Blah, blah, [distributable, testable, maintainable] "scripts"

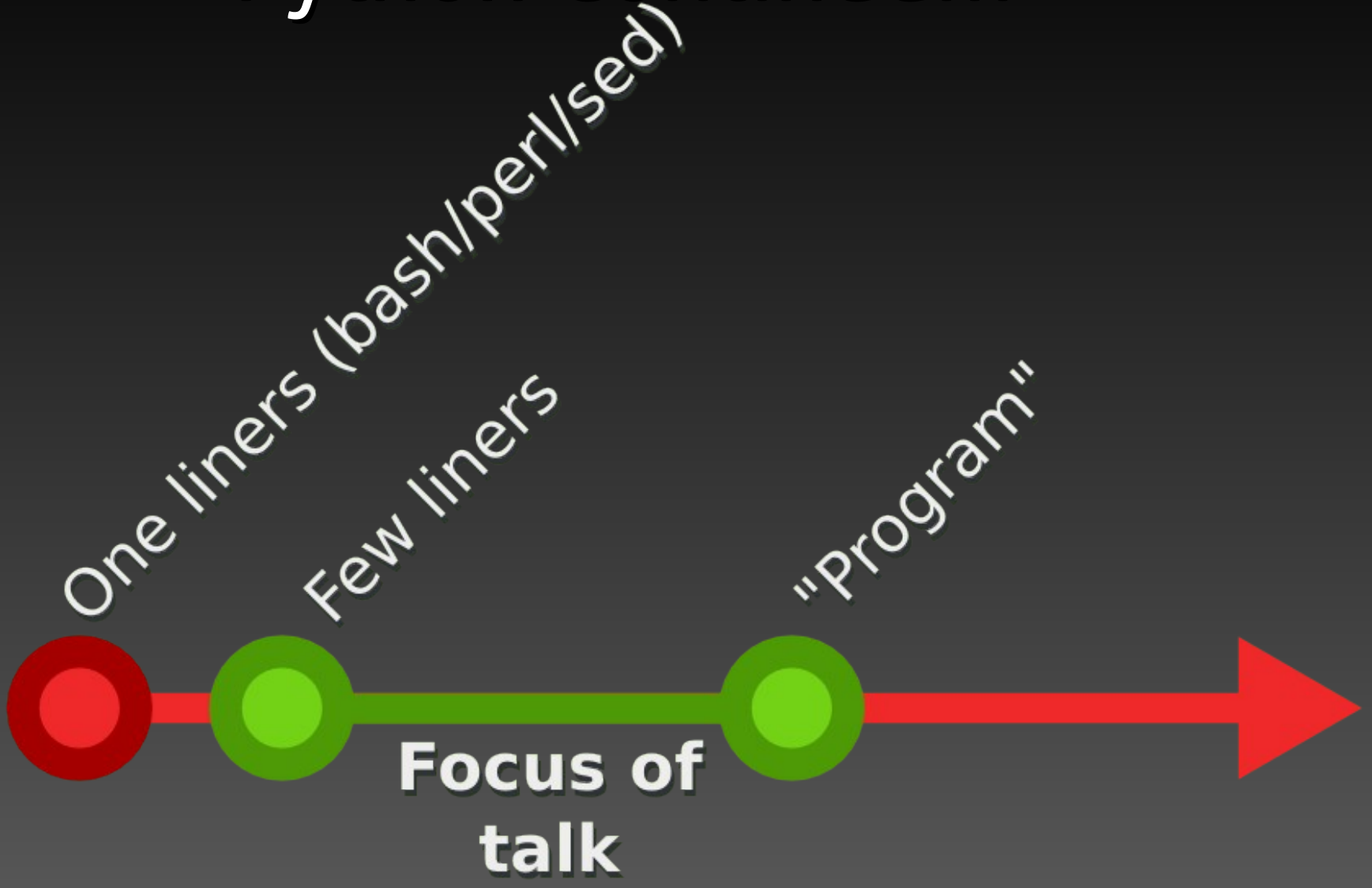
Matt Harrison
PYCON 2009

Beginner level

Cliff's Notes

See handout and `poachplate`.

Python Continuum



Focus on `stdlib` and pure python

```
>>> import this
    The Zen of Python, by Tim Peters
    There should be one-- and preferably
only one --obvious way to do it.
```

Problem: Some obvious things aren't obvious

Solution: Patterns

Scripting patterns in python

1 Compromise Layout

Scrapage/Packript

.py (module/package) vs *script*

PATH vs PYTHONPATH

PATH

- executable
- not (usually) importable

PATH vs PYTHONPATH

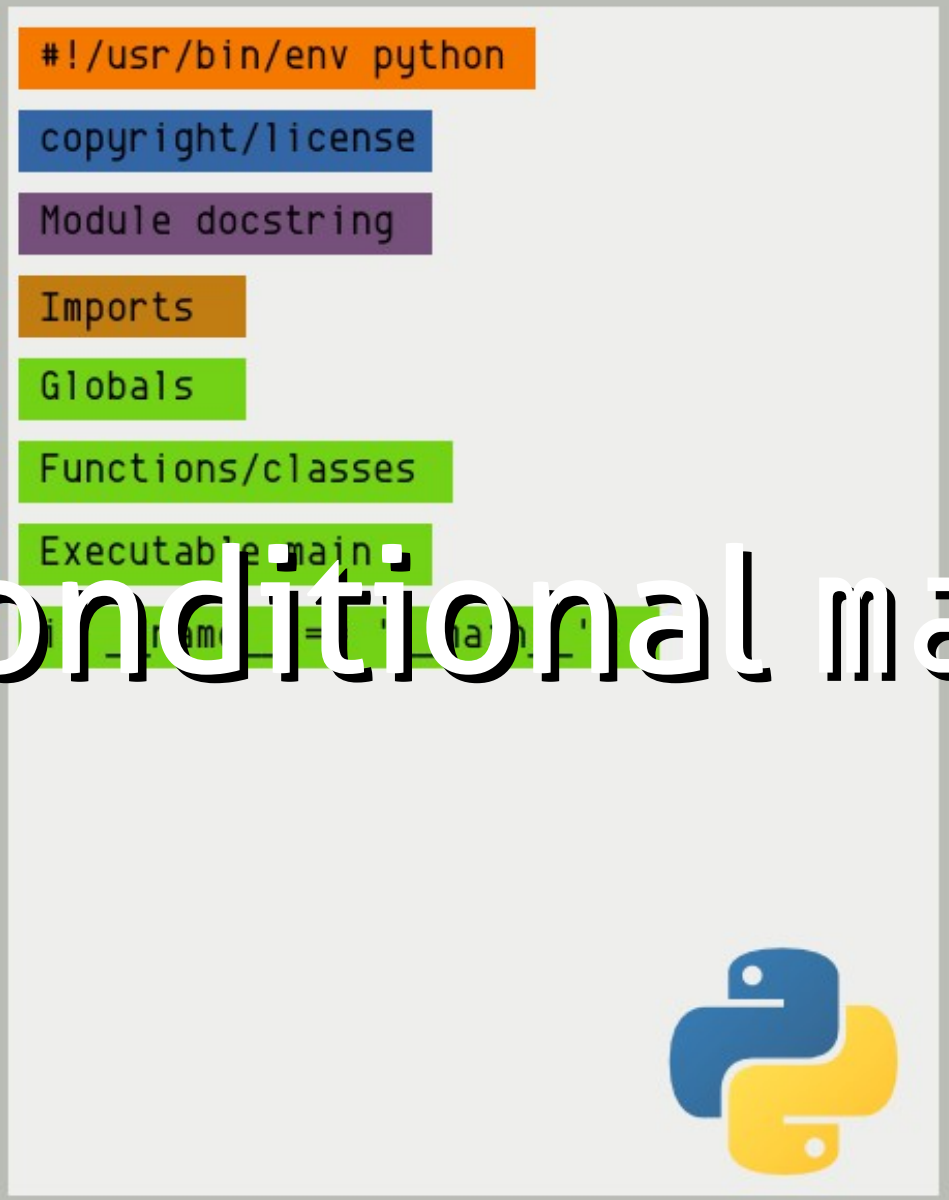
PYTHONPATH

- not (usually) executable
- importable

Compromise layout

Layout:

```
Project/
  bin/
    script(.py) (thin wrapper)
  scriptlib/
    __init__.py
    scriptimpl.py
  setup.py
```



2 Conditional main

Bad code

```
>>> lines = open(sys.argv[1]).readlines()
>>> for line in lines:
...     ..
>>> # More stuff
>>> # Other stuff
>>> # No functions
>>> sys.exit()
```

Good practices

- No side effects during import
- Globals are 'bad'
- Logic in grokable chunks (ie functions)

Conditional `main`

```
>>> # imports
>>> # classes/functions
>>> def main(prog_args):
...     # process args
...     # execute functions
...     # return exit code
>>> if __name__ == '__main__':
...     sys.exit(main(sys.argv))
```

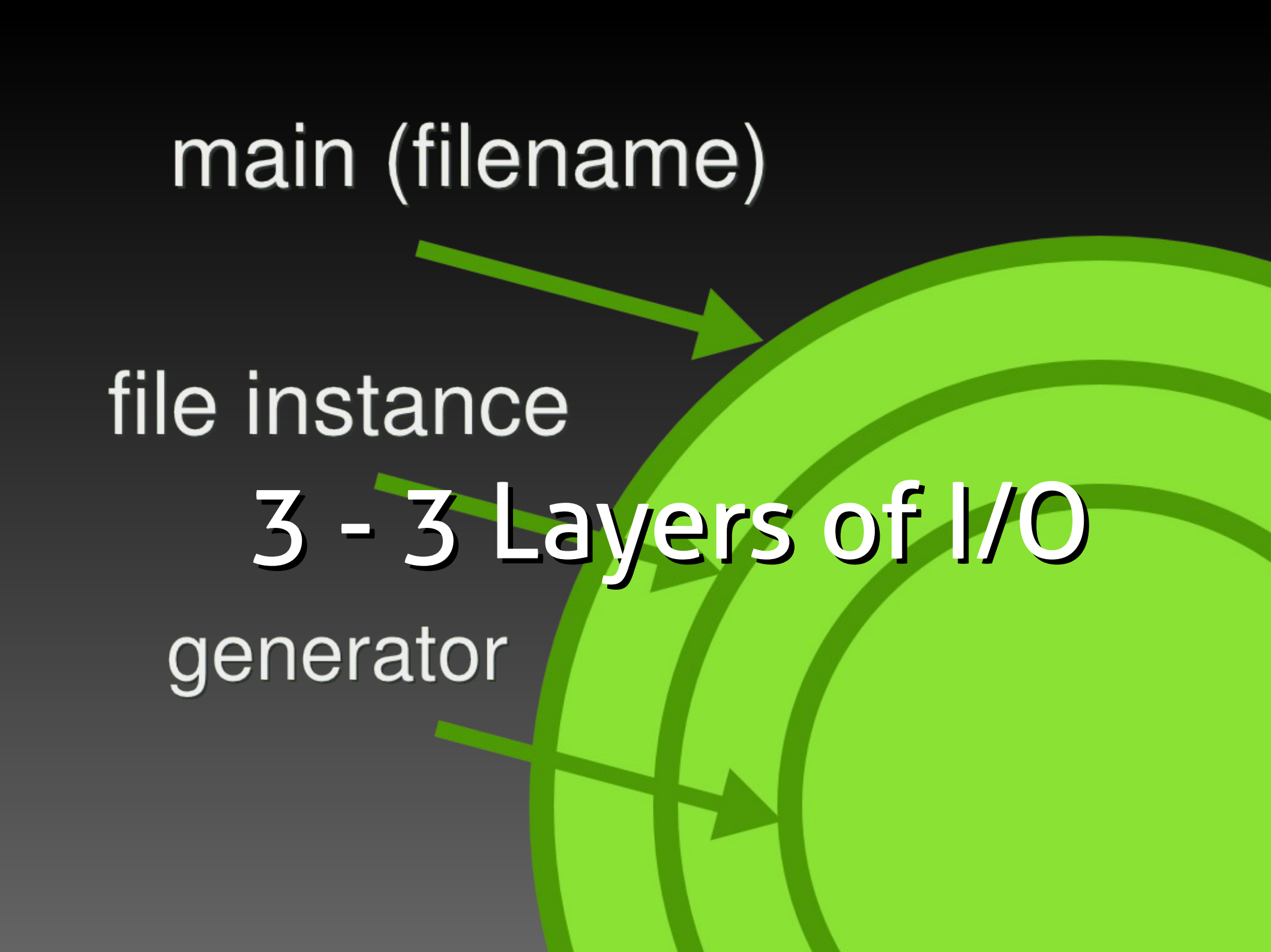
sys.exit

Limit use

- *0* - Sucess
- *Non-Zero* - Error

Results

- Modular code
- Testable chunks
- Code can be imported/reused
- Easier to modify


main (filename)
file instance
3 - 3 Layers of I/O
generator

What interface?

Common to deal with files, should the interface take:

- filenames?
- file instances?
- generators?

YES!

3 Layered interface

- `main` - filename
- file-like
- generator

`main`

- accepts filenames (defaults to stdin/stdout)
- Do file exception handling here
- Do `close` of files

file-like

Can take `open()`, `sys.stdin`, `StringIO`...

- Testing is easier

Generator

- Efficient

(Also use when dealing with dbs)

3 layers

```
>>> def gen_cat(line_iter):
...     for line in line_iter:
...         # business logic
...         yield line
>>> def file_cat(fin, fout):
...     for line in gen_cat(fin):
...         fout.write(line)
```

3 layers (cont)

```
>>> def main(pargs):
...     # optparse blah blah...
...     fin = sys.stdin
...     if opt.fin:
...       fin = open(opt.fin)
...     fout = sys.stdout
...     file_cat(fin, fout)
```

Testing generator

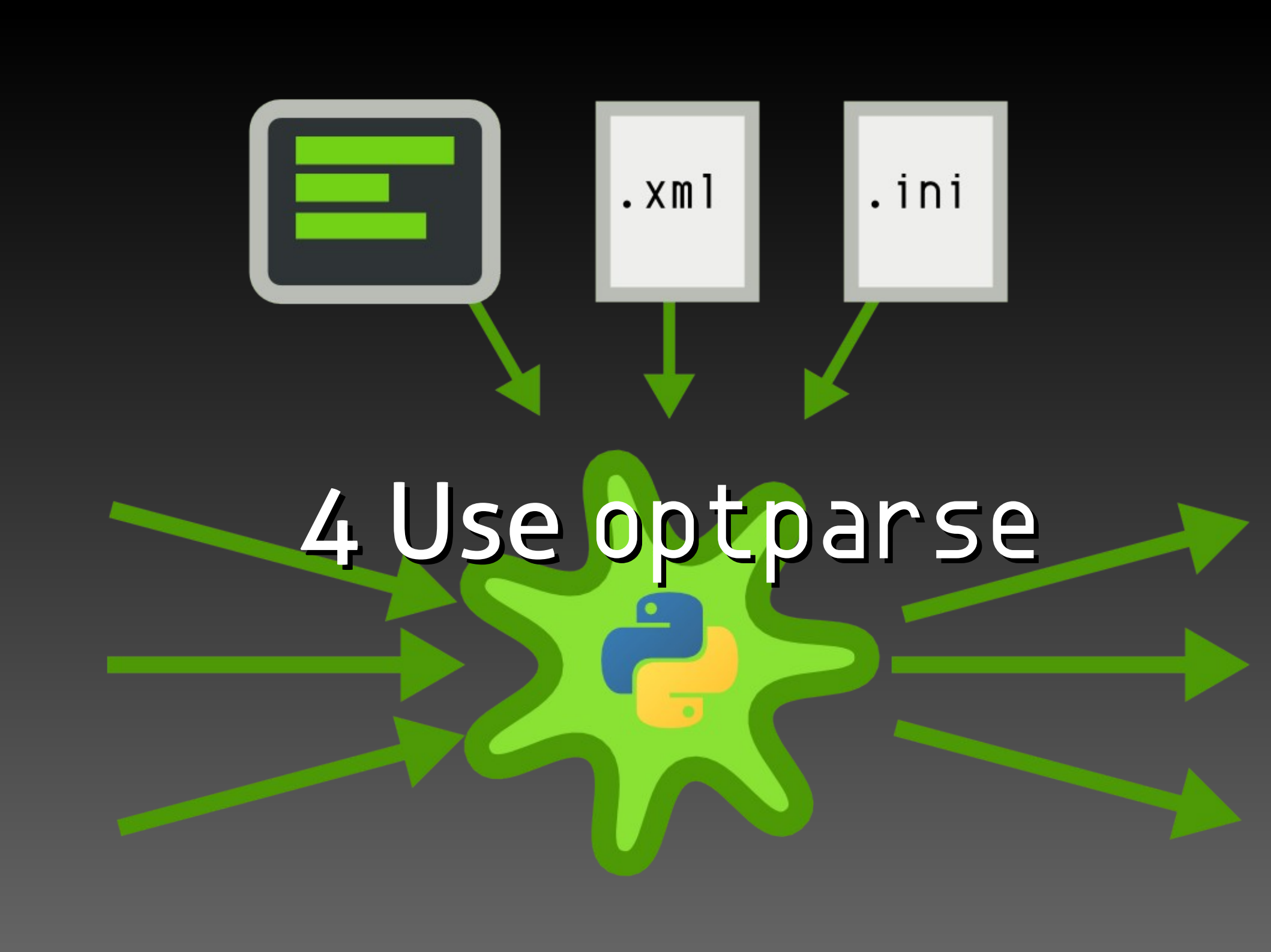
```
>>> list(gen_cat(['foo\n','bar\n']))
['foo\n', 'bar\n']
```

Testing file

```
>>> import StringIO
>>> fout = StringIO.StringIO()
>>> file_cat(StringIO.StringIO('foo\nbar\
n'), fout)
>>> print fout.getvalue()
    foo
    bar
```

Testing filename

```
>>> main(['--fin', '/tmp/foo', '--fout',
'/tmp/out'])
>>> print open('/tmp/out/').read()
    foo
    bar
```



4 Use optparse

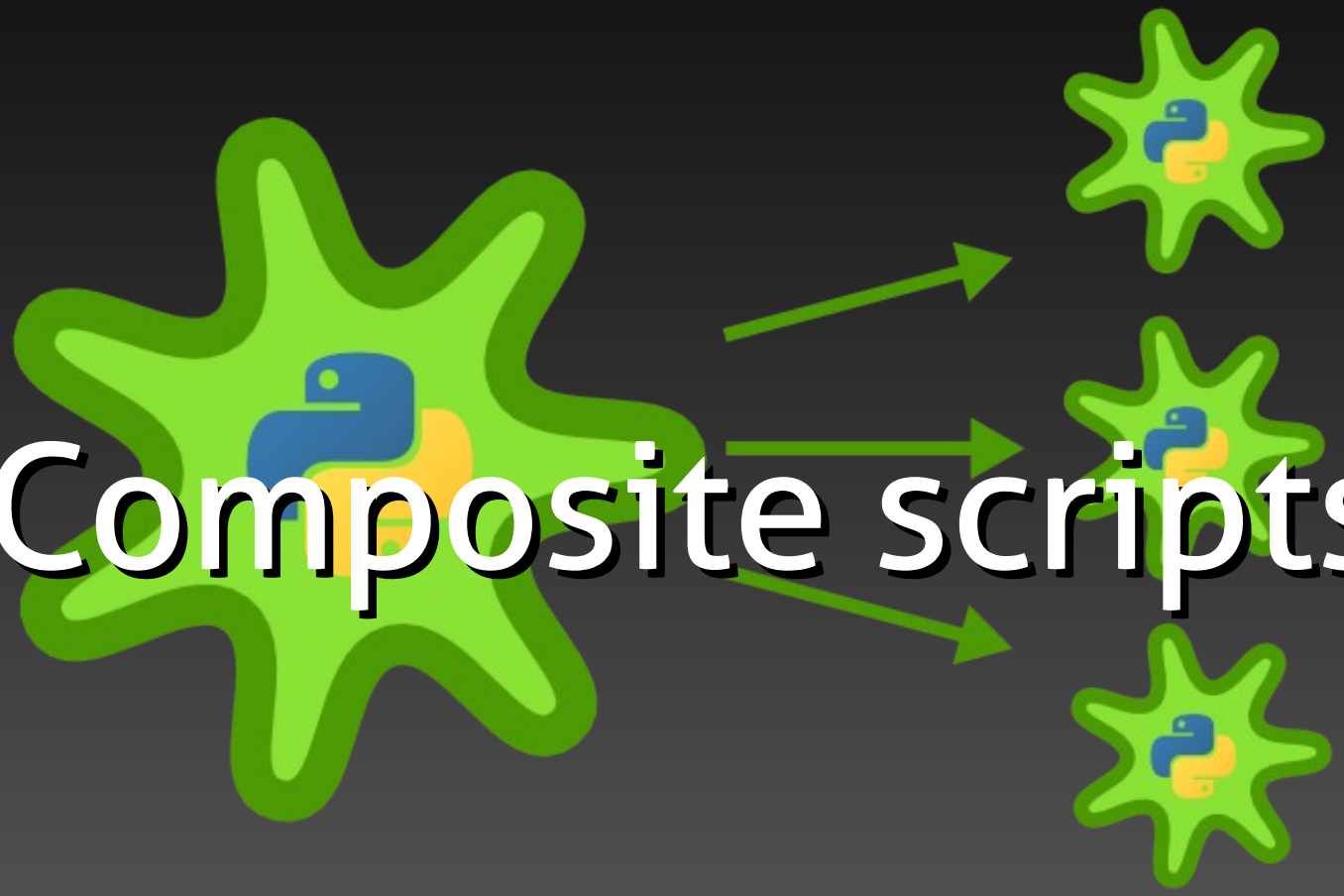
Commandline parsing options

- manual
- getopt
- optparse

`optparse` benefits

- Nice usage (*--help*)
- Provides *--version*

5 Composite scripts



Composite scripts

Series of commands:

```
git
  add
  bisect
  branch
  ...
```

Composite scripts

`git` style - wrapper script dispatches to "subscripts" `main`

```
>>> def main(pargs): # pargs =
['script.py', 'status', '--some-option']
...     if pargs[1] == 'status':
...         status.main(pargs[2:])
```

Bonus

If you want to have scripts support this, you get it for free from complying with *Conditional main* and *Use optparse*


print
6 Leave no trace

No `print`

How will I debug?

Use `logging`

logging boilerplate

```
>>> LOGFILE = os.path.expanduser('~/.script.log')
>>> logger = logging.getLogger('ScriptLogger')
>>> logger.setLevel(logging.DEBUG)
>>> handler = handlers.RotatingFileHandler(LOGFILE, maxBytes=500, backupCount=2)
>>> log_format = Formatter("%(asctime)s - %(name)s - %(levelname)s - %(message)s")
>>> handler.setFormatter(log_format)
>>> logger.addHandler(handler)
```

`atexit` is also your cleaning friend

Benefits

- Using 3 layers
- You'll have (proper) logging


7 Great artists steal
setup.py

setup.py

Non-pattern: Testing

Side note

Code reviews are usually more effective than testing

Figure out how to test

- None
- Manually
- Automated
  - unittest style
  - doctest
  - input/output checking

Testing is easier with well structured code

Globals make testing hard

No testing makes refactoring hard

No testing/refactoring · >

- crappy code
- harder to add features

crappy code -> unhappy co-workers

poachplate

poachplate

- Compromise Layout
- Conditional `main`
- Theft Packaging

handout

- Verbose file organization
- support for Unix configuration hierarchy
- `tempfile`
- Script chaining
- pid file
- `logging`

Thanks:

- docutils
- OOo
- inkscape
- pygments

Handout at http://panela.blog-city.com/