Advanced Python modules and packages

Reuven M. Lerner, PhD
reuven@lerner.co.il

File globbing

- Use the “glob” module, which returns a list of files matching the pattern

```
glob.glob("a*")

glob.glob("/etc/a*")
```

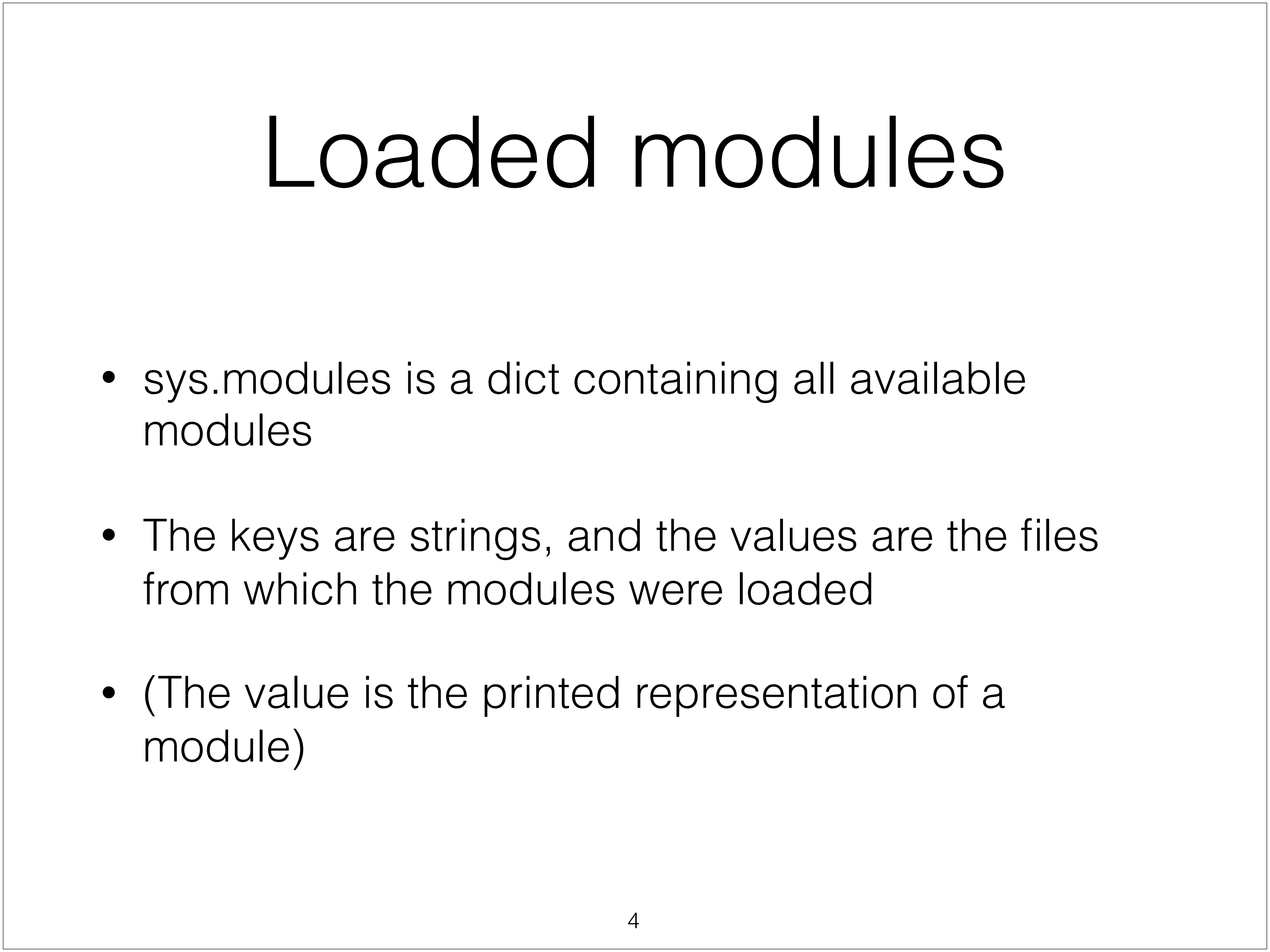
Reloading modules

- Only the first call to “import” has any effect
- Use “reload” to reload a module
- import is a statement, reload is a function:

```
reload(sys)
```

- Not:

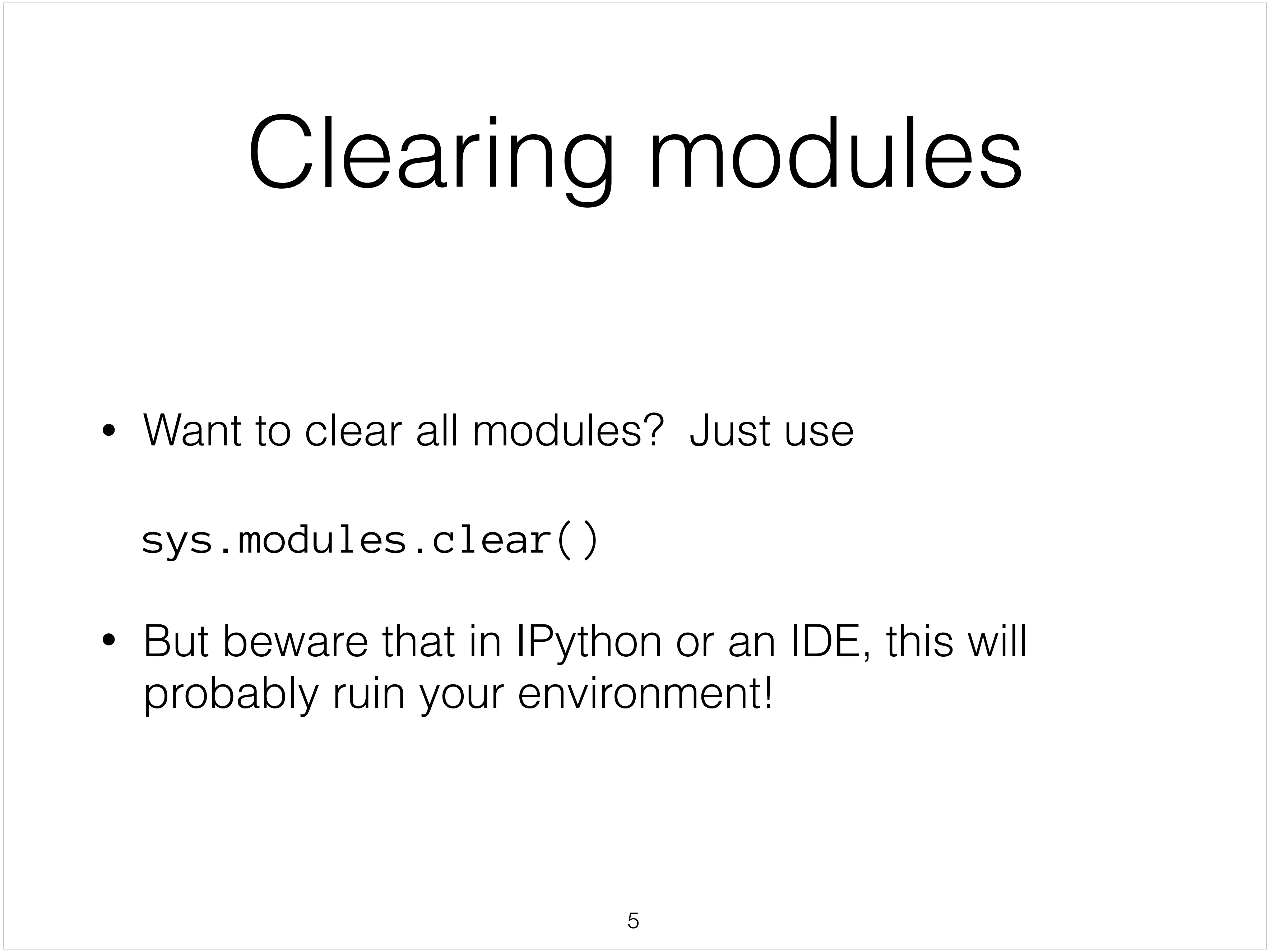
```
reload sys  # Syntax error
```



Loaded modules

- sys.modules is a dict containing all available modules
- The keys are strings, and the values are the files from which the modules were loaded
- (The value is the printed representation of a module)

4



Clearing modules

- Want to clear all modules? Just use

```
sys.modules.clear()
```

- But beware that in IPython or an IDE, this will probably ruin your environment!

A nicer approach

```
#!/usr/bin/env python
import sys
if globals().has_key('init_modules'):
    for m in sys.modules:
        if x not in init_modules:
            del(sys.modules[m])
else:
    init_modules = sys.modules.keys()
```

Globals and locals

- Two functions, globals() and locals(), return dictionaries
- These dictionaries contain the names and values of the current global and local state
- Very useful when debugging, looking at the current state, or trying to understand scoping issues

__builtin__

- The __builtin__ module always exists in Python, and works like any other module
- If you have (for some crazy reason) overwritten a built-in type, you can always use it by explicitly saying __builtin__.list (or whatever you overwrote)

__main__

- The __main__ module gives you access to the global namespace

```
import __main__
```

- Not the sort of thing you need every day, but it can be useful if you want to manipulate variables in __main__ without the "global" statement

__import__

- __import__ is the function behind the “import” statement
- You can say

```
__import__('os')
```

- This function takes optional parameters:

```
__import__( name[, globals[, locals[,
fromlist[, level]]]])
```

__future__

- Starting in Python 2.6, you can:

```
from __future__ import print_function
```

  - Now print is a function!

```
from __future__ import division
```

  - Now division works as you might expect!

operator

- This module provides all of Python's standard operators as functions
- So you can say

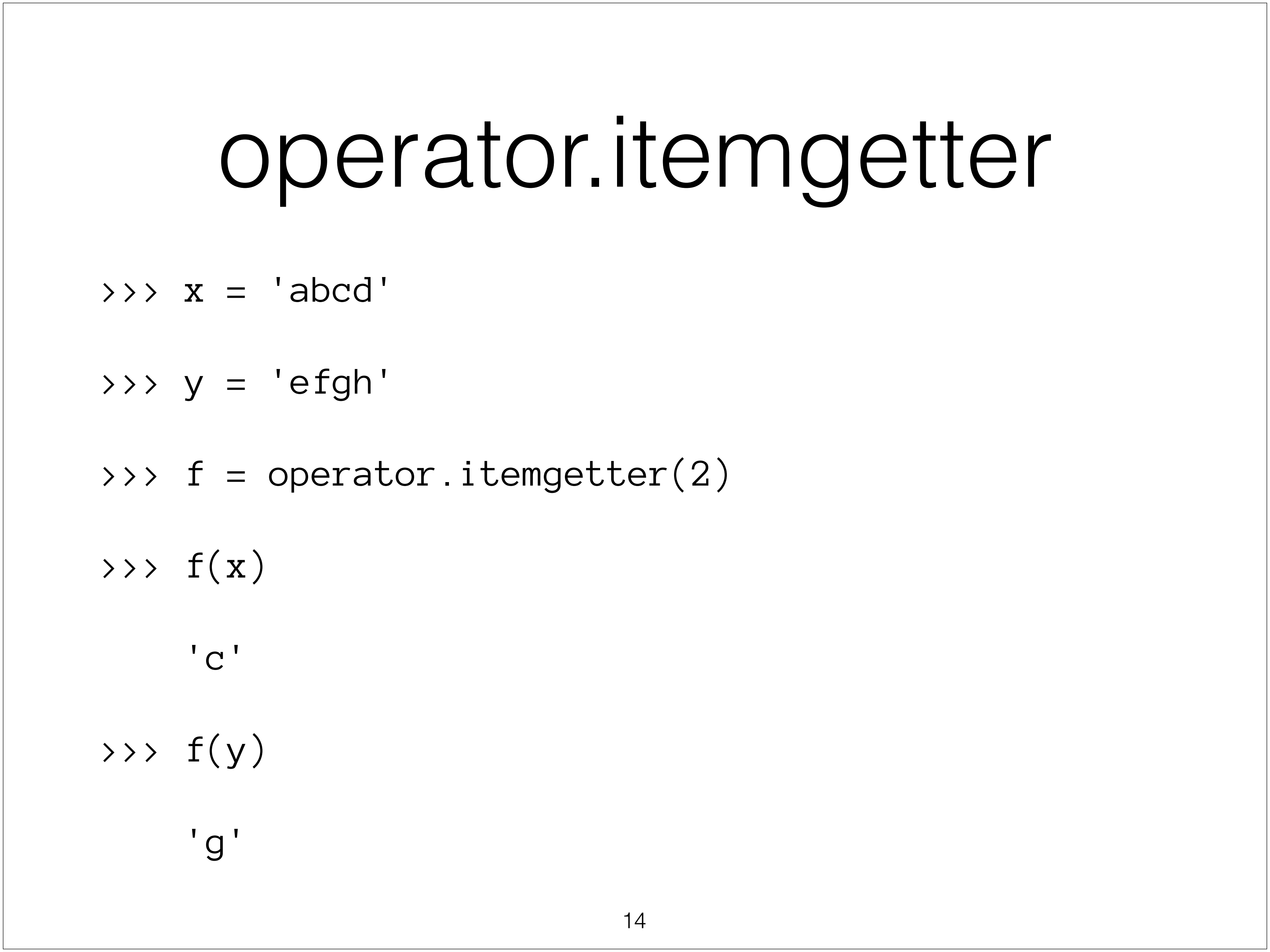
```
>>> import operator

>>> operator.add(1, 1)

    2

>>> operator.add('1', '1')

    '11'
```

Why operator?

- You could say:

```
def add(x,y):
   return type(x).__add__(x, y)
```

- But it's easier to use the module, which gives you all of Python's built-in operators

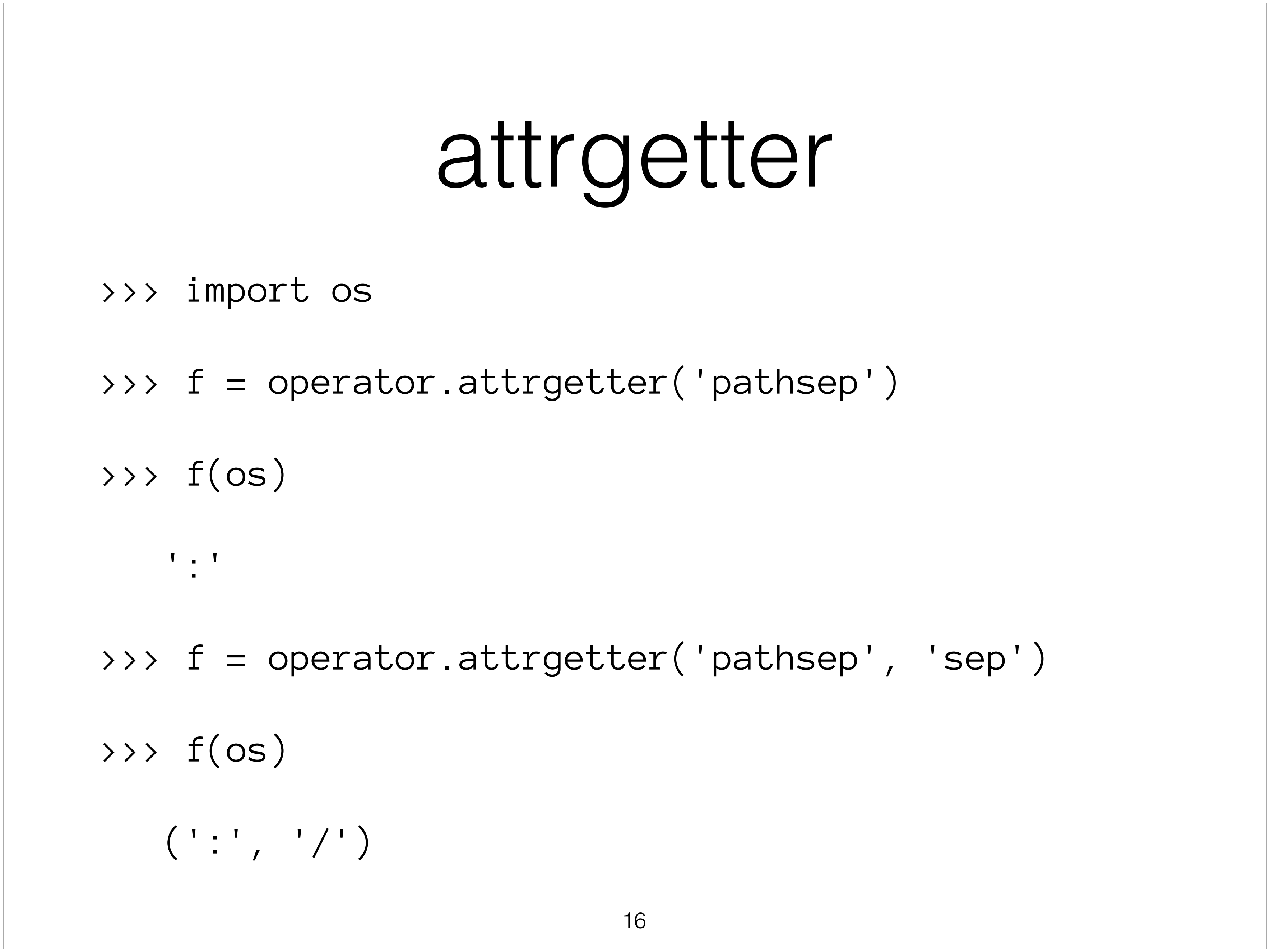


operator.itemgetter

```
>>> x = 'abcd'

>>> y = 'efgh'

>>> f = operator.itemgetter(2)

>>> f(x)

    'c'

>>> f(y)

    'g'
```

itemgetter and slices

```
>>> x = 'abcd'

>>> y = 'efgh'

>>> f = operator.itemgetter(0,1,3)

>>> f(x)

    ('a', 'b', 'd')

>>> f(y)

    ('e', 'f', 'h')
```



attrgetter

```
>>> import os

>>> f = operator.attrgetter('pathsep')

>>> f(os)

   ':'

>>> f = operator.attrgetter('pathsep', 'sep')

>>> f(os)

   (':', '/')
```

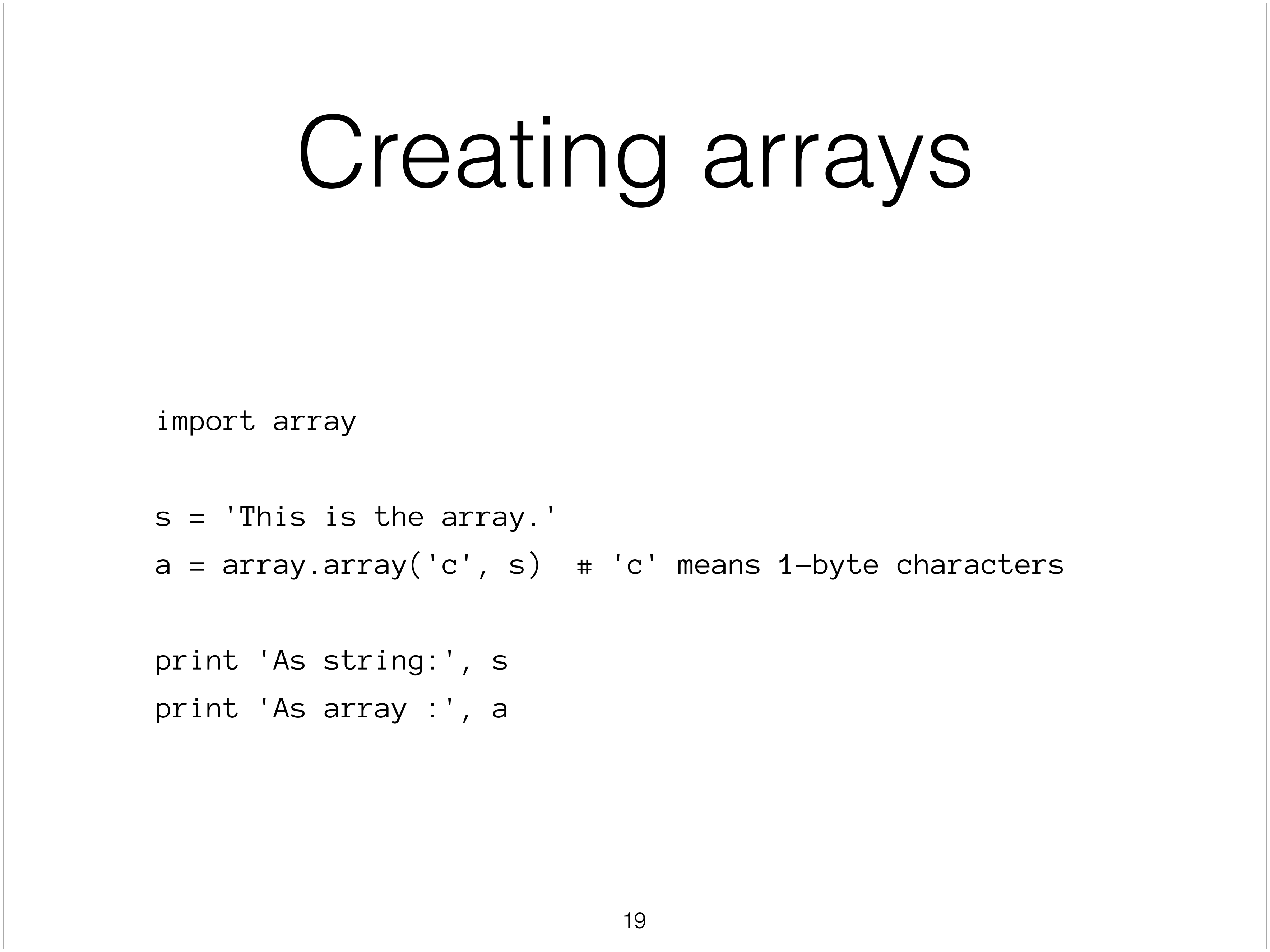
methodcaller

```
>>> x = 'abcd'

>>> y = 'efgh'

>>> f = operator.methodcaller('upper')

>>> f(x)

    'ABCD'

f(y)

    'EFGH'
```

array

- Similar to a list, but all of the elements need to be of the same type
- Any primitive type can be stored in an array
- We initialize the array with a type indicator (a one-character string), as well as the initial values

```
array('i', [1,2,3,4])

array('u', 'שלום')
```



Creating arrays

```
import array

s = 'This is the array.'
a = array.array('c', s)  # 'c' means 1-byte characters

print 'As string:', s
print 'As array :', a
```

Provides many list operations

```
append

extend

insert

remove

reverse

pop
```

Special methods

```
a = array('c', 'hello')

a.tostring()      # return a string

a.from_list('i', range(10))

a.tolist()        # List of characters
```

So, why arrays?

- The values are stored directly in the array. Python lists normally store values in objects, to which the array points

- Arrays use much less memory than lists — but their performance isn't that much better

- Arrays aren't designed for doing lots of math. For that, you should use NumPy.

random

- A very useful module is “random”. Some examples of what it can do:

```
random.randint(1,10)

    7

random.sample(['a', 'b', 'c', 'd'], 2)

    ['c', 'd']
```

Distributing Python packages

- Decide on a name for your package
- Create two nested directories with this name (e.g., “foo/foo”).
    - Top-level directory is for version control
- Start with a basic package, with __init__.py and the code within it.
- And then there’s setup.py...

setup.py

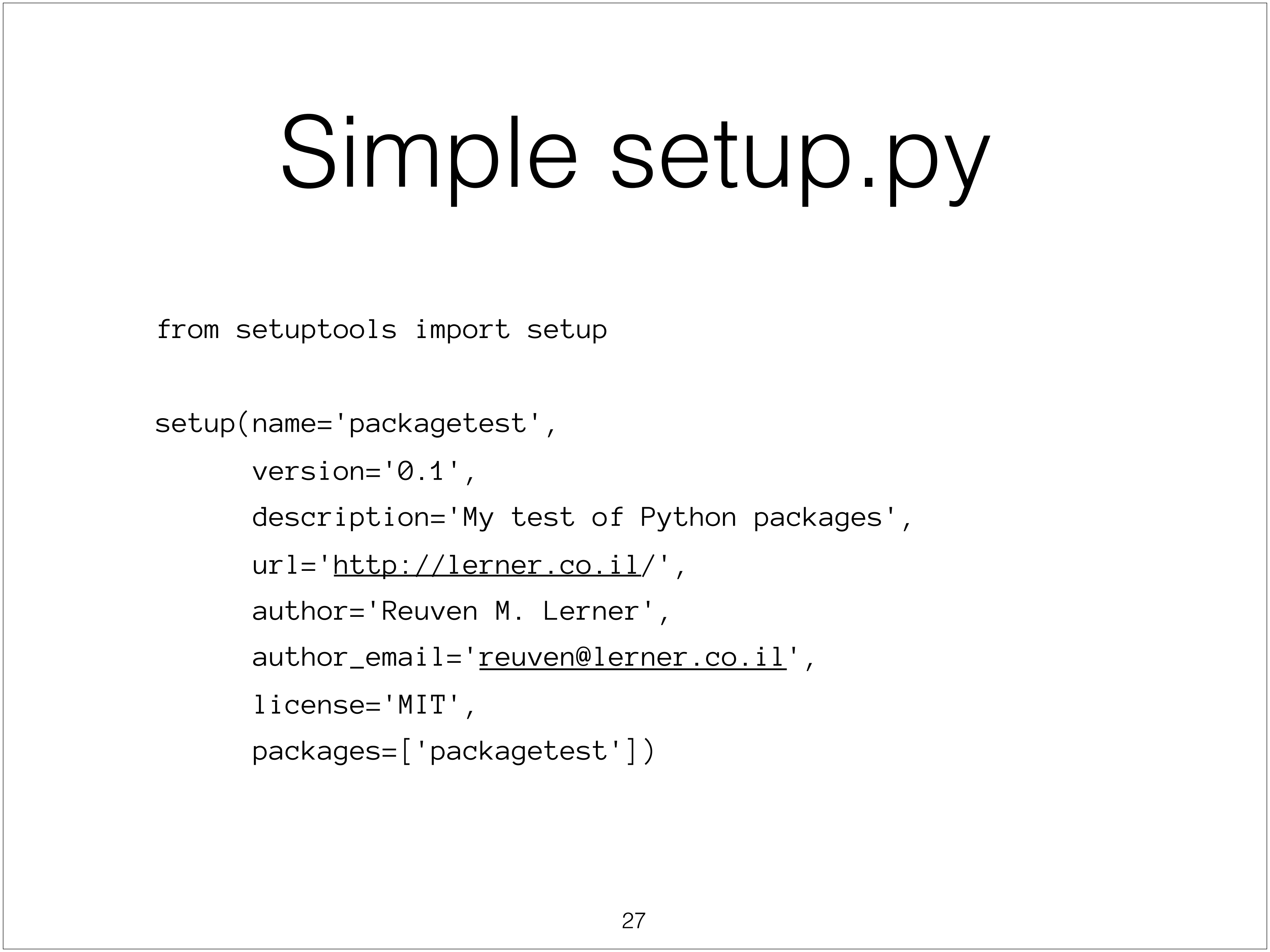
- Configuration file, written in Python, that describes the package you’re creating and distributing
- First line is always

```
from setuptools import setup
```

- Then you invoke the setup() function

setup()

- Function takes many keyword parameters, a large number of them optional
- name: name of the package
- version: version number (string)
- url, author, author_email, license
- packages: list of dependencies, including itself



Simple setup.py

```
from setuptools import setup

setup(name='packagetest',
      version='0.1',
      description='My test of Python packages',
      url='http://lerner.co.il/',
      author='Reuven M. Lerner',
      author_email='reuven@lerner.co.il',
      license='MIT',
      packages=['packagetest'])
```

Dependencies

- Dependencies are named in the “packages” and “install_requires” arguments to setup()
- packages is a list of package names provided; you can use find_packages(‘src’) from setuptools to handle this
- install_requires takes a list of strings, in the form NAME=VERSION

Learning more

- http://pythonhosted.org/distribute/setuptools.html

Installing our package

- Now we can go into our directory, and just as with a package we’ve downloaded manually, we can type:

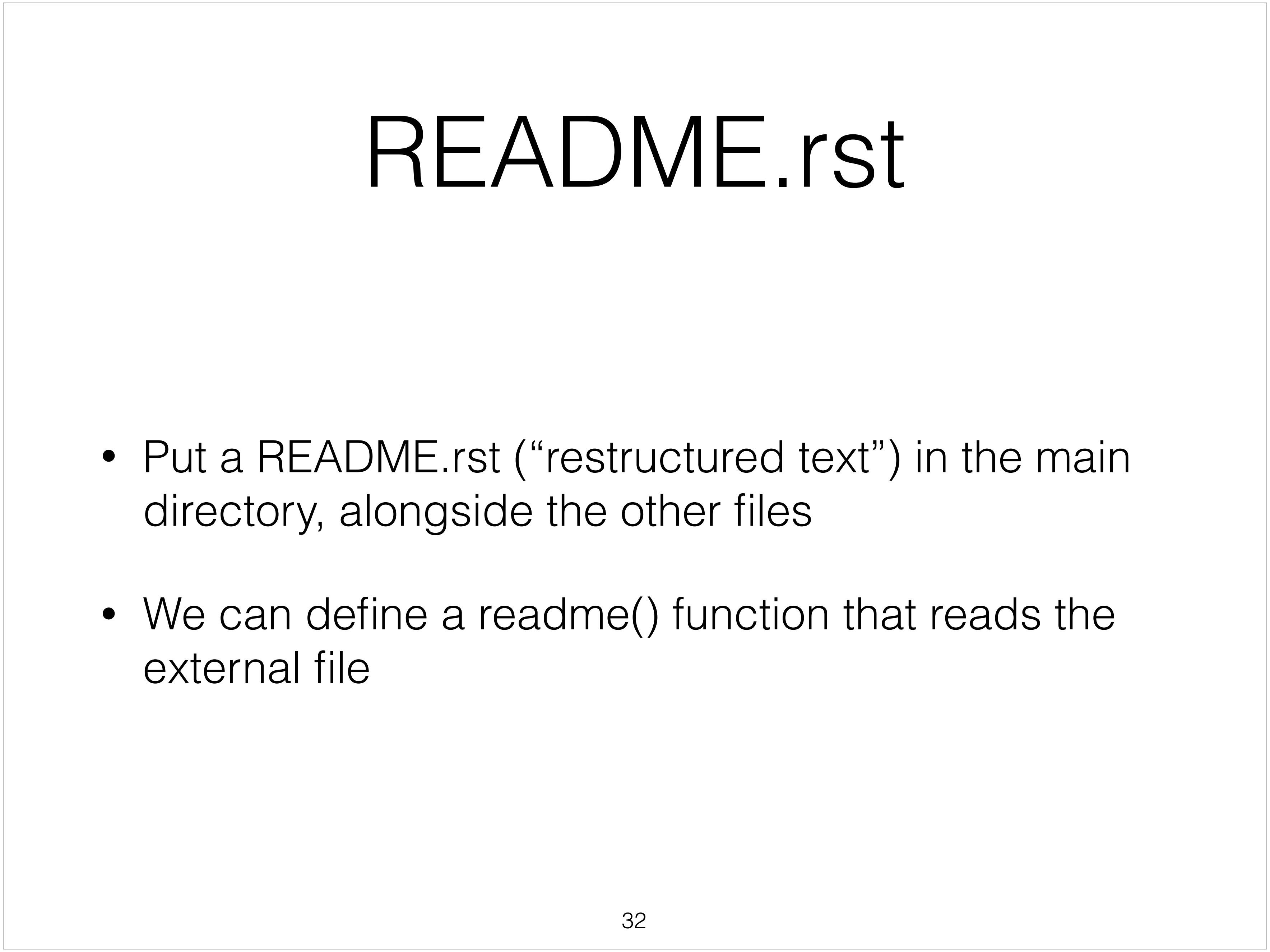
```
python setup.py install
```

MANIFEST.in

- Python packaging tools expect a MANIFEST.in file, describing which non-Python files should be included

- Each line can look like:

```
include filename.txt

include doc/*.txt

include *.rst
```



README.rst

- Put a README.rst ("restructured text") in the main directory, alongside the other files
- We can define a readme() function that reads the external file

Updated setup.py

```
from setuptools import setup

def readme():
    return open('README.rst').read()

setup(name='packagetest',
      version='0.1',
      description='My test of Python packages',
      long_description=readme(),
      url='http://lerner.co.il/',
      author='Reuven M. Lerner',
      author_email='reuven@lerner.co.il',
      license='MIT',
      packages=['packagetest'])
```

Eggs

- Sort of like Java Jarfiles
- Not obvious to me if they're still used that frequently
- About the format: http://peak.telecommunity.com/DevCenter/EggFormats

Simple command-line arguments

- Use argv list in sys module

```
import sys

for i,param in enumerate(sys.argv):

    print "[%d] %s" % (i, param)
```