4)                kwargs.setdefault("is_eager", True)
416: (4)                kwargs.setdefault("help", _("Show this message and exit."))
417: (4)                kwargs["callback"] = callback
418: (4)                return option(*param_decls, **kwargs)

----------------------------------------

File 8 - exceptions.py:

1: (0)              import typing as t
2: (0)              from gettext import gettext as _
3: (0)              from gettext import ngettext
4: (0)              from ._compat import get_text_stderr
5: (0)              from .utils import echo
6: (0)              from .utils import format_filename
7: (0)              if t.TYPE_CHECKING:
8: (4)                  from .core import Command
9: (4)                  from .core import Context
10: (4)                 from .core import Parameter
11: (0)             def _join_param_hints(
12: (4)                 param_hint: t.Optional[t.Union[t.Sequence[str], str]]
13: (0)             ) -> t.Optional[str]:
14: (4)                 if param_hint is not None and not isinstance(param_hint, str):
15: (8)                     return " / ".join(repr(x) for x in param_hint)
16: (4)                 return param_hint
17: (0)             class ClickException(Exception):
18: (4)                 """An exception that Click can handle and show to the user."""
19: (4)                 exit_code = 1
20: (4)                 def __init__(self, message: str) -> None:
21: (8)                     super().__init__(message)
22: (8)                     self.message = message
23: (4)                 def format_message(self) -> str:
```

```python
        return self.message
    def __str__(self) -> str:
        return self.message
    def show(self, file: t.Optional[t.IO[t.Any]] = None) -> None:
        if file is None:
            file = get_text_stderr()
        echo(_("Error: {message}").format(message=self.format_message()), file=file)
class UsageError(ClickException):
    """An internal exception that signals a usage error.  This typically
    aborts any further handling.
    :param message: the error message to display.
    :param ctx: optionally the context that caused this error.  Click will
                fill in the context automatically in some situations.
    """
    exit_code = 2
    def __init__(self, message: str, ctx: t.Optional["Context"] = None) -> None:
        super().__init__(message)
        self.ctx = ctx
        self.cmd: t.Optional["Command"] = self.ctx.command if self.ctx else None
    def show(self, file: t.Optional[t.IO[t.Any]] = None) -> None:
        if file is None:
            file = get_text_stderr()
        color = None
        hint = ""
        if (
            self.ctx is not None
            and self.ctx.command.get_help_option(self.ctx) is not None
        ):
            hint = _("Try '{command} {option}' for help.").format(
                command=self.ctx.command_path, option=self.ctx.help_option_names[0]
            )
            hint = f"{hint}\n"
        if self.ctx is not None:
            color = self.ctx.color
            echo(f"{self.ctx.get_usage()}\n{hint}", file=file, color=color)
        echo(
            _("Error: {message}").format(message=self.format_message()),
            file=file,
            color=color,
        )
class BadParameter(UsageError):
    """An exception that formats out a standardized error message for a
    bad parameter.  This is useful when thrown from a callback or type as
    Click will attach contextual information to it (for instance, which
    parameter it is).
    .. versionadded:: 2.0
    :param param: the parameter object that caused this error.  This can
                  be left out, and Click will attach this info itself
                  if possible.
    :param param_hint: a string that shows up as parameter name.  This
                       can be used as alternative to `param` in cases
                       where custom validation should happen.  If it is
                       a string it's used as such, if it's a list then
                       each item is quoted and separated.
    """
    def __init__(
        self,
        message: str,
        ctx: t.Optional["Context"] = None,
        param: t.Optional["Parameter"] = None,
        param_hint: t.Optional[str] = None,
    ) -> None:
        super().__init__(message, ctx)
        self.param = param
        self.param_hint = param_hint
    def format_message(self) -> str:
        if self.param_hint is not None:
            param_hint = self.param_hint
        elif self.param is not None:
            param_hint = self.param.get_error_hint(self.ctx)  # type: ignore
        else:
            return _("Invalid value: {message}").format(message=self.message)
        return _("Invalid value for {param_hint}: {message}").format(
            param_hint=_join_param_hints(param_hint), message=self.message
        )
class MissingParameter(BadParameter):
    """Raised if click required an option or argument but it was not
    provided when invoking the script.
    .. versionadded:: 4.0
    :param param_type: a string that indicates the type of the parameter.
                       The default is to inherit the parameter type from
                       the given `param`.  Valid values are ``'parameter'``,
                       ``'option'`` or ``'argument'``.
    """
    def __init__(
        self,
        message: t.Optional[str] = None,
        ctx: t.Optional["Context"] = None,
        param: t.Optional["Parameter"] = None,
        param_hint: t.Optional[str] = None,
        param_type: t.Optional[str] = None,
    ) -> None:
        super().__init__(message or "", ctx, param, param_hint)
        self.param_type = param_type
    def format_message(self) -> str:
        if self.param_hint is not None:
            param_hint: t.Optional[str] = self.param_hint
        elif self.param is not None:
            param_hint = self.param.get_error_hint(self.ctx)  # type: ignore
        else:
            param_hint = None
        param_hint = _join_param_hints(param_hint)
        param_hint = f" {param_hint}" if param_hint else ""
        param_type = self.param_type
        if param_type is None and self.param is not None:
            param_type = self.param.param_type_name
        msg = self.message
        if self.param is not None:
            msg_extra = self.param.type.get_missing_message(self.param)
            if msg_extra:
                if msg:
                    msg += f". {msg_extra}"
                else:
                    msg = msg_extra
        msg = f" {msg}" if msg else ""
        if param_type == "argument":
            missing = _("Missing argument")
        elif param_type == "option":
            missing = _("Missing option")
        elif param_type == "parameter":
            missing = _("Missing parameter")
        else:
            missing = _("Missing {param_type}").format(param_type=param_type)
        return f"{missing}{param_hint}.{msg}"
    def __str__(self) -> str:
        if not self.message:
            param_name = self.param.name if self.param else None
            return _("Missing parameter: {param_name}").format(param_name=param_name)
        else:
            return self.message
class NoSuchOption(UsageError):
    """Raised if click attempted to handle an option that does not
    exist.
    .. versionadded:: 4.0
    """
    def __init__(
        self,
        option_name: str,
        message: t.Optional[str] = None,
        possibilities: t.Optional[t.Sequence[str]] = None,
        ctx: t.Optional["Context"] = None,
    ) -> None:
        if message is None:
            message = _("No such option: {name}").format(name=option_name)
        super().__init__(message, ctx)
        self.option_name = option_name
        self.possibilities = possibilities
    def format_message(self) -> str:
        if not self.possibilities:
            return self.message
        possibility_str = ", ".join(sorted(self.possibilities))
        suggest = ngettext(
            "Did you mean {possibility}?",
            "(Possible options: {possibilities})",
            len(self.possibilities),
        ).format(possibility=possibility_str, possibilities=possibility_str)
        return f"{self.message} {suggest}"
class BadOptionUsage(UsageError):
    """Raised if an option is generally supplied but the use of the option
    was incorrect.  This is for instance raised if the number of arguments
    for an option is not correct.
    .. versionadded:: 4.0
    :param option_name: the name of the option being used incorrectly.
    """
    def __init__(
        self, option_name: str, message: str, ctx: t.Optional["Context"] = None
    ) -> None:
        super().__init__(message, ctx)
        self.option_name = option_name
class BadArgumentUsage(UsageError):
    """Raised if an argument is generally supplied but the use of the argument
    was incorrect.  This is for instance raised if the number of values
    for an argument is not correct.
    .. versionadded:: 6.0
    """
class FileError(ClickException):
    """Raised if a file cannot be opened."""
    def __init__(self, filename: str, hint: t.Optional[str] = None) -> None:
        if hint is None:
            hint = _("unknown error")
        super().__init__(hint)
        self.ui_filename: str = format_filename(filename)
        self.filename = filename
    def format_message(self) -> str:
        return _("Could not open file {filename!r}: {message}").format(
            filename=self.ui_filename, message=self.message
        )
class Abort(RuntimeError):
    """An internal signalling exception that signals Click to abort."""
class Exit(RuntimeError):
    """An exception that indicates that the application should exit with some
    status code.
    :param code: the status code to exit with.
    """
    __slots__ = ("exit_code",)
    def __init__(self, code: int = 0) -> None:
        self.exit_code: int = code
```

File 9 - globals.py:

```
(0)             import typing as t
(0)             from threading import local
(0)             if t.TYPE_CHECKING:
(4)                 import typing_extensions as te
(4)                 from .core import Context
(0)             _local = local()
(0)             @t.overload
(0)             def get_current_context(silent: "te.Literal[False]" = False) -> "Context":
(4)                 ...
(0)             @t.overload
(0)             def get_current_context(silent: bool = ...) -> t.Optional["Context"]:
(4)                 ...
(0)             def get_current_context(silent: bool = False) -> t.Optional["Context"]:
(4)                 """Returns the current click context.  This can be used as a way to
(4)                 access the current context object from anywhere.  This is a more implicit
(4)                 alternative to the :func:`pass_context` decorator.  This function is
(4)                 primarily useful for helpers such as :func:`echo` which might be
(4)                 interested in changing its behavior based on the current context.
(4)                 To push the current context, :meth:`Context.scope` can be used.
(4)                 .. versionadded:: 5.0
(4)                 :param silent: if set to `True` the return value is `None` if no context
(19)                               is available.  The default behavior is to raise a
(19)                               :exc:`RuntimeError`.
(4)                 """
(4)                 try:
(8)                     return t.cast("Context", _local.stack[-1])
(4)                 except (AttributeError, IndexError) as e:
(8)                     if not silent:
(12)                        raise RuntimeError("There is no active click context.") from e
(4)                 return None
(0)             def push_context(ctx: "Context") -> None:
(4)                 """Pushes a new context to the current stack."""
(4)                 _local.__dict__.setdefault("stack", []).append(ctx)
(0)             def pop_context() -> None:
(4)                 """Removes the top level from the stack."""
(4)                 _local.stack.pop()
(0)             def resolve_color_default(color: t.Optional[bool] = None) -> t.Optional[bool]:
(4)                 """Internal helper to get the default value of the color flag.  If a
(4)                 value is passed it's returned unchanged, otherwise it's looked up from
(4)                 the current context.
(4)                 """
(4)                 if color is not None:
(8)                     return color
(4)                 ctx = get_current_context(silent=True)
(4)                 if ctx is not None:
(8)                     return ctx.color
(4)                 return None
```

File 10 - formatting.py:

```
(0)             import typing as t
(0)             from contextlib import contextmanager
(0)             from gettext import gettext as _
(0)             from ._compat import term_len
(0)             from .parser import split_opt
(0)             FORCED_WIDTH: t.Optional[int] = None
(0)             def measure_table(rows: t.Iterable[t.Tuple[str, str]]) -> t.Tuple[int, ...]:
(4)                 widths: t.Dict[int, int] = {}
(4)                 for row in rows:
(8)                     for idx, col in enumerate(row):
(12)                        widths[idx] = max(widths.get(idx, 0), term_len(col))
(4)                 return tuple(y for x, y in sorted(widths.items()))
(0)             def iter_rows(
(4)                 rows: t.Iterable[t.Tuple[str, str]], col_count: int
(0)             ) -> t.Iterator[t.Tuple[str, ...]]:
(4)                 for row in rows:
        yield row + ("",) * (col_count - len(row))
def wrap_text(
    text: str,
    width: int = 78,
    initial_indent: str = "",
    subsequent_indent: str = "",
    preserve_paragraphs: bool = False,
) -> str:
    """A helper function that intelligently wraps text.  By default, it
    assumes that it operates on a single paragraph of text but if the
    `preserve_paragraphs` parameter is provided it will intelligently
    handle paragraphs (defined by two empty lines).
    If paragraphs are handled, a paragraph can be prefixed with an empty
    line containing the ``\\b`` character (``\\x08``) to indicate that
    no rewrapping should happen in that block.
    :param text: the text that should be rewrapped.
    :param width: the maximum width for the text.
    :param initial_indent: the initial indent that should be placed on the
                           first line as a string.
    :param subsequent_indent: the indent string that should be placed on
                              each consecutive line.
    :param preserve_paragraphs: if this flag is set then the wrapping will
                                intelligently handle paragraphs.
    """
    from ._textwrap import TextWrapper
    text = text.expandtabs()
    wrapper = TextWrapper(
        width,
        initial_indent=initial_indent,
        subsequent_indent=subsequent_indent,
        replace_whitespace=False,
    )
    if not preserve_paragraphs:
        return wrapper.fill(text)
    p: t.List[t.Tuple[int, bool, str]] = []
    buf: t.List[str] = []
    indent = None
    def _flush_par() -> None:
        if not buf:
            return
        if buf[0].strip() == "\b":
            p.append((indent or 0, True, "\n".join(buf[1:])))
        else:
            p.append((indent or 0, False, " ".join(buf)))
        del buf[:]
    for line in text.splitlines():
        if not line:
            _flush_par()
            indent = None
        else:
            if indent is None:
                orig_len = term_len(line)
                line = line.lstrip()
                indent = orig_len - term_len(line)
            buf.append(line)
    _flush_par()
    rv = []
    for indent, raw, text in p:
        with wrapper.extra_indent(" " * indent):
            if raw:
                rv.append(wrapper.indent_only(text))
            else:
                rv.append(wrapper.fill(text))
    return "\n\n".join(rv)
class HelpFormatter:
    """This class helps with formatting text-based help pages.  It's
    usually just needed for very special internal cases, but it's also
    exposed so that developers can write their own fancy outputs.
    At present, it always writes into memory.
```

```python
    :param indent_increment: the additional increment for each level.
    :param width: the width for the text.  This defaults to the terminal
                  width clamped to a maximum of 78.
    """
    def __init__(
        self,
        indent_increment: int = 2,
        width: t.Optional[int] = None,
        max_width: t.Optional[int] = None,
    ) -> None:
        import shutil
        self.indent_increment = indent_increment
        if max_width is None:
            max_width = 80
        if width is None:
            width = FORCED_WIDTH
            if width is None:
                width = max(min(shutil.get_terminal_size().columns, max_width) - 2, 50)
        self.width = width
        self.current_indent = 0
        self.buffer: t.List[str] = []
    def write(self, string: str) -> None:
        """Writes a unicode string into the internal buffer."""
        self.buffer.append(string)
    def indent(self) -> None:
        """Increases the indentation."""
        self.current_indent += self.indent_increment
    def dedent(self) -> None:
        """Decreases the indentation."""
        self.current_indent -= self.indent_increment
    def write_usage(
        self, prog: str, args: str = "", prefix: t.Optional[str] = None
    ) -> None:
        """Writes a usage line into the buffer.
        :param prog: the program name.
        :param args: whitespace separated list of arguments.
        :param prefix: The prefix for the first line. Defaults to
            ``"Usage: "``.
        """
        if prefix is None:
            prefix = f"{_('Usage:')} "
        usage_prefix = f"{prefix:>{self.current_indent}}{prog} "
        text_width = self.width - self.current_indent
        if text_width >= (term_len(usage_prefix) + 20):
            indent = " " * term_len(usage_prefix)
            self.write(
                wrap_text(
                    args,
                    text_width,
                    initial_indent=usage_prefix,
                    subsequent_indent=indent,
                )
            )
        else:
            self.write(usage_prefix)
            self.write("\n")
            indent = " " * (max(self.current_indent, term_len(prefix)) + 4)
            self.write(
                wrap_text(
                    args, text_width, initial_indent=indent, subsequent_indent=indent
                )
            )
        self.write("\n")
    def write_heading(self, heading: str) -> None:
        """Writes a heading into the buffer."""
        self.write(f"{'':>{self.current_indent}}{heading}:\n")
    def write_paragraph(self) -> None:
```

```
(8)                     """Writes a paragraph into the buffer."""
(8)                     if self.buffer:
(12)                        self.write("\n")
(4)                 def write_text(self, text: str) -> None:
(8)                     """Writes re-indented text into the buffer.  This rewraps and
(8)                     preserves paragraphs.
(8)                     """
(8)                     indent = " " * self.current_indent
(8)                     self.write(
(12)                        wrap_text(
(16)                            text,
(16)                            self.width,
(16)                            initial_indent=indent,
(16)                            subsequent_indent=indent,
(16)                            preserve_paragraphs=True,
(12)                        )
(8)                     )
(8)                     self.write("\n")
(4)                 def write_dl(
(8)                     self,
(8)                     rows: t.Sequence[t.Tuple[str, str]],
(8)                     col_max: int = 30,
(8)                     col_spacing: int = 2,
(4)                 ) -> None:
(8)                     """Writes a definition list into the buffer.  This is how options
(8)                     and commands are usually formatted.
(8)                     :param rows: a list of two item tuples for the terms and values.
(8)                     :param col_max: the maximum width of the first column.
(8)                     :param col_spacing: the number of spaces between the first and
(28)                                        second column.
(8)                     """
(8)                     rows = list(rows)
(8)                     widths = measure_table(rows)
(8)                     if len(widths) != 2:
(12)                        raise TypeError("Expected two columns for definition list")
(8)                     first_col = min(widths[0], col_max) + col_spacing
(8)                     for first, second in iter_rows(rows, len(widths)):
(12)                        self.write(f"{'':>{self.current_indent}}{first}")
(12)                        if not second:
(16)                            self.write("\n")
(16)                            continue
(12)                        if term_len(first) <= first_col - col_spacing:
(16)                            self.write(" " * (first_col - term_len(first)))
(12)                        else:
(16)                            self.write("\n")
(16)                            self.write(" " * (first_col + self.current_indent))
(12)                        text_width = max(self.width - first_col - 2, 10)
(12)                        wrapped_text = wrap_text(second, text_width, preserve_paragraphs=True)
(12)                        lines = wrapped_text.splitlines()
(12)                        if lines:
(16)                            self.write(f"{lines[0]}\n")
(16)                            for line in lines[1:]:
(20)                                self.write(f"{'':>{first_col + self.current_indent}}{line}\n")
(12)                        else:
(16)                            self.write("\n")
(4)                 @contextmanager
(4)                 def section(self, name: str) -> t.Iterator[None]:
(8)                     """Helpful context manager that writes a paragraph, a heading,
(8)                     and the indents.
(8)                     :param name: the section name that is written as heading.
(8)                     """
(8)                     self.write_paragraph()
(8)                     self.write_heading(name)
(8)                     self.indent()
(8)                     try:
(12)                        yield
(8)                     finally:
```

```
            self.dedent()
    @contextmanager
    def indentation(self) -> t.Iterator[None]:
        """A context manager that increases the indentation."""
        self.indent()
        try:
            yield
        finally:
            self.dedent()
    def getvalue(self) -> str:
        """Returns the buffer contents."""
        return "".join(self.buffer)
def join_options(options: t.Sequence[str]) -> t.Tuple[str, bool]:
    """Given a list of option strings this joins them in the most appropriate
    way and returns them in the form ``(formatted_string,
    any_prefix_is_slash)`` where the second item in the tuple is a flag that
    indicates if any of the option prefixes was a slash.
    """
    rv = []
    any_prefix_is_slash = False
    for opt in options:
        prefix = split_opt(opt)[0]
        if prefix == "/":
            any_prefix_is_slash = True
        rv.append((len(prefix), opt))
    rv.sort(key=lambda x: x[0])
    return ", ".join(x[1] for x in rv), any_prefix_is_slash
```

File 11 - parser.py:

```
"""
This module started out as largely a copy paste from the stdlib's
optparse module with the features removed that we do not need from
optparse because we implement them in Click on a higher level (for
instance type handling, help formatting and a lot more).
The plan is to remove more and more from here over time.
The reason this is a different module and not optparse from the stdlib
is that there are differences in 2.x and 3.x about the error messages
generated and optparse in the stdlib uses gettext for no good reason
and might cause us issues.
Click uses parts of optparse written by Gregory P. Ward and maintained
by the Python Software Foundation. This is limited to code in parser.py.
Copyright 2001-2006 Gregory P. Ward. All rights reserved.
Copyright 2002-2006 Python Software Foundation. All rights reserved.
"""
import typing as t
from collections import deque
from gettext import gettext as _
from gettext import ngettext
from .exceptions import BadArgumentUsage
from .exceptions import BadOptionUsage
from .exceptions import NoSuchOption
from .exceptions import UsageError
if t.TYPE_CHECKING:
    import typing_extensions as te
    from .core import Argument as CoreArgument
    from .core import Context
    from .core import Option as CoreOption
    from .core import Parameter as CoreParameter
V = t.TypeVar("V")
_flag_needs_value = object()
def _unpack_args(
    args: t.Sequence[str], nargs_spec: t.Sequence[int]
) -> t.Tuple[t.Sequence[t.Union[str, t.Sequence[t.Optional[str]], None]], t.List[str]]:
    """Given an iterable of arguments and an iterable of nargs specifications,
    it returns a tuple with all the unpacked arguments at the first index
    and all remaining arguments as the second.
    The nargs specification is the number of arguments that should be consumed
    or `-1` to indicate that this position should eat up all the remainders.
    Missing items are filled with `None`.
    """
    args = deque(args)
    nargs_spec = deque(nargs_spec)
    rv: t.List[t.Union[str, t.Tuple[t.Optional[str], ...], None]] = []
    spos: t.Optional[int] = None
    def _fetch(c: "te.Deque[V]") -> t.Optional[V]:
        try:
            if spos is None:
                return c.popleft()
            else:
                return c.pop()
        except IndexError:
            return None
    while nargs_spec:
        nargs = _fetch(nargs_spec)
        if nargs is None:
            continue
        if nargs == 1:
            rv.append(_fetch(args))
        elif nargs > 1:
            x = [_fetch(args) for _ in range(nargs)]
            if spos is not None:
                x.reverse()
            rv.append(tuple(x))
        elif nargs < 0:
            if spos is not None:
                raise TypeError("Cannot have two nargs < 0")
            spos = len(rv)
            rv.append(None)
    if spos is not None:
        rv[spos] = tuple(args)
        args = []
        rv[spos + 1 :] = reversed(rv[spos + 1 :])
    return tuple(rv), list(args)
def split_opt(opt: str) -> t.Tuple[str, str]:
    first = opt[:1]
    if first.isalnum():
        return "", opt
    if opt[1:2] == first:
        return opt[:2], opt[2:]
    return first, opt[1:]
def normalize_opt(opt: str, ctx: t.Optional["Context"]) -> str:
    if ctx is None or ctx.token_normalize_func is None:
        return opt
    prefix, opt = split_opt(opt)
    return f"{prefix}{ctx.token_normalize_func(opt)}"
def split_arg_string(string: str) -> t.List[str]:
    """Split an argument string as with :func:`shlex.split`, but don't
    fail if the string is incomplete. Ignores a missing closing quote or
    incomplete escape sequence and uses the partial token as-is.
    .. code-block:: python
        split_arg_string("example 'my file")
        ["example", "my file"]
        split_arg_string("example my\\")
        ["example", "my"]
    :param string: String to split.
    """
    import shlex
    lex = shlex.shlex(string, posix=True)
    lex.whitespace_split = True
    lex.commenters = ""
    out = []
    try:
        for token in lex:
            out.append(token)
    except ValueError:
        out.append(lex.token)
    return out
class Option:
    def __init__(
        self,
        obj: "CoreOption",
        opts: t.Sequence[str],
        dest: t.Optional[str],
        action: t.Optional[str] = None,
        nargs: int = 1,
        const: t.Optional[t.Any] = None,
    ):
        self._short_opts = []
        self._long_opts = []
        self.prefixes: t.Set[str] = set()
        for opt in opts:
            prefix, value = split_opt(opt)
            if not prefix:
                raise ValueError(f"Invalid start character for option ({opt})")
            self.prefixes.add(prefix[0])
            if len(prefix) == 1 and len(value) == 1:
                self._short_opts.append(opt)
            else:
                self._long_opts.append(opt)
                self.prefixes.add(prefix)
        if action is None:
            action = "store"
        self.dest = dest
        self.action = action
        self.nargs = nargs
        self.const = const
        self.obj = obj
    @property
    def takes_value(self) -> bool:
        return self.action in ("store", "append")
    def process(self, value: t.Any, state: "ParsingState") -> None:
        if self.action == "store":
            state.opts[self.dest] = value  # type: ignore
        elif self.action == "store_const":
            state.opts[self.dest] = self.const  # type: ignore
        elif self.action == "append":
            state.opts.setdefault(self.dest, []).append(value)  # type: ignore
        elif self.action == "append_const":
            state.opts.setdefault(self.dest, []).append(self.const)  # type: ignore
        elif self.action == "count":
            state.opts[self.dest] = state.opts.get(self.dest, 0) + 1  # type: ignore
        else:
            raise ValueError(f"unknown action '{self.action}'")
        state.order.append(self.obj)
class Argument:
    def __init__(self, obj: "CoreArgument", dest: t.Optional[str], nargs: int = 1):
        self.dest = dest
        self.nargs = nargs
        self.obj = obj
    def process(
        self,
        value: t.Union[t.Optional[str], t.Sequence[t.Optional[str]]],
        state: "ParsingState",
    ) -> None:
        if self.nargs > 1:
            assert value is not None
            holes = sum(1 for x in value if x is None)
            if holes == len(value):
                value = None
```

```
(12)            elif holes != 0:
(16)                raise BadArgumentUsage(
(20)                    _("Argument {name!r} takes {nargs} values.").format(
(24)                        name=self.dest, nargs=self.nargs
(20)                    )
(16)                )
(8)         if self.nargs == -1 and self.obj.envvar is not None and value == ():
(12)            value = None
(8)         state.opts[self.dest] = value  # type: ignore
(8)         state.order.append(self.obj)
(0) class ParsingState:
(4)     def __init__(self, rargs: t.List[str]) -> None:
(8)         self.opts: t.Dict[str, t.Any] = {}
(8)         self.largs: t.List[str] = []
(8)         self.rargs = rargs
(8)         self.order: t.List["CoreParameter"] = []
(0) class OptionParser:
(4)     """The option parser is an internal class that is ultimately used to
(4)     parse options and arguments.  It's modelled after optparse and brings
(4)     a similar but vastly simplified API.  It should generally not be used
(4)     directly as the high level Click classes wrap it for you.
(4)     It's not nearly as extensible as optparse or argparse as it does not
(4)     implement features that are implemented on a higher level (such as
(4)     types or defaults).
(4)     :param ctx: optionally the :class:`~click.Context` where this parser
(16)                should go with.
(4)     """
(4)     def __init__(self, ctx: t.Optional["Context"] = None) -> None:
(8)         self.ctx = ctx
(8)         self.allow_interspersed_args: bool = True
(8)         self.ignore_unknown_options: bool = False
(8)         if ctx is not None:
(12)            self.allow_interspersed_args = ctx.allow_interspersed_args
(12)            self.ignore_unknown_options = ctx.ignore_unknown_options
(8)         self._short_opt: t.Dict[str, Option] = {}
(8)         self._long_opt: t.Dict[str, Option] = {}
(8)         self._opt_prefixes = {"-", "--"}
(8)         self._args: t.List[Argument] = []
(4)     def add_option(
(8)         self,
(8)         obj: "CoreOption",
(8)         opts: t.Sequence[str],
(8)         dest: t.Optional[str],
(8)         action: t.Optional[str] = None,
(8)         nargs: int = 1,
(8)         const: t.Optional[t.Any] = None,
(4)     ) -> None:
(8)         """Adds a new option named `dest` to the parser.  The destination
(8)         is not inferred (unlike with optparse) and needs to be explicitly
(8)         provided.  Action can be any of ``store``, ``store_const``,
(8)         ``append``, ``append_const`` or ``count``.
(8)         The `obj` can be used to identify the option in the order list
(8)         that is returned from the parser.
(8)         """
(8)         opts = [normalize_opt(opt, self.ctx) for opt in opts]
(8)         option = Option(obj, opts, dest, action=action, nargs=nargs,
const=const)
(8)         self._opt_prefixes.update(option.prefixes)
(8)         for opt in option._short_opts:
(12)            self._short_opt[opt] = option
(8)         for opt in option._long_opts:
(12)            self._long_opt[opt] = option
(4)     def add_argument(
(8)         self, obj: "CoreArgument", dest: t.Optional[str], nargs: int = 1
(4)     ) -> None:
(8)         """Adds a positional argument named `dest` to the parser.
(8)         The `obj` can be used to identify the option in the order list
(8)         that is returned from the parser.
(8)         """
```

```
(8)                         self._args.append(Argument(obj, dest=dest, nargs=nargs))
(4)                     def parse_args(
(8)                         self, args: t.List[str]
(4)                     ) -> t.Tuple[t.Dict[str, t.Any], t.List[str], t.List["CoreParameter"]]:
(8)                         """Parses positional arguments and returns ``(values, args, order)``
(8)                         for the parsed options and arguments as well as the leftover
(8)                         arguments if there are any.  The order is a list of objects as they
(8)                         appear on the command line.  If arguments appear multiple times they
(8)                         will be memorized multiple times as well.
(8)                         """
(8)                         state = ParsingState(args)
(8)                         try:
(12)                            self._process_args_for_options(state)
(12)                            self._process_args_for_args(state)
(8)                         except UsageError:
(12)                            if self.ctx is None or not self.ctx.resilient_parsing:
(16)                                raise
(8)                         return state.opts, state.largs, state.order
(4)                     def _process_args_for_args(self, state: ParsingState) -> None:
(8)                         pargs, args = _unpack_args(
(12)                            state.largs + state.rargs, [x.nargs for x in self._args]
(8)                         )
(8)                         for idx, arg in enumerate(self._args):
(12)                            arg.process(pargs[idx], state)
(8)                         state.largs = args
(8)                         state.rargs = []
(4)                     def _process_args_for_options(self, state: ParsingState) -> None:
(8)                         while state.rargs:
(12)                            arg = state.rargs.pop(0)
(12)                            arglen = len(arg)
(12)                            if arg == "--":
(16)                                return
(12)                            elif arg[:1] in self._opt_prefixes and arglen > 1:
(16)                                self._process_opts(arg, state)
(12)                            elif self.allow_interspersed_args:
(16)                                state.largs.append(arg)
(12)                            else:
(16)                                state.rargs.insert(0, arg)
(16)                                return
(4)                     def _match_long_opt(
(8)                         self, opt: str, explicit_value: t.Optional[str], state: ParsingState
(4)                     ) -> None:
(8)                         if opt not in self._long_opt:
(12)                            from difflib import get_close_matches
(12)                            possibilities = get_close_matches(opt, self._long_opt)
(12)                            raise NoSuchOption(opt, possibilities=possibilities, ctx=self.ctx)
(8)                         option = self._long_opt[opt]
(8)                         if option.takes_value:
(12)                            if explicit_value is not None:
(16)                                state.rargs.insert(0, explicit_value)
(12)                            value = self._get_value_from_state(opt, option, state)
(8)                         elif explicit_value is not None:
(12)                            raise BadOptionUsage(
(16)                                opt, _("Option {name!r} does not take a
value.").format(name=opt)
(12)                            )
(8)                         else:
(12)                            value = None
(8)                         option.process(value, state)
(4)                     def _match_short_opt(self, arg: str, state: ParsingState) -> None:
(8)                         stop = False
(8)                         i = 1
(8)                         prefix = arg[0]
(8)                         unknown_options = []
(8)                         for ch in arg[1:]:
(12)                            opt = normalize_opt(f"{prefix}{ch}", self.ctx)
(12)                            option = self._short_opt.get(opt)
(12)                            i += 1
(12)                            if not option:
```

```
(16)                    if self.ignore_unknown_options:
(20)                        unknown_options.append(ch)
(20)                        continue
(16)                    raise NoSuchOption(opt, ctx=self.ctx)
(12)                if option.takes_value:
(16)                    if i < len(arg):
(20)                        state.rargs.insert(0, arg[i:])
(20)                        stop = True
(16)                    value = self._get_value_from_state(opt, option, state)
(12)                else:
(16)                    value = None
(12)                option.process(value, state)
(12)                if stop:
(16)                    break
(8)             if self.ignore_unknown_options and unknown_options:
(12)                state.largs.append(f"{prefix}{''.join(unknown_options)}")
(4)         def _get_value_from_state(
(8)             self, option_name: str, option: Option, state: ParsingState
(4)         ) -> t.Any:
(8)             nargs = option.nargs
(8)             if len(state.rargs) < nargs:
(12)                if option.obj._flag_needs_value:
(16)                    value = _flag_needs_value
(12)                else:
(16)                    raise BadOptionUsage(
(20)                        option_name,
(20)                        ngettext(
(24)                            "Option {name!r} requires an argument.",
(24)                            "Option {name!r} requires {nargs} arguments.",
(24)                            nargs,
(20)                        ).format(name=option_name, nargs=nargs),
(16)                    )
(8)             elif nargs == 1:
(12)                next_rarg = state.rargs[0]
(12)                if (
(16)                    option.obj._flag_needs_value
(16)                    and isinstance(next_rarg, str)
(16)                    and next_rarg[:1] in self._opt_prefixes
(16)                    and len(next_rarg) > 1
(12)                ):
(16)                    value = _flag_needs_value
(12)                else:
(16)                    value = state.rargs.pop(0)
(8)             else:
(12)                value = tuple(state.rargs[:nargs])
(12)                del state.rargs[:nargs]
(8)             return value
(4)         def _process_opts(self, arg: str, state: ParsingState) -> None:
(8)             explicit_value = None
(8)             if "=" in arg:
(12)                long_opt, explicit_value = arg.split("=", 1)
(8)             else:
(12)                long_opt = arg
(8)             norm_long_opt = normalize_opt(long_opt, self.ctx)
(8)             try:
(12)                self._match_long_opt(norm_long_opt, explicit_value, state)
(8)             except NoSuchOption:
(12)                if arg[:2] not in self._opt_prefixes:
(16)                    self._match_short_opt(arg, state)
(16)                    return
(12)                if not self.ignore_unknown_options:
(16)                    raise
(12)                state.largs.append(arg)
```

File 12 - shell_completion.py:

```
(0)         import os
```

```python
import re
import typing as t
from gettext import gettext as _
from .core import Argument
from .core import BaseCommand
from .core import Context
from .core import MultiCommand
from .core import Option
from .core import Parameter
from .core import ParameterSource
from .parser import split_arg_string
from .utils import echo
def shell_complete(
    cli: BaseCommand,
    ctx_args: t.MutableMapping[str, t.Any],
    prog_name: str,
    complete_var: str,
    instruction: str,
) -> int:
    """Perform shell completion for the given CLI program.
    :param cli: Command being called.
    :param ctx_args: Extra arguments to pass to
        ``cli.make_context``.
    :param prog_name: Name of the executable in the shell.
    :param complete_var: Name of the environment variable that holds
        the completion instruction.
    :param instruction: Value of ``complete_var`` with the completion
        instruction and shell, in the form ``instruction_shell``.
    :return: Status code to exit with.
    """
    shell, _, instruction = instruction.partition("_")
    comp_cls = get_completion_class(shell)
    if comp_cls is None:
        return 1
    comp = comp_cls(cli, ctx_args, prog_name, complete_var)
    if instruction == "source":
        echo(comp.source())
        return 0
    if instruction == "complete":
        echo(comp.complete())
        return 0
    return 1
class CompletionItem:
    """Represents a completion value and metadata about the value. The
    default metadata is ``type`` to indicate special shell handling,
    and ``help`` if a shell supports showing a help string next to the
    value.
    Arbitrary parameters can be passed when creating the object, and
    accessed using ``item.attr``. If an attribute wasn't passed,
    accessing it returns ``None``.
    :param value: The completion suggestion.
    :param type: Tells the shell script to provide special completion
        support for the type. Click uses ``"dir"`` and ``"file"``.
    :param help: String shown next to the value if supported.
    :param kwargs: Arbitrary metadata. The built-in implementations
        don't use this, but custom type completions paired with custom
        shell support could use it.
    """
    __slots__ = ("value", "type", "help", "_info")
    def __init__(
        self,
        value: t.Any,
        type: str = "plain",
        help: t.Optional[str] = None,
        **kwargs: t.Any,
    ) -> None:
        self.value: t.Any = value
        self.type: str = type
        self.help: t.Optional[str] = help
```

```
(8)                     self._info = kwargs
(4)                 def __getattr__(self, name: str) -> t.Any:
(8)                     return self._info.get(name)
(0)             _SOURCE_BASH = """\
(0)             %(complete_func)s() {
(4)                 local IFS=$'\\n'
(4)                 local response
(4)                 response=$(env COMP_WORDS="${COMP_WORDS[*]}" COMP_CWORD=$COMP_CWORD \
(0)             %(complete_var)s=bash_complete $1)
(4)                 for completion in $response; do
(8)                     IFS=',' read type value <<< "$completion"
(8)                     if [[ $type == 'dir' ]]; then
(12)                        COMPREPLY=()
(12)                        compopt -o dirnames
(8)                     elif [[ $type == 'file' ]]; then
(12)                        COMPREPLY=()
(12)                        compopt -o default
(8)                     elif [[ $type == 'plain' ]]; then
(12)                        COMPREPLY+=($value)
(8)                     fi
(4)                 done
(4)                 return 0
(0)             }
(0)             %(complete_func)s_setup() {
(4)                 complete -o nosort -F %(complete_func)s %(prog_name)s
(0)             }
(0)             %(complete_func)s_setup;
(0)             """
(0)             _SOURCE_ZSH = """\
(0)             %(complete_func)s() {
(4)                 local -a completions
(4)                 local -a completions_with_descriptions
(4)                 local -a response
(4)                 (( ! $+commands[%(prog_name)s] )) && return 1
(4)                 response=("${(@f)$(env COMP_WORDS="${words[*]}" COMP_CWORD=$((CURRENT-1)) \
(0)             %(complete_var)s=zsh_complete %(prog_name)s)}")
(4)                 for type key descr in ${response}; do
(8)                     if [[ "$type" == "plain" ]]; then
(12)                        if [[ "$descr" == "_" ]]; then
(16)                            completions+=("$key")
(12)                        else
(16)                            completions_with_descriptions+=("$key":"$descr")
(12)                        fi
(8)                     elif [[ "$type" == "dir" ]]; then
(12)                        _path_files -/
(8)                     elif [[ "$type" == "file" ]]; then
(12)                        _path_files -f
(8)                     fi
(4)                 done
(4)                 if [ -n "$completions_with_descriptions" ]; then
(8)                     _describe -V unsorted completions_with_descriptions -U
(4)                 fi
(4)                 if [ -n "$completions" ]; then
(8)                     compadd -U -V unsorted -a completions
(4)                 fi
(0)             }
(0)             if [[ $zsh_eval_context[-1] == loadautofunc ]]; then
(4)                 %(complete_func)s "$@"
(0)             else
(4)                 compdef %(complete_func)s %(prog_name)s
(0)             fi
(0)             """
(0)             _SOURCE_FISH = """\
(0)             function %(complete_func)s;
(4)                 set -l response (env %(complete_var)s=fish_complete COMP_WORDS=(commandline -cp) \
(0)             COMP_CWORD=(commandline -t) %(prog_name)s);
(4)                 for completion in $response;
```

```
        set -l metadata (string split "," $completion);
        if test $metadata[1] = "dir";
            __fish_complete_directories $metadata[2];
        else if test $metadata[1] = "file";
            __fish_complete_path $metadata[2];
        else if test $metadata[1] = "plain";
            echo $metadata[2];
        end;
    end;
end;
complete --no-files --command %(prog_name)s --arguments \
"(%(complete_func)s)";
"""
class ShellComplete:
    """Base class for providing shell completion support. A subclass for
    a given shell will override attributes and methods to implement the
    completion instructions (``source`` and ``complete``).
    :param cli: Command being called.
    :param prog_name: Name of the executable in the shell.
    :param complete_var: Name of the environment variable that holds
        the completion instruction.
    .. versionadded:: 8.0
    """
    name: t.ClassVar[str]
    """Name to register the shell as with :func:`add_completion_class`.
    This is used in completion instructions (``{name}_source`` and
    ``{name}_complete``).
    """
    source_template: t.ClassVar[str]
    """Completion script template formatted by :meth:`source`. This must
    be provided by subclasses.
    """
    def __init__(
        self,
        cli: BaseCommand,
        ctx_args: t.MutableMapping[str, t.Any],
        prog_name: str,
        complete_var: str,
    ) -> None:
        self.cli = cli
        self.ctx_args = ctx_args
        self.prog_name = prog_name
        self.complete_var = complete_var
    @property
    def func_name(self) -> str:
        """The name of the shell function defined by the completion
        script.
        """
        safe_name = re.sub(r"\W*", "", self.prog_name.replace("-", "_"), flags=re.ASCII)
        return f"_{safe_name}_completion"
    def source_vars(self) -> t.Dict[str, t.Any]:
        """Vars for formatting :attr:`source_template`.
        By default this provides ``complete_func``, ``complete_var``,
        and ``prog_name``.
        """
        return {
            "complete_func": self.func_name,
            "complete_var": self.complete_var,
            "prog_name": self.prog_name,
        }
    def source(self) -> str:
        """Produce the shell script that defines the completion
        function. By default this ``%``-style formats
        :attr:`source_template` with the dict returned by
        :meth:`source_vars`.
        """
        return self.source_template % self.source_vars()
    def get_completion_args(self) -> t.Tuple[t.List[str], str]:
        """Use the env vars defined by the shell script to return a
        tuple of ``args, incomplete``. This must be implemented by
        subclasses.
        """
        raise NotImplementedError
    def get_completions(
        self, args: t.List[str], incomplete: str
    ) -> t.List[CompletionItem]:
        """Determine the context and last complete command or parameter
        from the complete args. Call that object's ``shell_complete``
        method to get the completions for the incomplete value.
        :param args: List of complete args before the incomplete value.
        :param incomplete: Value being completed. May be empty.
        """
        ctx = _resolve_context(self.cli, self.ctx_args, self.prog_name, args)
        obj, incomplete = _resolve_incomplete(ctx, args, incomplete)
        return obj.shell_complete(ctx, incomplete)
    def format_completion(self, item: CompletionItem) -> str:
        """Format a completion item into the form recognized by the
        shell script. This must be implemented by subclasses.
        :param item: Completion item to format.
        """
        raise NotImplementedError
    def complete(self) -> str:
        """Produce the completion data to send back to the shell.
        By default this calls :meth:`get_completion_args`, gets the
        completions, then calls :meth:`format_completion` for each
        completion.
        """
        args, incomplete = self.get_completion_args()
        completions = self.get_completions(args, incomplete)
        out = [self.format_completion(item) for item in completions]
        return "\n".join(out)
class BashComplete(ShellComplete):
    """Shell completion for Bash."""
    name = "bash"
    source_template = _SOURCE_BASH
    @staticmethod
    def _check_version() -> None:
        import subprocess
        output = subprocess.run(
            ["bash", "-c", 'echo "${BASH_VERSION}"'], stdout=subprocess.PIPE
        )
        match = re.search(r"^(\d+)\.(\d+)\.\d+", output.stdout.decode())
        if match is not None:
            major, minor = match.groups()
            if major < "4" or major == "4" and minor < "4":
                echo(
                    _(
                        "Shell completion is not supported for Bash"
                        " versions older than 4.4."
                    ),
                    err=True,
                )
        else:
            echo(
                _("Couldn't detect Bash version, shell completion is not supported."),
                err=True,
            )
    def source(self) -> str:
        self._check_version()
        return super().source()
    def get_completion_args(self) -> t.Tuple[t.List[str], str]:
        cwords = split_arg_string(os.environ["COMP_WORDS"])
        cword = int(os.environ["COMP_CWORD"])
        args = cwords[1:cword]
        try:
            incomplete = cwords[cword]
```

```
(8)                  except IndexError:
(12)                     incomplete = ""
(8)                  return args, incomplete
(4)              def format_completion(self, item: CompletionItem) -> str:
(8)                  return f"{item.type},{item.value}"
(0)          class ZshComplete(ShellComplete):
(4)              """Shell completion for Zsh."""
(4)              name = "zsh"
(4)              source_template = _SOURCE_ZSH
(4)              def get_completion_args(self) -> t.Tuple[t.List[str], str]:
(8)                  cwords = split_arg_string(os.environ["COMP_WORDS"])
(8)                  cword = int(os.environ["COMP_CWORD"])
(8)                  args = cwords[1:cword]
(8)                  try:
(12)                     incomplete = cwords[cword]
(8)                  except IndexError:
(12)                     incomplete = ""
(8)                  return args, incomplete
(4)              def format_completion(self, item: CompletionItem) -> str:
(8)                  return f"{item.type}\n{item.value}\n{item.help if item.help else '_'}"
(0)          class FishComplete(ShellComplete):
(4)              """Shell completion for Fish."""
(4)              name = "fish"
(4)              source_template = _SOURCE_FISH
(4)              def get_completion_args(self) -> t.Tuple[t.List[str], str]:
(8)                  cwords = split_arg_string(os.environ["COMP_WORDS"])
(8)                  incomplete = os.environ["COMP_CWORD"]
(8)                  args = cwords[1:]
(8)                  if incomplete and args and args[-1] == incomplete:
(12)                     args.pop()
(8)                  return args, incomplete
(4)              def format_completion(self, item: CompletionItem) -> str:
(8)                  if item.help:
(12)                     return f"{item.type},{item.value}\t{item.help}"
(8)                  return f"{item.type},{item.value}"
(0)          ShellCompleteType = t.TypeVar("ShellCompleteType", bound=t.Type[ShellComplete])
(0)          _available_shells: t.Dict[str, t.Type[ShellComplete]] = {
(4)              "bash": BashComplete,
(4)              "fish": FishComplete,
(4)              "zsh": ZshComplete,
(0)          }
(0)          def add_completion_class(
(4)              cls: ShellCompleteType, name: t.Optional[str] = None
(0)          ) -> ShellCompleteType:
(4)              """Register a :class:`ShellComplete` subclass under the given name.
(4)              The name will be provided by the completion instruction environment
(4)              variable during completion.
(4)              :param cls: The completion class that will handle completion for the
(8)                  shell.
(4)              :param name: Name to register the class under. Defaults to the
(8)                  class's ``name`` attribute.
(4)              """
(4)              if name is None:
(8)                  name = cls.name
(4)              _available_shells[name] = cls
(4)              return cls
(0)          def get_completion_class(shell: str) -> t.Optional[t.Type[ShellComplete]]:
(4)              """Look up a registered :class:`ShellComplete` subclass by the name
(4)              provided by the completion instruction environment variable. If the
(4)              name isn't registered, returns ``None``.
(4)              :param shell: Name the class is registered under.
(4)              """
(4)              return _available_shells.get(shell)
(0)          def _is_incomplete_argument(ctx: Context, param: Parameter) -> bool:
(4)              """Determine if the given parameter is an argument that can still
(4)              accept values.
(4)              :param ctx: Invocation context for the command represented by the
(8)                  parsed complete args.
```

```
(4)                 :param param: Argument object being checked.
(4)                 """
(4)                 if not isinstance(param, Argument):
(8)                     return False
(4)                 assert param.name is not None
(4)                 value = ctx.params.get(param.name)
(4)                 return (
(8)                     param.nargs == -1
(8)                     or ctx.get_parameter_source(param.name) is not ParameterSource.COMMANDLINE
(8)                     or (
(12)                        param.nargs > 1
(12)                        and isinstance(value, (tuple, list))
(12)                        and len(value) < param.nargs
(8)                     )
(4)                 )
(0)             def _start_of_option(ctx: Context, value: str) -> bool:
(4)                 """Check if the value looks like the start of an option."""
(4)                 if not value:
(8)                     return False
(4)                 c = value[0]
(4)                 return c in ctx._opt_prefixes
(0)             def _is_incomplete_option(ctx: Context, args: t.List[str], param: Parameter) -> bool:
(4)                 """Determine if the given parameter is an option that needs a value.
(4)                 :param args: List of complete args before the incomplete value.
(4)                 :param param: Option object being checked.
(4)                 """
(4)                 if not isinstance(param, Option):
(8)                     return False
(4)                 if param.is_flag or param.count:
(8)                     return False
(4)                 last_option = None
(4)                 for index, arg in enumerate(reversed(args)):
(8)                     if index + 1 > param.nargs:
(12)                        break
(8)                     if _start_of_option(ctx, arg):
(12)                        last_option = arg
(4)                 return last_option is not None and last_option in param.opts
(0)             def _resolve_context(
(4)                 cli: BaseCommand,
(4)                 ctx_args: t.MutableMapping[str, t.Any],
(4)                 prog_name: str,
(4)                 args: t.List[str],
(0)             ) -> Context:
(4)                 """Produce the context hierarchy starting with the command and
(4)                 traversing the complete arguments. This only follows the commands,
(4)                 it doesn't trigger input prompts or callbacks.
(4)                 :param cli: Command being called.
(4)                 :param prog_name: Name of the executable in the shell.
(4)                 :param args: List of complete args before the incomplete value.
(4)                 """
(4)                 ctx_args["resilient_parsing"] = True
(4)                 ctx = cli.make_context(prog_name, args.copy(), **ctx_args)
(4)                 args = ctx.protected_args + ctx.args
(4)                 while args:
(8)                     command = ctx.command
(8)                     if isinstance(command, MultiCommand):
(12)                        if not command.chain:
(16)                            name, cmd, args = command.resolve_command(ctx, args)
(16)                            if cmd is None:
(20)                                return ctx
(16)                            ctx = cmd.make_context(name, args, parent=ctx, resilient_parsing=True)
(16)                            args = ctx.protected_args + ctx.args
(12)                        else:
(16)                            sub_ctx = ctx
(16)                            while args:
(20)                                name, cmd, args = command.resolve_command(ctx, args)
```

```
(20)                    if cmd is None:
(24)                        return ctx
(20)                    sub_ctx = cmd.make_context(
(24)                        name,
(24)                        args,
(24)                        parent=ctx,
(24)                        allow_extra_args=True,
(24)                        allow_interspersed_args=False,
(24)                        resilient_parsing=True,
(20)                    )
(20)                    args = sub_ctx.args
(16)                ctx = sub_ctx
(16)                args = [*sub_ctx.protected_args, *sub_ctx.args]
(8)         else:
(12)            break
(4)     return ctx
(0) def _resolve_incomplete(
(4)     ctx: Context, args: t.List[str], incomplete: str
(0) ) -> t.Tuple[t.Union[BaseCommand, Parameter], str]:
(4)     """Find the Click object that will handle the completion of the
(4)     incomplete value. Return the object and the incomplete value.
(4)     :param ctx: Invocation context for the command represented by
(8)         the parsed complete args.
(4)     :param args: List of complete args before the incomplete value.
(4)     :param incomplete: Value being completed. May be empty.
(4)     """
(4)     if incomplete == "=":
(8)         incomplete = ""
(4)     elif "=" in incomplete and _start_of_option(ctx, incomplete):
(8)         name, _, incomplete = incomplete.partition("=")
(8)         args.append(name)
(4)     if "--" not in args and _start_of_option(ctx, incomplete):
(8)         return ctx.command, incomplete
(4)     params = ctx.command.get_params(ctx)
(4)     for param in params:
(8)         if _is_incomplete_option(ctx, args, param):
(12)            return param, incomplete
(4)     for param in params:
(8)         if _is_incomplete_argument(ctx, param):
(12)            return param, incomplete
(4)     return ctx.command, incomplete
```

File 13 - termui.py:

```
(0) import inspect
(0) import io
(0) import itertools
(0) import sys
(0) import typing as t
(0) from gettext import gettext as _
(0) from ._compat import isatty
(0) from ._compat import strip_ansi
(0) from .exceptions import Abort
(0) from .exceptions import UsageError
(0) from .globals import resolve_color_default
(0) from .types import Choice
(0) from .types import convert_type
(0) from .types import ParamType
(0) from .utils import echo
(0) from .utils import LazyFile
(0) if t.TYPE_CHECKING:
(4)     from ._termui_impl import ProgressBar
(0) V = t.TypeVar("V")
(0) visible_prompt_func: t.Callable[[str], str] = input
(0) _ansi_colors = {
(4)     "black": 30,
(4)     "red": 31,
(4)                 "green": 32,
(4)                 "yellow": 33,
(4)                 "blue": 34,
(4)                 "magenta": 35,
(4)                 "cyan": 36,
(4)                 "white": 37,
(4)                 "reset": 39,
(4)                 "bright_black": 90,
(4)                 "bright_red": 91,
(4)                 "bright_green": 92,
(4)                 "bright_yellow": 93,
(4)                 "bright_blue": 94,
(4)                 "bright_magenta": 95,
(4)                 "bright_cyan": 96,
(4)                 "bright_white": 97,
(0)             }
(0)             _ansi_reset_all = "\033[0m"
(0)             def hidden_prompt_func(prompt: str) -> str:
(4)                 import getpass
(4)                 return getpass.getpass(prompt)
(0)             def _build_prompt(
(4)                 text: str,
(4)                 suffix: str,
(4)                 show_default: bool = False,
(4)                 default: t.Optional[t.Any] = None,
(4)                 show_choices: bool = True,
(4)                 type: t.Optional[ParamType] = None,
(0)             ) -> str:
(4)                 prompt = text
(4)                 if type is not None and show_choices and isinstance(type, Choice):
(8)                     prompt += f" ({', '.join(map(str, type.choices))})"
(4)                 if default is not None and show_default:
(8)                     prompt = f"{prompt} [{_format_default(default)}]"
(4)                 return f"{prompt}{suffix}"
(0)             def _format_default(default: t.Any) -> t.Any:
(4)                 if isinstance(default, (io.IOBase, LazyFile)) and hasattr(default, "name"):
(8)                     return default.name
(4)                 return default
(0)             def prompt(
(4)                 text: str,
(4)                 default: t.Optional[t.Any] = None,
(4)                 hide_input: bool = False,
(4)                 confirmation_prompt: t.Union[bool, str] = False,
(4)                 type: t.Optional[t.Union[ParamType, t.Any]] = None,
(4)                 value_proc: t.Optional[t.Callable[[str], t.Any]] = None,
(4)                 prompt_suffix: str = ": ",
(4)                 show_default: bool = True,
(4)                 err: bool = False,
(4)                 show_choices: bool = True,
(0)             ) -> t.Any:
(4)                 """Prompts a user for input.  This is a convenience function that can
(4)                 be used to prompt a user for input later.
(4)                 If the user aborts the input by sending an interrupt signal, this
(4)                 function will catch it and raise a :exc:`Abort` exception.
(4)                 :param text: the text to show for the prompt.
(4)                 :param default: the default value to use if no input happens.  If this
(20)                                 is not given it will prompt until it's aborted.
(4)                 :param hide_input: if this is set to true then the input value will
(23)                                    be hidden.
(4)                 :param confirmation_prompt: Prompt a second time to confirm the
(8)                     value. Can be set to a string instead of ``True`` to customize
(8)                     the message.
(4)                 :param type: the type to use to check the value against.
(4)                 :param value_proc: if this parameter is provided it's a function that
(23)                                    is invoked instead of the type conversion to
(23)                                    convert a value.
(4)                 :param prompt_suffix: a suffix that should be added to the prompt.
(4)                 :param show_default: shows or hides the default value in the prompt.
```

```
(4)         :param err: if set to true the file defaults to ``stderr`` instead of
(16)                    ``stdout``, the same as with echo.
(4)         :param show_choices: Show or hide choices if the passed type is a Choice.
(25)                             For example if type is a Choice of either day or week,
(25)                             show_choices is true and text is "Group by" then the
(25)                             prompt will be "Group by (day, week): ".
(4)         .. versionadded:: 8.0
(8)             ``confirmation_prompt`` can be a custom string.
(4)         .. versionadded:: 7.0
(8)             Added the ``show_choices`` parameter.
(4)         .. versionadded:: 6.0
(8)             Added unicode support for cmd.exe on Windows.
(4)         .. versionadded:: 4.0
(8)             Added the `err` parameter.
(4)         """
(4)         def prompt_func(text: str) -> str:
(8)             f = hidden_prompt_func if hide_input else visible_prompt_func
(8)             try:
(12)                echo(text.rstrip(" "), nl=False, err=err)
(12)                return f(" ")
(8)             except (KeyboardInterrupt, EOFError):
(12)                if hide_input:
(16)                    echo(None, err=err)
(12)                raise Abort() from None
(4)         if value_proc is None:
(8)             value_proc = convert_type(type, default)
(4)         prompt = _build_prompt(
(8)             text, prompt_suffix, show_default, default, show_choices, type
(4)         )
(4)         if confirmation_prompt:
(8)             if confirmation_prompt is True:
(12)                confirmation_prompt = _("Repeat for confirmation")
(8)             confirmation_prompt = _build_prompt(confirmation_prompt, prompt_suffix)
(4)         while True:
(8)             while True:
(12)                value = prompt_func(prompt)
(12)                if value:
(16)                    break
(12)                elif default is not None:
(16)                    value = default
(16)                    break
(8)             try:
(12)                result = value_proc(value)
(8)             except UsageError as e:
(12)                if hide_input:
(16)                    echo(_("Error: The value you entered was invalid."), err=err)
(12)                else:
(16)                    echo(_("Error: {e.message}").format(e=e), err=err)  # noqa: B306
(12)                continue
(8)             if not confirmation_prompt:
(12)                return result
(8)             while True:
(12)                value2 = prompt_func(confirmation_prompt)
(12)                is_empty = not value and not value2
(12)                if value2 or is_empty:
(16)                    break
(8)             if value == value2:
(12)                return result
(8)             echo(_("Error: The two entered values do not match."), err=err)
(0)     def confirm(
(4)         text: str,
(4)         default: t.Optional[bool] = False,
(4)         abort: bool = False,
(4)         prompt_suffix: str = ": ",
(4)         show_default: bool = True,
(4)         err: bool = False,
(0)     ) -> bool:
(4)         """Prompts for confirmation (yes/no question).
(4)         If the user aborts the input by sending a interrupt signal this
(4)         function will catch it and raise a :exc:`Abort` exception.
(4)         :param text: the question to ask.
(4)         :param default: The default value to use when no input is given. If
(8)             ``None``, repeat until input is given.
(4)         :param abort: if this is set to `True` a negative answer aborts the
(18)                      exception by raising :exc:`Abort`.
(4)         :param prompt_suffix: a suffix that should be added to the prompt.
(4)         :param show_default: shows or hides the default value in the prompt.
(4)         :param err: if set to true the file defaults to ``stderr`` instead of
(16)                    ``stdout``, the same as with echo.
(4)         .. versionchanged:: 8.0
(8)             Repeat until input is given if ``default`` is ``None``.
(4)         .. versionadded:: 4.0
(8)             Added the ``err`` parameter.
(4)         """
(4)         prompt = _build_prompt(
(8)             text,
(8)             prompt_suffix,
(8)             show_default,
(8)             "y/n" if default is None else ("Y/n" if default else "y/N"),
(4)         )
(4)         while True:
(8)             try:
(12)                echo(prompt.rstrip(" "), nl=False, err=err)
(12)                value = visible_prompt_func(" ").lower().strip()
(8)             except (KeyboardInterrupt, EOFError):
(12)                raise Abort() from None
(8)             if value in ("y", "yes"):
(12)                rv = True
(8)             elif value in ("n", "no"):
(12)                rv = False
(8)             elif default is not None and value == "":
(12)                rv = default
(8)             else:
(12)                echo(_("Error: invalid input"), err=err)
(12)                continue
(8)             break
(4)         if abort and not rv:
(8)             raise Abort()
(4)         return rv
(0)     def echo_via_pager(
(4)         text_or_generator: t.Union[t.Iterable[str], t.Callable[[],
t.Iterable[str]], str],
(4)         color: t.Optional[bool] = None,
(0)     ) -> None:
(4)         """This function takes a text and shows it via an environment specific
(4)         pager on stdout.
(4)         .. versionchanged:: 3.0
(7)            Added the `color` flag.
(4)         :param text_or_generator: the text to page, or alternatively, a
(30)                                  generator emitting the text to page.
(4)         :param color: controls if the pager supports ANSI colors or not.  The
(18)                      default is autodetection.
(4)         """
(4)         color = resolve_color_default(color)
(4)         if inspect.isgeneratorfunction(text_or_generator):
(8)             i = t.cast(t.Callable[[], t.Iterable[str]], text_or_generator)()
(4)         elif isinstance(text_or_generator, str):
(8)             i = [text_or_generator]
(4)         else:
(8)             i = iter(t.cast(t.Iterable[str], text_or_generator))
(4)         text_generator = (el if isinstance(el, str) else str(el) for el in i)
(4)         from ._termui_impl import pager
(4)         return pager(itertools.chain(text_generator, "\n"), color)
(0)     def progressbar(
(4)         iterable: t.Optional[t.Iterable[V]] = None,
(4)                 length: t.Optional[int] = None,
(4)                 label: t.Optional[str] = None,
(4)                 show_eta: bool = True,
(4)                 show_percent: t.Optional[bool] = None,
(4)                 show_pos: bool = False,
(4)                 item_show_func: t.Optional[t.Callable[[t.Optional[V]], t.Optional[str]]] = None,
(4)                 fill_char: str = "#",
(4)                 empty_char: str = "-",
(4)                 bar_template: str = "%(label)s  [%(bar)s]  %(info)s",
(4)                 info_sep: str = "  ",
(4)                 width: int = 36,
(4)                 file: t.Optional[t.TextIO] = None,
(4)                 color: t.Optional[bool] = None,
(4)                 update_min_steps: int = 1,
(0)             ) -> "ProgressBar[V]":
(4)                 """This function creates an iterable context manager that can be used
(4)                 to iterate over something while showing a progress bar.  It will
(4)                 either iterate over the `iterable` or `length` items (that are counted
(4)                 up).  While iteration happens, this function will print a rendered
(4)                 progress bar to the given `file` (defaults to stdout) and will attempt
(4)                 to calculate remaining time and more.  By default, this progress bar
(4)                 will not be rendered if the file is not a terminal.
(4)                 The context manager creates the progress bar.  When the context
(4)                 manager is entered the progress bar is already created.  With every
(4)                 iteration over the progress bar, the iterable passed to the bar is
(4)                 advanced and the bar is updated.  When the context manager exits,
(4)                 a newline is printed and the progress bar is finalized on screen.
(4)                 Note: The progress bar is currently designed for use cases where the
(4)                 total progress can be expected to take at least several seconds.
(4)                 Because of this, the ProgressBar class object won't display
(4)                 progress that is considered too fast, and progress where the time
(4)                 between steps is less than a second.
(4)                 No printing must happen or the progress bar will be unintentionally
(4)                 destroyed.
(4)                 Example usage::
(8)                     with progressbar(items) as bar:
(12)                        for item in bar:
(16)                            do_something_with(item)
(4)                 Alternatively, if no iterable is specified, one can manually update the
(4)                 progress bar through the `update()` method instead of directly
(4)                 iterating over the progress bar.  The update method accepts the number
(4)                 of steps to increment the bar with::
(8)                     with progressbar(length=chunks.total_bytes) as bar:
(12)                        for chunk in chunks:
(16)                            process_chunk(chunk)
(16)                            bar.update(chunks.bytes)
(4)                 The ``update()`` method also takes an optional value specifying the
(4)                 ``current_item`` at the new position. This is useful when used
(4)                 together with ``item_show_func`` to customize the output for each
(4)                 manual step::
(8)                     with click.progressbar(
(12)                        length=total_size,
(12)                        label='Unzipping archive',
(12)                        item_show_func=lambda a: a.filename
(8)                     ) as bar:
(12)                        for archive in zip_file:
(16)                            archive.extract()
(16)                            bar.update(archive.size, archive)
(4)                 :param iterable: an iterable to iterate over.  If not provided the length
(21)                                  is required.
(4)                 :param length: the number of items to iterate over.  By default the
(19)                               progressbar will attempt to ask the iterator about its
(19)                               length, which might or might not work.  If an iterable is
(19)                               also provided this parameter can be used to override the
(19)                               length.  If an iterable is not provided the progress bar
(19)                               will iterate over a range of that length.
(4)                 :param label: the label to show next to the progress bar.
(4)                 :param show_eta: enables or disables the estimated time display.  This is
```

```
(21)                     automatically disabled if the length cannot be
(21)                     determined.
(4)     :param show_percent: enables or disables the percentage display.  The
(25)                         default is `True` if the iterable has a length or
(25)                         `False` if not.
(4)     :param show_pos: enables or disables the absolute position display.  The
(21)                     default is `False`.
(4)     :param item_show_func: A function called with the current item which
(8)         can return a string to show next to the progress bar. If the
(8)         function returns ``None`` nothing is shown. The current item can
(8)         be ``None``, such as when entering and exiting the bar.
(4)     :param fill_char: the character to use to show the filled part of the
(22)                      progress bar.
(4)     :param empty_char: the character to use to show the non-filled part of
(23)                       the progress bar.
(4)     :param bar_template: the format string to use as template for the bar.
(25)                         The parameters in it are ``label`` for the label,
(25)                         ``bar`` for the progress bar and ``info`` for the
(25)                         info section.
(4)     :param info_sep: the separator between multiple info items (eta etc.)
(4)     :param width: the width of the progress bar in characters, 0 means full
(18)                  terminal width
(4)     :param file: The file to write to. If this is not a terminal then
(8)         only the label is printed.
(4)     :param color: controls if the terminal supports ANSI colors or not.  The
(18)                  default is autodetection.  This is only needed if ANSI
(18)                  codes are included anywhere in the progress bar output
(18)                  which is not the case by default.
(4)     :param update_min_steps: Render only when this many updates have
(8)         completed. This allows tuning for very fast iterators.
(4)     .. versionchanged:: 8.0
(8)         Output is shown even if execution time is less than 0.5 seconds.
(4)     .. versionchanged:: 8.0
(8)         ``item_show_func`` shows the current item, not the previous one.
(4)     .. versionchanged:: 8.0
(8)         Labels are echoed if the output is not a TTY. Reverts a change
(8)         in 7.0 that removed all output.
(4)     .. versionadded:: 8.0
(7)        Added the ``update_min_steps`` parameter.
(4)     .. versionchanged:: 4.0
(8)         Added the ``color`` parameter. Added the ``update`` method to
(8)         the object.
(4)     .. versionadded:: 2.0
(4)     """
(4)     from ._termui_impl import ProgressBar
(4)     color = resolve_color_default(color)
(4)     return ProgressBar(
(8)         iterable=iterable,
(8)         length=length,
(8)         show_eta=show_eta,
(8)         show_percent=show_percent,
(8)         show_pos=show_pos,
(8)         item_show_func=item_show_func,
(8)         fill_char=fill_char,
(8)         empty_char=empty_char,
(8)         bar_template=bar_template,
(8)         info_sep=info_sep,
(8)         file=file,
(8)         label=label,
(8)         width=width,
(8)         color=color,
(8)         update_min_steps=update_min_steps,
(4)     )
(0) def clear() -> None:
(4)     """Clears the terminal screen.  This will have the effect of clearing
(4)     the whole visible space of the terminal and moving the cursor to the
(4)     top left.  This does not do anything if not connected to a terminal.
(4)     .. versionadded:: 2.0
(4)     """
```

```
(4)                 if not isatty(sys.stdout):
(8)                     return
(4)                 echo("\033[2J\033[1;1H", nl=False)
(0)             def _interpret_color(
(4)                 color: t.Union[int, t.Tuple[int, int, int], str], offset: int = 0
(0)             ) -> str:
(4)                 if isinstance(color, int):
(8)                     return f"{38 + offset};5;{color:d}"
(4)                 if isinstance(color, (tuple, list)):
(8)                     r, g, b = color
(8)                     return f"{38 + offset};2;{r:d};{g:d};{b:d}"
(4)                 return str(_ansi_colors[color] + offset)
(0)             def style(
(4)                 text: t.Any,
(4)                 fg: t.Optional[t.Union[int, t.Tuple[int, int, int], str]] = None,
(4)                 bg: t.Optional[t.Union[int, t.Tuple[int, int, int], str]] = None,
(4)                 bold: t.Optional[bool] = None,
(4)                 dim: t.Optional[bool] = None,
(4)                 underline: t.Optional[bool] = None,
(4)                 overline: t.Optional[bool] = None,
(4)                 italic: t.Optional[bool] = None,
(4)                 blink: t.Optional[bool] = None,
(4)                 reverse: t.Optional[bool] = None,
(4)                 strikethrough: t.Optional[bool] = None,
(4)                 reset: bool = True,
(0)             ) -> str:
(4)                 """Styles a text with ANSI styles and returns the new string.  By
(4)                 default the styling is self contained which means that at the end
(4)                 of the string a reset code is issued.  This can be prevented by
(4)                 passing ``reset=False``.
(4)                 Examples::
(8)                     click.echo(click.style('Hello World!', fg='green'))
(8)                     click.echo(click.style('ATTENTION!', blink=True))
(8)                     click.echo(click.style('Some things', reverse=True, fg='cyan'))
(8)                     click.echo(click.style('More colors', fg=(255, 12, 128), bg=117))
(4)                 Supported color names:
(4)                 * ``black`` (might be a gray)
(4)                 * ``red``
(4)                 * ``green``
(4)                 * ``yellow`` (might be an orange)
(4)                 * ``blue``
(4)                 * ``magenta``
(4)                 * ``cyan``
(4)                 * ``white`` (might be light gray)
(4)                 * ``bright_black``
(4)                 * ``bright_red``
(4)                 * ``bright_green``
(4)                 * ``bright_yellow``
(4)                 * ``bright_blue``
(4)                 * ``bright_magenta``
(4)                 * ``bright_cyan``
(4)                 * ``bright_white``
(4)                 * ``reset`` (reset the color code only)
(4)                 If the terminal supports it, color may also be specified as:
(4)                 -   An integer in the interval [0, 255]. The terminal must support
(8)                     8-bit/256-color mode.
(4)                 -   An RGB tuple of three integers in [0, 255]. The terminal must
(8)                     support 24-bit/true-color mode.
(4)                 See https://en.wikipedia.org/wiki/ANSI_color and
(4)                 https://gist.github.com/XVilka/8346728 for more information.
(4)                 :param text: the string to style with ansi codes.
(4)                 :param fg: if provided this will become the foreground color.
(4)                 :param bg: if provided this will become the background color.
(4)                 :param bold: if provided this will enable or disable bold mode.
(4)                 :param dim: if provided this will enable or disable dim mode.  This is
(16)                                badly supported.
(4)                 :param underline: if provided this will enable or disable underline.
(4)                 :param overline: if provided this will enable or disable overline.
(4)                 :param italic: if provided this will enable or disable italic.
```

```
    :param blink: if provided this will enable or disable blinking.
    :param reverse: if provided this will enable or disable inverse
                    rendering (foreground becomes background and the
                    other way round).
    :param strikethrough: if provided this will enable or disable
        striking through text.
    :param reset: by default a reset-all code is added at the end of the
                  string which means that styles do not carry over.  This
                  can be disabled to compose styles.
    .. versionchanged:: 8.0
        A non-string ``message`` is converted to a string.
    .. versionchanged:: 8.0
       Added support for 256 and RGB color codes.
    .. versionchanged:: 8.0
        Added the ``strikethrough``, ``italic``, and ``overline``
        parameters.
    .. versionchanged:: 7.0
        Added support for bright colors.
    .. versionadded:: 2.0
    """
    if not isinstance(text, str):
        text = str(text)
    bits = []
    if fg:
        try:
            bits.append(f"\033[{_interpret_color(fg)}m")
        except KeyError:
            raise TypeError(f"Unknown color {fg!r}") from None
    if bg:
        try:
            bits.append(f"\033[{_interpret_color(bg, 10)}m")
        except KeyError:
            raise TypeError(f"Unknown color {bg!r}") from None
    if bold is not None:
        bits.append(f"\033[{1 if bold else 22}m")
    if dim is not None:
        bits.append(f"\033[{2 if dim else 22}m")
    if underline is not None:
        bits.append(f"\033[{4 if underline else 24}m")
    if overline is not None:
        bits.append(f"\033[{53 if overline else 55}m")
    if italic is not None:
        bits.append(f"\033[{3 if italic else 23}m")
    if blink is not None:
        bits.append(f"\033[{5 if blink else 25}m")
    if reverse is not None:
        bits.append(f"\033[{7 if reverse else 27}m")
    if strikethrough is not None:
        bits.append(f"\033[{9 if strikethrough else 29}m")
    bits.append(text)
    if reset:
        bits.append(_ansi_reset_all)
    return "".join(bits)
def unstyle(text: str) -> str:
    """Removes ANSI styling information from a string.  Usually it's not
    necessary to use this function as Click's echo function will
    automatically remove styling if necessary.
    .. versionadded:: 2.0
    :param text: the text to remove style information from.
    """
    return strip_ansi(text)
def secho(
    message: t.Optional[t.Any] = None,
    file: t.Optional[t.IO[t.AnyStr]] = None,
    nl: bool = True,
    err: bool = False,
    color: t.Optional[bool] = None,
    **styles: t.Any,
) -> None:
```

```
(4)             """This function combines :func:`echo` and :func:`style` into one
(4)             call.  As such the following two calls are the same::
(8)                 click.secho('Hello World!', fg='green')
(8)                 click.echo(click.style('Hello World!', fg='green'))
(4)             All keyword arguments are forwarded to the underlying functions
(4)             depending on which one they go with.
(4)             Non-string types will be converted to :class:`str`. However,
(4)             :class:`bytes` are passed directly to :meth:`echo` without applying
(4)             style. If you want to style bytes that represent text, call
(4)             :meth:`bytes.decode` first.
(4)             .. versionchanged:: 8.0
(8)                 A non-string ``message`` is converted to a string. Bytes are
(8)                 passed through without style applied.
(4)             .. versionadded:: 2.0
(4)             """
(4)             if message is not None and not isinstance(message, (bytes, bytearray)):
(8)                 message = style(message, **styles)
(4)             return echo(message, file=file, nl=nl, err=err, color=color)
(0)         def edit(
(4)             text: t.Optional[t.AnyStr] = None,
(4)             editor: t.Optional[str] = None,
(4)             env: t.Optional[t.Mapping[str, str]] = None,
(4)             require_save: bool = True,
(4)             extension: str = ".txt",
(4)             filename: t.Optional[str] = None,
(0)         ) -> t.Optional[t.AnyStr]:
(4)             r"""Edits the given text in the defined editor.  If an editor is given
(4)             (should be the full path to the executable but the regular operating
(4)             system search path is used for finding the executable) it overrides
(4)             the detected editor.  Optionally, some environment variables can be
(4)             used.  If the editor is closed without changes, `None` is returned.  In
(4)             case a file is edited directly the return value is always `None` and
(4)             `require_save` and `extension` are ignored.
(4)             If the editor cannot be opened a :exc:`UsageError` is raised.
(4)             Note for Windows: to simplify cross-platform usage, the newlines are
(4)             automatically converted from POSIX to Windows and vice versa.  As such,
(4)             the message here will have ``\n`` as newline markers.
(4)             :param text: the text to edit.
(4)             :param editor: optionally the editor to use.  Defaults to automatic
(19)                           detection.
(4)             :param env: environment variables to forward to the editor.
(4)             :param require_save: if this is true, then not saving in the editor
(25)                                 will make the return value become `None`.
(4)             :param extension: the extension to tell the editor about.  This defaults
(22)                              to `.txt` but changing this might change syntax
(22)                              highlighting.
(4)             :param filename: if provided it will edit this file instead of the
(21)                             provided text contents.  It will not use a temporary
(21)                             file as an indirection in that case.
(4)             """
(4)             from ._termui_impl import Editor
(4)             ed = Editor(editor=editor, env=env, require_save=require_save,
extension=extension)
(4)             if filename is None:
(8)                 return ed.edit(text)
(4)             ed.edit_file(filename)
(4)             return None
(0)         def launch(url: str, wait: bool = False, locate: bool = False) -> int:
(4)             """This function launches the given URL (or filename) in the default
(4)             viewer application for this file type.  If this is an executable, it
(4)             might launch the executable in a new session.  The return value is
(4)             the exit code of the launched application.  Usually, ``0`` indicates
(4)             success.
(4)             Examples::
(8)                 click.launch('https://click.palletsprojects.com/')
(8)                 click.launch('/my/downloaded/file', locate=True)
(4)             .. versionadded:: 2.0
(4)             :param url: URL or filename of the thing to launch.
(4)             :param wait: Wait for the program to exit before returning. This
```

```
569: (8)                        only works if the launched program blocks. In particular,
570: (8)                        ``xdg-open`` on Linux does not block.
571: (4)                    :param locate: if this is set to `True` then instead of launching the
572: (19)                                   application associated with the URL it will attempt to
573: (19)                                   launch a file manager with the file located.  This
574: (19)                                   might have weird effects if the URL does not point to
575: (19)                                   the filesystem.
576: (4)                    """
577: (4)                    from ._termui_impl import open_url
578: (4)                    return open_url(url, wait=wait, locate=locate)
579: (0)                _getchar: t.Optional[t.Callable[[bool], str]] = None
580: (0)                def getchar(echo: bool = False) -> str:
581: (4)                    """Fetches a single character from the terminal and returns it.  This
582: (4)                    will always return a unicode character and under certain rare
583: (4)                    circumstances this might return more than one character.  The
584: (4)                    situations which more than one character is returned is when for
585: (4)                    whatever reason multiple characters end up in the terminal buffer or
586: (4)                    standard input was not actually a terminal.
587: (4)                    Note that this will always read from the terminal, even if something
588: (4)                    is piped into the standard input.
589: (4)                    Note for Windows: in rare cases when typing non-ASCII characters, this
590: (4)                    function might wait for a second character and then return both at once.
591: (4)                    This is because certain Unicode characters look like special-key markers.
592: (4)                    .. versionadded:: 2.0
593: (4)                    :param echo: if set to `True`, the character read will also show up on
594: (17)                                 the terminal.  The default is to not show it.
595: (4)                    """
596: (4)                    global _getchar
597: (4)                    if _getchar is None:
598: (8)                        from ._termui_impl import getchar as f
599: (8)                        _getchar = f
600: (4)                    return _getchar(echo)
601: (0)                def raw_terminal() -> t.ContextManager[int]:
602: (4)                    from ._termui_impl import raw_terminal as f
603: (4)                    return f()
604: (0)                def pause(info: t.Optional[str] = None, err: bool = False) -> None:
605: (4)                    """This command stops execution and waits for the user to press any
606: (4)                    key to continue.  This is similar to the Windows batch "pause"
607: (4)                    command.  If the program is not run through a terminal, this command
608: (4)                    will instead do nothing.
609: (4)                    .. versionadded:: 2.0
610: (4)                    .. versionadded:: 4.0
611: (7)                       Added the `err` parameter.
612: (4)                    :param info: The message to print before pausing. Defaults to
613: (8)                        ``"Press any key to continue..."``.
614: (4)                    :param err: if set to message goes to ``stderr`` instead of
615: (16)                               ``stdout``, the same as with echo.
616: (4)                    """
617: (4)                    if not isatty(sys.stdin) or not isatty(sys.stdout):
618: (8)                        return
619: (4)                    if info is None:
620: (8)                        info = _("Press any key to continue...")
621: (4)                    try:
622: (8)                        if info:
623: (12)                           echo(info, nl=False, err=err)
624: (8)                        try:
625: (12)                           getchar()
626: (8)                        except (KeyboardInterrupt, EOFError):
627: (12)                           pass
628: (4)                    finally:
629: (8)                        if info:
630: (12)                           echo(err=err)

----------------------------------------

File 14 - testing.py:

1: (0)                  import contextlib
2: (0)                  import io
```

```
(0)             import os
(0)             import shlex
(0)             import shutil
(0)             import sys
(0)             import tempfile
(0)             import typing as t
(0)             from types import TracebackType
(0)             from . import formatting
(0)             from . import termui
(0)             from . import utils
(0)             from ._compat import _find_binary_reader
(0)             if t.TYPE_CHECKING:
(4)                 from .core import BaseCommand
(0)             class EchoingStdin:
(4)                 def __init__(self, input: t.BinaryIO, output: t.BinaryIO) -> None:
(8)                     self._input = input
(8)                     self._output = output
(8)                     self._paused = False
(4)                 def __getattr__(self, x: str) -> t.Any:
(8)                     return getattr(self._input, x)
(4)                 def _echo(self, rv: bytes) -> bytes:
(8)                     if not self._paused:
(12)                        self._output.write(rv)
(8)                     return rv
(4)                 def read(self, n: int = -1) -> bytes:
(8)                     return self._echo(self._input.read(n))
(4)                 def read1(self, n: int = -1) -> bytes:
(8)                     return self._echo(self._input.read1(n))  # type: ignore
(4)                 def readline(self, n: int = -1) -> bytes:
(8)                     return self._echo(self._input.readline(n))
(4)                 def readlines(self) -> t.List[bytes]:
(8)                     return [self._echo(x) for x in self._input.readlines()]
(4)                 def __iter__(self) -> t.Iterator[bytes]:
(8)                     return iter(self._echo(x) for x in self._input)
(4)                 def __repr__(self) -> str:
(8)                     return repr(self._input)
(0)             @contextlib.contextmanager
(0)             def _pause_echo(stream: t.Optional[EchoingStdin]) -> t.Iterator[None]:
(4)                 if stream is None:
(8)                     yield
(4)                 else:
(8)                     stream._paused = True
(8)                     yield
(8)                     stream._paused = False
(0)             class _NamedTextIOWrapper(io.TextIOWrapper):
(4)                 def __init__(
(8)                     self, buffer: t.BinaryIO, name: str, mode: str, **kwargs: t.Any
(4)                 ) -> None:
(8)                     super().__init__(buffer, **kwargs)
(8)                     self._name = name
(8)                     self._mode = mode
(4)                 @property
(4)                 def name(self) -> str:
(8)                     return self._name
(4)                 @property
(4)                 def mode(self) -> str:
(8)                     return self._mode
(0)             def make_input_stream(
(4)                 input: t.Optional[t.Union[str, bytes, t.IO[t.Any]]], charset: str
(0)             ) -> t.BinaryIO:
(4)                 if hasattr(input, "read"):
(8)                     rv = _find_binary_reader(t.cast(t.IO[t.Any], input))
(8)                     if rv is not None:
(12)                        return rv
(8)                     raise TypeError("Could not find binary reader for input stream.")
(4)                 if input is None:
(8)                     input = b""
(4)                 elif isinstance(input, str):
(8)                     input = input.encode(charset)
```

```
    return io.BytesIO(input)
class Result:
    """Holds the captured result of an invoked CLI script."""
    def __init__(
        self,
        runner: "CliRunner",
        stdout_bytes: bytes,
        stderr_bytes: t.Optional[bytes],
        return_value: t.Any,
        exit_code: int,
        exception: t.Optional[BaseException],
        exc_info: t.Optional[
            t.Tuple[t.Type[BaseException], BaseException, TracebackType]
        ] = None,
    ):
        self.runner = runner
        self.stdout_bytes = stdout_bytes
        self.stderr_bytes = stderr_bytes
        self.return_value = return_value
        self.exit_code = exit_code
        self.exception = exception
        self.exc_info = exc_info
    @property
    def output(self) -> str:
        """The (standard) output as unicode string."""
        return self.stdout
    @property
    def stdout(self) -> str:
        """The standard output as unicode string."""
        return self.stdout_bytes.decode(self.runner.charset, "replace").replace(
            "\r\n", "\n"
        )
    @property
    def stderr(self) -> str:
        """The standard error as unicode string."""
        if self.stderr_bytes is None:
            raise ValueError("stderr not separately captured")
        return self.stderr_bytes.decode(self.runner.charset, "replace").replace(
            "\r\n", "\n"
        )
    def __repr__(self) -> str:
        exc_str = repr(self.exception) if self.exception else "okay"
        return f"<{type(self).__name__} {exc_str}>"
class CliRunner:
    """The CLI runner provides functionality to invoke a Click command line
    script for unittesting purposes in a isolated environment.  This only
    works in single-threaded systems without any concurrency as it changes the
    global interpreter state.
    :param charset: the character set for the input and output data.
    :param env: a dictionary with environment variables for overriding.
    :param echo_stdin: if this is set to `True`, then reading from stdin writes
                       to stdout.  This is useful for showing examples in
                       some circumstances.  Note that regular prompts
                       will automatically echo the input.
    :param mix_stderr: if this is set to `False`, then stdout and stderr are
                       preserved as independent streams.  This is useful for
                       Unix-philosophy apps that have predictable stdout and
                       noisy stderr, such that each may be measured
                       independently
    """
    def __init__(
        self,
        charset: str = "utf-8",
        env: t.Optional[t.Mapping[str, t.Optional[str]]] = None,
        echo_stdin: bool = False,
        mix_stderr: bool = True,
    ) -> None:
        self.charset = charset
        self.env: t.Mapping[str, t.Optional[str]] = env or {}
        self.echo_stdin = echo_stdin
        self.mix_stderr = mix_stderr
    def get_default_prog_name(self, cli: "BaseCommand") -> str:
        """Given a command object it will return the default program name
        for it.  The default is the `name` attribute or ``"root"`` if not
        set.
        """
        return cli.name or "root"
    def make_env(
        self, overrides: t.Optional[t.Mapping[str, t.Optional[str]]] = None
    ) -> t.Mapping[str, t.Optional[str]]:
        """Returns the environment overrides for invoking a script."""
        rv = dict(self.env)
        if overrides:
            rv.update(overrides)
        return rv
    @contextlib.contextmanager
    def isolation(
        self,
        input: t.Optional[t.Union[str, bytes, t.IO[t.Any]]] = None,
        env: t.Optional[t.Mapping[str, t.Optional[str]]] = None,
        color: bool = False,
    ) -> t.Iterator[t.Tuple[io.BytesIO, t.Optional[io.BytesIO]]]:
        """A context manager that sets up the isolation for invoking of a
        command line tool.  This sets up stdin with the given input data
        and `os.environ` with the overrides from the given dictionary.
        This also rebinds some internals in Click to be mocked (like the
        prompt functionality).
        This is automatically done in the :meth:`invoke` method.
        :param input: the input stream to put into sys.stdin.
        :param env: the environment overrides as dictionary.
        :param color: whether the output should contain color codes. The
                      application can still override this explicitly.
        .. versionchanged:: 8.0
            ``stderr`` is opened with ``errors="backslashreplace"``
            instead of the default ``"strict"``.
        .. versionchanged:: 4.0
            Added the ``color`` parameter.
        """
        bytes_input = make_input_stream(input, self.charset)
        echo_input = None
        old_stdin = sys.stdin
        old_stdout = sys.stdout
        old_stderr = sys.stderr
        old_forced_width = formatting.FORCED_WIDTH
        formatting.FORCED_WIDTH = 80
        env = self.make_env(env)
        bytes_output = io.BytesIO()
        if self.echo_stdin:
            bytes_input = echo_input = t.cast(
                t.BinaryIO, EchoingStdin(bytes_input, bytes_output)
            )
        sys.stdin = text_input = _NamedTextIOWrapper(
            bytes_input, encoding=self.charset, name="<stdin>", mode="r"
        )
        if self.echo_stdin:
            text_input._CHUNK_SIZE = 1  # type: ignore
        sys.stdout = _NamedTextIOWrapper(
            bytes_output, encoding=self.charset, name="<stdout>", mode="w"
        )
        bytes_error = None
        if self.mix_stderr:
            sys.stderr = sys.stdout
        else:
            bytes_error = io.BytesIO()
            sys.stderr = _NamedTextIOWrapper(
(16)                        bytes_error,
(16)                        encoding=self.charset,
(16)                        name="<stderr>",
(16)                        mode="w",
(16)                        errors="backslashreplace",
(12)                    )
(8)                 @_pause_echo(echo_input)  # type: ignore
(8)                 def visible_input(prompt: t.Optional[str] = None) -> str:
(12)                    sys.stdout.write(prompt or "")
(12)                    val = text_input.readline().rstrip("\r\n")
(12)                    sys.stdout.write(f"{val}\n")
(12)                    sys.stdout.flush()
(12)                    return val
(8)                 @_pause_echo(echo_input)  # type: ignore
(8)                 def hidden_input(prompt: t.Optional[str] = None) -> str:
(12)                    sys.stdout.write(f"{prompt or ''}\n")
(12)                    sys.stdout.flush()
(12)                    return text_input.readline().rstrip("\r\n")
(8)                 @_pause_echo(echo_input)  # type: ignore
(8)                 def _getchar(echo: bool) -> str:
(12)                    char = sys.stdin.read(1)
(12)                    if echo:
(16)                        sys.stdout.write(char)
(12)                    sys.stdout.flush()
(12)                    return char
(8)                 default_color = color
(8)                 def should_strip_ansi(
(12)                    stream: t.Optional[t.IO[t.Any]] = None, color: t.Optional[bool] = None
(8)                 ) -> bool:
(12)                    if color is None:
(16)                        return not default_color
(12)                    return not color
(8)                 old_visible_prompt_func = termui.visible_prompt_func
(8)                 old_hidden_prompt_func = termui.hidden_prompt_func
(8)                 old__getchar_func = termui._getchar
(8)                 old_should_strip_ansi = utils.should_strip_ansi  # type: ignore
(8)                 termui.visible_prompt_func = visible_input
(8)                 termui.hidden_prompt_func = hidden_input
(8)                 termui._getchar = _getchar
(8)                 utils.should_strip_ansi = should_strip_ansi  # type: ignore
(8)                 old_env = {}
(8)                 try:
(12)                    for key, value in env.items():
(16)                        old_env[key] = os.environ.get(key)
(16)                        if value is None:
(20)                            try:
(24)                                del os.environ[key]
(20)                            except Exception:
(24)                                pass
(16)                        else:
(20)                            os.environ[key] = value
(12)                    yield (bytes_output, bytes_error)
(8)                 finally:
(12)                    for key, value in old_env.items():
(16)                        if value is None:
(20)                            try:
(24)                                del os.environ[key]
(20)                            except Exception:
(24)                                pass
(16)                        else:
(20)                            os.environ[key] = value
(12)                    sys.stdout = old_stdout
(12)                    sys.stderr = old_stderr
(12)                    sys.stdin = old_stdin
(12)                    termui.visible_prompt_func = old_visible_prompt_func
(12)                    termui.hidden_prompt_func = old_hidden_prompt_func
(12)                    termui._getchar = old__getchar_func
(12)                    utils.should_strip_ansi = old_should_strip_ansi  # type: ignore
```

```
(12)              formatting.FORCED_WIDTH = old_forced_width
(4)       def invoke(
(8)           self,
(8)           cli: "BaseCommand",
(8)           args: t.Optional[t.Union[str, t.Sequence[str]]] = None,
(8)           input: t.Optional[t.Union[str, bytes, t.IO[t.Any]]] = None,
(8)           env: t.Optional[t.Mapping[str, t.Optional[str]]] = None,
(8)           catch_exceptions: bool = True,
(8)           color: bool = False,
(8)           **extra: t.Any,
(4)       ) -> Result:
(8)           """Invokes a command in an isolated environment.  The arguments are
(8)           forwarded directly to the command line script, the `extra` keyword
(8)           arguments are passed to the :meth:`~clickpkg.Command.main` function of
(8)           the command.
(8)           This returns a :class:`Result` object.
(8)           :param cli: the command to invoke
(8)           :param args: the arguments to invoke. It may be given as an iterable
(21)                       or a string. When given as string it will be interpreted
(21)                       as a Unix shell command. More details at
(21)                       :func:`shlex.split`.
(8)           :param input: the input data for `sys.stdin`.
(8)           :param env: the environment overrides.
(8)           :param catch_exceptions: Whether to catch any other exceptions than
(33)                                  ``SystemExit``.
(8)           :param extra: the keyword arguments to pass to :meth:`main`.
(8)           :param color: whether the output should contain color codes. The
(22)                        application can still override this explicitly.
(8)           .. versionchanged:: 8.0
(12)              The result object has the ``return_value`` attribute with
(12)              the value returned from the invoked command.
(8)           .. versionchanged:: 4.0
(12)              Added the ``color`` parameter.
(8)           .. versionchanged:: 3.0
(12)              Added the ``catch_exceptions`` parameter.
(8)           .. versionchanged:: 3.0
(12)              The result object has the ``exc_info`` attribute with the
(12)              traceback if available.
(8)           """
(8)           exc_info = None
(8)           with self.isolation(input=input, env=env, color=color) as outstreams:
(12)              return_value = None
(12)              exception: t.Optional[BaseException] = None
(12)              exit_code = 0
(12)              if isinstance(args, str):
(16)                  args = shlex.split(args)
(12)              try:
(16)                  prog_name = extra.pop("prog_name")
(12)              except KeyError:
(16)                  prog_name = self.get_default_prog_name(cli)
(12)              try:
(16)                  return_value = cli.main(args=args or (), prog_name=prog_name,
**extra)
(12)              except SystemExit as e:
(16)                  exc_info = sys.exc_info()
(16)                  e_code = t.cast(t.Optional[t.Union[int, t.Any]], e.code)
(16)                  if e_code is None:
(20)                      e_code = 0
(16)                  if e_code != 0:
(20)                      exception = e
(16)                  if not isinstance(e_code, int):
(20)                      sys.stdout.write(str(e_code))
(20)                      sys.stdout.write("\n")
(20)                      e_code = 1
(16)                  exit_code = e_code
(12)              except Exception as e:
(16)                  if not catch_exceptions:
(20)                      raise
(16)                  exception = e
```

```
(16)                    exit_code = 1
(16)                    exc_info = sys.exc_info()
(12)                finally:
(16)                    sys.stdout.flush()
(16)                    stdout = outstreams[0].getvalue()
(16)                    if self.mix_stderr:
(20)                        stderr = None
(16)                    else:
(20)                        stderr = outstreams[1].getvalue()  # type: ignore
(8)             return Result(
(12)                runner=self,
(12)                stdout_bytes=stdout,
(12)                stderr_bytes=stderr,
(12)                return_value=return_value,
(12)                exit_code=exit_code,
(12)                exception=exception,
(12)                exc_info=exc_info,  # type: ignore
(8)             )
(4)         @contextlib.contextmanager
(4)         def isolated_filesystem(
(8)             self, temp_dir: t.Optional[t.Union[str, "os.PathLike[str]"]] = None
(4)         ) -> t.Iterator[str]:
(8)             """A context manager that creates a temporary directory and
(8)             changes the current working directory to it. This isolates tests
(8)             that affect the contents of the CWD to prevent them from
(8)             interfering with each other.
(8)             :param temp_dir: Create the temporary directory under this
(12)                directory. If given, the created directory is not removed
(12)                when exiting.
(8)             .. versionchanged:: 8.0
(12)                Added the ``temp_dir`` parameter.
(8)             """
(8)             cwd = os.getcwd()
(8)             dt = tempfile.mkdtemp(dir=temp_dir)
(8)             os.chdir(dt)
(8)             try:
(12)                yield dt
(8)             finally:
(12)                os.chdir(cwd)
(12)                if temp_dir is None:
(16)                    try:
(20)                        shutil.rmtree(dt)
(16)                    except OSError:  # noqa: B014
(20)                        pass
```

File 15 - types.py:

```
(0)     import os
(0)     import stat
(0)     import sys
(0)     import typing as t
(0)     from datetime import datetime
(0)     from gettext import gettext as _
(0)     from gettext import ngettext
(0)     from ._compat import _get_argv_encoding
(0)     from ._compat import open_stream
(0)     from .exceptions import BadParameter
(0)     from .utils import format_filename
(0)     from .utils import LazyFile
(0)     from .utils import safecall
(0)     if t.TYPE_CHECKING:
(4)         import typing_extensions as te
(4)         from .core import Context
(4)         from .core import Parameter
(4)         from .shell_completion import CompletionItem
(0)     class ParamType:
(4)         """Represents the type of a parameter. Validates and converts values
```

```
    from the command line or Python into the correct type.
    To implement a custom type, subclass and implement at least the
    following:
    -   The :attr:`name` class attribute must be set.
    -   Calling an instance of the type with ``None`` must return
        ``None``. This is already implemented by default.
    -   :meth:`convert` must convert string values to the correct type.
    -   :meth:`convert` must accept values that are already the correct
        type.
    -   It must be able to convert a value if the ``ctx`` and ``param``
        arguments are ``None``. This can occur when converting prompt
        input.
    """
    is_composite: t.ClassVar[bool] = False
    arity: t.ClassVar[int] = 1
    name: str
    envvar_list_splitter: t.ClassVar[t.Optional[str]] = None
    def to_info_dict(self) -> t.Dict[str, t.Any]:
        """Gather information that could be useful for a tool generating
        user-facing documentation.
        Use :meth:`click.Context.to_info_dict` to traverse the entire
        CLI structure.
        .. versionadded:: 8.0
        """
        param_type = type(self).__name__.partition("ParamType")[0]
        param_type = param_type.partition("ParameterType")[0]
        if hasattr(self, "name"):
            name = self.name
        else:
            name = param_type
        return {"param_type": param_type, "name": name}
    def __call__(
        self,
        value: t.Any,
        param: t.Optional["Parameter"] = None,
        ctx: t.Optional["Context"] = None,
    ) -> t.Any:
        if value is not None:
            return self.convert(value, param, ctx)
    def get_metavar(self, param: "Parameter") -> t.Optional[str]:
        """Returns the metavar default for this param if it provides one."""
    def get_missing_message(self, param: "Parameter") -> t.Optional[str]:
        """Optionally might return extra information about a missing
        parameter.
        .. versionadded:: 2.0
        """
    def convert(
        self, value: t.Any, param: t.Optional["Parameter"], ctx: t.Optional["Context"]
    ) -> t.Any:
        """Convert the value to the correct type. This is not called if
        the value is ``None`` (the missing value).
        This must accept string values from the command line, as well as
        values that are already the correct type. It may also convert
        other compatible types.
        The ``param`` and ``ctx`` arguments may be ``None`` in certain
        situations, such as when converting prompt input.
        If the value cannot be converted, call :meth:`fail` with a
        descriptive message.
        :param value: The value to convert.
        :param param: The parameter that is using this type to convert
            its value. May be ``None``.
        :param ctx: The current context that arrived at this value. May
            be ``None``.
        """
        return value
    def split_envvar_value(self, rv: str) -> t.Sequence[str]:
        """Given a value from an environment variable this splits it up
        into small chunks depending on the defined envvar list splitter.
```

```
(8)                 If the splitter is set to `None`, which means that whitespace splits,
(8)                 then leading and trailing whitespace is ignored.  Otherwise, leading
(8)                 and trailing splitters usually lead to empty items being included.
(8)                 """
(8)                 return (rv or "").split(self.envvar_list_splitter)
(4)             def fail(
(8)                 self,
(8)                 message: str,
(8)                 param: t.Optional["Parameter"] = None,
(8)                 ctx: t.Optional["Context"] = None,
(4)             ) -> "t.NoReturn":
(8)                 """Helper method to fail with an invalid value message."""
(8)                 raise BadParameter(message, ctx=ctx, param=param)
(4)             def shell_complete(
(8)                 self, ctx: "Context", param: "Parameter", incomplete: str
(4)             ) -> t.List["CompletionItem"]:
(8)                 """Return a list of
(8)                 :class:`~click.shell_completion.CompletionItem` objects for the
(8)                 incomplete value. Most types do not provide completions, but
(8)                 some do, and this allows custom types to provide custom
(8)                 completions as well.
(8)                 :param ctx: Invocation context for this command.
(8)                 :param param: The parameter that is requesting completion.
(8)                 :param incomplete: Value being completed. May be empty.
(8)                 .. versionadded:: 8.0
(8)                 """
(8)                 return []
(0)         class CompositeParamType(ParamType):
(4)             is_composite = True
(4)             @property
(4)             def arity(self) -> int:  # type: ignore
(8)                 raise NotImplementedError()
(0)         class FuncParamType(ParamType):
(4)             def __init__(self, func: t.Callable[[t.Any], t.Any]) -> None:
(8)                 self.name: str = func.__name__
(8)                 self.func = func
(4)             def to_info_dict(self) -> t.Dict[str, t.Any]:
(8)                 info_dict = super().to_info_dict()
(8)                 info_dict["func"] = self.func
(8)                 return info_dict
(4)             def convert(
(8)                 self, value: t.Any, param: t.Optional["Parameter"], ctx:
t.Optional["Context"]
(4)             ) -> t.Any:
(8)                 try:
(12)                    return self.func(value)
(8)                 except ValueError:
(12)                    try:
(16)                        value = str(value)
(12)                    except UnicodeError:
(16)                        value = value.decode("utf-8", "replace")
(12)                    self.fail(value, param, ctx)
(0)         class UnprocessedParamType(ParamType):
(4)             name = "text"
(4)             def convert(
(8)                 self, value: t.Any, param: t.Optional["Parameter"], ctx:
t.Optional["Context"]
(4)             ) -> t.Any:
(8)                 return value
(4)             def __repr__(self) -> str:
(8)                 return "UNPROCESSED"
(0)         class StringParamType(ParamType):
(4)             name = "text"
(4)             def convert(
(8)                 self, value: t.Any, param: t.Optional["Parameter"], ctx:
t.Optional["Context"]
(4)             ) -> t.Any:
(8)                 if isinstance(value, bytes):
(12)                    enc = _get_argv_encoding()
```

```
(12)                    try:
(16)                        value = value.decode(enc)
(12)                    except UnicodeError:
(16)                        fs_enc = sys.getfilesystemencoding()
(16)                        if fs_enc != enc:
(20)                            try:
(24)                                value = value.decode(fs_enc)
(20)                            except UnicodeError:
(24)                                value = value.decode("utf-8", "replace")
(16)                        else:
(20)                            value = value.decode("utf-8", "replace")
(12)                    return value
(8)                 return str(value)
(4)             def __repr__(self) -> str:
(8)                 return "STRING"
(0)         class Choice(ParamType):
(4)             """The choice type allows a value to be checked against a fixed set
(4)             of supported values. All of these values have to be strings.
(4)             You should only pass a list or tuple of choices. Other iterables
(4)             (like generators) may lead to surprising results.
(4)             The resulting value will always be one of the originally passed choices
(4)             regardless of ``case_sensitive`` or any ``ctx.token_normalize_func``
(4)             being specified.
(4)             See :ref:`choice-opts` for an example.
(4)             :param case_sensitive: Set to false to make choices case
(8)                 insensitive. Defaults to true.
(4)             """
(4)             name = "choice"
(4)             def __init__(self, choices: t.Sequence[str], case_sensitive: bool = True) -> None:
(8)                 self.choices = choices
(8)                 self.case_sensitive = case_sensitive
(4)             def to_info_dict(self) -> t.Dict[str, t.Any]:
(8)                 info_dict = super().to_info_dict()
(8)                 info_dict["choices"] = self.choices
(8)                 info_dict["case_sensitive"] = self.case_sensitive
(8)                 return info_dict
(4)             def get_metavar(self, param: "Parameter") -> str:
(8)                 choices_str = "|".join(self.choices)
(8)                 if param.required and param.param_type_name == "argument":
(12)                    return f"{{{choices_str}}}"
(8)                 return f"[{choices_str}]"
(4)             def get_missing_message(self, param: "Parameter") -> str:
(8)                 return _("Choose from:\n\t{choices}").format(choices=",\n\t".join(self.choices))
(4)             def convert(
(8)                 self, value: t.Any, param: t.Optional["Parameter"], ctx: t.Optional["Context"]
(4)             ) -> t.Any:
(8)                 normed_value = value
(8)                 normed_choices = {choice: choice for choice in self.choices}
(8)                 if ctx is not None and ctx.token_normalize_func is not None:
(12)                    normed_value = ctx.token_normalize_func(value)
(12)                    normed_choices = {
(16)                        ctx.token_normalize_func(normed_choice): original
(16)                        for normed_choice, original in normed_choices.items()
(12)                    }
(8)                 if not self.case_sensitive:
(12)                    normed_value = normed_value.casefold()
(12)                    normed_choices = {
(16)                        normed_choice.casefold(): original
(16)                        for normed_choice, original in normed_choices.items()
(12)                    }
(8)                 if normed_value in normed_choices:
(12)                    return normed_choices[normed_value]
(8)                 choices_str = ", ".join(map(repr, self.choices))
(8)                 self.fail(
(12)                    ngettext(
(16)                        "{value!r} is not {choice}.",
(16)                 "{value!r} is not one of {choices}.",
(16)                 len(self.choices),
(12)             ).format(value=value, choice=choices_str, choices=choices_str),
(12)             param,
(12)             ctx,
(8)          )
(4)      def __repr__(self) -> str:
(8)          return f"Choice({list(self.choices)})"
(4)      def shell_complete(
(8)          self, ctx: "Context", param: "Parameter", incomplete: str
(4)      ) -> t.List["CompletionItem"]:
(8)          """Complete choices that start with the incomplete value.
(8)          :param ctx: Invocation context for this command.
(8)          :param param: The parameter that is requesting completion.
(8)          :param incomplete: Value being completed. May be empty.
(8)          .. versionadded:: 8.0
(8)          """
(8)          from click.shell_completion import CompletionItem
(8)          str_choices = map(str, self.choices)
(8)          if self.case_sensitive:
(12)             matched = (c for c in str_choices if c.startswith(incomplete))
(8)          else:
(12)             incomplete = incomplete.lower()
(12)             matched = (c for c in str_choices if c.lower().startswith(incomplete))
(8)          return [CompletionItem(c) for c in matched]
(0)  class DateTime(ParamType):
(4)      """The DateTime type converts date strings into `datetime` objects.
(4)      The format strings which are checked are configurable, but default to some
(4)      common (non-timezone aware) ISO 8601 formats.
(4)      When specifying *DateTime* formats, you should only pass a list or a tuple.
(4)      Other iterables, like generators, may lead to surprising results.
(4)      The format strings are processed using ``datetime.strptime``, and this
(4)      consequently defines the format strings which are allowed.
(4)      Parsing is tried using each format, in order, and the first format which
(4)      parses successfully is used.
(4)      :param formats: A list or tuple of date format strings, in the order in
(20)                         which they should be tried. Defaults to
(20)                         ``'%Y-%m-%d'``, ``'%Y-%m-%dT%H:%M:%S'``,
(20)                         ``'%Y-%m-%d %H:%M:%S'``.
(4)      """
(4)      name = "datetime"
(4)      def __init__(self, formats: t.Optional[t.Sequence[str]] = None):
(8)          self.formats: t.Sequence[str] = formats or [
(12)             "%Y-%m-%d",
(12)             "%Y-%m-%dT%H:%M:%S",
(12)             "%Y-%m-%d %H:%M:%S",
(8)          ]
(4)      def to_info_dict(self) -> t.Dict[str, t.Any]:
(8)          info_dict = super().to_info_dict()
(8)          info_dict["formats"] = self.formats
(8)          return info_dict
(4)      def get_metavar(self, param: "Parameter") -> str:
(8)          return f"[{'|'.join(self.formats)}]"
(4)      def _try_to_convert_date(self, value: t.Any, format: str) -> t.Optional[datetime]:
(8)          try:
(12)             return datetime.strptime(value, format)
(8)          except ValueError:
(12)             return None
(4)      def convert(
(8)          self, value: t.Any, param: t.Optional["Parameter"], ctx: t.Optional["Context"]
(4)      ) -> t.Any:
(8)          if isinstance(value, datetime):
(12)             return value
(8)          for format in self.formats:
(12)             converted = self._try_to_convert_date(value, format)
```

```
(12)                if converted is not None:
(16)                    return converted
(8)             formats_str = ", ".join(map(repr, self.formats))
(8)             self.fail(
(12)                ngettext(
(16)                    "{value!r} does not match the format {format}.",
(16)                    "{value!r} does not match the formats {formats}.",
(16)                    len(self.formats),
(12)                ).format(value=value, format=formats_str, formats=formats_str),
(12)                param,
(12)                ctx,
(8)             )
(4)         def __repr__(self) -> str:
(8)             return "DateTime"
(0)     class _NumberParamTypeBase(ParamType):
(4)         _number_class: t.ClassVar[t.Type[t.Any]]
(4)         def convert(
(8)             self, value: t.Any, param: t.Optional["Parameter"], ctx:
t.Optional["Context"]
(4)         ) -> t.Any:
(8)             try:
(12)                return self._number_class(value)
(8)             except ValueError:
(12)                self.fail(
(16)                    _("{value!r} is not a valid {number_type}.").format(
(20)                        value=value, number_type=self.name
(16)                    ),
(16)                    param,
(16)                    ctx,
(12)                )
(0)     class _NumberRangeBase(_NumberParamTypeBase):
(4)         def __init__(
(8)             self,
(8)             min: t.Optional[float] = None,
(8)             max: t.Optional[float] = None,
(8)             min_open: bool = False,
(8)             max_open: bool = False,
(8)             clamp: bool = False,
(4)         ) -> None:
(8)             self.min = min
(8)             self.max = max
(8)             self.min_open = min_open
(8)             self.max_open = max_open
(8)             self.clamp = clamp
(4)         def to_info_dict(self) -> t.Dict[str, t.Any]:
(8)             info_dict = super().to_info_dict()
(8)             info_dict.update(
(12)                min=self.min,
(12)                max=self.max,
(12)                min_open=self.min_open,
(12)                max_open=self.max_open,
(12)                clamp=self.clamp,
(8)             )
(8)             return info_dict
(4)         def convert(
(8)             self, value: t.Any, param: t.Optional["Parameter"], ctx:
t.Optional["Context"]
(4)         ) -> t.Any:
(8)             import operator
(8)             rv = super().convert(value, param, ctx)
(8)             lt_min: bool = self.min is not None and (
(12)                operator.le if self.min_open else operator.lt
(8)             )(rv, self.min)
(8)             gt_max: bool = self.max is not None and (
(12)                operator.ge if self.max_open else operator.gt
(8)             )(rv, self.max)
(8)             if self.clamp:
(12)                if lt_min:
(16)                    return self._clamp(self.min, 1, self.min_open)  # type: ignore
```

```
(12)            if gt_max:
(16)                return self._clamp(self.max, -1, self.max_open)  # type: ignore
(8)         if lt_min or gt_max:
(12)            self.fail(
(16)                _("{value} is not in the range {range}.").format(
(20)                    value=rv, range=self._describe_range()
(16)                ),
(16)                param,
(16)                ctx,
(12)            )
(8)         return rv
(4)     def _clamp(self, bound: float, dir: "te.Literal[1, -1]", open: bool) -> float:
(8)         """Find the valid value to clamp to bound in the given
(8)         direction.
(8)         :param bound: The boundary value.
(8)         :param dir: 1 or -1 indicating the direction to move.
(8)         :param open: If true, the range does not include the bound.
(8)         """
(8)         raise NotImplementedError
(4)     def _describe_range(self) -> str:
(8)         """Describe the range for use in help text."""
(8)         if self.min is None:
(12)            op = "<" if self.max_open else "<="
(12)            return f"x{op}{self.max}"
(8)         if self.max is None:
(12)            op = ">" if self.min_open else ">="
(12)            return f"x{op}{self.min}"
(8)         lop = "<" if self.min_open else "<="
(8)         rop = "<" if self.max_open else "<="
(8)         return f"{self.min}{lop}x{rop}{self.max}"
(4)     def __repr__(self) -> str:
(8)         clamp = " clamped" if self.clamp else ""
(8)         return f"<{type(self).__name__} {self._describe_range()}{clamp}>"
(0) class IntParamType(_NumberParamTypeBase):
(4)     name = "integer"
(4)     _number_class = int
(4)     def __repr__(self) -> str:
(8)         return "INT"
(0) class IntRange(_NumberRangeBase, IntParamType):
(4)     """Restrict an :data:`click.INT` value to a range of accepted
(4)     values. See :ref:`ranges`.
(4)     If ``min`` or ``max`` are not passed, any value is accepted in that
(4)     direction. If ``min_open`` or ``max_open`` are enabled, the
(4)     corresponding boundary is not included in the range.
(4)     If ``clamp`` is enabled, a value outside the range is clamped to the
(4)     boundary instead of failing.
(4)     .. versionchanged:: 8.0
(8)         Added the ``min_open`` and ``max_open`` parameters.
(4)     """
(4)     name = "integer range"
(4)     def _clamp(  # type: ignore
(8)         self, bound: int, dir: "te.Literal[1, -1]", open: bool
(4)     ) -> int:
(8)         if not open:
(12)            return bound
(8)         return bound + dir
(0) class FloatParamType(_NumberParamTypeBase):
(4)     name = "float"
(4)     _number_class = float
(4)     def __repr__(self) -> str:
(8)         return "FLOAT"
(0) class FloatRange(_NumberRangeBase, FloatParamType):
(4)     """Restrict a :data:`click.FLOAT` value to a range of accepted
(4)     values. See :ref:`ranges`.
(4)     If ``min`` or ``max`` are not passed, any value is accepted in that
(4)     direction. If ``min_open`` or ``max_open`` are enabled, the
(4)     corresponding boundary is not included in the range.
```

```
(4)         If ``clamp`` is enabled, a value outside the range is clamped to the
(4)         boundary instead of failing. This is not supported if either
(4)         boundary is marked ``open``.
(4)         .. versionchanged:: 8.0
(8)             Added the ``min_open`` and ``max_open`` parameters.
(4)         """
(4)         name = "float range"
(4)         def __init__(
(8)             self,
(8)             min: t.Optional[float] = None,
(8)             max: t.Optional[float] = None,
(8)             min_open: bool = False,
(8)             max_open: bool = False,
(8)             clamp: bool = False,
(4)         ) -> None:
(8)             super().__init__(
(12)                min=min, max=max, min_open=min_open, max_open=max_open, clamp=clamp
(8)             )
(8)             if (min_open or max_open) and clamp:
(12)                raise TypeError("Clamping is not supported for open bounds.")
(4)         def _clamp(self, bound: float, dir: "te.Literal[1, -1]", open: bool) -> float:
(8)             if not open:
(12)                return bound
(8)             raise RuntimeError("Clamping is not supported for open bounds.")
(0)     class BoolParamType(ParamType):
(4)         name = "boolean"
(4)         def convert(
(8)             self, value: t.Any, param: t.Optional["Parameter"], ctx: t.Optional["Context"]
(4)         ) -> t.Any:
(8)             if value in {False, True}:
(12)                return bool(value)
(8)             norm = value.strip().lower()
(8)             if norm in {"1", "true", "t", "yes", "y", "on"}:
(12)                return True
(8)             if norm in {"0", "false", "f", "no", "n", "off"}:
(12)                return False
(8)             self.fail(
(12)                _("{value!r} is not a valid boolean.").format(value=value), param, ctx
(8)             )
(4)         def __repr__(self) -> str:
(8)             return "BOOL"
(0)     class UUIDParameterType(ParamType):
(4)         name = "uuid"
(4)         def convert(
(8)             self, value: t.Any, param: t.Optional["Parameter"], ctx: t.Optional["Context"]
(4)         ) -> t.Any:
(8)             import uuid
(8)             if isinstance(value, uuid.UUID):
(12)                return value
(8)             value = value.strip()
(8)             try:
(12)                return uuid.UUID(value)
(8)             except ValueError:
(12)                self.fail(
(16)                    _("{value!r} is not a valid UUID.").format(value=value), param, ctx
(12)                )
(4)         def __repr__(self) -> str:
(8)             return "UUID"
(0)     class File(ParamType):
(4)         """Declares a parameter to be a file for reading or writing.  The file
(4)         is automatically closed once the context tears down (after the command
(4)         finished working).
(4)         Files can be opened for reading or writing.  The special value ``-``
```

```
    indicates stdin or stdout depending on the mode.
    By default, the file is opened for reading text data, but it can also be
    opened in binary mode or for writing.  The encoding parameter can be used
    to force a specific encoding.
    The `lazy` flag controls if the file should be opened immediately or upon
    first IO. The default is to be non-lazy for standard input and output
    streams as well as files opened for reading, `lazy` otherwise. When opening a
    file lazily for reading, it is still opened temporarily for validation, but
    will not be held open until first IO. lazy is mainly useful when opening
    for writing to avoid creating the file until it is needed.
    Starting with Click 2.0, files can also be opened atomically in which
    case all writes go into a separate file in the same folder and upon
    completion the file will be moved over to the original location.  This
    is useful if a file regularly read by other users is modified.
    See :ref:`file-args` for more information.
    """
    name = "filename"
    envvar_list_splitter: t.ClassVar[str] = os.path.pathsep
    def __init__(
        self,
        mode: str = "r",
        encoding: t.Optional[str] = None,
        errors: t.Optional[str] = "strict",
        lazy: t.Optional[bool] = None,
        atomic: bool = False,
    ) -> None:
        self.mode = mode
        self.encoding = encoding
        self.errors = errors
        self.lazy = lazy
        self.atomic = atomic
    def to_info_dict(self) -> t.Dict[str, t.Any]:
        info_dict = super().to_info_dict()
        info_dict.update(mode=self.mode, encoding=self.encoding)
        return info_dict
    def resolve_lazy_flag(self, value: "t.Union[str, os.PathLike[str]]") -> bool:
        if self.lazy is not None:
            return self.lazy
        if os.fspath(value) == "-":
            return False
        elif "w" in self.mode:
            return True
        return False
    def convert(
        self,
        value: t.Union[str, "os.PathLike[str]", t.IO[t.Any]],
        param: t.Optional["Parameter"],
        ctx: t.Optional["Context"],
    ) -> t.IO[t.Any]:
        if _is_file_like(value):
            return value
        value = t.cast("t.Union[str, os.PathLike[str]]", value)
        try:
            lazy = self.resolve_lazy_flag(value)
            if lazy:
                lf = LazyFile(
                    value, self.mode, self.encoding, self.errors, atomic=self.atomic
                )
                if ctx is not None:
                    ctx.call_on_close(lf.close_intelligently)
                return t.cast(t.IO[t.Any], lf)
            f, should_close = open_stream(
                value, self.mode, self.encoding, self.errors, atomic=self.atomic
            )
```

```
(12)                if ctx is not None:
(16)                    if should_close:
(20)                        ctx.call_on_close(safecall(f.close))
(16)                    else:
(20)                        ctx.call_on_close(safecall(f.flush))
(12)                return f
(8)             except OSError as e:  # noqa: B014
(12)                self.fail(f"'{format_filename(value)}': {e.strerror}", param, ctx)
(4)         def shell_complete(
(8)             self, ctx: "Context", param: "Parameter", incomplete: str
(4)         ) -> t.List["CompletionItem"]:
(8)             """Return a special completion marker that tells the completion
(8)             system to use the shell to provide file path completions.
(8)             :param ctx: Invocation context for this command.
(8)             :param param: The parameter that is requesting completion.
(8)             :param incomplete: Value being completed. May be empty.
(8)             .. versionadded:: 8.0
(8)             """
(8)             from click.shell_completion import CompletionItem
(8)             return [CompletionItem(incomplete, type="file")]
(0)     def _is_file_like(value: t.Any) -> "te.TypeGuard[t.IO[t.Any]]":
(4)         return hasattr(value, "read") or hasattr(value, "write")
(0)     class Path(ParamType):
(4)         """The ``Path`` type is similar to the :class:`File` type, but
(4)         returns the filename instead of an open file. Various checks can be
(4)         enabled to validate the type of file and permissions.
(4)         :param exists: The file or directory needs to exist for the value to
(8)             be valid. If this is not set to ``True``, and the file does not
(8)             exist, then all further checks are silently skipped.
(4)         :param file_okay: Allow a file as a value.
(4)         :param dir_okay: Allow a directory as a value.
(4)         :param readable: if true, a readable check is performed.
(4)         :param writable: if true, a writable check is performed.
(4)         :param executable: if true, an executable check is performed.
(4)         :param resolve_path: Make the value absolute and resolve any
(8)             symlinks. A ``~`` is not expanded, as this is supposed to be
(8)             done by the shell only.
(4)         :param allow_dash: Allow a single dash as a value, which indicates
(8)             a standard stream (but does not open it). Use
(8)             :func:`~click.open_file` to handle opening this value.
(4)         :param path_type: Convert the incoming path value to this type. If
(8)             ``None``, keep Python's default, which is ``str``. Useful to
(8)             convert to :class:`pathlib.Path`.
(4)         .. versionchanged:: 8.1
(8)             Added the ``executable`` parameter.
(4)         .. versionchanged:: 8.0
(8)             Allow passing ``path_type=pathlib.Path``.
(4)         .. versionchanged:: 6.0
(8)             Added the ``allow_dash`` parameter.
(4)         """
(4)         envvar_list_splitter: t.ClassVar[str] = os.path.pathsep
(4)         def __init__(
(8)             self,
(8)             exists: bool = False,
(8)             file_okay: bool = True,
(8)             dir_okay: bool = True,
(8)             writable: bool = False,
(8)             readable: bool = True,
(8)             resolve_path: bool = False,
(8)             allow_dash: bool = False,
(8)             path_type: t.Optional[t.Type[t.Any]] = None,
(8)             executable: bool = False,
(4)         ):
(8)             self.exists = exists
(8)             self.file_okay = file_okay
(8)             self.dir_okay = dir_okay
(8)             self.readable = readable
(8)             self.writable = writable
(8)             self.executable = executable
```

```
(8)                     self.resolve_path = resolve_path
(8)                     self.allow_dash = allow_dash
(8)                     self.type = path_type
(8)                     if self.file_okay and not self.dir_okay:
(12)                        self.name: str = _("file")
(8)                     elif self.dir_okay and not self.file_okay:
(12)                        self.name = _("directory")
(8)                     else:
(12)                        self.name = _("path")
(4)                 def to_info_dict(self) -> t.Dict[str, t.Any]:
(8)                     info_dict = super().to_info_dict()
(8)                     info_dict.update(
(12)                        exists=self.exists,
(12)                        file_okay=self.file_okay,
(12)                        dir_okay=self.dir_okay,
(12)                        writable=self.writable,
(12)                        readable=self.readable,
(12)                        allow_dash=self.allow_dash,
(8)                     )
(8)                     return info_dict
(4)                 def coerce_path_result(
(8)                     self, value: "t.Union[str, os.PathLike[str]]"
(4)                 ) -> "t.Union[str, bytes, os.PathLike[str]]":
(8)                     if self.type is not None and not isinstance(value, self.type):
(12)                        if self.type is str:
(16)                            return os.fsdecode(value)
(12)                        elif self.type is bytes:
(16)                            return os.fsencode(value)
(12)                        else:
(16)                            return t.cast("os.PathLike[str]", self.type(value))
(8)                     return value
(4)                 def convert(
(8)                     self,
(8)                     value: "t.Union[str, os.PathLike[str]]",
(8)                     param: t.Optional["Parameter"],
(8)                     ctx: t.Optional["Context"],
(4)                 ) -> "t.Union[str, bytes, os.PathLike[str]]":
(8)                     rv = value
(8)                     is_dash = self.file_okay and self.allow_dash and rv in (b"-", "-")
(8)                     if not is_dash:
(12)                        if self.resolve_path:
(16)                            import pathlib
(16)                            rv = os.fsdecode(pathlib.Path(rv).resolve())
(12)                        try:
(16)                            st = os.stat(rv)
(12)                        except OSError:
(16)                            if not self.exists:
(20)                                return self.coerce_path_result(rv)
(16)                            self.fail(
(20)                                _("{name} {filename!r} does not exist.").format(
(24)                                    name=self.name.title(),
filename=format_filename(value)
(20)                                ),
(20)                                param,
(20)                                ctx,
(16)                            )
(12)                        if not self.file_okay and stat.S_ISREG(st.st_mode):
(16)                            self.fail(
(20)                                _("{name} {filename!r} is a file.").format(
(24)                                    name=self.name.title(),
filename=format_filename(value)
(20)                                ),
(20)                                param,
(20)                                ctx,
(16)                            )
(12)                        if not self.dir_okay and stat.S_ISDIR(st.st_mode):
(16)                            self.fail(
(20)                                _("{name} '{filename}' is a directory.").format(
(24)                                    name=self.name.title(),
filename=format_filename(value)
(20)                                ),
(20)                                param,
(20)                                ctx,
(16)                            )
(12)                        if self.readable and not os.access(rv, os.R_OK):
(16)                            self.fail(
(20)                                _("{name} {filename!r} is not readable.").format(
(24)                                    name=self.name.title(),
filename=format_filename(value)
(20)                                ),
(20)                                param,
(20)                                ctx,
(16)                            )
(12)                        if self.writable and not os.access(rv, os.W_OK):
(16)                            self.fail(
(20)                                _("{name} {filename!r} is not writable.").format(
(24)                                    name=self.name.title(),
filename=format_filename(value)
(20)                                ),
(20)                                param,
(20)                                ctx,
(16)                            )
(12)                        if self.executable and not os.access(value, os.X_OK):
(16)                            self.fail(
(20)                                _("{name} {filename!r} is not executable.").format(
(24)                                    name=self.name.title(),
filename=format_filename(value)
(20)                                ),
(20)                                param,
(20)                                ctx,
(16)                            )
(8)                     return self.coerce_path_result(rv)
(4)                 def shell_complete(
(8)                     self, ctx: "Context", param: "Parameter", incomplete: str
(4)                 ) -> t.List["CompletionItem"]:
(8)                     """Return a special completion marker that tells the completion
(8)                     system to use the shell to provide path completions for only
(8)                     directories or any paths.
(8)                     :param ctx: Invocation context for this command.
(8)                     :param param: The parameter that is requesting completion.
(8)                     :param incomplete: Value being completed. May be empty.
(8)                     .. versionadded:: 8.0
(8)                     """
(8)                     from click.shell_completion import CompletionItem
(8)                     type = "dir" if self.dir_okay and not self.file_okay else "file"
(8)                     return [CompletionItem(incomplete, type=type)]
(0)             class Tuple(CompositeParamType):
(4)                 """The default behavior of Click is to apply a type on a value directly.
(4)                 This works well in most cases, except for when `nargs` is set to a fixed
(4)                 count and different types should be used for different items.  In this
(4)                 case the :class:`Tuple` type can be used.  This type can only be used
(4)                 if `nargs` is set to a fixed number.
(4)                 For more information see :ref:`tuple-type`.
(4)                 This can be selected by using a Python tuple literal as a type.
(4)                 :param types: a list of types that should be used for the tuple items.
(4)                 """
(4)                 def __init__(self, types: t.Sequence[t.Union[t.Type[t.Any], ParamType]]) -
> None:
(8)                     self.types: t.Sequence[ParamType] = [convert_type(ty) for ty in types]
(4)                 def to_info_dict(self) -> t.Dict[str, t.Any]:
(8)                     info_dict = super().to_info_dict()
(8)                     info_dict["types"] = [t.to_info_dict() for t in self.types]
(8)                     return info_dict
(4)                 @property
(4)                 def name(self) -> str:  # type: ignore
(8)                     return f"<{' '.join(ty.name for ty in self.types)}>"
(4)                 @property
(4)                 def arity(self) -> int:  # type: ignore
```

```python
        return len(self.types)
    def convert(
        self, value: t.Any, param: t.Optional["Parameter"], ctx: t.Optional["Context"]
    ) -> t.Any:
        len_type = len(self.types)
        len_value = len(value)
        if len_value != len_type:
            self.fail(
                ngettext(
                    "{len_type} values are required, but {len_value} was given.",
                    "{len_type} values are required, but {len_value} were given.",
                    len_value,
                ).format(len_type=len_type, len_value=len_value),
                param=param,
                ctx=ctx,
            )
        return tuple(ty(x, param, ctx) for ty, x in zip(self.types, value))
def convert_type(ty: t.Optional[t.Any], default: t.Optional[t.Any] = None) -> ParamType:
    """Find the most appropriate :class:`ParamType` for the given Python
    type. If the type isn't provided, it can be inferred from a default
    value.
    """
    guessed_type = False
    if ty is None and default is not None:
        if isinstance(default, (tuple, list)):
            if default:
                item = default[0]
                if isinstance(item, (tuple, list)):
                    ty = tuple(map(type, item))
                else:
                    ty = type(item)
        else:
            ty = type(default)
        guessed_type = True
    if isinstance(ty, tuple):
        return Tuple(ty)
    if isinstance(ty, ParamType):
        return ty
    if ty is str or ty is None:
        return STRING
    if ty is int:
        return INT
    if ty is float:
        return FLOAT
    if ty is bool:
        return BOOL
    if guessed_type:
        return STRING
    if __debug__:
        try:
            if issubclass(ty, ParamType):
                raise AssertionError(
                    f"Attempted to use an uninstantiated parameter type ({ty})."
                )
        except TypeError:
            pass
    return FuncParamType(ty)
UNPROCESSED = UnprocessedParamType()
STRING = StringParamType()
INT = IntParamType()
FLOAT = FloatParamType()
BOOL = BoolParamType()
UUID = UUIDParameterType()
```

```
----------------------------------------

File 16 - utils.py:

1: (0)              import os
2: (0)              import re
3: (0)              import sys
4: (0)              import typing as t
5: (0)              from functools import update_wrapper
6: (0)              from types import ModuleType
7: (0)              from types import TracebackType
8: (0)              from ._compat import _default_text_stderr
9: (0)              from ._compat import _default_text_stdout
10: (0)             from ._compat import _find_binary_writer
11: (0)             from ._compat import auto_wrap_for_ansi
12: (0)             from ._compat import binary_streams
13: (0)             from ._compat import open_stream
14: (0)             from ._compat import should_strip_ansi
15: (0)             from ._compat import strip_ansi
16: (0)             from ._compat import text_streams
17: (0)             from ._compat import WIN
18: (0)             from .globals import resolve_color_default
19: (0)             if t.TYPE_CHECKING:
20: (4)                 import typing_extensions as te
21: (4)                 P = te.ParamSpec("P")
22: (0)             R = t.TypeVar("R")
23: (0)             def _posixify(name: str) -> str:
24: (4)                 return "-".join(name.split()).lower()
25: (0)             def safecall(func: "t.Callable[P, R]") -> "t.Callable[P, t.Optional[R]]":
26: (4)                 """Wraps a function so that it swallows exceptions."""
27: (4)                 def wrapper(*args: "P.args", **kwargs: "P.kwargs") -> t.Optional[R]:
28: (8)                     try:
29: (12)                        return func(*args, **kwargs)
30: (8)                     except Exception:
31: (12)                        pass
32: (8)                     return None
33: (4)                 return update_wrapper(wrapper, func)
34: (0)             def make_str(value: t.Any) -> str:
35: (4)                 """Converts a value into a valid string."""
36: (4)                 if isinstance(value, bytes):
37: (8)                     try:
38: (12)                        return value.decode(sys.getfilesystemencoding())
39: (8)                     except UnicodeError:
40: (12)                        return value.decode("utf-8", "replace")
41: (4)                 return str(value)
42: (0)             def make_default_short_help(help: str, max_length: int = 45) -> str:
43: (4)                 """Returns a condensed version of help string."""
44: (4)                 paragraph_end = help.find("\n\n")
45: (4)                 if paragraph_end != -1:
46: (8)                     help = help[:paragraph_end]
47: (4)                 words = help.split()
48: (4)                 if not words:
49: (8)                     return ""
50: (4)                 if words[0] == "\b":
51: (8)                     words = words[1:]
52: (4)                 total_length = 0
53: (4)                 last_index = len(words) - 1
54: (4)                 for i, word in enumerate(words):
55: (8)                     total_length += len(word) + (i > 0)
56: (8)                     if total_length > max_length:  # too long, truncate
57: (12)                        break
58: (8)                     if word[-1] == ".":  # sentence end, truncate without "..."
59: (12)                        return " ".join(words[: i + 1])
60: (8)                     if total_length == max_length and i != last_index:
61: (12)                        break  # not at sentence end, truncate with "..."
62: (4)                 else:
63: (8)                     return " ".join(words)  # no truncation needed
64: (4)                 total_length += len("...")
65: (4)                 while i > 0:
```

```
(8)             total_length -= len(words[i]) + (i > 0)
(8)             if total_length <= max_length:
(12)                break
(8)             i -= 1
(4)         return " ".join(words[:i]) + "..."
(0)     class LazyFile:
(4)         """A lazy file works like a regular file but it does not fully open
(4)         the file but it does perform some basic checks early to see if the
(4)         filename parameter does make sense.  This is useful for safely opening
(4)         files for writing.
(4)         """
(4)         def __init__(
(8)             self,
(8)             filename: t.Union[str, "os.PathLike[str]"],
(8)             mode: str = "r",
(8)             encoding: t.Optional[str] = None,
(8)             errors: t.Optional[str] = "strict",
(8)             atomic: bool = False,
(4)         ):
(8)             self.name: str = os.fspath(filename)
(8)             self.mode = mode
(8)             self.encoding = encoding
(8)             self.errors = errors
(8)             self.atomic = atomic
(8)             self._f: t.Optional[t.IO[t.Any]]
(8)             self.should_close: bool
(8)             if self.name == "-":
(12)                self._f, self.should_close = open_stream(filename, mode, encoding, errors)
(8)             else:
(12)                if "r" in mode:
(16)                    open(filename, mode).close()
(12)                self._f = None
(12)                self.should_close = True
(4)         def __getattr__(self, name: str) -> t.Any:
(8)             return getattr(self.open(), name)
(4)         def __repr__(self) -> str:
(8)             if self._f is not None:
(12)                return repr(self._f)
(8)             return f"<unopened file '{format_filename(self.name)}' {self.mode}>"
(4)         def open(self) -> t.IO[t.Any]:
(8)             """Opens the file if it's not yet open.  This call might fail with
(8)             a :exc:`FileError`.  Not handling this error will produce an error
(8)             that Click shows.
(8)             """
(8)             if self._f is not None:
(12)                return self._f
(8)             try:
(12)                rv, self.should_close = open_stream(
(16)                    self.name, self.mode, self.encoding, self.errors, atomic=self.atomic
(12)                )
(8)             except OSError as e:  # noqa: E402
(12)                from .exceptions import FileError
(12)                raise FileError(self.name, hint=e.strerror) from e
(8)             self._f = rv
(8)             return rv
(4)         def close(self) -> None:
(8)             """Closes the underlying file, no matter what."""
(8)             if self._f is not None:
(12)                self._f.close()
(4)         def close_intelligently(self) -> None:
(8)             """This function only closes the file if it was opened by the lazy
(8)             file wrapper.  For instance this will never close stdin.
(8)             """
(8)             if self.should_close:
(12)                self.close()
(4)         def __enter__(self) -> "LazyFile":
(8)             return self
```

```
    def __exit__(
        self,
        exc_type: t.Optional[t.Type[BaseException]],
        exc_value: t.Optional[BaseException],
        tb: t.Optional[TracebackType],
    ) -> None:
        self.close_intelligently()
    def __iter__(self) -> t.Iterator[t.AnyStr]:
        self.open()
        return iter(self._f)  # type: ignore
class KeepOpenFile:
    def __init__(self, file: t.IO[t.Any]) -> None:
        self._file: t.IO[t.Any] = file
    def __getattr__(self, name: str) -> t.Any:
        return getattr(self._file, name)
    def __enter__(self) -> "KeepOpenFile":
        return self
    def __exit__(
        self,
        exc_type: t.Optional[t.Type[BaseException]],
        exc_value: t.Optional[BaseException],
        tb: t.Optional[TracebackType],
    ) -> None:
        pass
    def __repr__(self) -> str:
        return repr(self._file)
    def __iter__(self) -> t.Iterator[t.AnyStr]:
        return iter(self._file)
def echo(
    message: t.Optional[t.Any] = None,
    file: t.Optional[t.IO[t.Any]] = None,
    nl: bool = True,
    err: bool = False,
    color: t.Optional[bool] = None,
) -> None:
    """Print a message and newline to stdout or a file. This should be
    used instead of :func:`print` because it provides better support
    for different data, files, and environments.
    Compared to :func:`print`, this does the following:
    -   Ensures that the output encoding is not misconfigured on Linux.
    -   Supports Unicode in the Windows console.
    -   Supports writing to binary outputs, and supports writing bytes
        to text outputs.
    -   Supports colors and styles on Windows.
    -   Removes ANSI color and style codes if the output does not look
        like an interactive terminal.
    -   Always flushes the output.
    :param message: The string or bytes to output. Other objects are
        converted to strings.
    :param file: The file to write to. Defaults to ``stdout``.
    :param err: Write to ``stderr`` instead of ``stdout``.
    :param nl: Print a newline after the message. Enabled by default.
    :param color: Force showing or hiding colors and other styles. By
        default Click will remove color if the output does not look like
        an interactive terminal.
    .. versionchanged:: 6.0
        Support Unicode output on the Windows console. Click does not
        modify ``sys.stdout``, so ``sys.stdout.write()`` and ``print()``
        will still not support Unicode.
    .. versionchanged:: 4.0
        Added the ``color`` parameter.
    .. versionadded:: 3.0
        Added the ``err`` parameter.
    .. versionchanged:: 2.0
        Support colors on Windows if colorama is installed.
    """
    if file is None:
        if err:
            file = _default_text_stderr()
        else:
            file = _default_text_stdout()
        if file is None:
            return
    if message is not None and not isinstance(message, (str, bytes, bytearray)):
        out: t.Optional[t.Union[str, bytes]] = str(message)
    else:
        out = message
    if nl:
        out = out or ""
        if isinstance(out, str):
            out += "\n"
        else:
            out += b"\n"
    if not out:
        file.flush()
        return
    if isinstance(out, (bytes, bytearray)):
        binary_file = _find_binary_writer(file)
        if binary_file is not None:
            file.flush()
            binary_file.write(out)
            binary_file.flush()
            return
    else:
        color = resolve_color_default(color)
        if should_strip_ansi(file, color):
            out = strip_ansi(out)
        elif WIN:
            if auto_wrap_for_ansi is not None:
                file = auto_wrap_for_ansi(file)  # type: ignore
            elif not color:
                out = strip_ansi(out)
    file.write(out)  # type: ignore
    file.flush()
def get_binary_stream(name: "te.Literal['stdin', 'stdout', 'stderr']") -> t.BinaryIO:
    """Returns a system stream for byte processing.
    :param name: the name of the stream to open.  Valid names are ``'stdin'``,
                 ``'stdout'`` and ``'stderr'``
    """
    opener = binary_streams.get(name)
    if opener is None:
        raise TypeError(f"Unknown standard stream '{name}'")
    return opener()
def get_text_stream(
    name: "te.Literal['stdin', 'stdout', 'stderr']",
    encoding: t.Optional[str] = None,
    errors: t.Optional[str] = "strict",
) -> t.TextIO:
    """Returns a system stream for text processing.  This usually returns
    a wrapped stream around a binary stream returned from
    :func:`get_binary_stream` but it also can take shortcuts for already
    correctly configured streams.
    :param name: the name of the stream to open.  Valid names are ``'stdin'``,
                 ``'stdout'`` and ``'stderr'``
    :param encoding: overrides the detected default encoding.
    :param errors: overrides the default error mode.
    """
    opener = text_streams.get(name)
    if opener is None:
        raise TypeError(f"Unknown standard stream '{name}'")
    return opener(encoding, errors)
def open_file(
    filename: str,
    mode: str = "r",
    encoding: t.Optional[str] = None,
    errors: t.Optional[str] = "strict",
    lazy: bool = False,
    atomic: bool = False,
) -> t.IO[t.Any]:
    """Open a file, with extra behavior to handle ``'-'`` to indicate
    a standard stream, lazy open on write, and atomic write. Similar to
    the behavior of the :class:`~click.File` param type.
    If ``'-'`` is given to open ``stdout`` or ``stdin``, the stream is
    wrapped so that using it in a context manager will not close it.
    This makes it possible to use the function without accidentally
    closing a standard stream:
    .. code-block:: python
        with open_file(filename) as f:
            ...
    :param filename: The name of the file to open, or ``'-'`` for
        ``stdin``/``stdout``.
    :param mode: The mode in which to open the file.
    :param encoding: The encoding to decode or encode a file opened in
        text mode.
    :param errors: The error handling mode.
    :param lazy: Wait to open the file until it is accessed. For read
        mode, the file is temporarily opened to raise access errors
        early, then closed until it is read again.
    :param atomic: Write to a temporary file and replace the given file
        on close.
    .. versionadded:: 3.0
    """
    if lazy:
        return t.cast(
            t.IO[t.Any], LazyFile(filename, mode, encoding, errors, atomic=atomic)
        )
    f, should_close = open_stream(filename, mode, encoding, errors, atomic=atomic)
    if not should_close:
        f = t.cast(t.IO[t.Any], KeepOpenFile(f))
    return f
def format_filename(
    filename: "t.Union[str, bytes, os.PathLike[str], os.PathLike[bytes]]",
    shorten: bool = False,
) -> str:
    """Format a filename as a string for display. Ensures the filename can be
    displayed by replacing any invalid bytes or surrogate escapes in the name
    with the replacement character ``ï¿½``.
    Invalid bytes or surrogate escapes will raise an error when written to a
    stream with ``errors="strict". This will typically happen with ``stdout``
    when the locale is something like ``en_GB.UTF-8``.
    Many scenarios *are* safe to write surrogates though, due to PEP 538 and
    PEP 540, including:
    -   Writing to ``stderr``, which uses ``errors="backslashreplace"``.
    -   The system has ``LANG=C.UTF-8``, ``C``, or ``POSIX``. Python opens
        stdout and stderr with ``errors="surrogateescape"``.
    -   None of ``LANG/LC_*`` are set. Python assumes ``LANG=C.UTF-8``.
    -   Python is started in UTF-8 mode  with  ``PYTHONUTF8=1`` or ``-X utf8``.
        Python opens stdout and stderr with ``errors="surrogateescape"``.
    :param filename: formats a filename for UI display.  This will also convert
                     the filename into unicode without failing.
    :param shorten: this optionally shortens the filename to strip of the
                    path that leads up to it.
    """
    if shorten:
        filename = os.path.basename(filename)
    else:
        filename = os.fspath(filename)
    if isinstance(filename, bytes):
        filename = filename.decode(sys.getfilesystemencoding(), "replace")
    else:
        filename = filename.encode("utf-8", "surrogateescape").decode(
            "utf-8", "replace"
        )
    return filename
def get_app_dir(app_name: str, roaming: bool = True, force_posix: bool = False) -> str:
    r"""Returns the config folder for the application.  The default behavior
    is to return whatever is most appropriate for the operating system.
    To give you an idea, for an app called ``"Foo Bar"``, something like
    the following folders could be returned:
    Mac OS X:
      ``~/Library/Application Support/Foo Bar``
    Mac OS X (POSIX):
      ``~/.foo-bar``
    Unix:
      ``~/.config/foo-bar``
    Unix (POSIX):
      ``~/.foo-bar``
    Windows (roaming):
      ``C:\Users\<user>\AppData\Roaming\Foo Bar``
    Windows (not roaming):
      ``C:\Users\<user>\AppData\Local\Foo Bar``
    .. versionadded:: 2.0
    :param app_name: the application name.  This should be properly capitalized
                     and can contain whitespace.
    :param roaming: controls if the folder should be roaming or not on Windows.
                    Has no effect otherwise.
    :param force_posix: if this is set to `True` then on any POSIX system the
                        folder will be stored in the home folder with a leading
                        dot instead of the XDG config home or darwin's
                        application support folder.
    """
    if WIN:
        key = "APPDATA" if roaming else "LOCALAPPDATA"
        folder = os.environ.get(key)
        if folder is None:
            folder = os.path.expanduser("~")
        return os.path.join(folder, app_name)
    if force_posix:
        return os.path.join(os.path.expanduser(f"~/.{_posixify(app_name)}"))
    if sys.platform == "darwin":
        return os.path.join(
            os.path.expanduser("~/Library/Application Support"), app_name
        )
    return os.path.join(
        os.environ.get("XDG_CONFIG_HOME", os.path.expanduser("~/.config")),
        _posixify(app_name),
    )
class PacifyFlushWrapper:
    """This wrapper is used to catch and suppress BrokenPipeErrors resulting
    from ``.flush()`` being called on broken pipe during the shutdown/final-GC
    of the Python interpreter. Notably ``.flush()`` is always called on
    ``sys.stdout`` and ``sys.stderr``. So as to have minimal impact on any
    other cleanup code, and the case where the underlying file is not a broken
    pipe, all calls and attributes are proxied.
    """
    def __init__(self, wrapped: t.IO[t.Any]) -> None:
        self.wrapped = wrapped
    def flush(self) -> None:
        try:
            self.wrapped.flush()
        except OSError as e:
            import errno
            if e.errno != errno.EPIPE:
                raise
    def __getattr__(self, attr: str) -> t.Any:
        return getattr(self.wrapped, attr)
```

```python
def _detect_program_name(
    path: t.Optional[str] = None, _main: t.Optional[ModuleType] = None
) -> str:
    """Determine the command used to run the program, for use in help
    text. If a file or entry point was executed, the file name is
    returned. If ``python -m`` was used to execute a module or package,
    ``python -m name`` is returned.
    This doesn't try to be too precise, the goal is to give a concise
    name for help text. Files are only shown as their name without the
    path. ``python`` is only shown for modules, and the full path to
    ``sys.executable`` is not shown.
    :param path: The Python file being executed. Python puts this in
        ``sys.argv[0]``, which is used by default.
    :param _main: The ``__main__`` module. This should only be passed
        during internal testing.
    .. versionadded:: 8.0
        Based on command args detection in the Werkzeug reloader.
    :meta private:
    """
    if _main is None:
        _main = sys.modules["__main__"]
    if not path:
        path = sys.argv[0]
    if getattr(_main, "__package__", None) in {None, ""} or (
        os.name == "nt"
        and _main.__package__ == ""
        and not os.path.exists(path)
        and os.path.exists(f"{path}.exe")
    ):
        return os.path.basename(path)
    py_module = t.cast(str, _main.__package__)
    name = os.path.splitext(os.path.basename(path))[0]
    if name != "__main__":
        py_module = f"{py_module}.{name}"
    return f"python -m {py_module.lstrip('.')}"
def _expand_args(
    args: t.Iterable[str],
    *,
    user: bool = True,
    env: bool = True,
    glob_recursive: bool = True,
) -> t.List[str]:
    """Simulate Unix shell expansion with Python functions.
    See :func:`glob.glob`, :func:`os.path.expanduser`, and
    :func:`os.path.expandvars`.
    This is intended for use on Windows, where the shell does not do any
    expansion. It may not exactly match what a Unix shell would do.
    :param args: List of command line arguments to expand.
    :param user: Expand user home directory.
    :param env: Expand environment variables.
    :param glob_recursive: ``**`` matches directories recursively.
    .. versionchanged:: 8.1
        Invalid glob patterns are treated as empty expansions rather
        than raising an error.
    .. versionadded:: 8.0
    :meta private:
    """
    from glob import glob
    out = []
    for arg in args:
        if user:
            arg = os.path.expanduser(arg)
        if env:
            arg = os.path.expandvars(arg)
        try:
            matches = glob(arg, recursive=glob_recursive)
        except re.error:
            matches = []
        if not matches:
```

```
468: (12)                   out.append(arg)
469: (8)                else:
470: (12)                   out.extend(matches)
471: (4)            return out

----------------------------------------

File 17 -
SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRYCOMBINER_aligner_20_characters_for_pythons_codes.p
y:

1: (0)              import os
2: (0)              from datetime import datetime
3: (0)              def get_file_info(root_folder):
4: (4)                  file_info_list = []
5: (4)                  for root, dirs, files in os.walk(root_folder):
6: (8)                      for file in files:
7: (12)                         try:
8: (16)                             if file.endswith('.py'):
9: (20)                                 file_path = os.path.join(root, file)
10: (20)                                creation_time =
datetime.fromtimestamp(os.path.getctime(file_path))
11: (20)                                modified_time =
datetime.fromtimestamp(os.path.getmtime(file_path))
12: (20)                                file_extension = os.path.splitext(file)[1].lower()
13: (20)                                file_info_list.append([file, file_path, creation_time,
modified_time, file_extension, root])
14: (12)                        except Exception as e:
15: (16)                            print(f"Error processing file {file}: {e}")
16: (4)                 file_info_list.sort(key=lambda x: (x[2], x[3], len(x[0]), x[4]))  # Sort
by creation, modification time, name length, extension
17: (4)                 return file_info_list
18: (0)             def process_file(file_info_list):
19: (4)                 combined_output = []
20: (4)                 for idx, (file_name, file_path, creation_time, modified_time,
file_extension, root) in enumerate(file_info_list):
21: (8)                     with open(file_path, 'r', encoding='utf-8', errors='ignore') as f:
22: (12)                        content = f.read()
23: (12)                        content = "\n".join([line for line in content.split('\n') if
line.strip() and not line.strip().startswith("#")])
24: (12)                        content = content.replace('\t', '    ')
25: (12)                        processed_lines = []
26: (12)                        for i, line in enumerate(content.split('\n')):
27: (16)                            leading_spaces = len(line) - len(line.lstrip(' '))
28: (16)                            line_number_str = f"{i+1}: ({leading_spaces})"
29: (16)                            padding = ' ' * (20 - len(line_number_str))
30: (16)                            processed_line = f"{line_number_str}{padding}{line}"
31: (16)                            processed_lines.append(processed_line)
32: (12)                        content_with_line_numbers = "\n".join(processed_lines)
33: (12)                        combined_output.append(f"File {idx + 1} - {file_name}:\n")
34: (12)                        combined_output.append(content_with_line_numbers)
35: (12)                        combined_output.append("\n" + "-"*40 + "\n")
36: (4)                 return combined_output
37: (0)             root_folder_path = '.'  # Set this to the desired folder
38: (0)             file_info_list = get_file_info(root_folder_path)
39: (0)             combined_output = process_file(file_info_list)
40: (0)             output_file =
'SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY_combined_python_files_20_chars.txt'
41: (0)             with open(output_file, 'w', encoding='utf-8') as logfile:
42: (4)                 logfile.write("\n".join(combined_output))
43: (0)             print(f"Processed file info logged to {output_file}")

----------------------------------------
```