Duct

Dec 03, 2020

Contents

Duct is a library for running child processes. Duct makes it easy to build pipelines and redirect IO like a shell. At the same time, Duct helps you write correct, portable code: whitespace is never significant, errors from child processes get reported by default, and a variety of gotchas, bugs, and platform inconsistencies are handled for you the Right Way™.

- GitHub repo
- PyPI package
- the same library, in Rust

Examples

Run a command without capturing any output. Here "hi" is printed directly to the terminal:

```
>>> from duct import cmd
>>> cmd("echo", "hi").run() # doctest: +SKIP
hi
Output(status=0, stdout=None, stderr=None)
```

Capture the standard output of a command. Here "hi" is returned as a string:

```
>>> cmd("echo", "hi").read()
'hi'
```

Capture the standard output of a pipeline:

```
>>> cmd("echo", "hi").pipe(cmd("sed", "s/i/o/")).read()
'ho'
```

Merge standard error into standard output and read both incrementally:

```
>>> big_cmd = cmd("bash", "-c", "echo out && echo err 1>&2")
>>> reader = big_cmd.stderr_to_stdout().reader()
>>> with reader:
...     reader.readlines()
[b'out\n', b'err\n']
```

Children that exit with a non-zero status raise an exception by default:

```
>>> cmd("false").run()
Traceback (most recent call last):
...
duct.StatusError: Expression cmd('false') returned non-zero exit status:␣
↪Output(status=1, stdout=None, stderr=None)
>>> cmd("false").unchecked().run()
Output(status=1, stdout=None, stderr=None)
```

CHAPTER 1

Functions

duct.**cmd**(*prog*, **args*)

Build a command *Expression* from a program name and any number of arguments.

This is the sole entry point to Duct. All the types below are built with methods on the *Expression* returned by this function.

```
>>> cmd("echo", "hi").read()
'hi'
```

CHAPTER 2

Types

class duct.**Expression**

An expression object representing a command or a pipeline of commands.

Build command expressions with the *cmd()* function. Build pipelines with the *pipe()* method. Methods like *stdout_path()* and *env()* also return new expressions representing the modified execution environment. Execute expressions with *run()*, *read()*, *start()*, or *reader()*.

before_spawn(*callback*)

Add a callback for modifying the arguments to `Popen()` right before it's called. The callback will be passed a command list (the program followed by its arguments) and a keyword arguments dictionary, and it may modify either. The callback's return value is ignored.

The callback is called for each command in its sub-expression, and each time the expression is executed. That call happens after other features like `stdout()` and *env()* have been applied, so any changes made by the callback take priority. More than one callback can be added, in which case the innermost is executed last. For example, if one call to *before_spawn()* is applied to an entire *pipe()* expression, and another call is applied to just one command within the pipeline, the callback for the entire pipeline will be called first over the command where both hooks apply.

This is intended for rare and tricky cases, like callers who want to change the group ID of their child processes, or who want to run code in `Popen.preexec_fn()`. Most callers shouldn't need to use it.

```
>>> def add_sneaky_arg(command, kwargs):
...     command.append("sneaky!")
>>> cmd("echo", "being").before_spawn(add_sneaky_arg).read()
'being sneaky!'
```

dir(*path*)

Set the working directory for the expression.

```
>>> cmd("pwd").dir("/").read()
'/'
```

Note that *dir()* does *not* affect the meaning of relative exe paths. For example in the expression `cmd("./foo.sh").dir("bar")`, the script `./foo.sh` will execute, *not* the script `./bar/foo.sh`. How-

ever, it usually *does* affect how the child process interprets relative paths in command arguments.

env(*name*, *val*)
Set an environment variable in the expression's environment.

```
>>> cmd("bash", "-c", "echo $FOO").env("FOO", "bar").read()
'bar'
```

env_remove(*name*)
Unset an environment variable in the expression's environment.

```
>>> os.environ["FOO"] = "bar"
>>> cmd("bash", "-c", "echo $FOO").env_remove("FOO").read()
''
```

Note that all of Duct's `env` functions follow OS rules for environment variable case sensitivity. That means that `env_remove("foo")` will unset `FOO` on Windows (where `foo` and `FOO` are equivalent) but not on Unix (where they are separate variables). Portable programs should restrict themselves to uppercase environment variable names for that reason.

full_env(*env_dict*)
Set the entire environment for the expression, from a dictionary of name-value pairs.

```
>>> os.environ["FOO"] = "bar"
>>> os.environ["BAZ"] = "bing"
>>> cmd("bash", "-c", "echo $FOO$BAZ").full_env({"FOO": "xyz"}).read()
'xyz'
```

Note that some environment variables are required for normal program execution (like SystemRoot on Windows), so copying the parent's environment is usually preferable to starting with an empty one.

pipe(*right_side*)
Combine two expressions to form a pipeline.

```
>>> cmd("echo", "hi").pipe(cmd("sed", "s/i/o/")).read()
'ho'
```

During execution, if one side of the pipe returns a non-zero exit status, that becomes the status of the whole pipe, similar to Bash's `pipefail` option. If both sides return non-zero, and one of them is *`unchecked()`*, then the checked side wins. Otherwise the right side wins.

During spawning, if the left side of the pipe spawns successfully, but the right side fails to spawn, the left side will be killed and awaited. That's necessary to return the spawn errors immediately, without leaking the left side as a zombie.

read()
Execute the expression and capture its output, similar to backticks or $() in the shell.

This is a wrapper around reader() which reads to EOF, decodes UTF-8, trims newlines, and returns the resulting string.

```
>>> cmd("echo", "hi").read()
'hi'
```

reader()
Start executing the expression with its stdout captured, and return a *`ReaderHandle`* wrapping the capture pipe.

Note that while *start()* uses background threads to do IO, *reader()* does not, and it's the caller's responsibility to read the child's output promptly. Otherwise the child's stdout pipe buffer can fill up, causing the child to block and potentially leading to performance issues or deadlocks.

```
>>> reader = cmd("echo", "hi").reader()
>>> with reader:
...     reader.read()
b'hi\n'
```

run()

Execute the expression and return an *Output*, which includes the exit status and any captured output. Raise an exception if the status is non-zero.

```
>>> cmd("true").run()
Output(status=0, stdout=None, stderr=None)
```

start()

Start executing the expression and return a *Handle*.

Calling *start()* followed by *Handle.wait()* is equivalent to *run()*.

```
>>> handle = cmd("echo", "hi").stdout_capture().start()
>>> # Do some other stuff.
>>> handle.wait()
Output(status=0, stdout=b'hi\n', stderr=None)
```

Note that leaking a *Handle* without calling *Handle.wait()* will turn the children into zombie processes. In a long-running program, that could be serious resource leak.

stderr_capture()

Capture the standard error of the expression. The captured bytes become the `stderr` field of the returned *Output*.

```
>>> cmd("bash", "-c", "echo hi 1>&2").stderr_capture().run()
Output(status=0, stdout=None, stderr=b'hi\n')
```

stderr_file(*file_*)

Redirect the standard error of the expression to the supplied file. This works with any file-like object accepted by `Popen`, including raw file descriptors.

```
>>> f = open("/dev/null", "w")
>>> cmd("bash", "-c", "echo hi 1>&2").stderr_file(f).run()
Output(status=0, stdout=None, stderr=None)
```

stderr_null()

Redirect the standard error of the expression to `/dev/null`.

```
>>> cmd("bash", "-c", "echo hi 1>&2").stderr_null().run()
Output(status=0, stdout=None, stderr=None)
```

stderr_path(*path*)

Redirect the standard error of the expression to a file opened from the supplied filepath.

This works with strings, bytes, and pathlib `Path` objects.

```
>>> cmd("bash", "-c", "echo hi 1>&2").stderr_path("/tmp/outfile").run()
Output(status=0, stdout=None, stderr=None)
```

(continues on next page)

(continued from previous page)

```
>>> open("/tmp/outfile").read()
'hi\n'
```

stderr_to_stdout()
Merge the standard error of the expression with its stdout.

```
>>> bash_cmd = cmd("bash", "-c", "echo out && echo err 1>&2")
>>> bash_cmd.stderr_to_stdout().stdout_capture().stderr_capture().run()
Output(status=0, stdout=b'out\nerr\n', stderr=b'')
```

stdin_bytes(*buf*)
Redirect the standard input of the expression to a pipe, and write the supplied bytes to the pipe using a background thread.

This also accepts a string, in which case it converts any `\n` characters to `os.linesep` and encodes the result as UTF-8.

```
>>> cmd("cat").stdin_bytes(b"foo").read()
'foo'
```

stdin_file(*file_*)
Redirect the standard input of the expression to the supplied file. This works with any file-like object accepted by `Popen`, including raw file descriptors.

```
>>> f = open("/dev/zero")
>>> cmd("head", "-c10").stdin_file(f).read()
'\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00'
```

stdin_null()
Redirect the standard input of the expression to `/dev/null`.

```
>>> cmd("cat").stdin_null().read()
''
```

stdin_path(*path*)
Redirect the standard input of the expression to a file opened from the supplied filepath.

This works with strings, bytes, and pathlib `Path` objects.

```
>>> cmd("head", "-c10").stdin_path("/dev/zero").read()
'\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00'
```

stdout_capture()
Capture the standard output of the expression. The captured bytes become the `stdout` field of the returned *`Output`*.

```
>>> cmd("echo", "hi").stdout_capture().run()
Output(status=0, stdout=b'hi\n', stderr=None)
```

stdout_file(*file_*)
Redirect the standard output of the expression to the supplied file. This works with any file-like object accepted by `Popen`, including raw file descriptors.

```
>>> f = open("/dev/null", "w")
>>> cmd("echo", "hi").stdout_file(f).run()
Output(status=0, stdout=None, stderr=None)
```

stdout_null()
Redirect the standard output of the expression to `/dev/null`.

```
>>> cmd("echo", "hi").stdout_null().run()
Output(status=0, stdout=None, stderr=None)
```

stdout_path(*path*)
Redirect the standard output of the expression to a file opened from the supplied filepath.

This works with strings, bytes, and pathlib `Path` objects.

```
>>> cmd("echo", "hi").stdout_path("/tmp/outfile").run()
Output(status=0, stdout=None, stderr=None)
>>> open("/tmp/outfile").read()
'hi\n'
```

stdout_stderr_swap()
Swap the standard output and standard error of the expression.

```
>>> bash_cmd = cmd("bash", "-c", "echo out && echo err 1>&2")
>>> swapped_cmd = bash_cmd.stdout_stderr_swap()
>>> swapped_cmd.stdout_capture().stderr_capture().run()
Output(status=0, stdout=b'err\n', stderr=b'out\n')
```

stdout_to_stderr()
Merge the standard output of the expression with its stderr.

```
>>> bash_cmd = cmd("bash", "-c", "echo out && echo err 1>&2")
>>> bash_cmd.stdout_to_stderr().stdout_capture().stderr_capture().run()
Output(status=0, stdout=b'', stderr=b'out\nerr\n')
```

unchecked()
Prevent a non-zero exit status from raising a *StatusError*. The unchecked exit code will still be there on the *Output* returned by *run()*; its value doesn't change.

```
>>> cmd("false").run()
Traceback (most recent call last):
...
duct.StatusError: Expression cmd('false') returned non-zero exit status:␣
↪Output(status=1, stdout=None, stderr=None)
>>> cmd("false").unchecked().run()
Output(status=1, stdout=None, stderr=None)
```

"Uncheckedness" sticks to an exit code as it propagates up from part of a pipeline, but it doesn't "infect" other exit codes. So for example, if only one sub-expression in a pipe is *unchecked()*, then errors returned by the other side will still be checked.

```
>>> cmd("false").pipe(cmd("true")).unchecked().run()
Output(status=1, stdout=None, stderr=None)
>>> cmd("false").unchecked().pipe(cmd("true")).run()
Output(status=1, stdout=None, stderr=None)
>>> cmd("false").pipe(cmd("true").unchecked()).run()
Traceback (most recent call last):
...
duct.StatusError: Expression cmd('false').pipe(cmd('true').unchecked())␣
↪returned non-zero exit status: Output(status=1, stdout=None, stderr=None)
```

class duct.**Handle**

A handle representing one or more running child processes, returned by the *Expression.start()* method.

Note that leaking a *Handle* without calling *wait()* will turn the children into zombie processes. In a long-running program, that could be serious resource leak.

kill()

Send a kill signal to the child process(es). This is equivalent to `Popen.kill()`, which uses `SIGKILL` on Unix. After sending the signal, wait for the child to finish and free the OS resources associated with it. If the child has already been waited on, this has no effect.

This function does not return an *Output*, and it does not raise *StatusError*. However, subsequent calls to *wait()* or *try_wait()* are likely to raise *StatusError* if you didn't use *Expression.unchecked()*.

```
>>> handle = cmd("sleep", "1000").start()
>>> handle.kill()
```

pids()

Return the PIDs of all the running child processes. The order of the PIDs in the returned list is the same as the pipeline order, from left to right.

try_wait()

Check whether the child process(es) have finished, and if so return an *Output* containing the exit status and any captured output. If the child has exited, this frees the OS resources associated with it.

```
>>> handle = cmd("sleep", "1000").unchecked().start()
>>> assert handle.try_wait() is None
>>> handle.kill()
>>> handle.try_wait()
Output(status=-9, stdout=None, stderr=None)
```

wait()

Wait for the child process(es) to finish and return an *Output* containing the exit status and any captured output. This frees the OS resources associated with the child.

```
>>> handle = cmd("true").start()
>>> handle.wait()
Output(status=0, stdout=None, stderr=None)
```

class duct.**ReaderHandle**

A stdout reader that automatically closes its read pipe and awaits child processes once EOF is reached.

This inherits from `io.IOBase`, and you can call *read()* and related methods like `readlines()` on it. When *ReaderHandle* is used as a context manager with the `with` keyword, context exit will automatically call *close()*.

Note that if you don't read to EOF, and you don't call *close()* or use a `with` statement, then the child will become a zombie. Using a `with` statement is recommended for exception safety.

If one thread is blocked on a call to *read()*, then calling *kill()* from another thread is an effective way to unblock the reader. However, note that killed child processes return a non-zero exit status, which turns into an exception for the reader by default, unless you use *Expression.unchecked()*.

close()

Close the read pipe and call *kill()* on the inner *Handle*.

ReaderHandle is a context manager, and if you use it with the *with* keyword, context exit will automatically call *close()*. Using a `with` statement is recommended, for exception safety.

```
>>> reader = cmd("echo", "hi").reader()
>>> reader.close()
```

kill()
Call *kill()* on the inner *Handle*.

This function does not raise *StatusError*. However, subsequent calls to *read()* are likely to raise *StatusError* if you didn't use *Expression.unchecked()*.

```
>>> reader = cmd("bash", "-c", "echo hi && sleep 1000000").unchecked().
↪reader()
>>> with reader:
...     reader.read(3)
...     reader.kill()
...     reader.read()
b'hi\n'
b''
```

pids()
Return the PIDs of all the running child processes. The order of the PIDs in the returned list is the same as the pipeline order, from left to right.

read(*size=-1*)
Read bytes from the child's standard output. Because *ReaderHandle* inherits from `io.IOBase`, related methods like `readlines()` are also available.

```
>>> reader = cmd("printf", r"a\nb\nc\n").reader()
>>> with reader:
...     reader.read(2)
...     reader.readlines()
b'a\n'
[b'b\n', b'c\n']
```

If *read()* reaches EOF and awaits the child, and the child exits with a non-zero status, and *Expression.unchecked()* was not used, *read()* will raise a *StatusError*.

```
>>> with cmd("false").reader() as reader:
...     reader.read()
Traceback (most recent call last):
...
duct.StatusError: Expression cmd('false').stdout_capture() returned non-zero␣
↪exit status: Output(status=1, stdout=None, stderr=None)
```

try_wait()
Check whether the child process(es) have finished, and if so return an *Output* containing the exit status and any captured output. This is equivalent to *Handle.try_wait()*.

Note that the `stdout` field of the returned *Output* will always be `None`, because the *ReaderHandle* itself owns the child's stdout pipe.

```
>>> input_bytes = bytes([42]) * 1000000
>>> reader = cmd("cat").stdin_bytes(input_bytes).reader()
>>> with reader:
...     assert reader.try_wait() is None
...     output_bytes = reader.read()
...     assert reader.try_wait() is not None
...     assert input_bytes == output_bytes
```

class duct.**Output**

The return type of *Expression.run()* and *Handle.wait()*. It carries the public fields `status`, `stdout`, and `stderr`. If *Expression.stdout_capture()* and `Expression:stderr_capture()` aren't used, `stdout` and `stderr` respectively will be `None`.

```
>>> cmd("bash", "-c", "echo hi 1>&2").stderr_capture().run()
Output(status=0, stdout=None, stderr=b'hi\n')
```

class duct.**StatusError**

The exception raised by default when a child exits with a non-zero exit status. See *Expression.unchecked()* for suppressing this. If the exception is caught, the `output` field contains the *Output*.

```
>>> from duct import StatusError
>>> try:
...     cmd("bash", "-c", "echo hi 1>&2 && false").stderr_capture().run()
... except StatusError as e:
...     e.output
Output(status=1, stdout=None, stderr=b'hi\n')
```

Python Module Index

d

Index