Python Tutorial

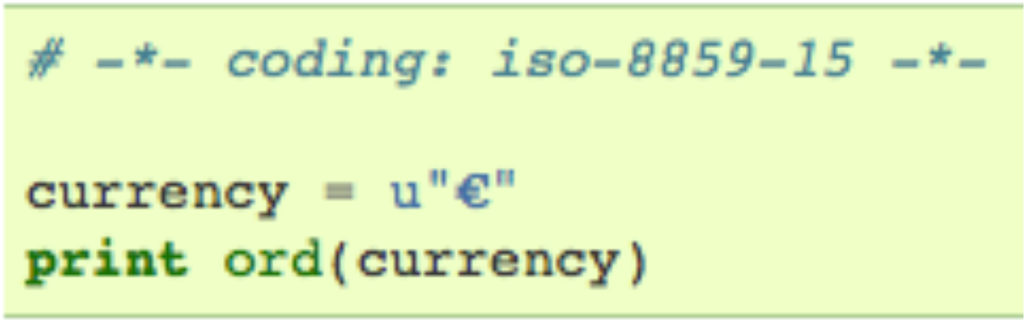
Haitham El-Ghareeb, Ph.D.

May 2012

Twitter: @helghareeb

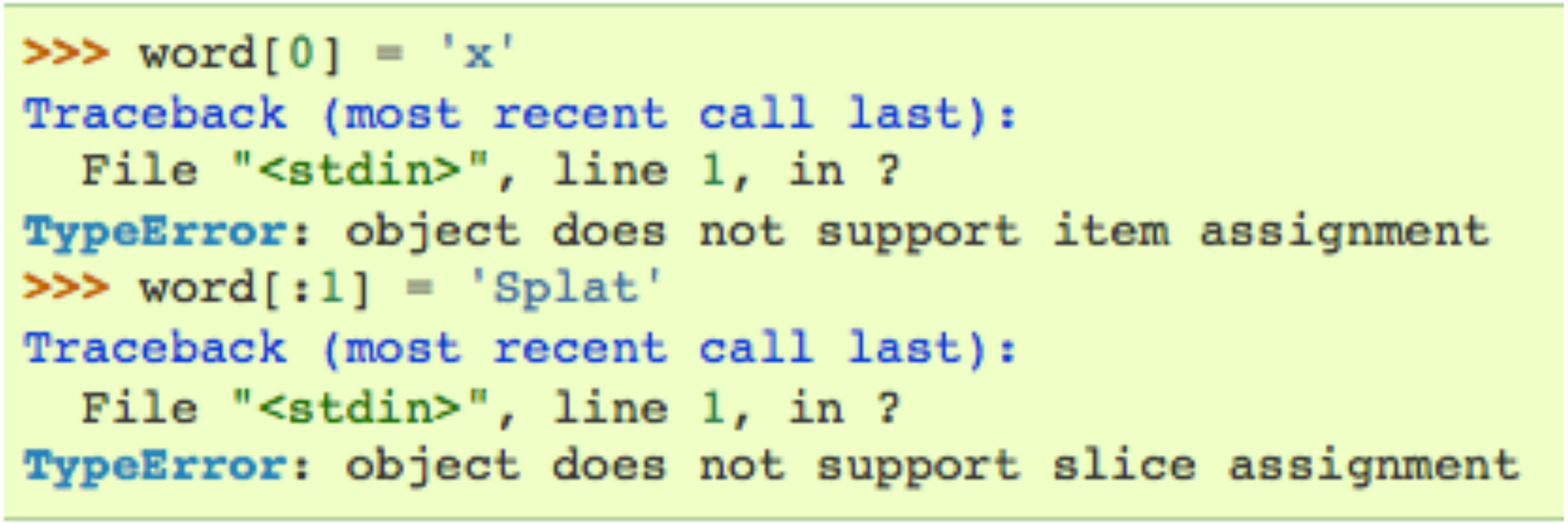
Source Code Encoding

- It is possible to use encodings different than ASCII in Python source files.
- The best way to do it is to put one more special comment line right after the #! line to define the source file encoding:



```
# -*- coding: iso-8859-15 -*-

currency = u"€"
print ord(currency)
```

Strings are Immutable!



```
>>> word[0] = 'x'
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: object does not support item assignment
>>> word[:1] = 'Splat'
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: object does not support slice assignment
```

Pass

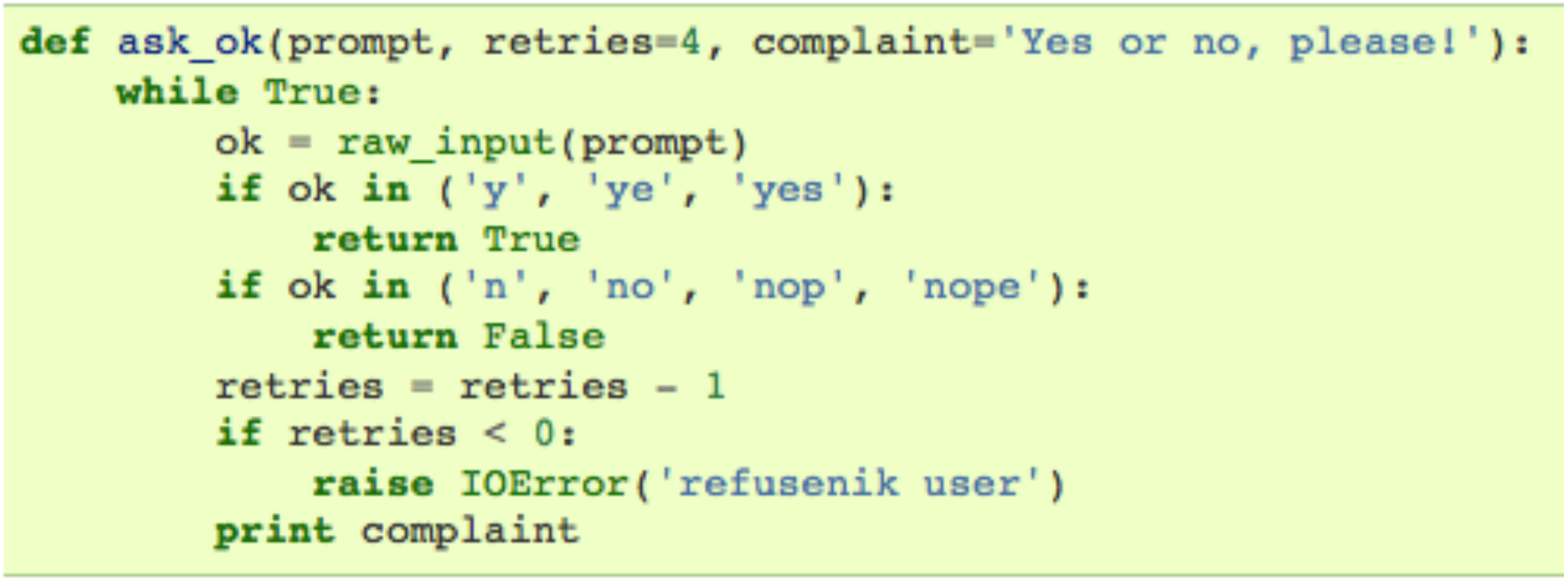
- Does Nothing!

```
>>> while True:
...     pass  # Busy-wait for keyboard interrupt (Ctrl+C)
...
```

```
>>> class MyEmptyClass:
...     pass
...
```

```
>>> def initlog(*args):
...     pass   # Remember to implement this!
...
```

Default Argument Value



```
def ask_ok(prompt, retries=4, complaint='Yes or no, please!'):
    while True:
        ok = raw_input(prompt)
        if ok in ('y', 'ye', 'yes'):
            return True
        if ok in ('n', 'no', 'nop', 'nope'):
            return False
        retries = retries - 1
        if retries < 0:
            raise IOError('refusenik user')
        print complaint
```

Keyword Arguments

```
def parrot(voltage, state='a stiff', action='voom', type='Norwegian Blue'):
    print "-- This parrot wouldn't", action,
    print "if you put", voltage, "volts through it."
    print "-- Lovely plumage, the", type
    print "-- It's", state, "!"
```

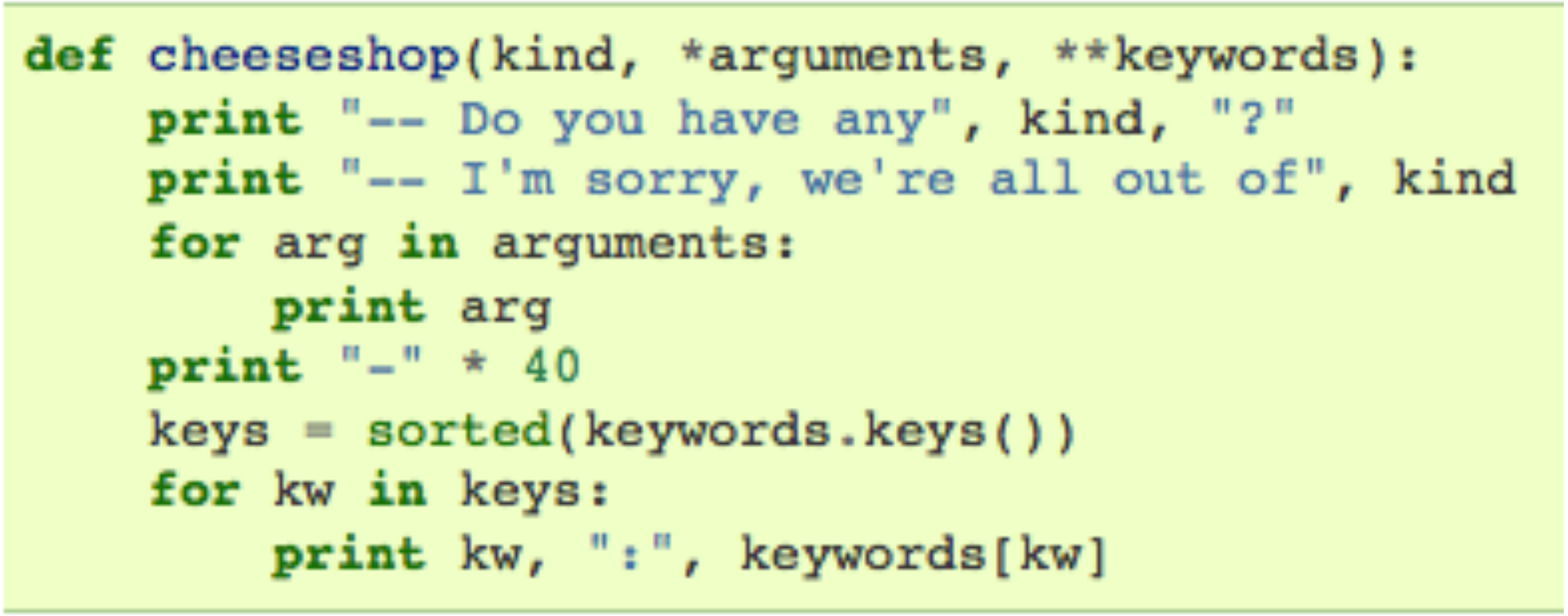
```
parrot(1000)                                          # 1 positional argument
parrot(voltage=1000)                                  # 1 keyword argument
parrot(voltage=1000000, action='VOOOOOM')             # 2 keyword arguments
parrot(action='VOOOOOM', voltage=1000000)             # 2 keyword arguments
parrot('a million', 'bereft of life', 'jump')         # 3 positional arguments
parrot('a thousand', state='pushing up the daisies')  # 1 positional, 1 keyword
```

```
parrot()                     # required argument missing
parrot(voltage=5.0, 'dead')  # non-keyword argument after a keyword argument
parrot(110, voltage=220)     # duplicate value for the same argument
parrot(actor='John Cleese')  # unknown keyword argument
```

Keyword Arguments

- When a final formal parameter of the form **name is present, it receives a dictionary containing all keyword arguments except for those corresponding to a formal parameter.
- This may be combined with a formal parameter of the form *name which receives a tuple containing the positional arguments beyond the formal parameter list.
- (*name must occur before **name.)

Declaration

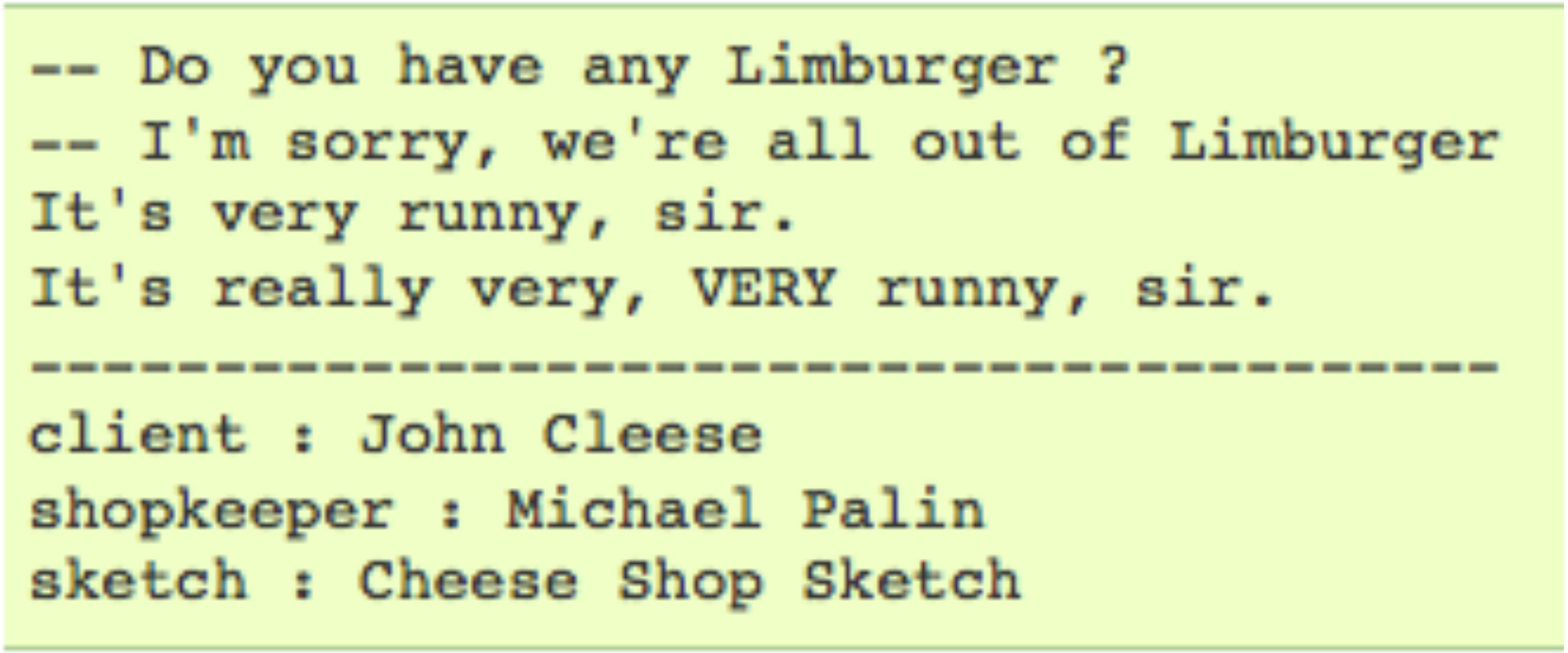


```python
def cheeseshop(kind, *arguments, **keywords):
    print "-- Do you have any", kind, "?"
    print "-- I'm sorry, we're all out of", kind
    for arg in arguments:
        print arg
    print "-" * 40
    keys = sorted(keywords.keys())
    for kw in keys:
        print kw, ":", keywords[kw]
```

Calling

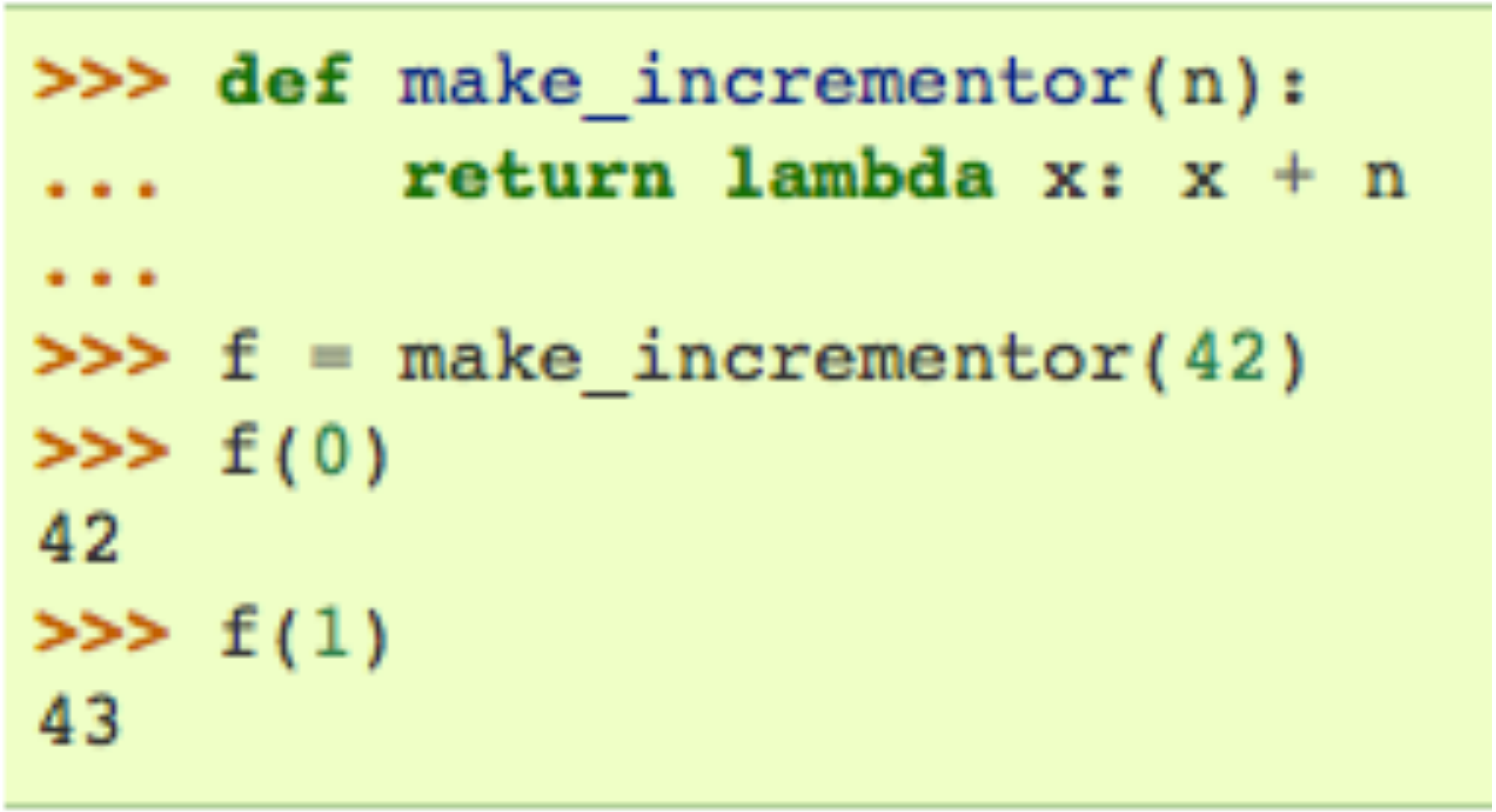
```
cheeseshop("Limburger", "It's very runny, sir.",
           "It's really very, VERY runny, sir.",
           shopkeeper='Michael Palin',
           client="John Cleese",
           sketch="Cheese Shop Sketch")
```

Output



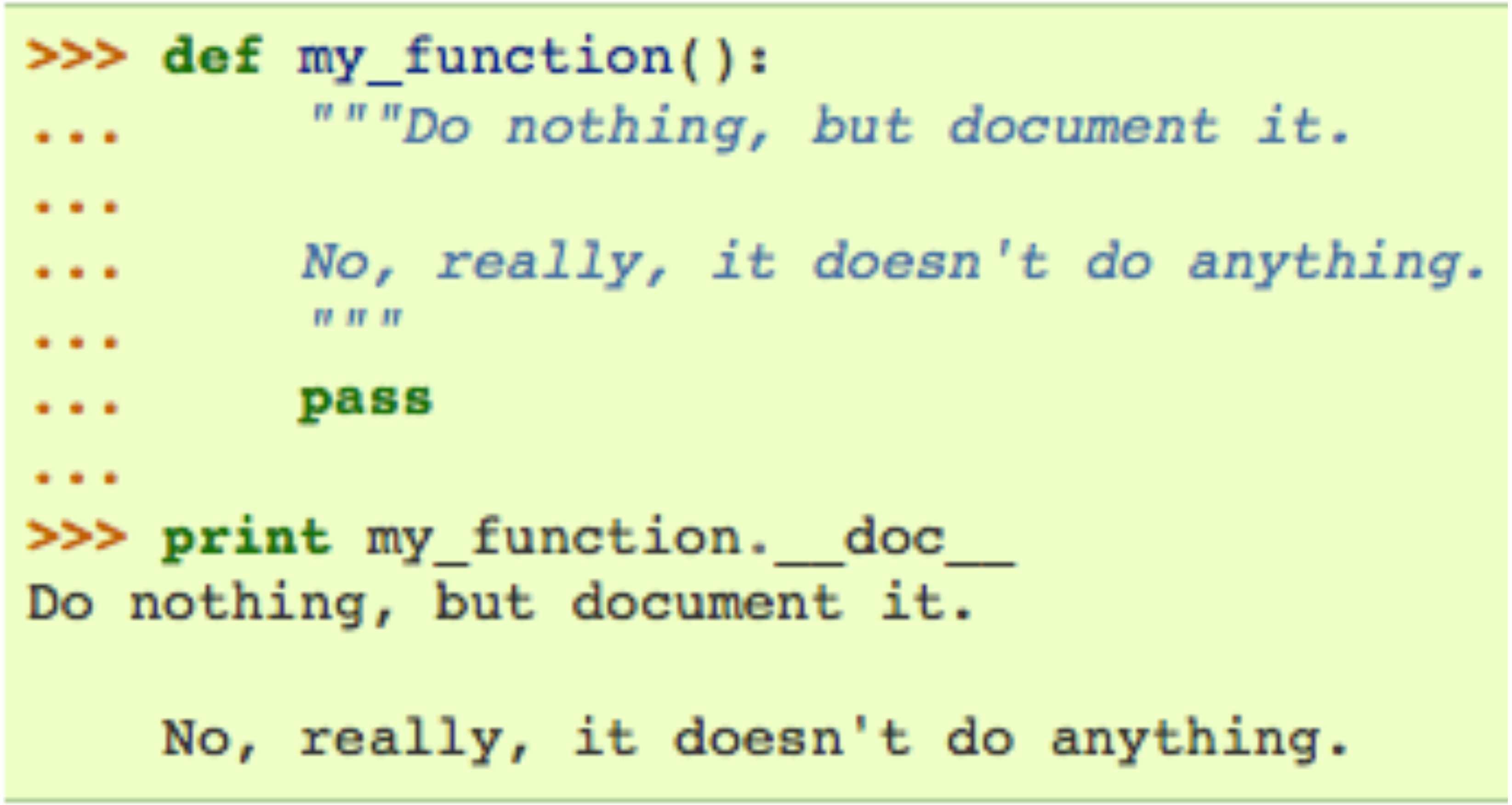
```
-- Do you have any Limburger ?
-- I'm sorry, we're all out of Limburger
It's very runny, sir.
It's really very, VERY runny, sir.
----------------------------------------
client : John Cleese
shopkeeper : Michael Palin
sketch : Cheese Shop Sketch
```

Lambda Strings



```
>>> def make_incrementor(n):
...     return lambda x: x + n
...
>>> f = make_incrementor(42)
>>> f(0)
42
>>> f(1)
43
```

Documentation String



```
>>> def my_function():
...     """Do nothing, but document it.
...
...     No, really, it doesn't do anything.
...     """
...     pass
...
>>> print my_function.__doc__
Do nothing, but document it.

    No, really, it doesn't do anything.
```

Coding Style - PEP8

- Use 4-space indentation, and no tabs.
- 4 spaces are a good compromise between small indentation (allows greater nesting depth) and large indentation (easier to read). Tabs introduce confusion, and are best left out.
- Wrap lines so that they don’t exceed 79 characters.
- This helps users with small displays and makes it possible to have several code files side-by-side on larger displays.
- Use blank lines to separate functions and classes, and larger blocks of code inside functions.

Coding Style - PEP8

- When possible, put comments on a line of their own.
- Use docstrings.
- Use spaces around operators and after commas, but not directly inside bracketing constructs: a = f(1, 2) + g(3, 4).
- Name your classes and functions consistently; the convention is to use CamelCase for classes and lower_case_with_underscores for functions and methods. Always use self as the name for the first method argument.
- Don’t use fancy encodings if your code is meant to be used in international environments. Plain ASCII works best in any case.

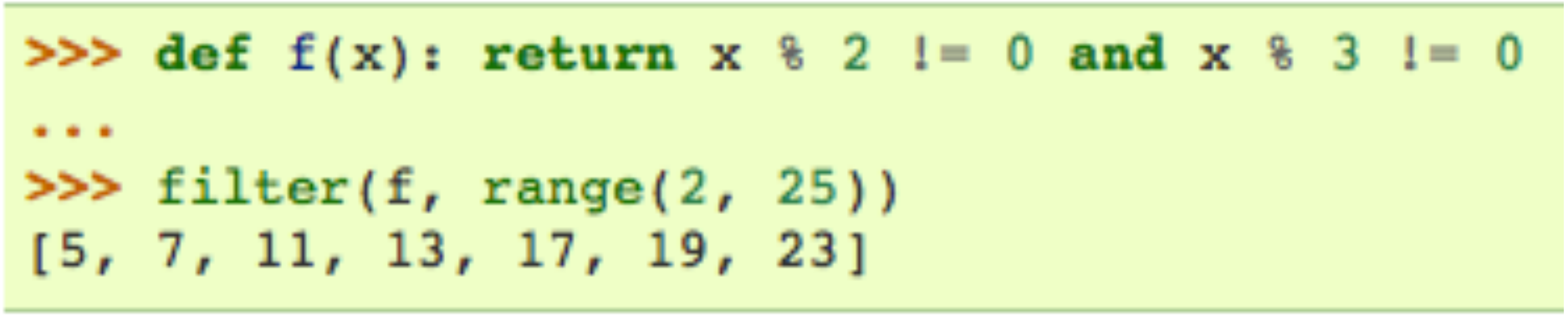
Functional Programming Tools

- There are three built-in functions that are very useful when used with lists: filter(), map(), and reduce().

Filter()

- filter(function, sequence) returns a sequence consisting of those items from the sequence for which function(item) is true.
- If sequence is a string or tuple, the result will be of the same type; otherwise, it is always a list.
- For example, to compute a sequence of numbers not divisible by 2 and 3:

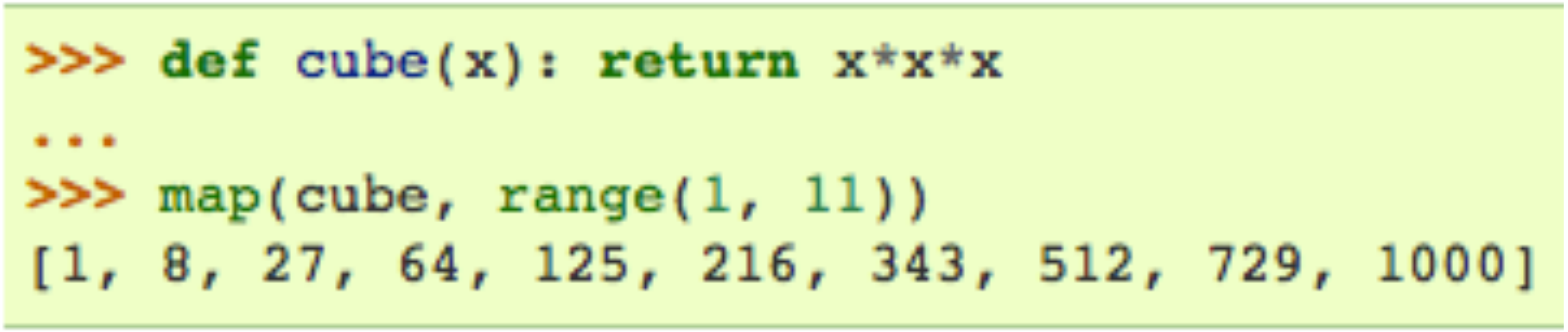
Filter()



```
>>> def f(x): return x % 2 != 0 and x % 3 != 0
...
>>> filter(f, range(2, 25))
[5, 7, 11, 13, 17, 19, 23]
```

Map()

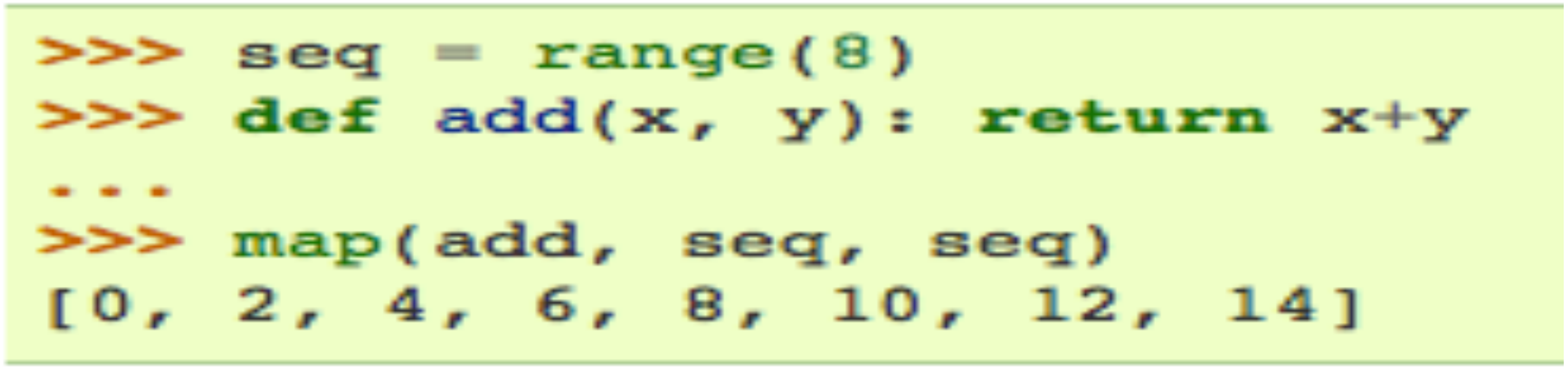
- map(function, sequence) calls function(item) for each of the sequence's items and returns a list of the return values.
- For example, to compute some cubes:



```
>>> def cube(x): return x*x*x
...
>>> map(cube, range(1, 11))
[1, 8, 27, 64, 125, 216, 343, 512, 729, 1000]
```

Map()

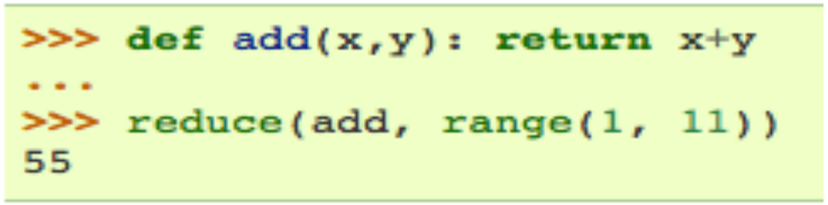
- More than one sequence may be passed; the function must then have as many arguments as there are sequences and is called with the corresponding item from each sequence (or None if some sequence is shorter than another). For example:



```
>>> seq = range(8)
>>> def add(x, y): return x+y
...
>>> map(add, seq, seq)
[0, 2, 4, 6, 8, 10, 12, 14]
```

Reduce()

- reduce(function, sequence) returns a single value constructed by calling the binary function *function* on the first two items of the sequence, then on the result and the next item, and so on.
- For example, to compute the sum of the numbers 1 through 10:

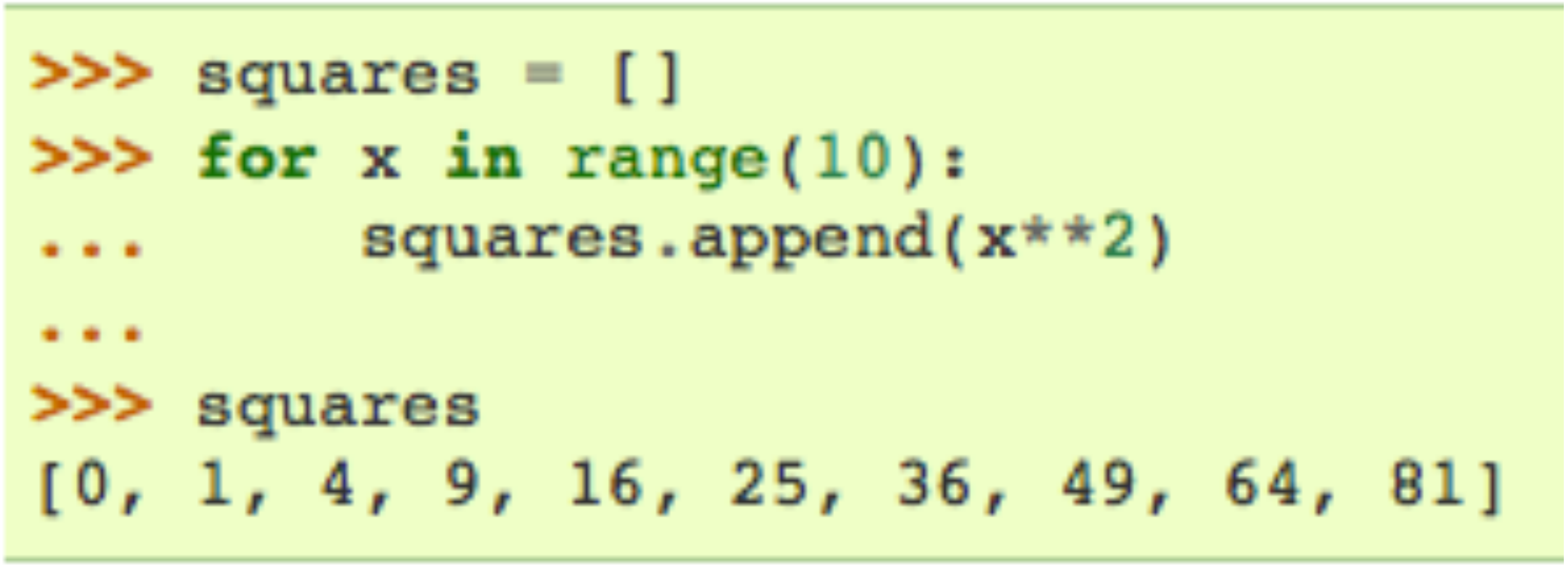


```
>>> def add(x,y): return x+y
...
>>> reduce(add, range(1, 11))
55
```

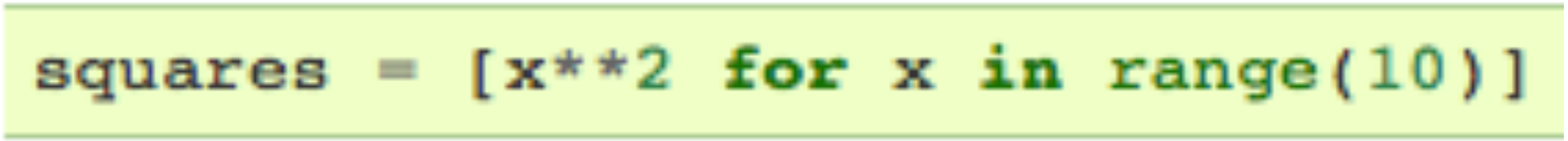
List Comprehension

- List comprehensions provide a concise way to create lists.
- Common applications are to make new lists where each element is the result of some operations applied to each member of another sequence or iterable, or to create a subsequence of those elements that satisfy a certain condition.

List of Squares



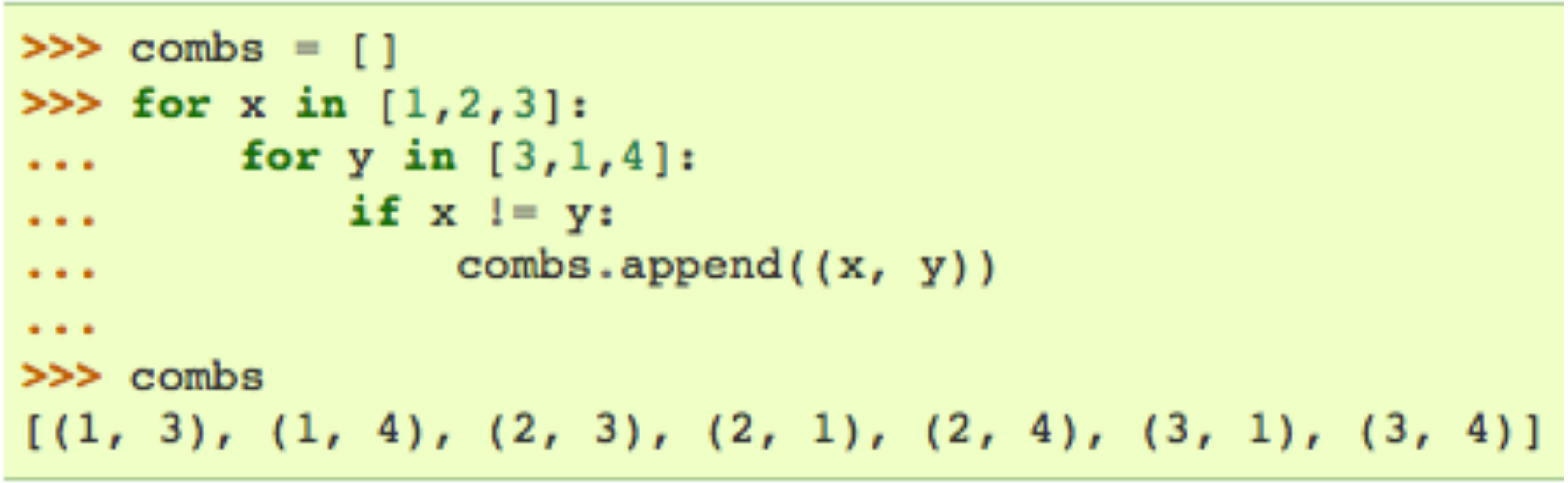
```
>>> squares = []
>>> for x in range(10):
...     squares.append(x**2)
...
>>> squares
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```



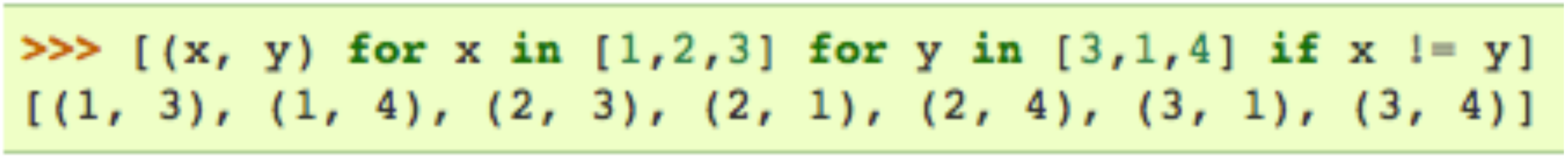
```
squares = [x**2 for x in range(10)]
```

List Comprehension

- A list comprehension consists of brackets containing an expression followed by a for clause, then zero or more for or if clauses. The result will be a new list resulting from evaluating the expression in the context of the for and if clauses which follow it.
- For example, this listcomp combines the elements of two lists if they are not equal:

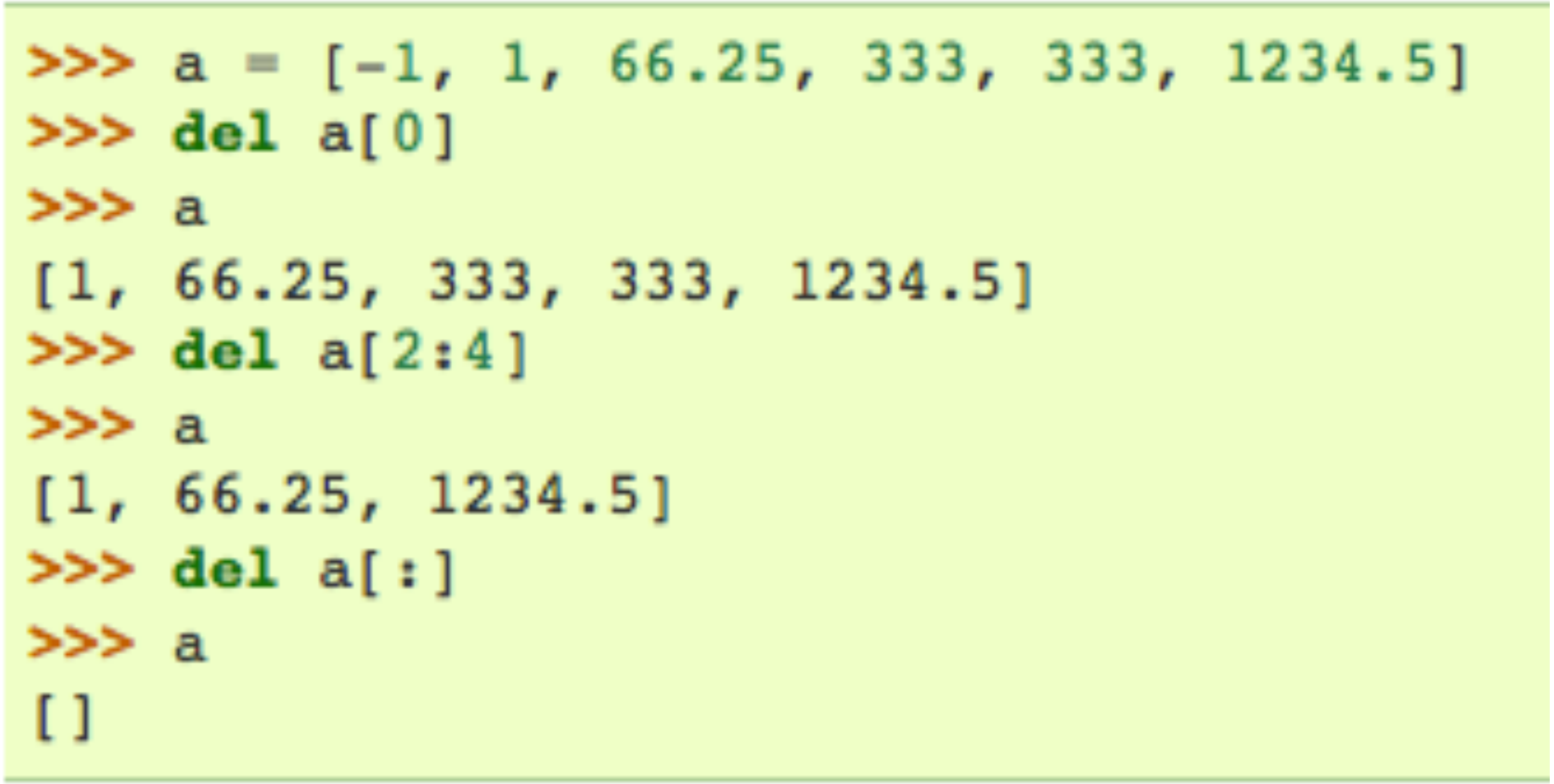


```
>>> combs = []
>>> for x in [1,2,3]:
...     for y in [3,1,4]:
...         if x != y:
...             combs.append((x, y))
...
>>> combs
[(1, 3), (1, 4), (2, 3), (2, 1), (2, 4), (3, 1), (3, 4)]
```



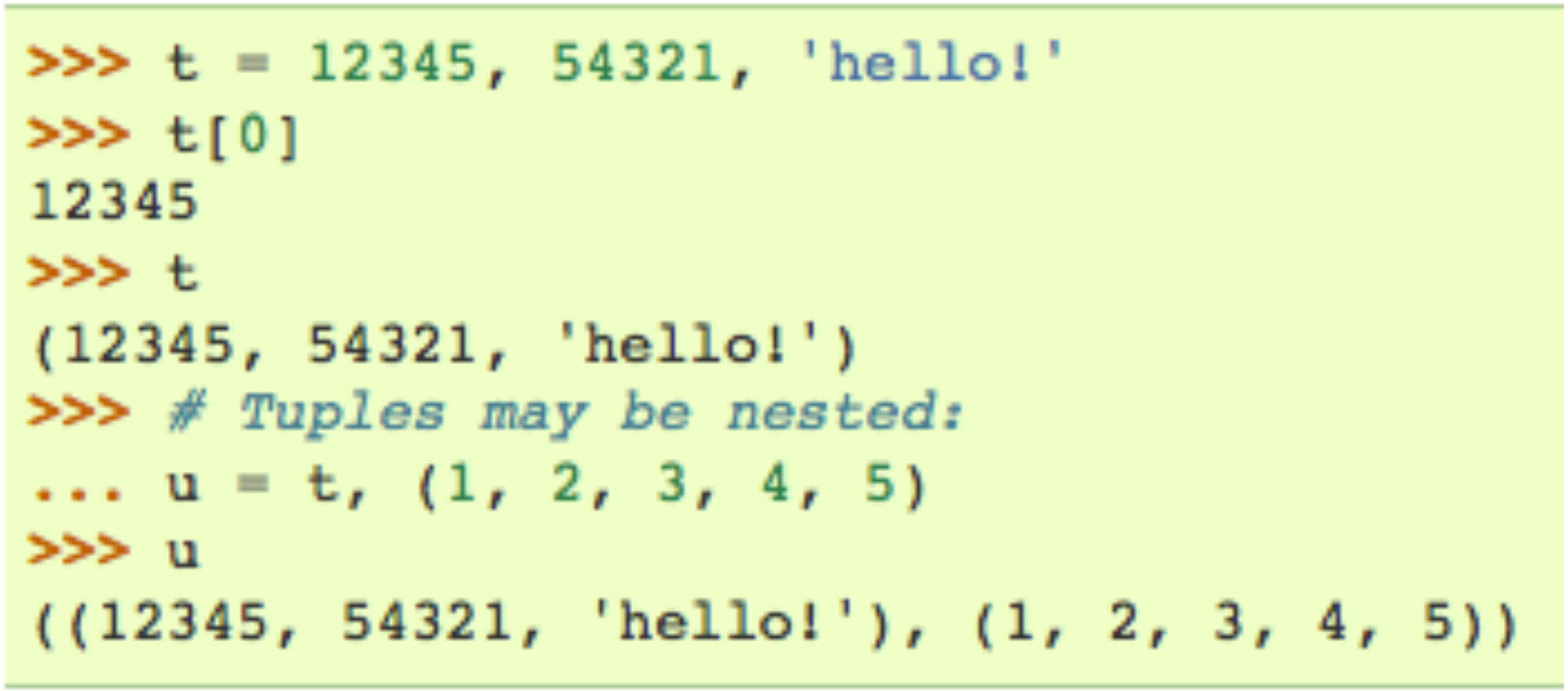
```
>>> [(x, y) for x in [1,2,3] for y in [3,1,4] if x != y]
[(1, 3), (1, 4), (2, 3), (2, 1), (2, 4), (3, 1), (3, 4)]
```

Del()



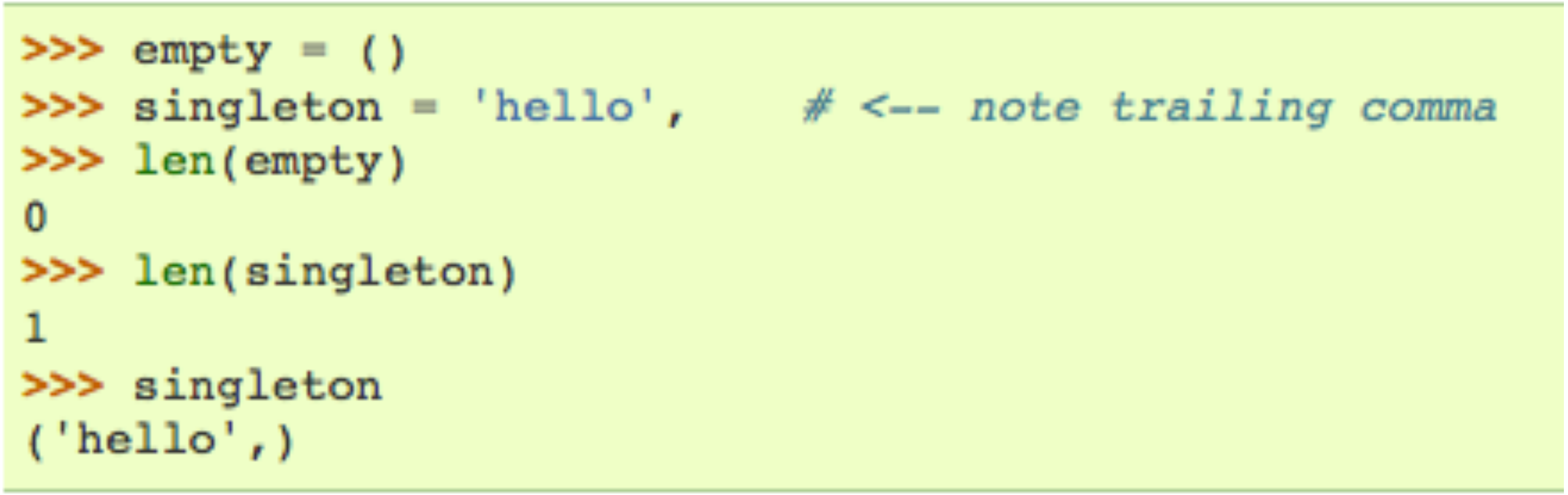
```
>>> a = [-1, 1, 66.25, 333, 333, 1234.5]
>>> del a[0]
>>> a
[1, 66.25, 333, 333, 1234.5]
>>> del a[2:4]
>>> a
[1, 66.25, 1234.5]
>>> del a[:]
>>> a
[]
```

Tuples and Sequences



```
>>> t = 12345, 54321, 'hello!'
>>> t[0]
12345
>>> t
(12345, 54321, 'hello!')
>>> # Tuples may be nested:
... u = t, (1, 2, 3, 4, 5)
>>> u
((12345, 54321, 'hello!'), (1, 2, 3, 4, 5))
```

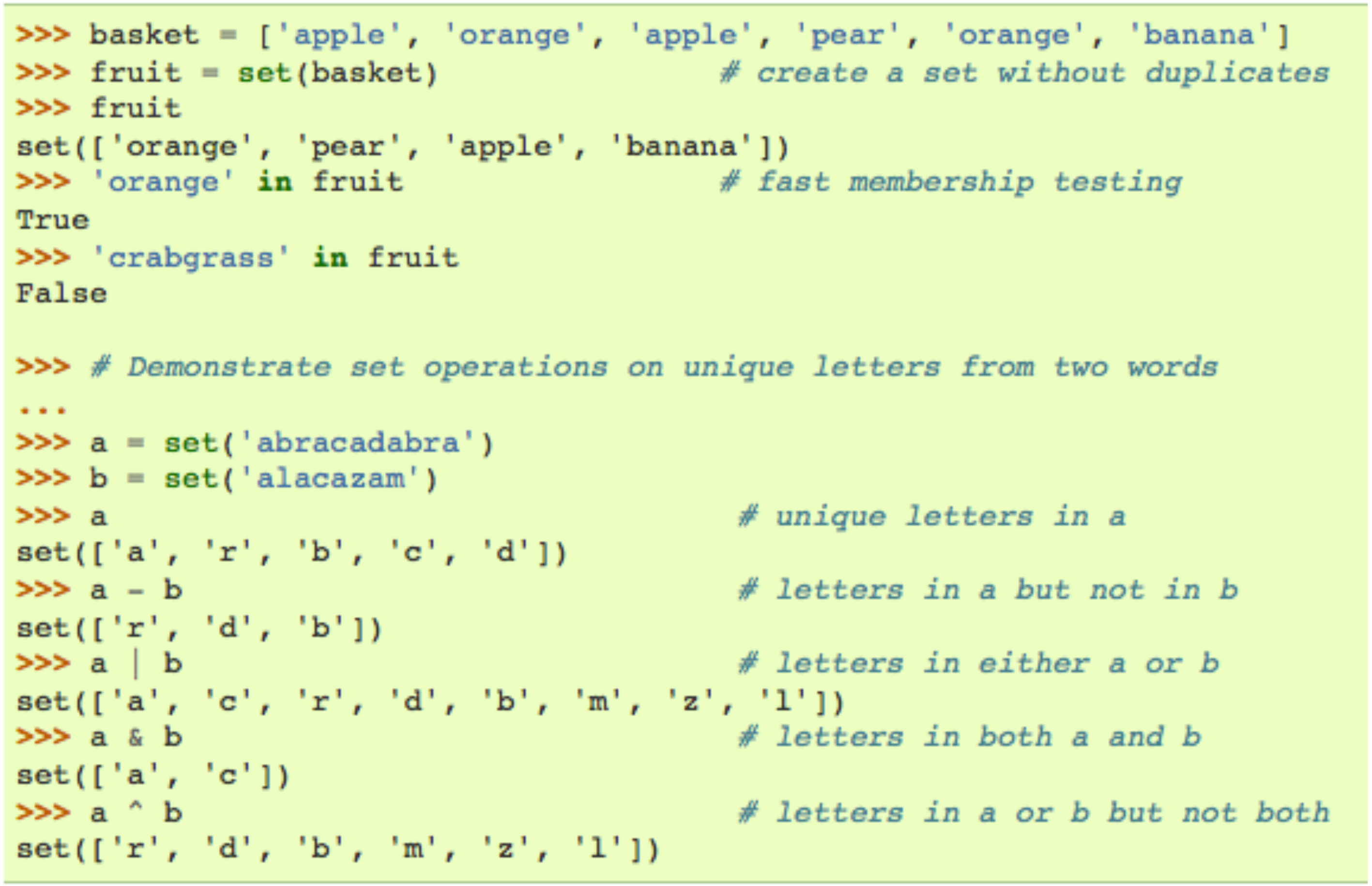
Comma at the end!



```
>>> empty = ()
>>> singleton = 'hello',    # <-- note trailing comma
>>> len(empty)
0
>>> len(singleton)
1
>>> singleton
('hello',)
```

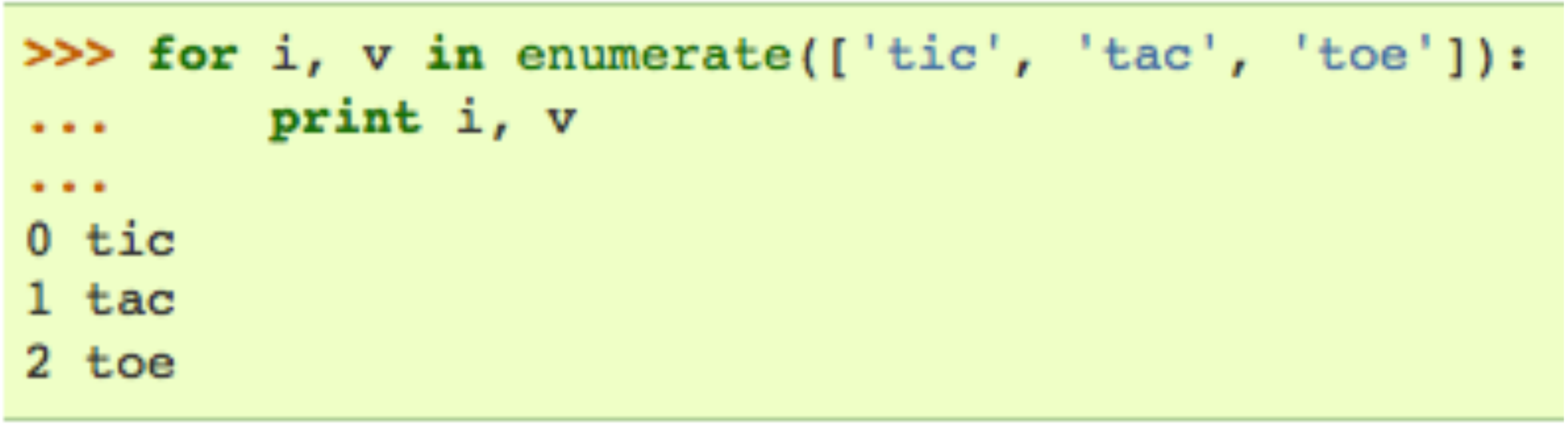
Sets

- A set is an unordered collection with no duplicate elements.
- Basic uses include membership testing and eliminating duplicate entries.
- Set objects also support mathematical operations like union, intersection, difference, and symmetric difference.



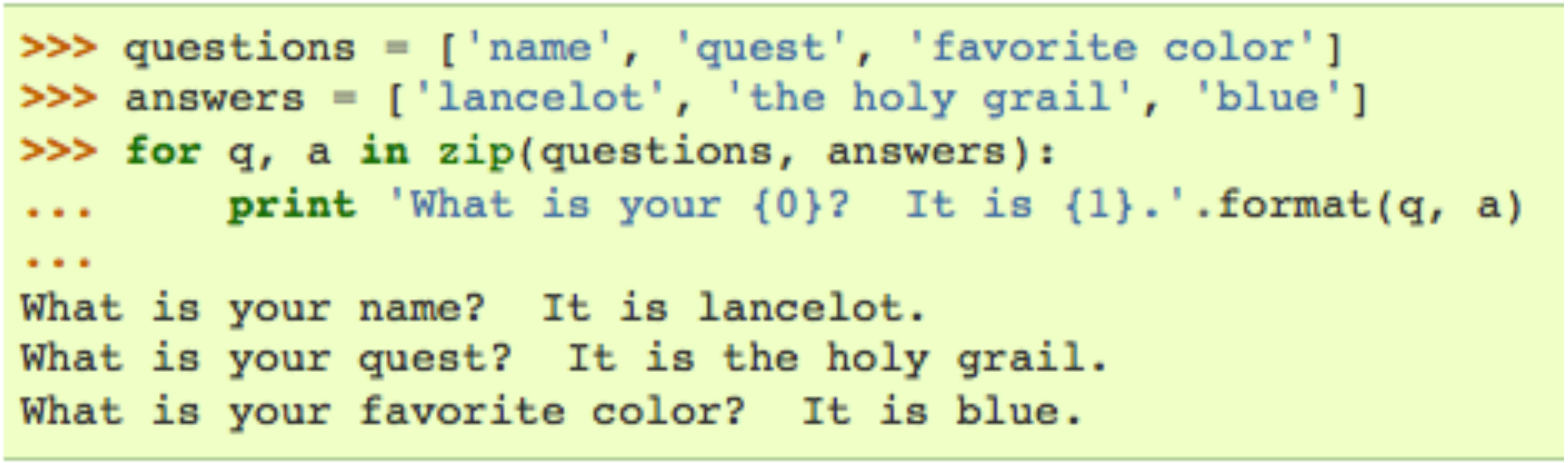
```
>>> basket = ['apple', 'orange', 'apple', 'pear', 'orange', 'banana']
>>> fruit = set(basket)               # create a set without duplicates
>>> fruit
set(['orange', 'pear', 'apple', 'banana'])
>>> 'orange' in fruit                 # fast membership testing
True
>>> 'crabgrass' in fruit
False

>>> # Demonstrate set operations on unique letters from two words
...
>>> a = set('abracadabra')
>>> b = set('alacazam')
>>> a                                  # unique letters in a
set(['a', 'r', 'b', 'c', 'd'])
>>> a - b                              # letters in a but not in b
set(['r', 'd', 'b'])
>>> a | b                              # letters in either a or b
set(['a', 'c', 'r', 'd', 'b', 'm', 'z', 'l'])
>>> a & b                              # letters in both a and b
set(['a', 'c'])
>>> a ^ b                              # letters in a or b but not both
set(['r', 'd', 'b', 'm', 'z', 'l'])
```

Looping Techniques



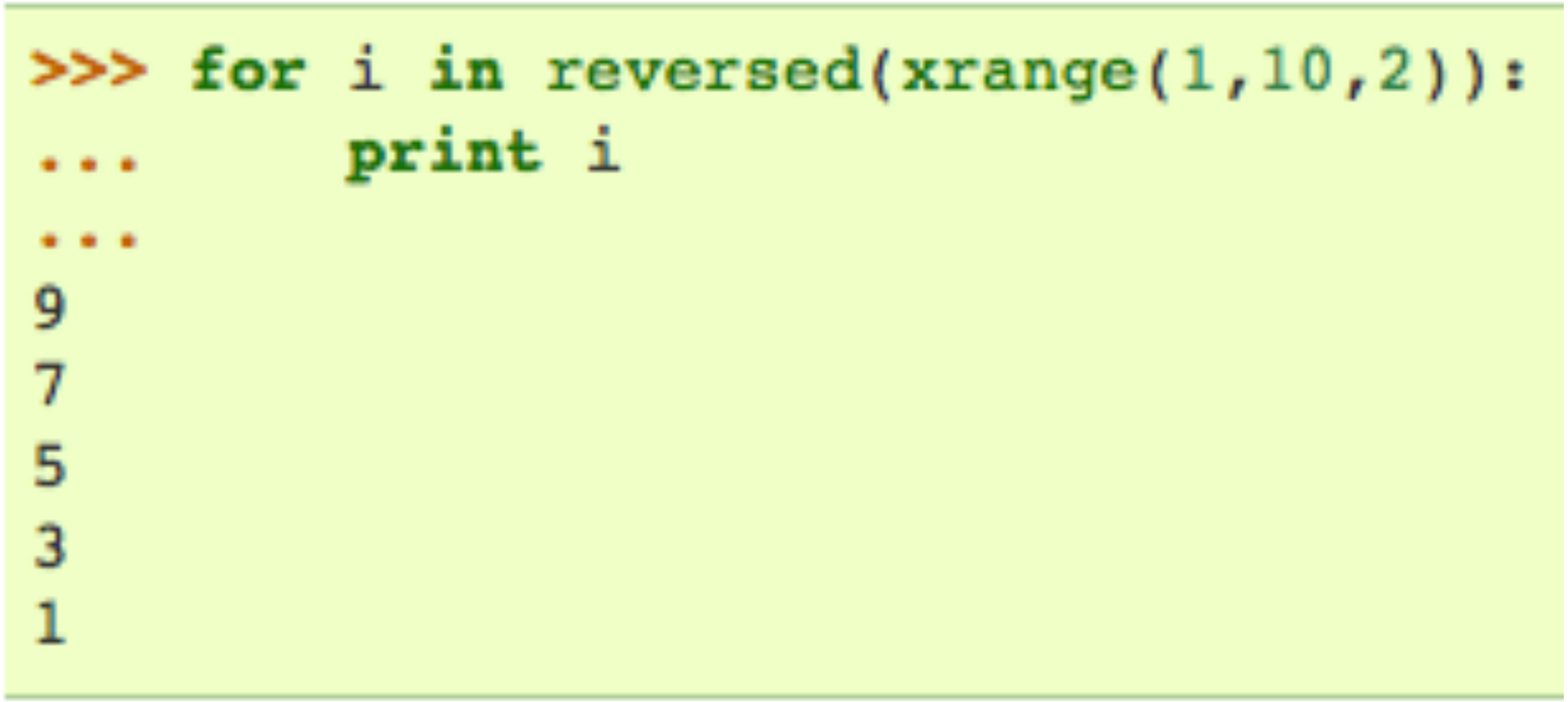
```
>>> for i, v in enumerate(['tic', 'tac', 'toe']):
...     print i, v
...
0 tic
1 tac
2 toe
```

Looping over Two Sequences



```
>>> questions = ['name', 'quest', 'favorite color']
>>> answers = ['lancelot', 'the holy grail', 'blue']
>>> for q, a in zip(questions, answers):
...     print 'What is your {0}?  It is {1}.'.format(q, a)
...
What is your name?  It is lancelot.
What is your quest?  It is the holy grail.
What is your favorite color?  It is blue.
```

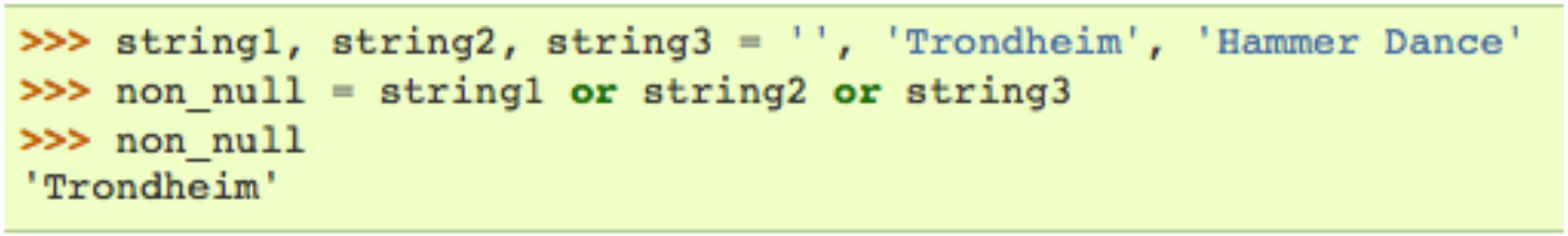
Loop in Reverse



```
>>> for i in reversed(xrange(1,10,2)):
...     print i
...
9
7
5
3
1
```

More on Conditions

- Include and, or



```
>>> string1, string2, string3 = '', 'Trondheim', 'Hammer Dance'
>>> non_null = string1 or string2 or string3
>>> non_null
'Trondheim'
```

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.

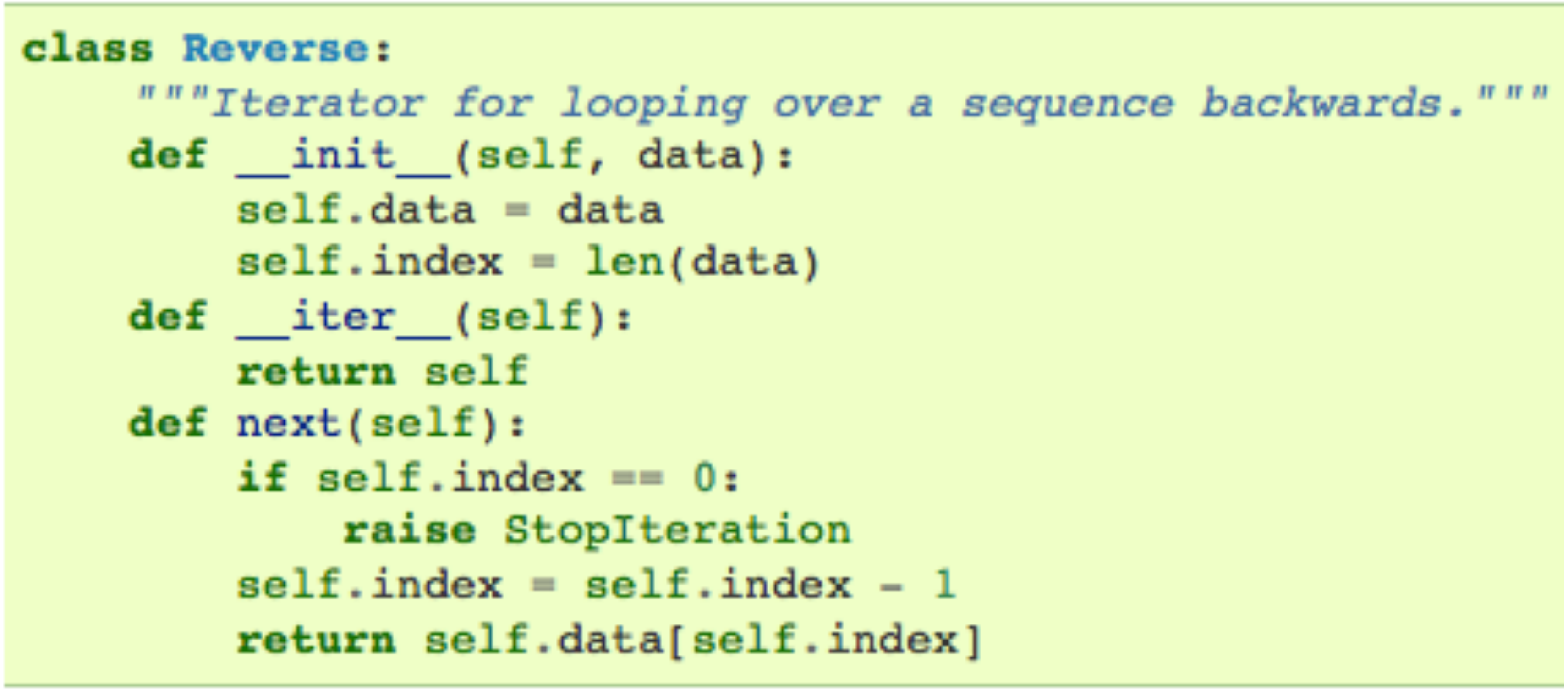
Some Tips for Experts

- 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.

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__