Modules

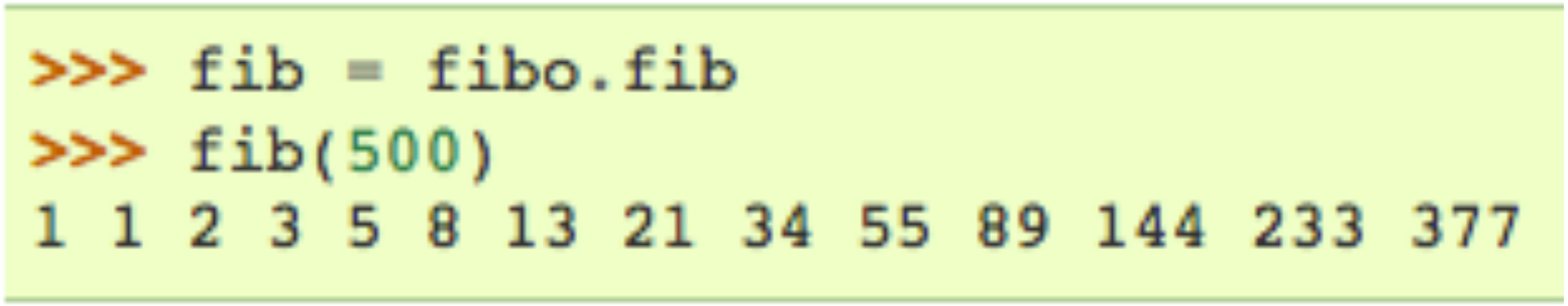
- A module is a file containing Python definitions and statements.
- The file name is the module name with the suffix .py appended.
- Within a module, the module’s name (as a string) is available as the value of the global variable __name__.

```
>>> import fibo
```

```
# Fibonacci numbers module

def fib(n):    # write Fibonacci series up to n
    a, b = 0, 1
    while b < n:
        print b,
        a, b = b, a+b

def fib2(n): # return Fibonacci series up to n
    result = []
    a, b = 0, 1
    while b < n:
        result.append(b)
        a, b = b, a+b
    return result
```

```
>>> fibo.fib(1000)
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
>>> fibo.fib2(100)
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
>>> fibo.__name__
'fibo'
```

Assign Local Name



```
>>> fib = fibo.fib
>>> fib(500)
1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

```
>>> from fibo import fib, fib2
>>> fib(500)
1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

```
>>> from fibo import *
>>> fib(500)
1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

Compiled Python Files

- As an important speed-up of the start-up time for short programs that use a lot of standard modules,
- if a file called spam.pyc exists in the directory where spam.py is found, this is assumed to contain an already-“byte-compiled” version of the module spam.
- The modification time of the version of spam.py used to create spam.pyc is recorded in spam.pyc, and the .pyc file is ignored if these don’t match.

Compiled Python Files

- Normally, you don’t need to do anything to create the spam.pyc file.
- Whenever spam.py is successfully compiled, an attempt is made to write the compiled version to spam.pyc.
- It is not an error if this attempt fails; if for any reason the file is not written completely, the resulting spam.pyc file will be recognized as invalid and thus ignored later.
- The contents of the spam.pyc file are platform independent, so a Python module directory can be shared by machines of different architectures.

Some Tips for Experts

- When the Python interpreter is invoked with the -O flag, optimized code is generated and stored in .pyo files.
- The optimizer currently doesn’t help much; it only removes assert statements.
- When -O is used, all bytecode is optimized; .pyc files are ignored and .py files are compiled to optimized bytecode.

Some Tips for Experts

- Passing two -O flags to the Python interpreter (-OO) will cause the bytecode compiler to perform optimizations that could in some rare cases result in malfunctioning programs.
- Currently only __doc__ strings are removed from the bytecode, resulting in more compact .pyo files.
- Since some programs may rely on having these available, you should only use this option if you know what you’re doing.

Some Tips for Experts

- A program doesn’t run any faster when it is read from a .pyc or .pyo file than when it is read from a .py file; the only thing that’s faster about .pyc or .pyo files is the speed with which they are loaded.

Some Tips for Experts

- When a script is run by giving its name on the command line, the bytecode for the script is never written to a .pyc or .pyo file.
- Thus, the startup time of a script may be reduced by moving most of its code to a module and having a small bootstrap script that imports that module. It is also possible to name a .pyc or .pyo file directly on the command line.

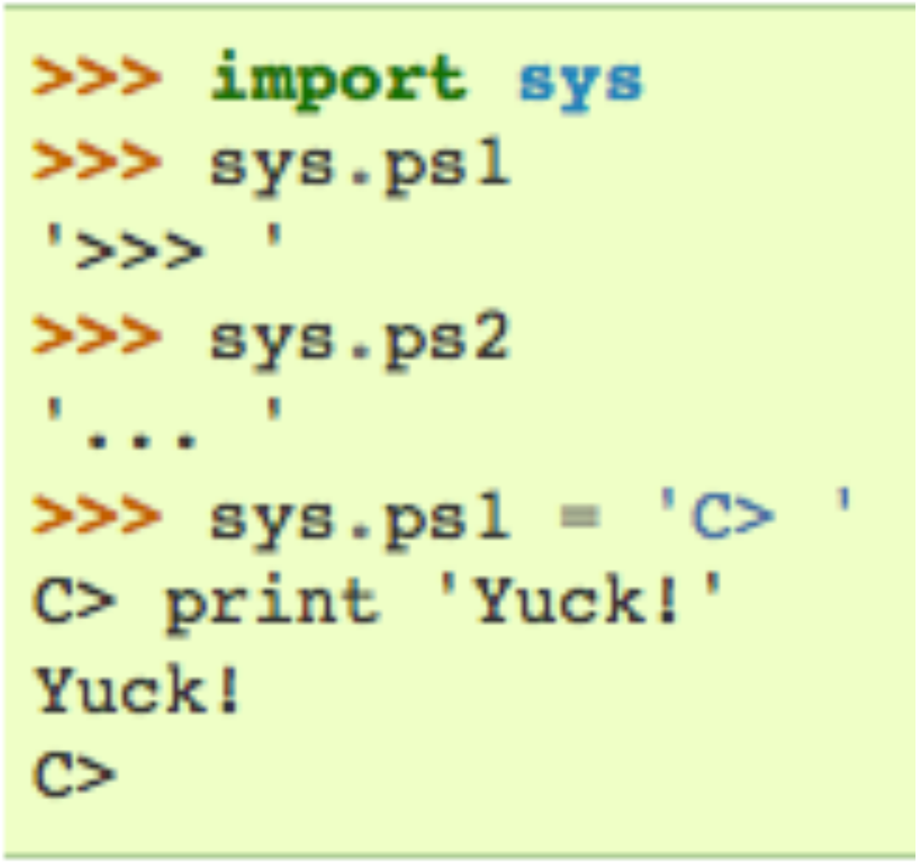
Some Tips for Experts

- It is possible to have a file called spam.pyc (or spam.pyo when -O is used) without a file spam.py for the same module.
- This can be used to distribute a library of Python code in a form that is moderately hard to reverse engineer.

Some Tips for Experts

- The module compileall can create .pyc files (or .pyo files when -O is used) for all modules in a directory.

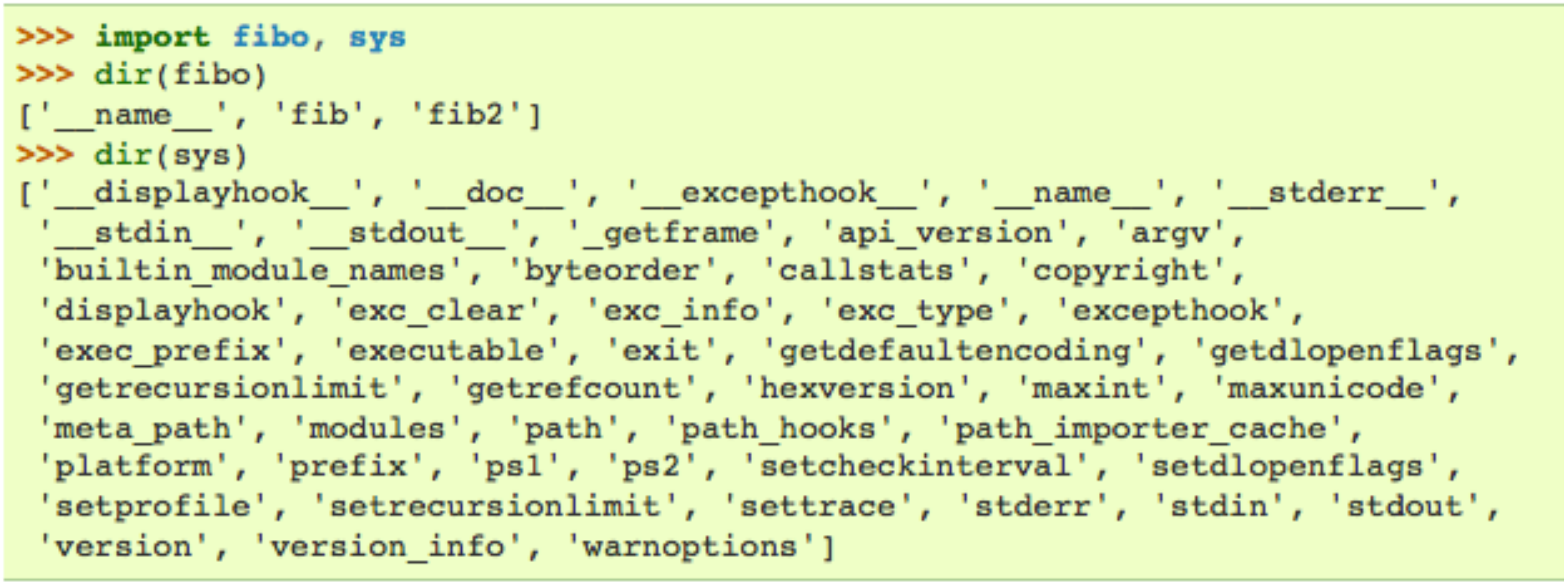
Standard Modules



```
>>> import sys
>>> sys.ps1
'>>> '
>>> sys.ps2
'... '
>>> sys.ps1 = 'C> '
C> print 'Yuck!'
Yuck!
C>
```

Dir()

- The built-in function dir() is used to find out which names a module defines. It returns a sorted list of strings.



```
>>> import fibo, sys
>>> dir(fibo)
['__name__', 'fib', 'fib2']
>>> dir(sys)
['__displayhook__', '__doc__', '__excepthook__', '__name__', '__stderr__',
 '__stdin__', '__stdout__', '_getframe', 'api_version', 'argv',
 'builtin_module_names', 'byteorder', 'callstats', 'copyright',
 'displayhook', 'exc_clear', 'exc_info', 'exc_type', 'excepthook',
 'exec_prefix', 'executable', 'exit', 'getdefaultencoding', 'getdlopenflags',
 'getrecursionlimit', 'getrefcount', 'hexversion', 'maxint', 'maxunicode',
 'meta_path', 'modules', 'path', 'path_hooks', 'path_importer_cache',
 'platform', 'prefix', 'ps1', 'ps2', 'setcheckinterval', 'setdlopenflags',
 'setprofile', 'setrecursionlimit', 'settrace', 'stderr', 'stdin', 'stdout',
 'version', 'version_info', 'warnoptions']
```

Packages

- Packages are a way of structuring Python’s module namespace by using “dotted module names”.
- For example, the module name A.B designates a submodule named B in a package named A.
- Just like the use of modules saves the authors of different modules from having to worry about each other’s global variable names, the use of dotted module names saves the authors of multi-module packages like NumPy or the Python Imaging Library from having to worry about each other’s module names.

```
sound/                          Top-level package
      __init__.py               Initialize the sound package
      formats/                  Subpackage for file format conversions
              __init__.py
              wavread.py
              wavwrite.py
              aiffread.py
              aiffwrite.py
              auread.py
              auwrite.py
              ...
      effects/                  Subpackage for sound effects
              __init__.py
              echo.py
              surround.py
              reverse.py
              ...
      filters/                  Subpackage for filters
              __init__.py
              equalizer.py
              vocoder.py
              karaoke.py
              ...
```

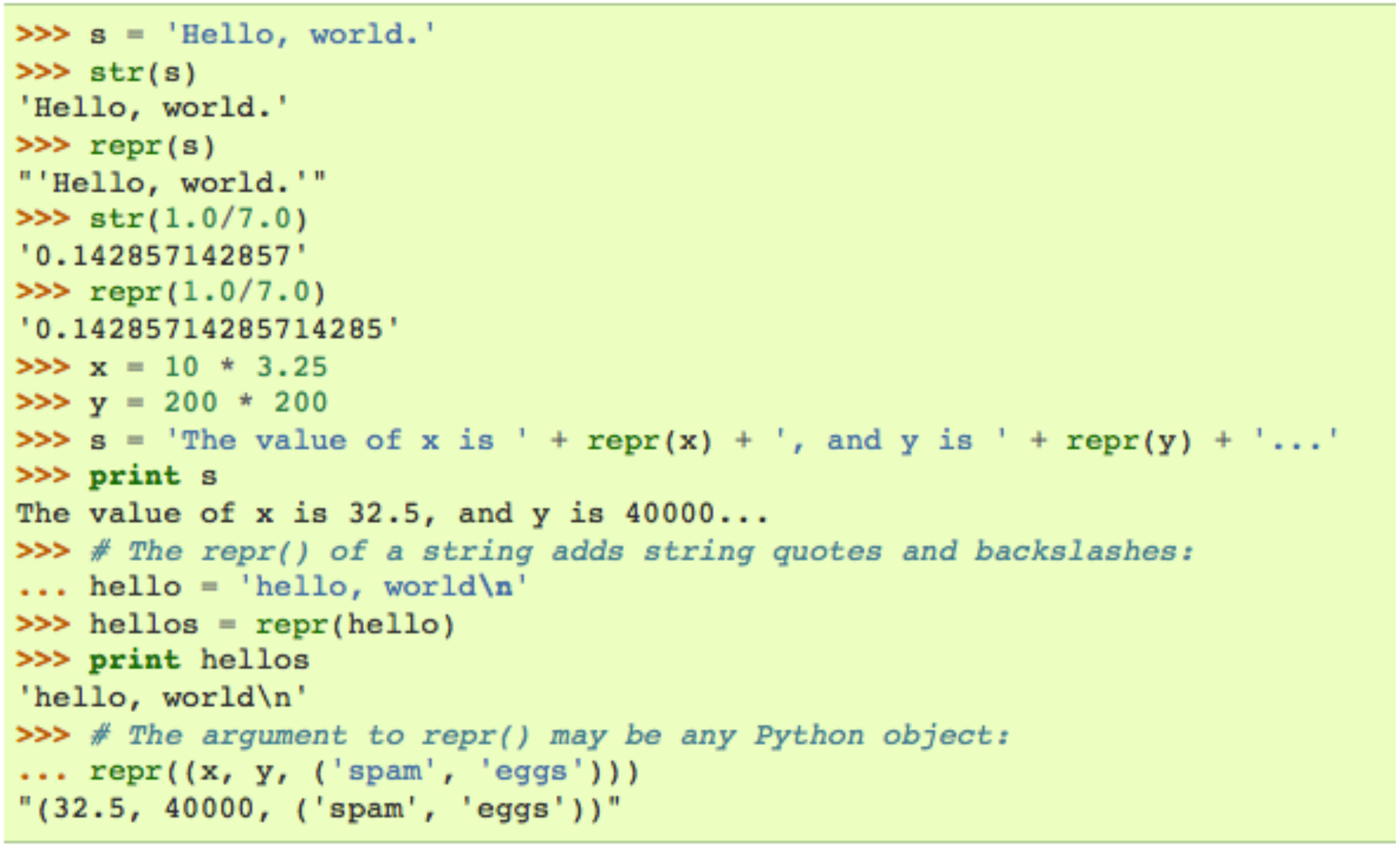
Input and Output

- There are several ways to present the output of a program; data can be printed in a human-readable form, or written to a file for future use. This chapter will discuss some of the possibilities.
- How do you convert values to strings?

Representation

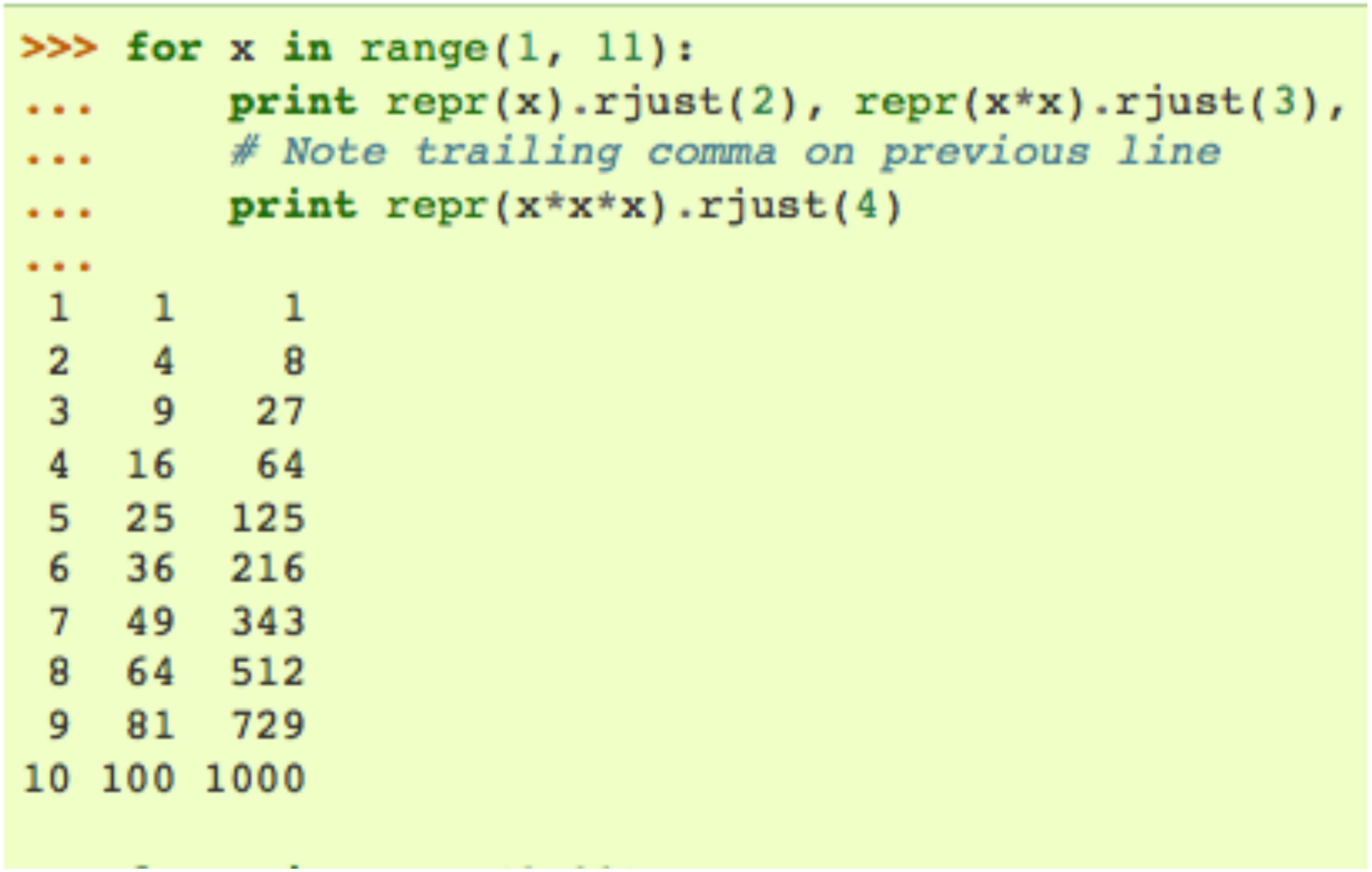
- The str() function is meant to return representations of values which are fairly human-readable.
- repr() is meant to generate representations which can be read by the interpreter (or will force a SyntaxError if there is not equivalent syntax).

Str() and Repr()



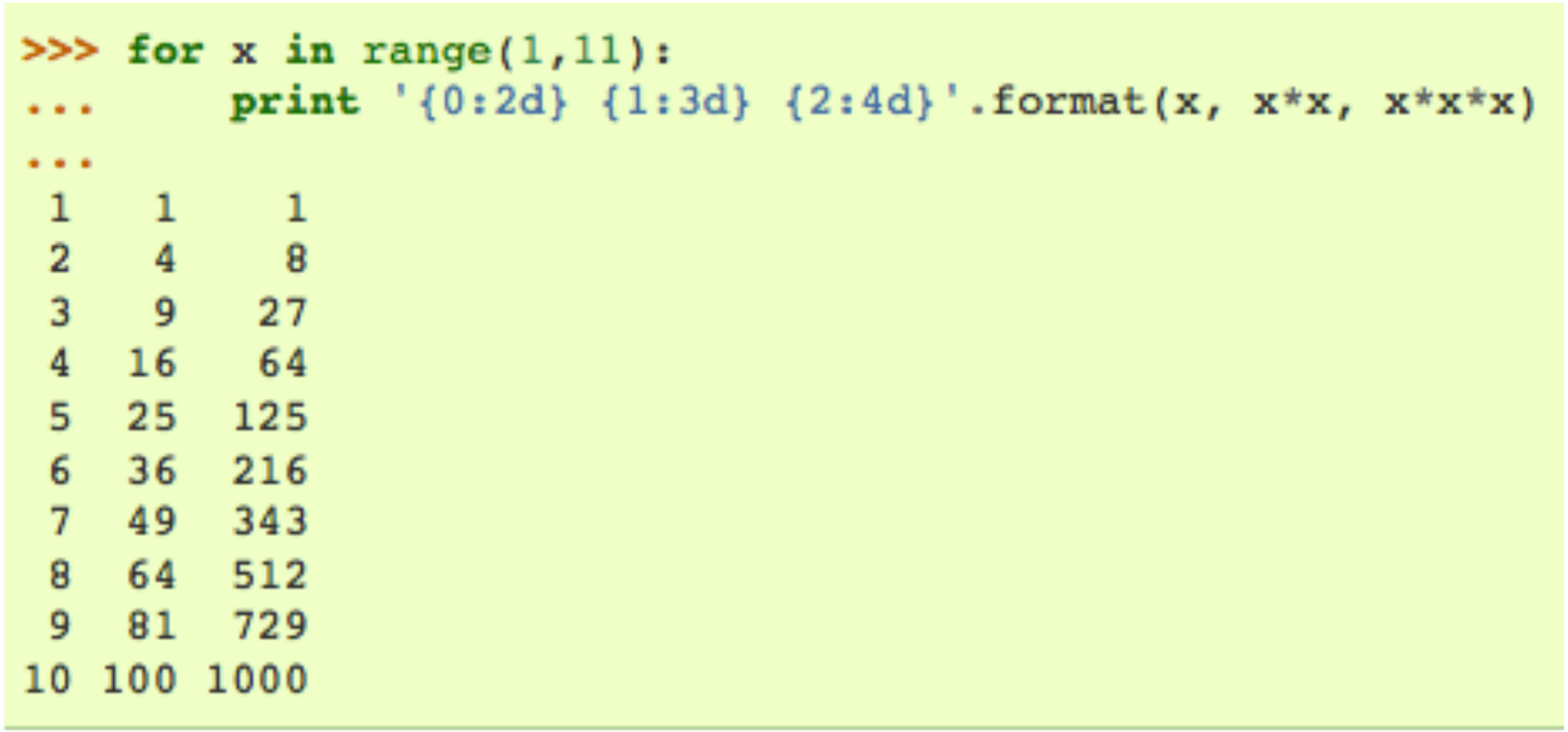
```
>>> s = 'Hello, world.'
>>> str(s)
'Hello, world.'
>>> repr(s)
"'Hello, world.'"
>>> str(1.0/7.0)
'0.142857142857'
>>> repr(1.0/7.0)
'0.14285714285714285'
>>> x = 10 * 3.25
>>> y = 200 * 200
>>> s = 'The value of x is ' + repr(x) + ', and y is ' + repr(y) + '...'
>>> print s
The value of x is 32.5, and y is 40000...
>>> # The repr() of a string adds string quotes and backslashes:
... hello = 'hello, world\n'
>>> hellos = repr(hello)
>>> print hellos
'hello, world\n'
>>> # The argument to repr() may be any Python object:
... repr((x, y, ('spam', 'eggs')))
"(32.5, 40000, ('spam', 'eggs'))"
```

Formatting Output



```
>>> for x in range(1, 11):
...     print repr(x).rjust(2), repr(x*x).rjust(3),
...     # Note trailing comma on previous line
...     print repr(x*x*x).rjust(4)
...
 1   1    1
 2   4    8
 3   9   27
 4  16   64
 5  25  125
 6  36  216
 7  49  343
 8  64  512
 9  81  729
10 100 1000
```

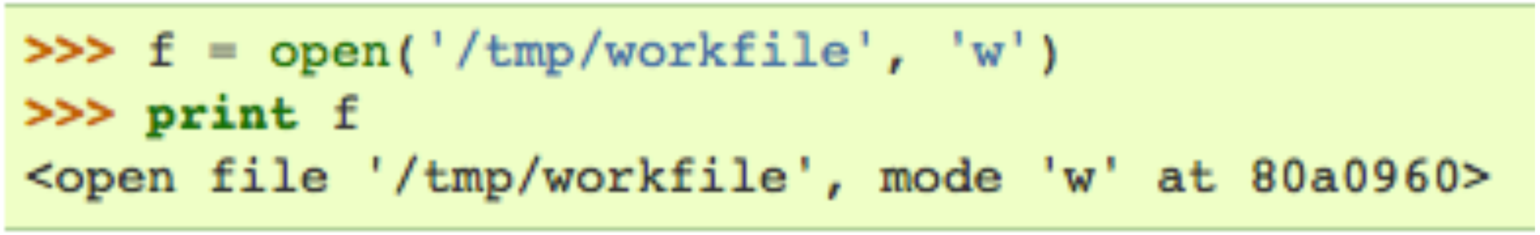
Formatting Output



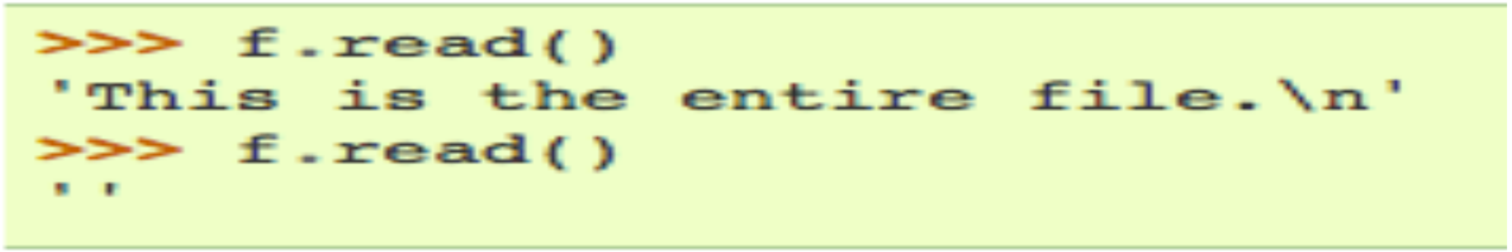
```
>>> for x in range(1,11):
...     print '{0:2d} {1:3d} {2:4d}'.format(x, x*x, x*x*x)
...
 1   1    1
 2   4    8
 3   9   27
 4  16   64
 5  25  125
 6  36  216
 7  49  343
 8  64  512
 9  81  729
10 100 1000
```

Reading and Writing Files

- Open() returns a file object

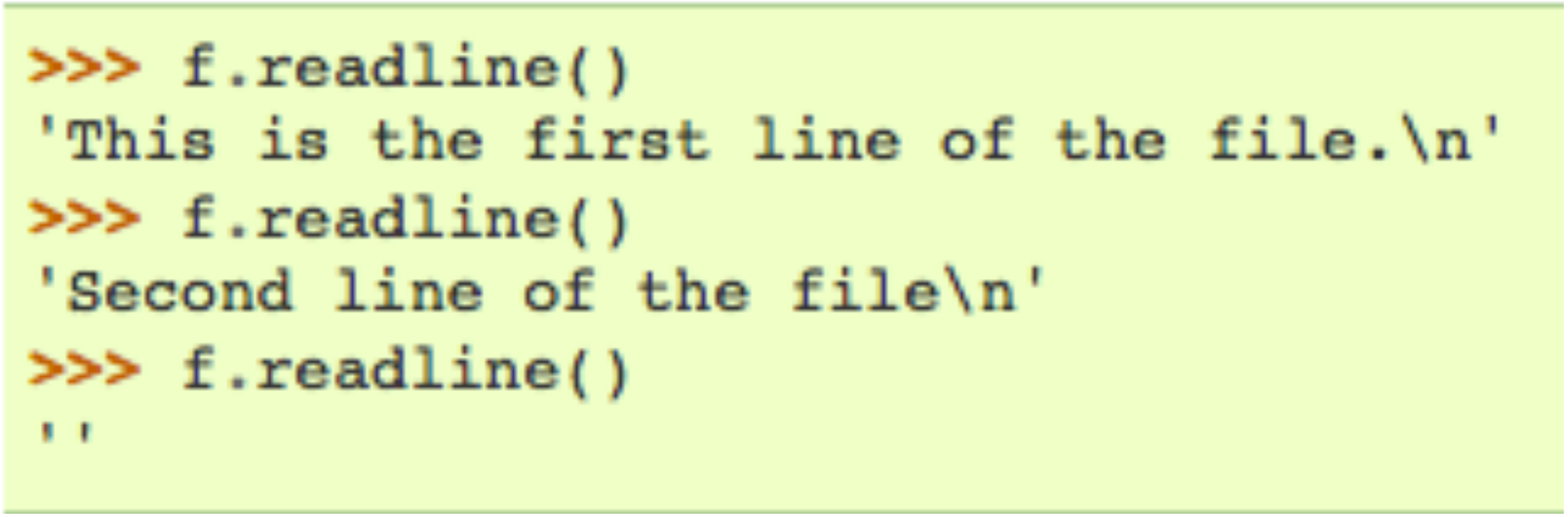


```
>>> f = open('/tmp/workfile', 'w')
>>> print f
<open file '/tmp/workfile', mode 'w' at 80a0960>
```

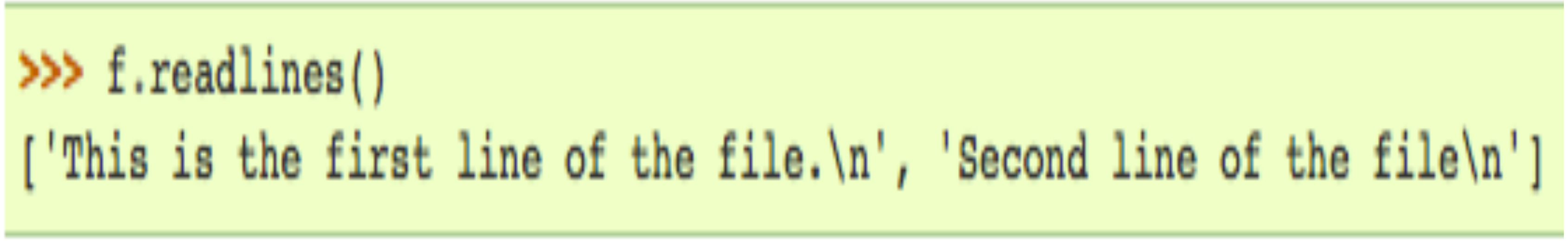


```
>>> f.read()
'This is the entire file.\n'
>>> f.read()
''
```

Reading and Writing Files

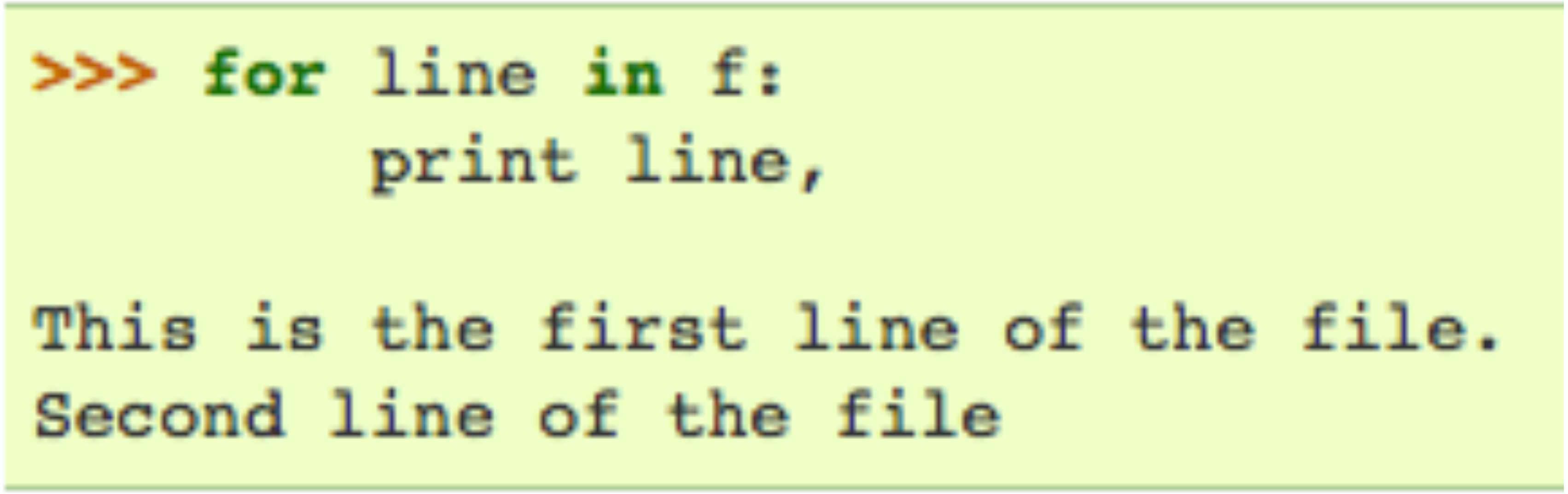


```
>>> f.readline()
'This is the first line of the file.\n'
>>> f.readline()
'Second line of the file\n'
>>> f.readline()
''
```



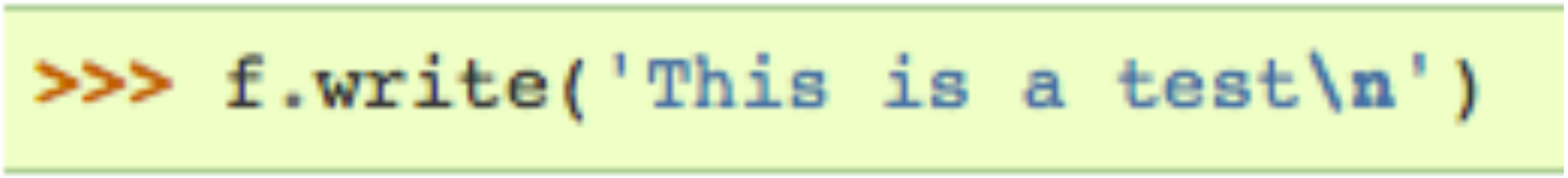
```
>>> f.readlines()
['This is the first line of the file.\n', 'Second line of the file\n']
```

Alternative Method



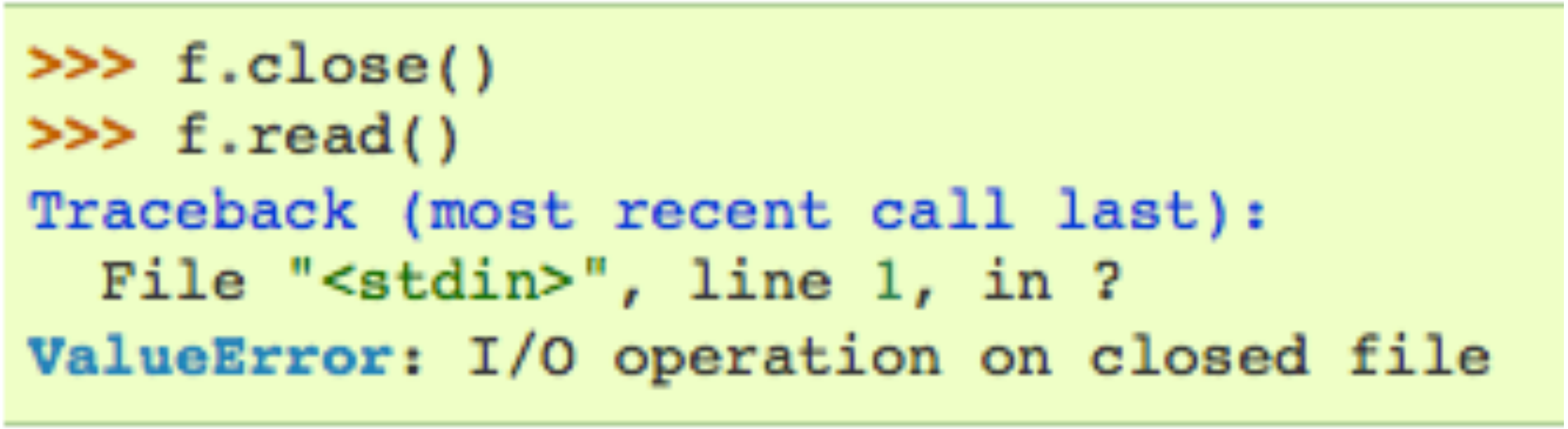
```
>>> for line in f:
        print line,

This is the first line of the file.
Second line of the file
```

Write to File



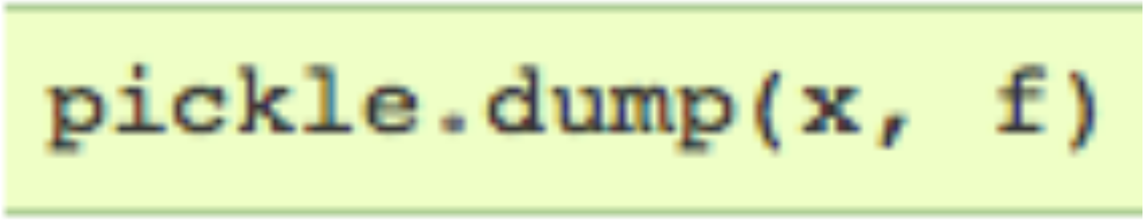
```
>>> f.write('This is a test\n')
```

Close

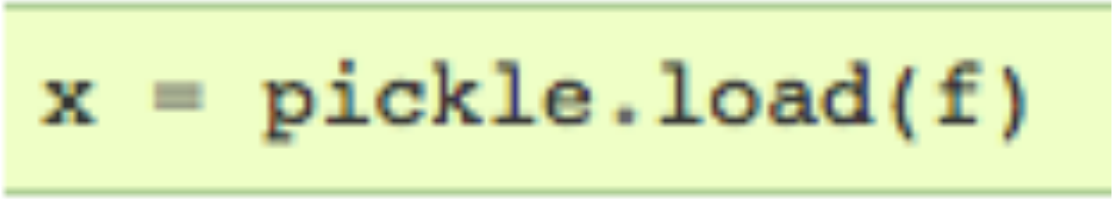


```
>>> f.close()
>>> f.read()
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ValueError: I/O operation on closed file
```

Pickle Module



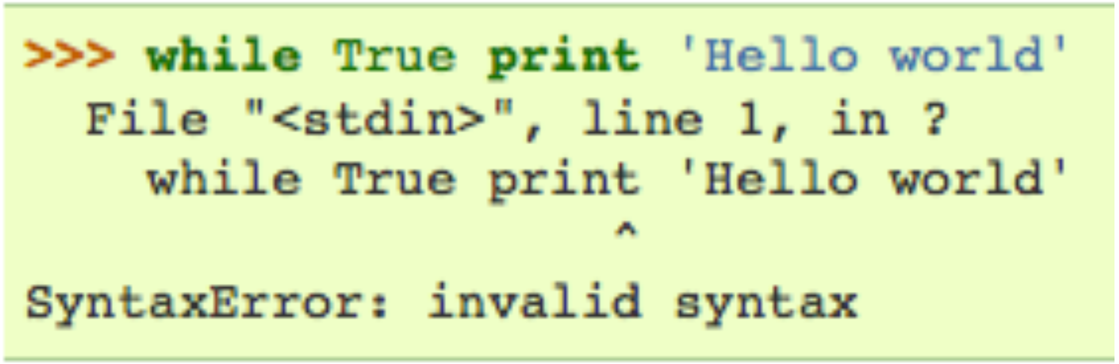
```
pickle.dump(x, f)
```



```
x = pickle.load(f)
```

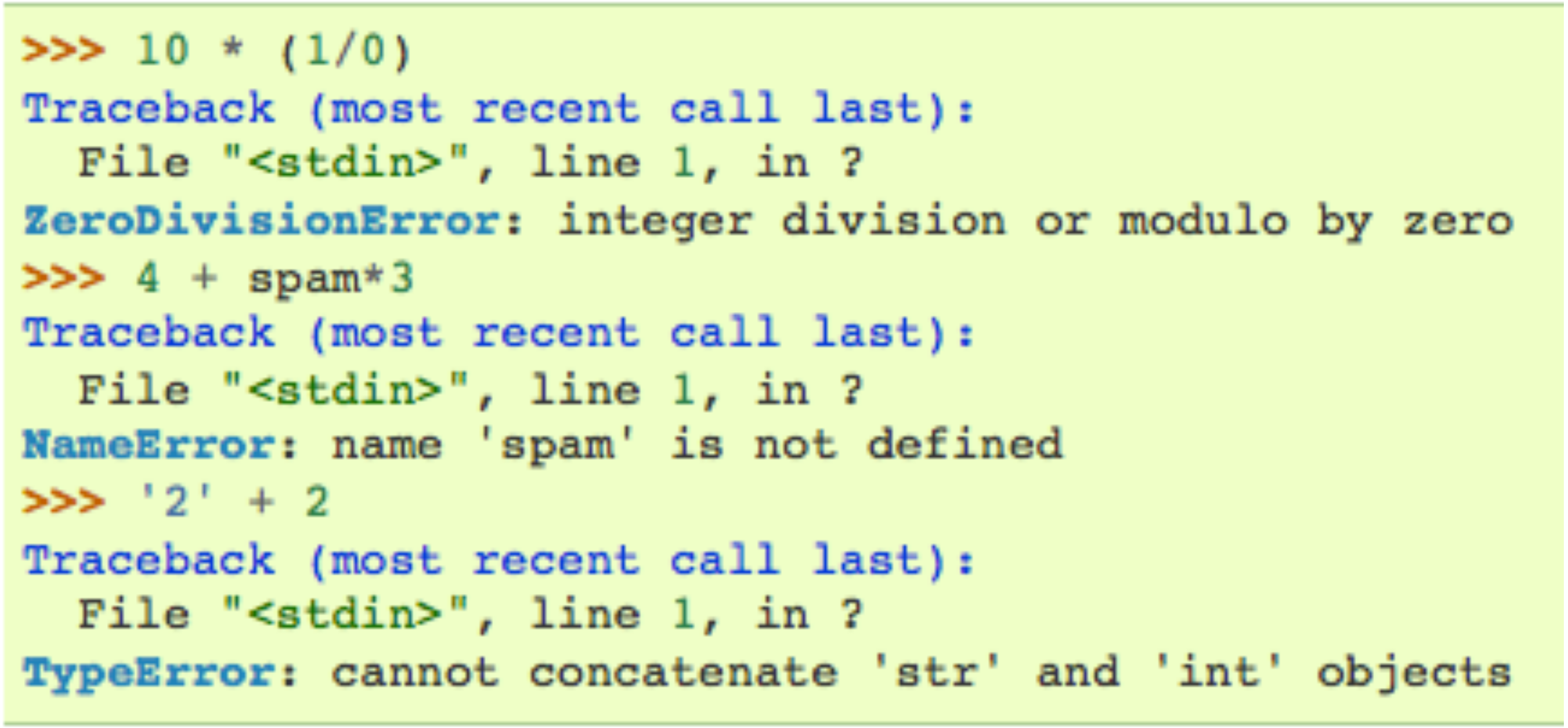
Errors and Exceptions

- Syntax Errors



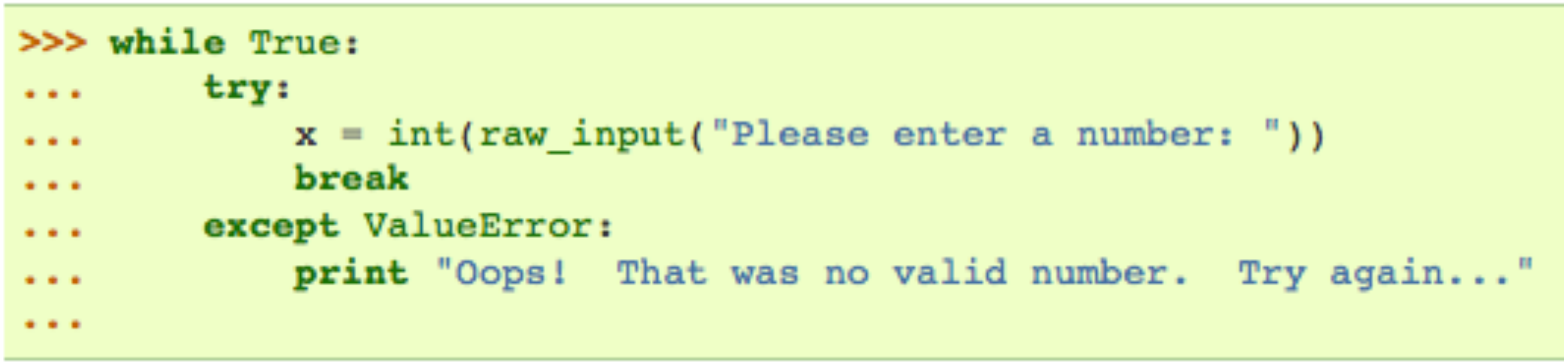
```
>>> while True print 'Hello world'
  File "<stdin>", line 1, in ?
    while True print 'Hello world'
                   ^
SyntaxError: invalid syntax
```

Exceptions



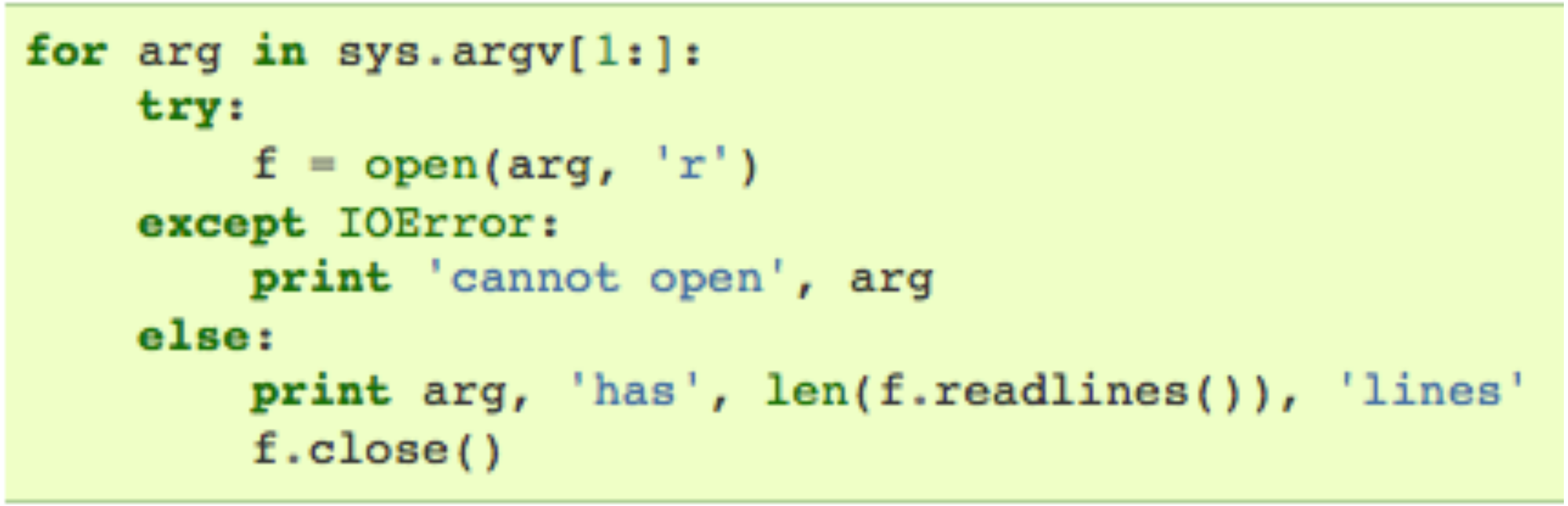
```
>>> 10 * (1/0)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ZeroDivisionError: integer division or modulo by zero
>>> 4 + spam*3
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
NameError: name 'spam' is not defined
>>> '2' + 2
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: cannot concatenate 'str' and 'int' objects
```

Handling Exceptions



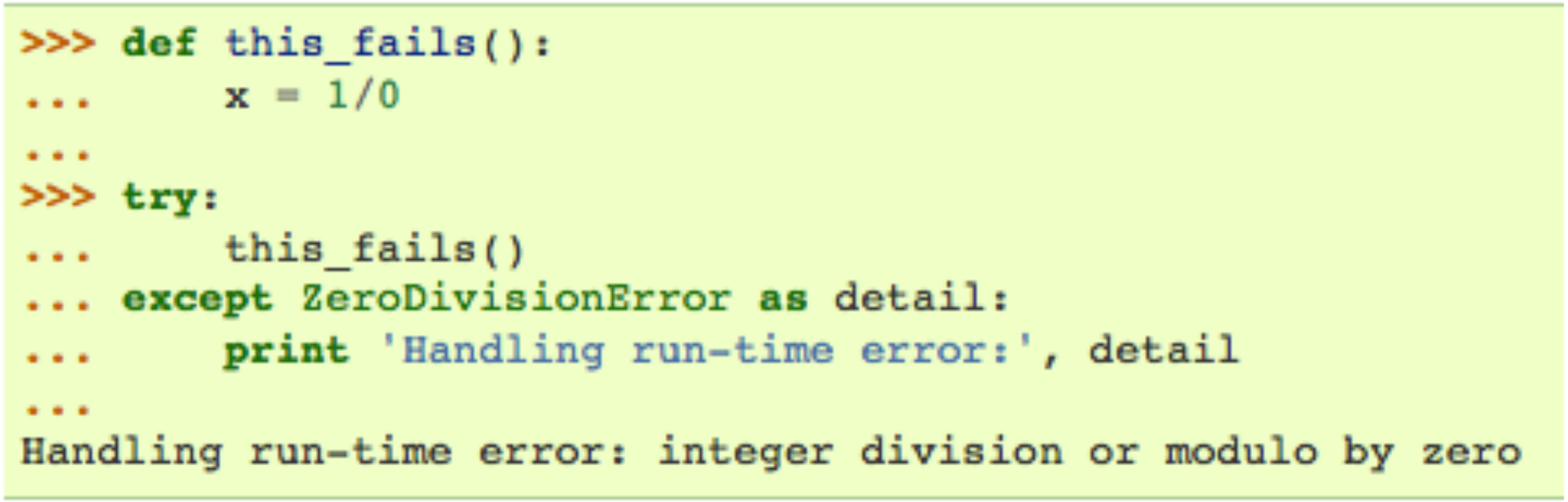
```
>>> while True:
...     try:
...         x = int(raw_input("Please enter a number: "))
...         break
...     except ValueError:
...         print "Oops!  That was no valid number.  Try again..."
...
```

Else?!



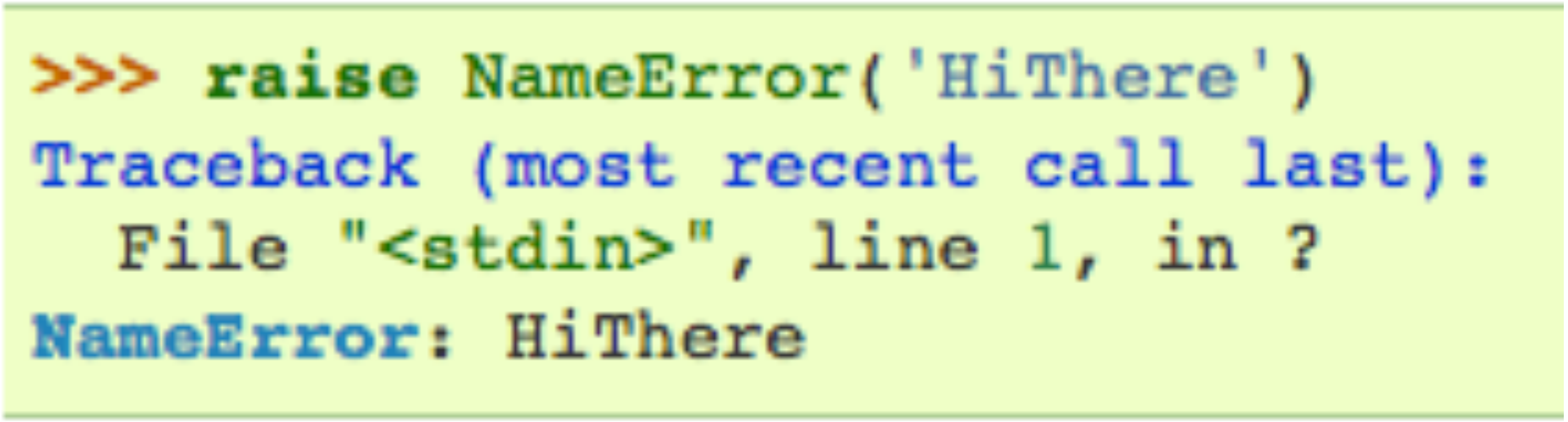
```
for arg in sys.argv[1:]:
    try:
        f = open(arg, 'r')
    except IOError:
        print 'cannot open', arg
    else:
        print arg, 'has', len(f.readlines()), 'lines'
        f.close()
```

Another Exception



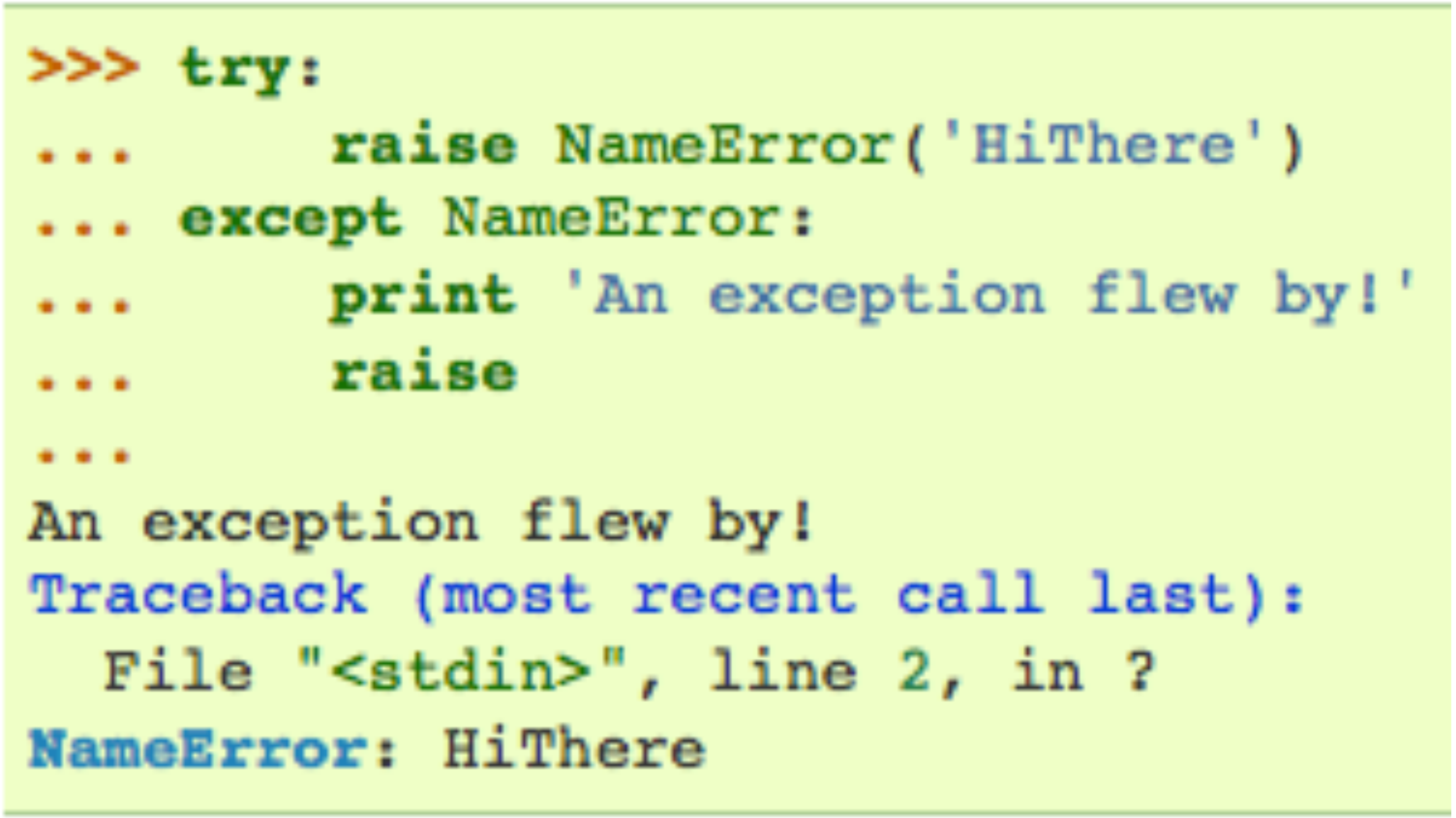
```
>>> def this_fails():
...     x = 1/0
...
>>> try:
...     this_fails()
... except ZeroDivisionError as detail:
...     print 'Handling run-time error:', detail
...
Handling run-time error: integer division or modulo by zero
```

Raise Exception



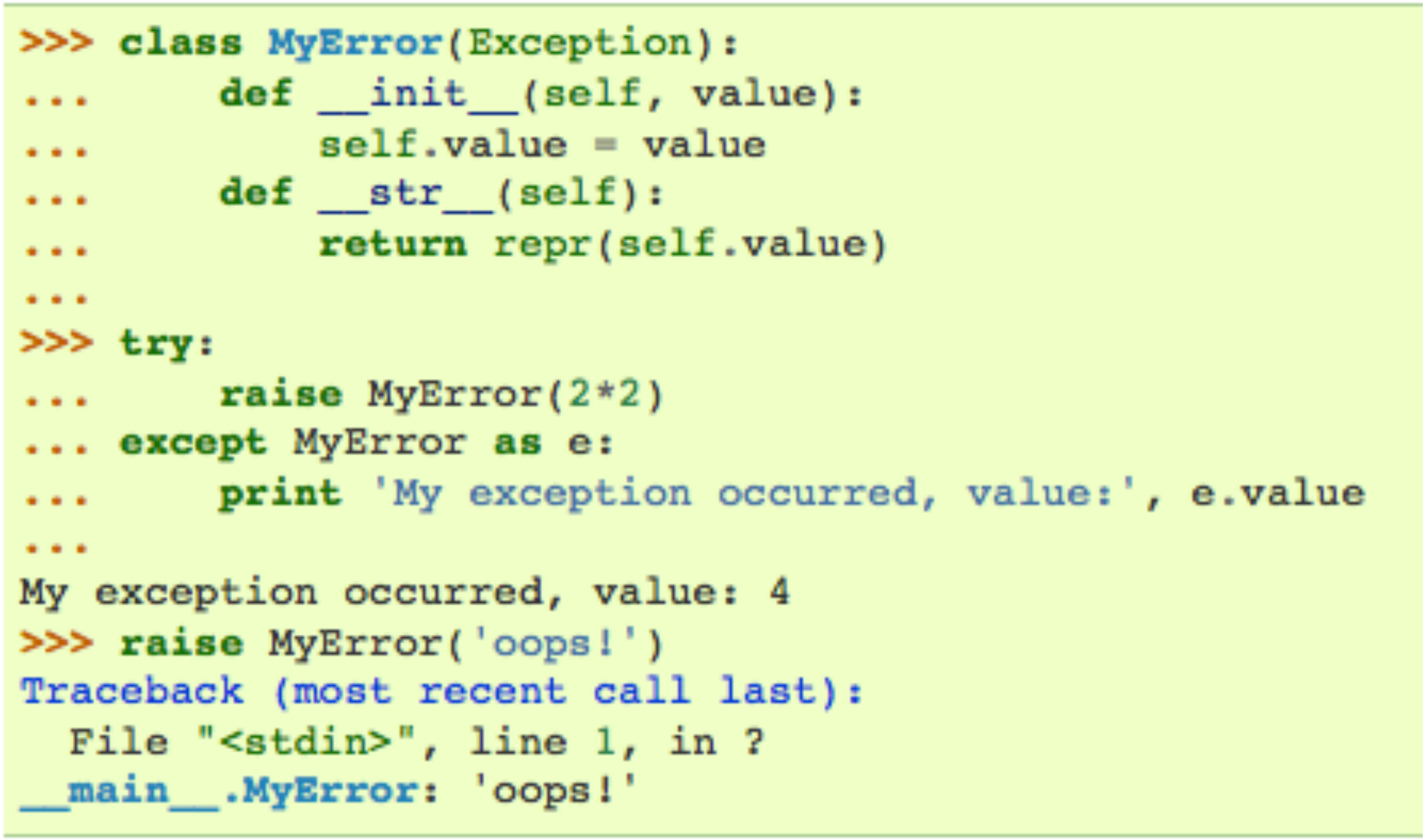
```
>>> raise NameError('HiThere')
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
NameError: HiThere
```

Catching it!



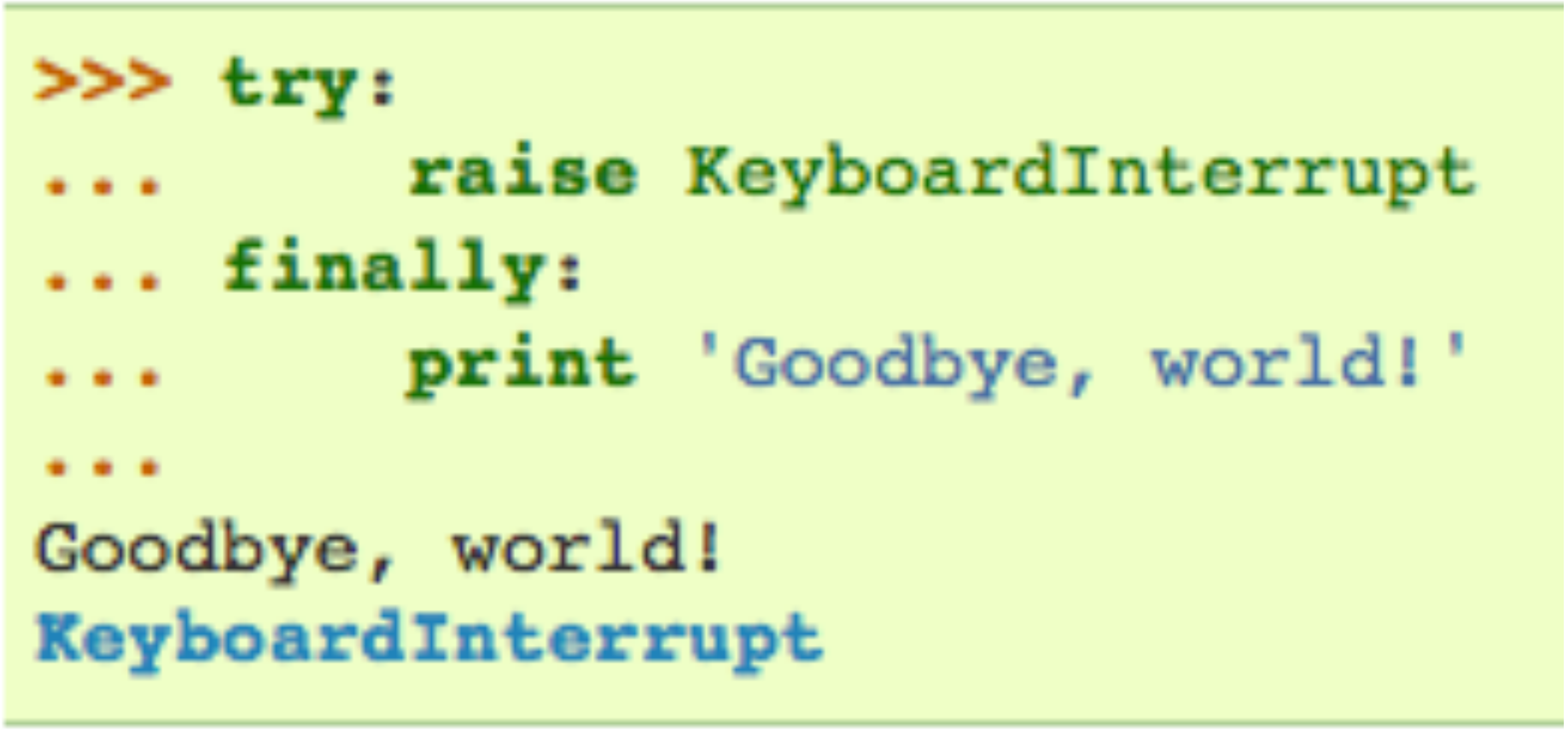
```
>>> try:
...     raise NameError('HiThere')
... except NameError:
...     print 'An exception flew by!'
...     raise
...
An exception flew by!
Traceback (most recent call last):
  File "<stdin>", line 2, in ?
NameError: HiThere
```

User Defined Exception



```
>>> class MyError(Exception):
...     def __init__(self, value):
...         self.value = value
...     def __str__(self):
...         return repr(self.value)
...
>>> try:
...     raise MyError(2*2)
... except MyError as e:
...     print 'My exception occurred, value:', e.value
...
My exception occurred, value: 4
>>> raise MyError('oops!')
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
__main__.MyError: 'oops!'
```

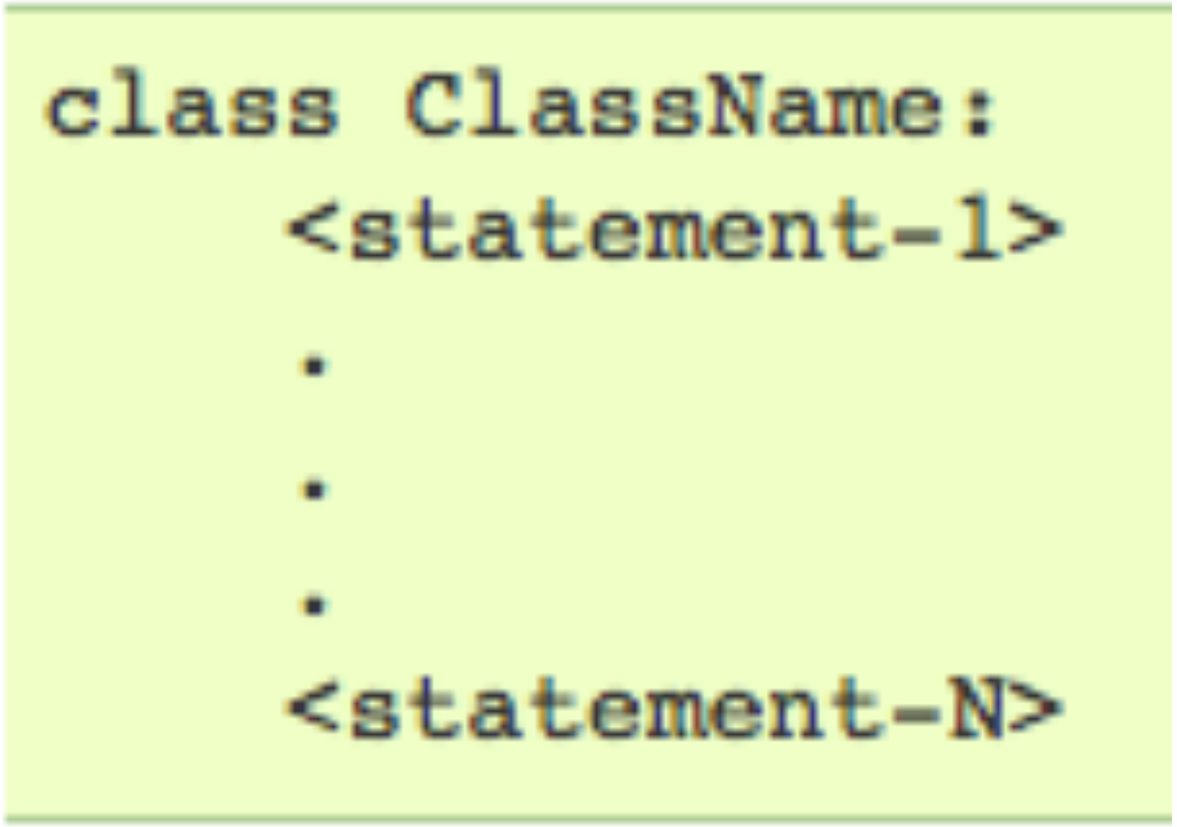
Clean Up Actions



```
>>> try:
...     raise KeyboardInterrupt
... finally:
...     print 'Goodbye, world!'
...
Goodbye, world!
KeyboardInterrupt
```

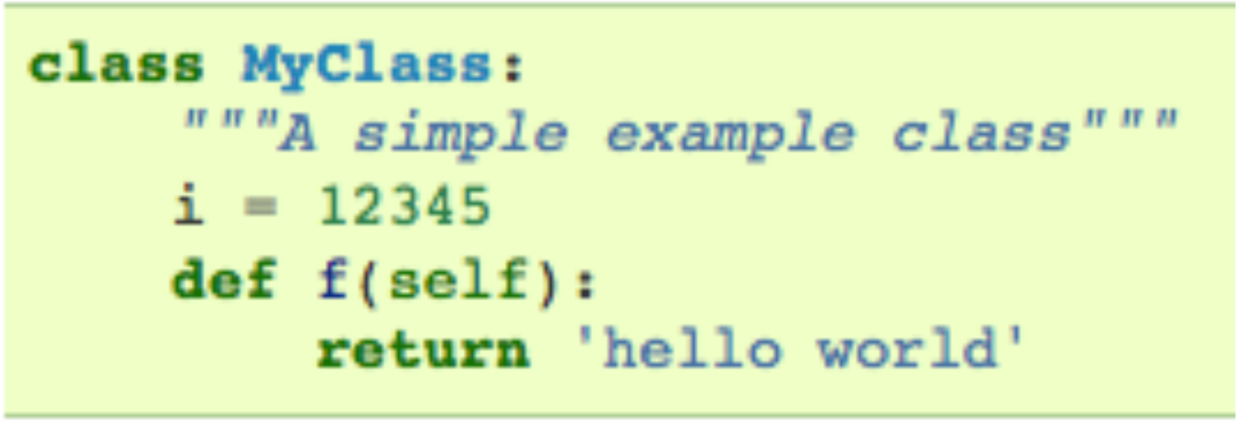
Classes

Class Definition Syntax

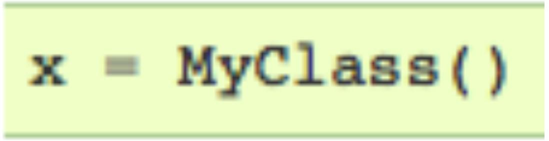


```
class ClassName:
    <statement-1>
    .
    .
    .
    <statement-N>
```

Class Objects

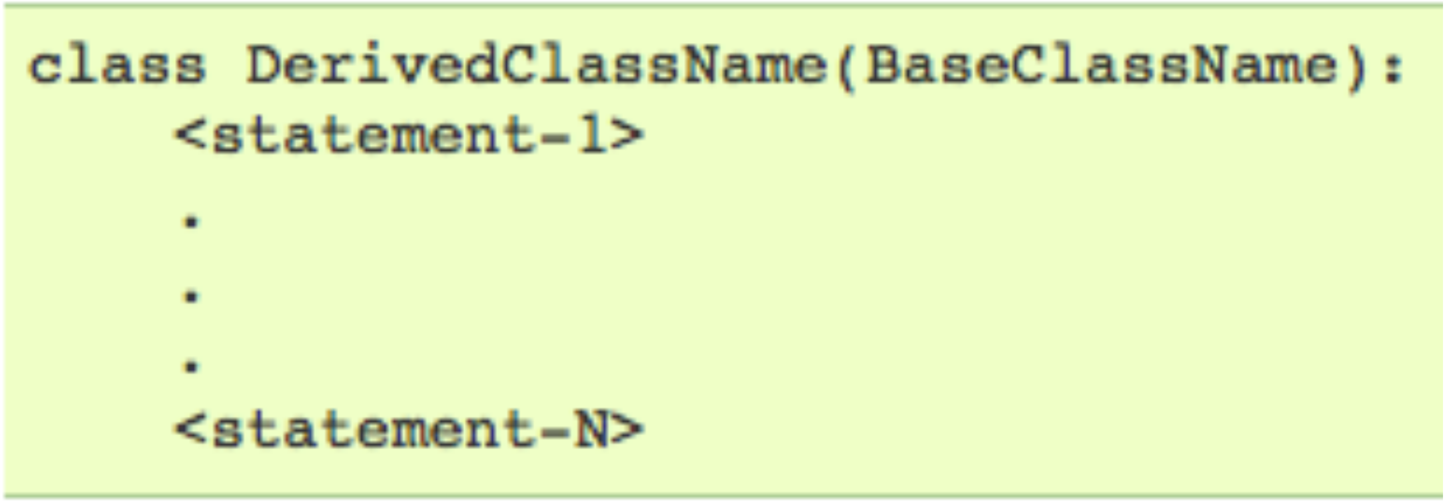


```python
class MyClass:
    """A simple example class"""
    i = 12345
    def f(self):
        return 'hello world'
```

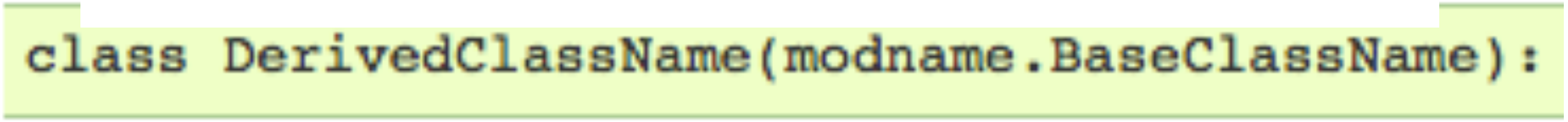


```python
x = MyClass()
```

Inheritance



```
class DerivedClassName(BaseClassName):
    <statement-1>
    .
    .
    .
    <statement-N>
```



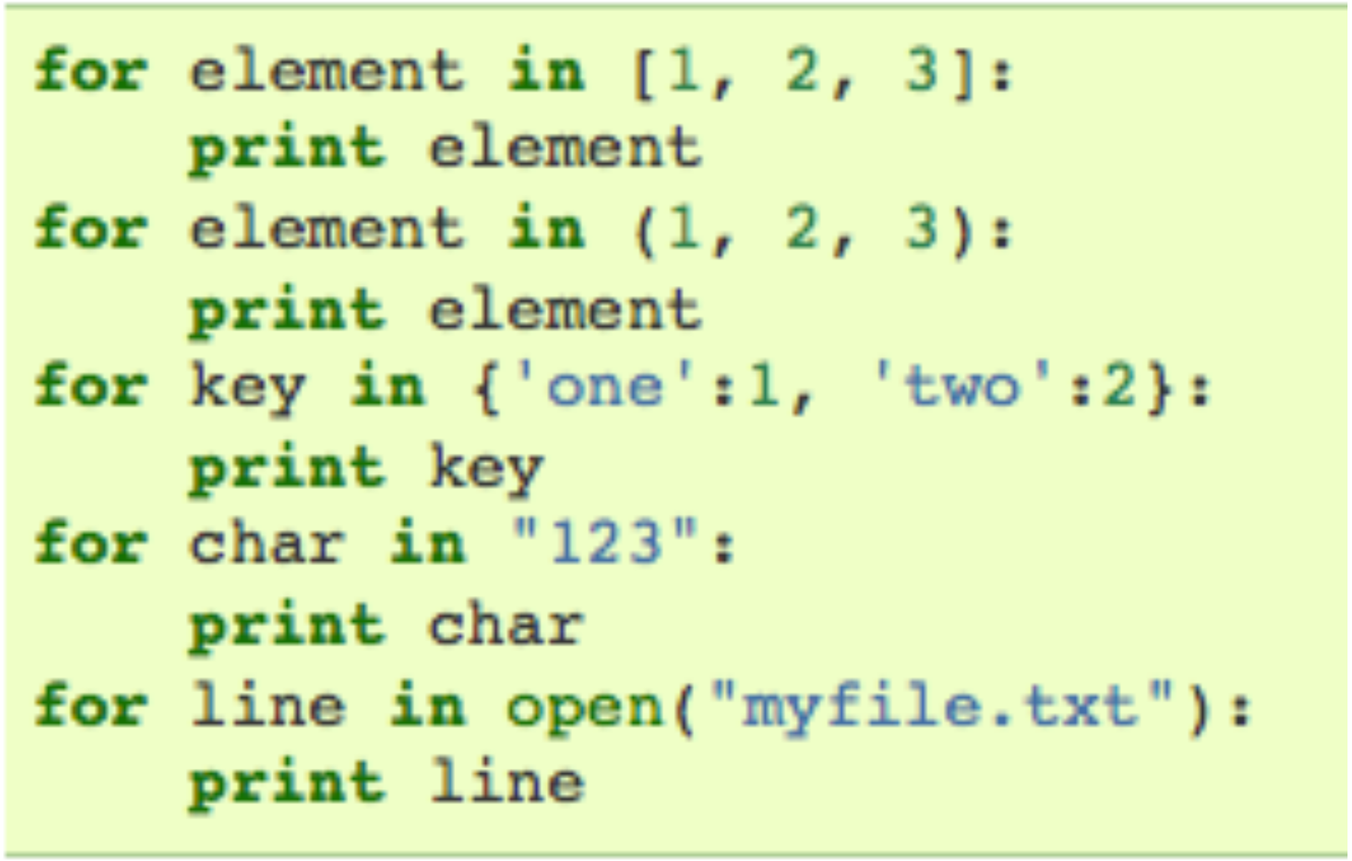
```
class DerivedClassName(modname.BaseClassName):
```

Multiple Inheritance

```
class DerivedClassName(Base1, Base2, Base3):
    <statement-1>
    .
    .
    .
    <statement-N>
```

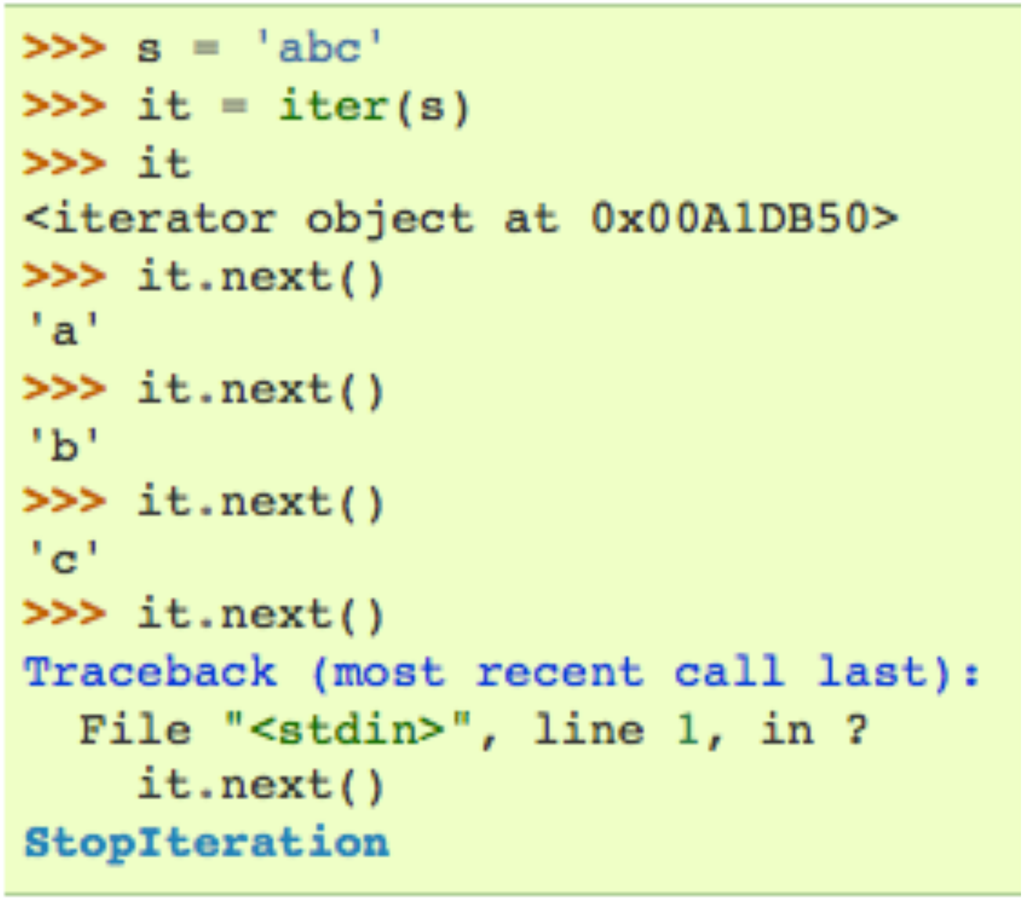
```python
class Employee:
    pass

john = Employee() # Create an empty employee record

# Fill the fields of the record
john.name = 'John Doe'
john.dept = 'computer lab'
john.salary = 1000
```

Iterators



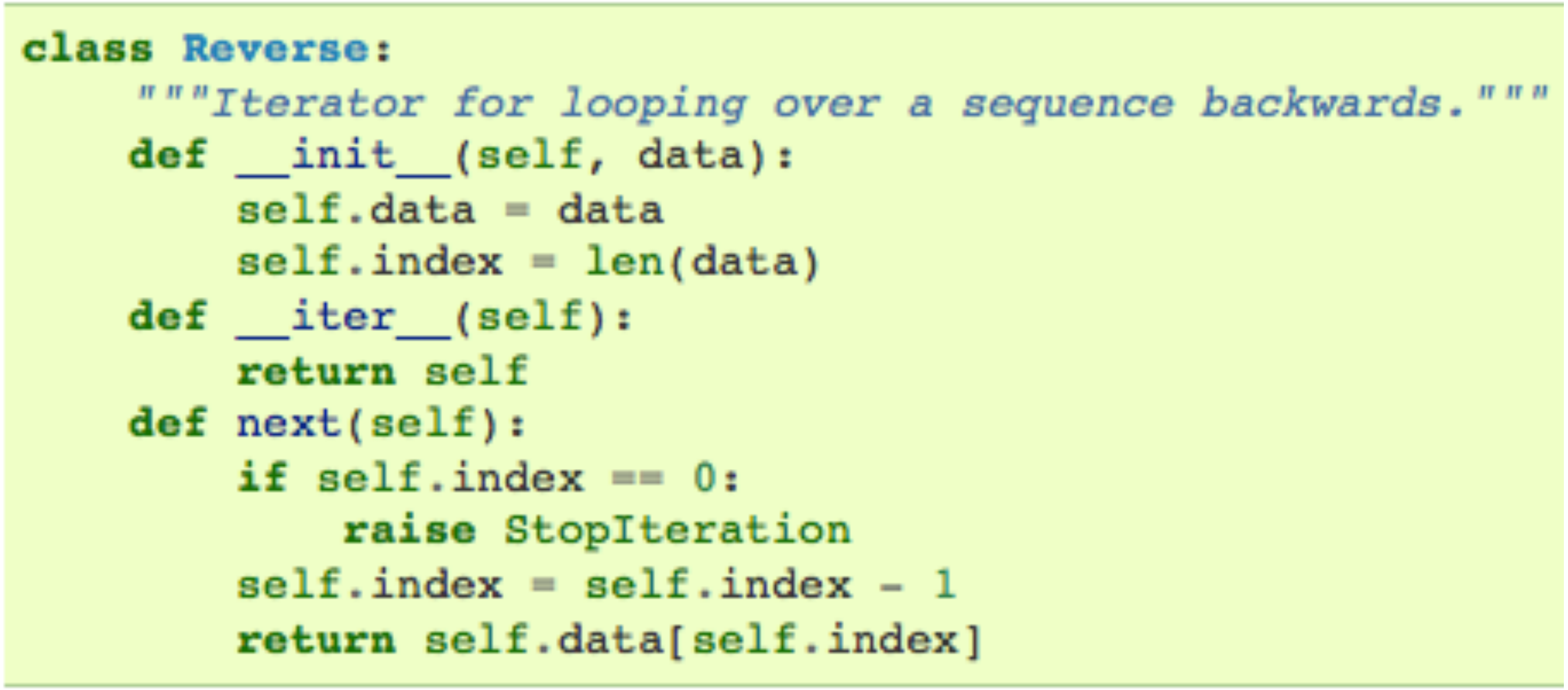
```
for element in [1, 2, 3]:
    print element
for element in (1, 2, 3):
    print element
for key in {'one':1, 'two':2}:
    print key
for char in "123":
    print char
for line in open("myfile.txt"):
    print line
```

Style of Access



```
>>> s = 'abc'
>>> it = iter(s)
>>> it
<iterator object at 0x00A1DB50>
>>> it.next()
'a'
>>> it.next()
'b'
>>> it.next()
'c'
>>> it.next()
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
    it.next()
StopIteration
```

Make your Class Iterable



```
class Reverse:
    """Iterator for looping over a sequence backwards."""
    def __init__(self, data):
        self.data = data
        self.index = len(data)
    def __iter__(self):
        return self
    def next(self):
        if self.index == 0:
            raise StopIteration
        self.index = self.index - 1
        return self.data[self.index]
```

Len()

- Implement __len__