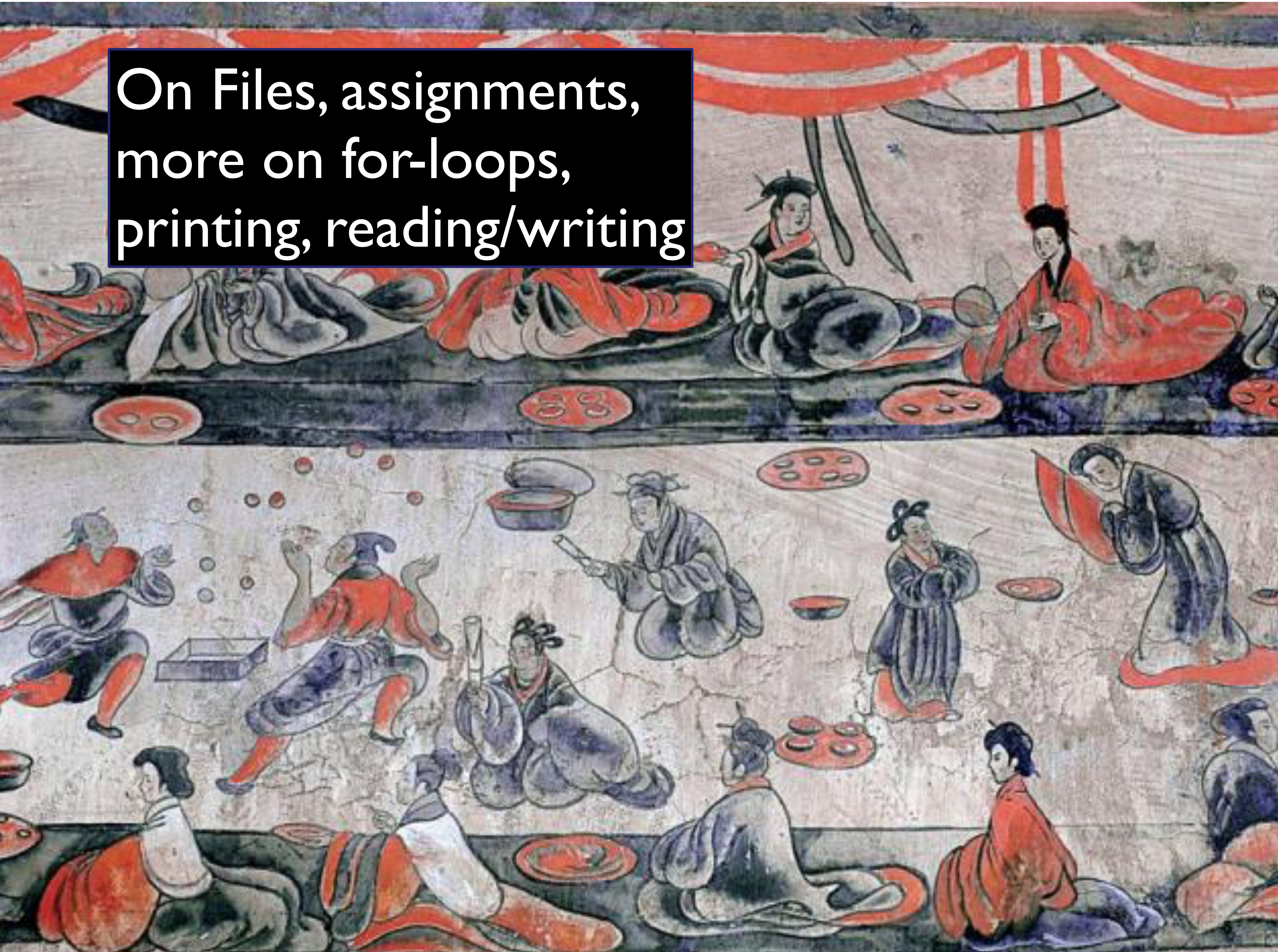
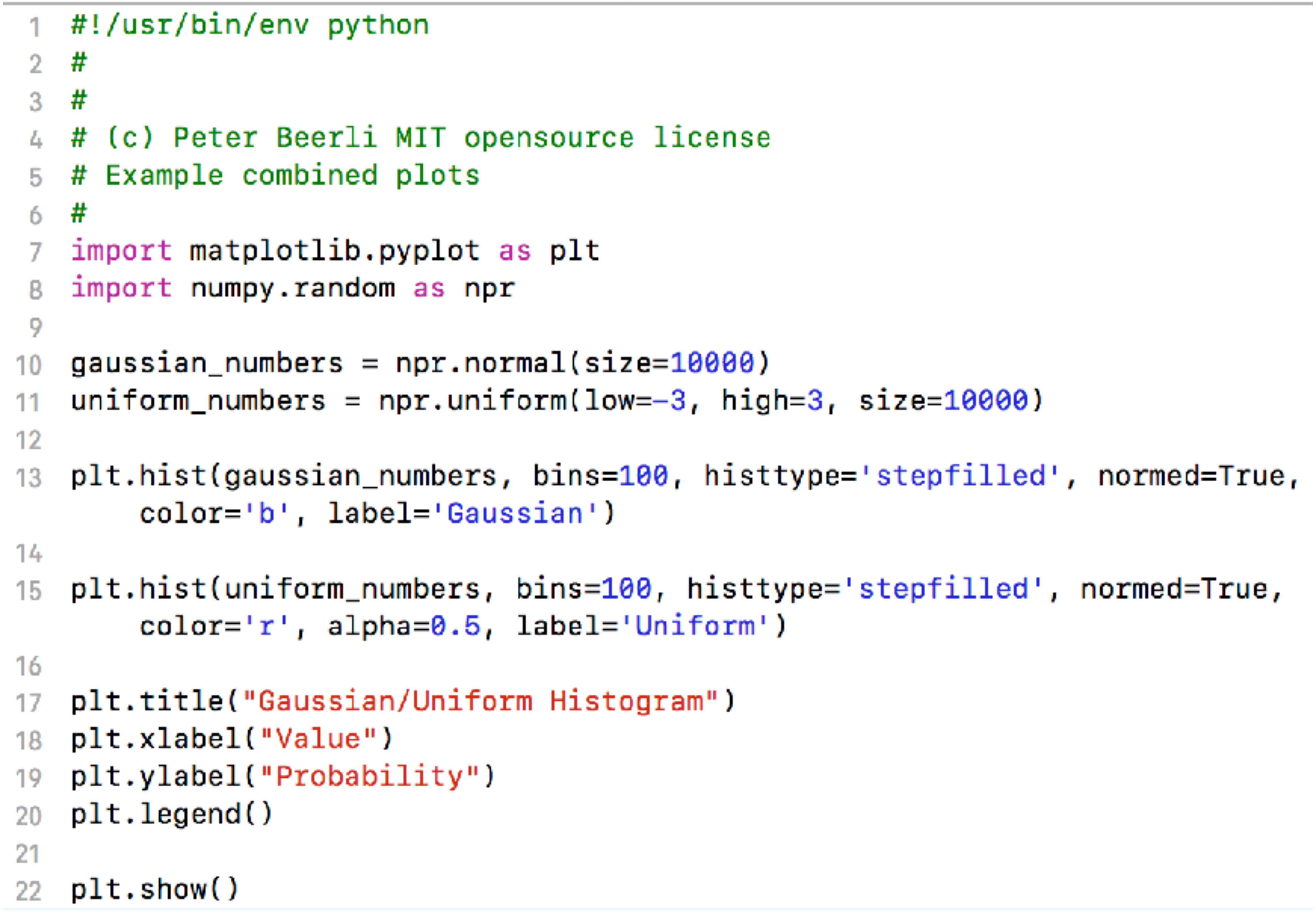
On Files, assignments, more on for-loops, printing, reading/writing





```
#!/usr/bin/env python
#
#
# (c) Peter Beerli MIT opensource license
# Example combined plots
#
import matplotlib.pyplot as plt
import numpy.random as npr

gaussian_numbers = npr.normal(size=10000)
uniform_numbers = npr.uniform(low=-3, high=3, size=10000)

plt.hist(gaussian_numbers, bins=100, histtype='stepfilled', normed=True,
    color='b', label='Gaussian')

plt.hist(uniform_numbers, bins=100, histtype='stepfilled', normed=True,
    color='r', alpha=0.5, label='Uniform')

plt.title("Gaussian/Uniform Histogram")
plt.xlabel("Value")
plt.ylabel("Probability")
plt.legend()

plt.show()
```

4.4. **break** and **continue** Statements, and **else** Clauses on Loops¶

The **break** statement, like in C, breaks out of the smallest enclosing **for** or **while** loop.

```
>>> for n in range(2, 10):
...     for x in range(2, n):
...         if n % x == 0:
...             print n, 'equals', x, '*', n/x
...             break
...     else:
...         # loop fell through without finding a factor
...         print n, 'is a prime number'
...
2 is a prime number
3 is a prime number
4 equals 2 * 2
5 is a prime number
6 equals 2 * 3
7 is a prime number
8 equals 2 * 4
9 equals 3 * 3
```

(Yes, this is the correct code. Look closely: the `else` clause belongs to the **for** loop, **not** the **if** statement.)

The **continue** statement, also borrowed from C, continues with the next iteration of the loop:

```
>>>
>>> for num in range(2, 10):
...     if num % 2 == 0:
...         print "Found an even number", num
...         continue
...     print "Found a number", num
Found an even number 2
Found a number 3
Found an even number 4
Found a number 5
Found an even number 6
Found a number 7
Found an even number 8
Found a number 9
```

7.1. Fancier Output Formatting¶

So far we’ve encountered two ways of writing values: *expression statements* and the **print** statement. (A third way is using the **write()** method of file objects; the standard output file can be referenced as `sys.stdout`. See the Library Reference for more information on this.)

Often you’ll want more control over the formatting of your output than simply printing space-separated values. There are two ways to format your output; the first way is to do all the string handling yourself; using string slicing and concatenation operations you can create any layout you can imagine. The string types have some methods that perform useful operations for padding strings to a given column width; these will be discussed shortly. The second way is to use the **str.format()** method.

The **string** module contains a **Template** class which offers yet another way to substitute values into strings.

One question remains, of course: how do you convert values to strings? Luckily, Python has ways to convert any value to a string: pass it to the **repr()** or **str()** functions.

The **str()** function is meant to return representations of values which are fairly human-readable, while **repr()** is meant to generate representations which can be read by the interpreter (or will force a **SyntaxError** if there is no equivalent syntax). For objects which don’t have a particular representation for human consumption, **str()** will return the same value as **repr()**. Many values, such as numbers or structures like lists and dictionaries, have the same representation using either function. Strings and floating point numbers, in particular, have two distinct representations.

Some examples:

```
>>>
>>> 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'))"
```

```
>>> 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
```

Basic usage of the **str.format()** method looks like this:

```
>>>
>>> print 'We are the {} who say "{}!"'.format('knights',
'Ni')
We are the knights who say "Ni!"
```

The brackets and characters within them (called format fields) are replaced with the objects passed into the **str.format()** method. A number in the brackets refers to the position of the object passed into the **str.format()** method.

```
>>>
>>> print '{0} and {1}'.format('spam', 'eggs')
spam and eggs
>>> print '{1} and {0}'.format('spam', 'eggs')
eggs and spam
```

If keyword arguments are used in the **str.format()** method, their values are referred to by using the name of the argument.

```
>>>
>>> print 'This {food} is {adjective}.'.format(
...       food='spam', adjective='absolutely horrible')
This spam is absolutely horrible.
```

Positional and keyword arguments can be arbitrarily combined:

```
>>>
>>> print 'The story of {0}, {1}, and
{other}.'.format('Bill', 'Manfred',
...
other='Georg')
The story of Bill, Manfred, and Georg.
```

`open()` returns a file object, and is most commonly used with two arguments: `open(filename, mode)`.

```
>>>

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

The first argument is a string containing the filename. The second argument is another string containing a few characters describing the way in which the file will be used. *mode* can be `'r'` when the file will only be read, `'w'` for only writing (an existing file with the same name will be erased), and `'a'` opens the file for appending; any data written to the file is automatically added to the end. `'r+'` opens the file for both reading and writing. The *mode* argument is optional; `'r'` will be assumed if it's omitted.

On Windows, `'b'` appended to the mode opens the file in binary mode, so there are also modes like `'rb'`, `'wb'`, and `'r+b'`. Python on Windows makes a distinction between text and binary files; the end-of-line characters in text files are automatically altered slightly when data is read or written. This behind-the-scenes modification to file data is fine for ASCII text files, but it'll corrupt binary data like that in `JPEG` or `EXE` files. Be very careful to use binary mode when reading and writing such files. On Unix, it doesn't hurt to append a `'b'` to the mode, so you can use it platform-independently for all binary files.

To read a file’s contents, call `f.read(size)`, which reads some quantity of data and returns it as a string. *size* is an optional numeric argument. When *size* is omitted or negative, the entire contents of the file will be read and returned; it’s your problem if the file is twice as large as your machine’s memory. Otherwise, at most *size* bytes are read and returned. If the end of the file has been reached, `f.read()` will return an empty string (`""`).

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

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

For reading lines from a file, you can loop over the file object. This is efficient, fast, and leads to simple code:

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

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

`f.write(string)` writes the contents of *string* to the file, returning `None`.

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

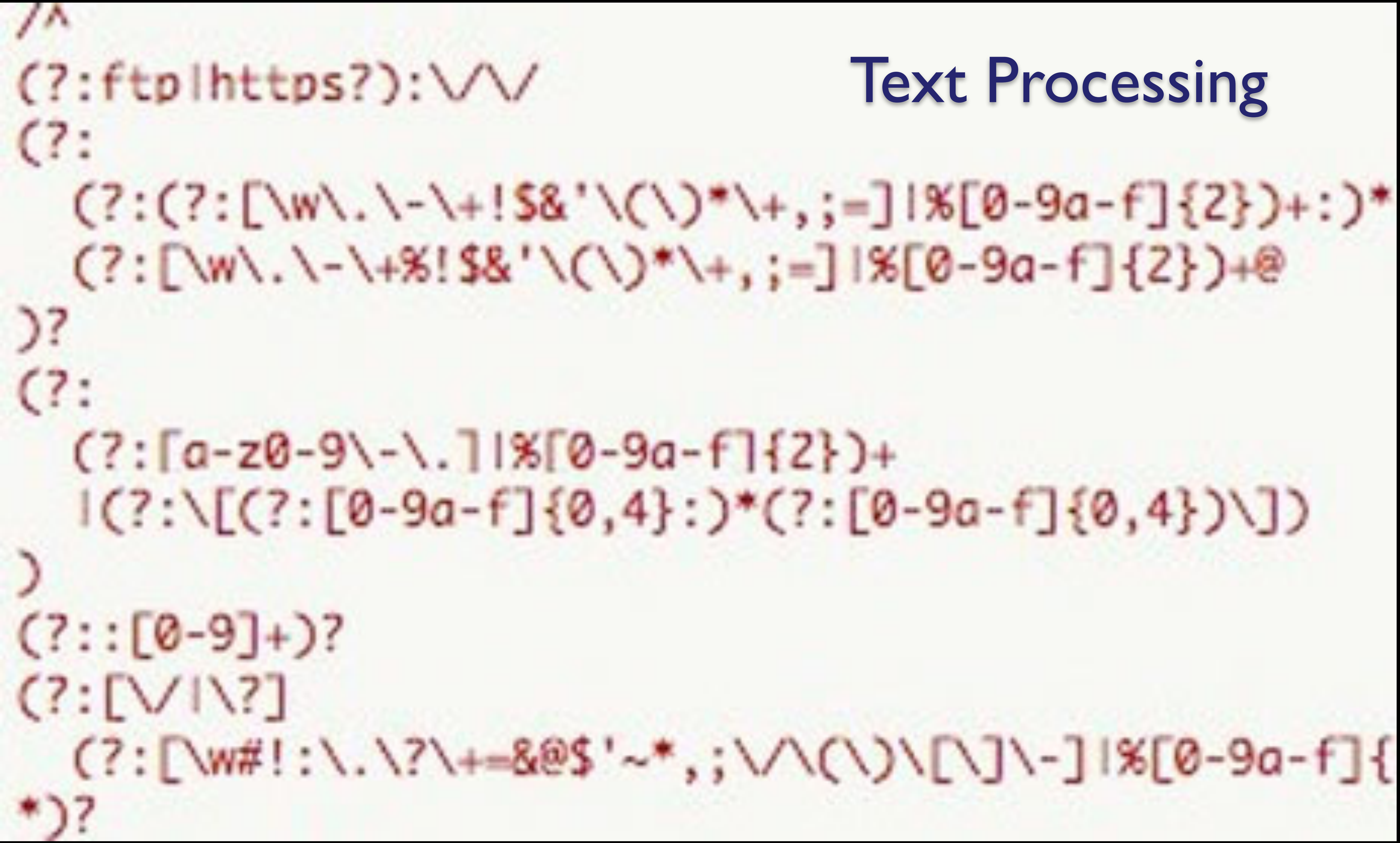
To write something other than a string, it needs to be converted to a string first:

```
>>> value = ('the answer', 42)
>>> s = str(value)
>>> f.write(s)
```

It is good practice to use the **with** keyword when dealing with file objects. This has the advantage that the file is properly closed after its suite finishes, even if an exception is raised on the way. It is also much shorter than writing equivalent **try**-**finally** blocks:

```
>>>
>>> with open('workfile', 'r') as f:
...     read_data = f.read()
>>> f.closed
True

>>> with open('workfile', 'r') as f:
...     for line in f:
...         read_data.append(line)
>>> f.closed
True
```

Text Processing



```
/^
(?:ftp|https?):\/\/
(?:
  (?:(?:[\w\.\-\+!$&'\(\)*\+,;=]|%[0-9a-f]{2})+:)*
  (?:[\w\.\-\+%!$&'\(\)*\+,;=]|%[0-9a-f]{2})+@
)?
(?:
  (?:[a-z0-9\-\.]|%[0-9a-f]{2})+
  |(?:\[(?:[0-9a-f]{0,4}:)*(?:[0-9a-f]{0,4})\])
)
(?::[0-9]+)?
(?:[\/|\?]
  (?:[\w#!:\.\?\+=&@$'~*,;\/\(\)\[\]\-]|%[0-9a-f]{
*)?
```

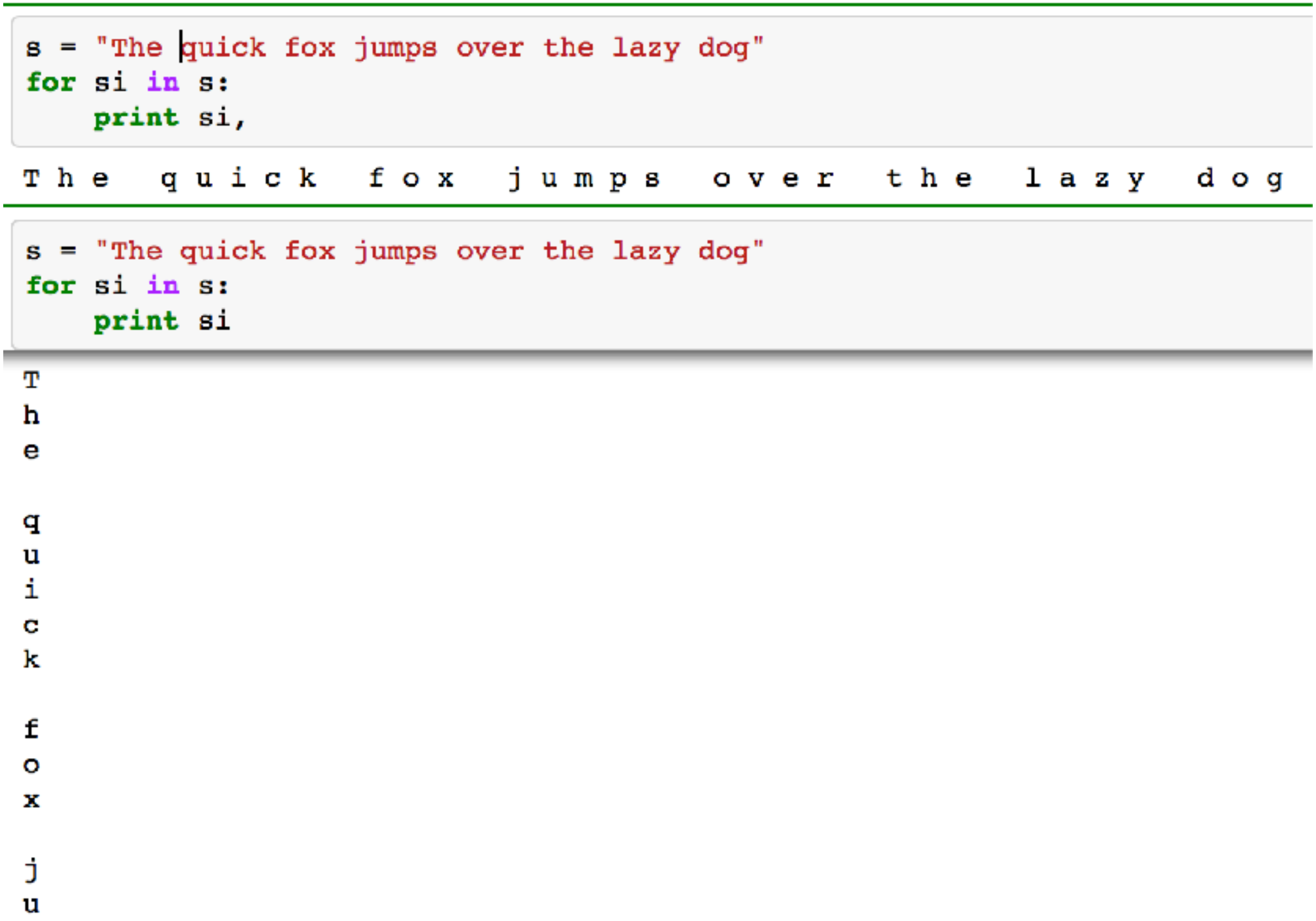
Text Manipulation

- Parsing text
  - extract tokens and understand their significance
- Text transformation
  - change all words to lower case
  - replace multiple consecutive spaces by a single space
- Regular expressions
  - identify text with specific structure

Strings

- s = “hello world”
- s = “”””go home
  class dismissed”
- s = ‘gone with the wind’
- Strings are immutable.
- Strings are similar to sequences
  - s[3]   # ‘n’
  - s[3:6]   # ‘e wi’

Loops with String



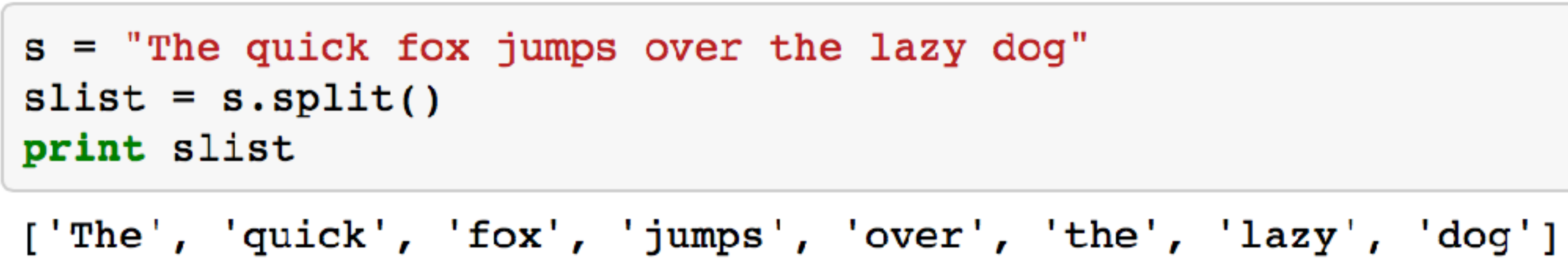
```
s = "The quick fox jumps over the lazy dog"
for si in s:
    print si,
```

```
T h e   q u i c k   f o x   j u m p s   o v e r   t h e   l a z y   d o g
```

```
s = "The quick fox jumps over the lazy dog"
for si in s:
    print si
```

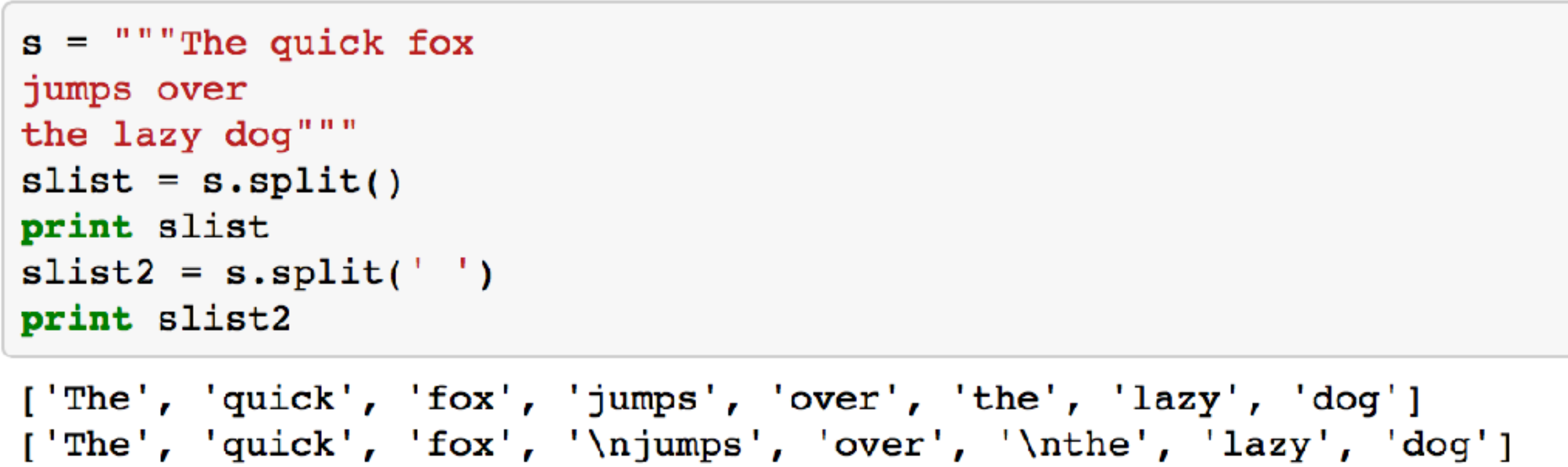
```
T
h
e

q
u
i
c
k

f
o
x

j
u
```

Splitting text



```
s = "The quick fox jumps over the lazy dog"
slist = s.split()
print slist
```

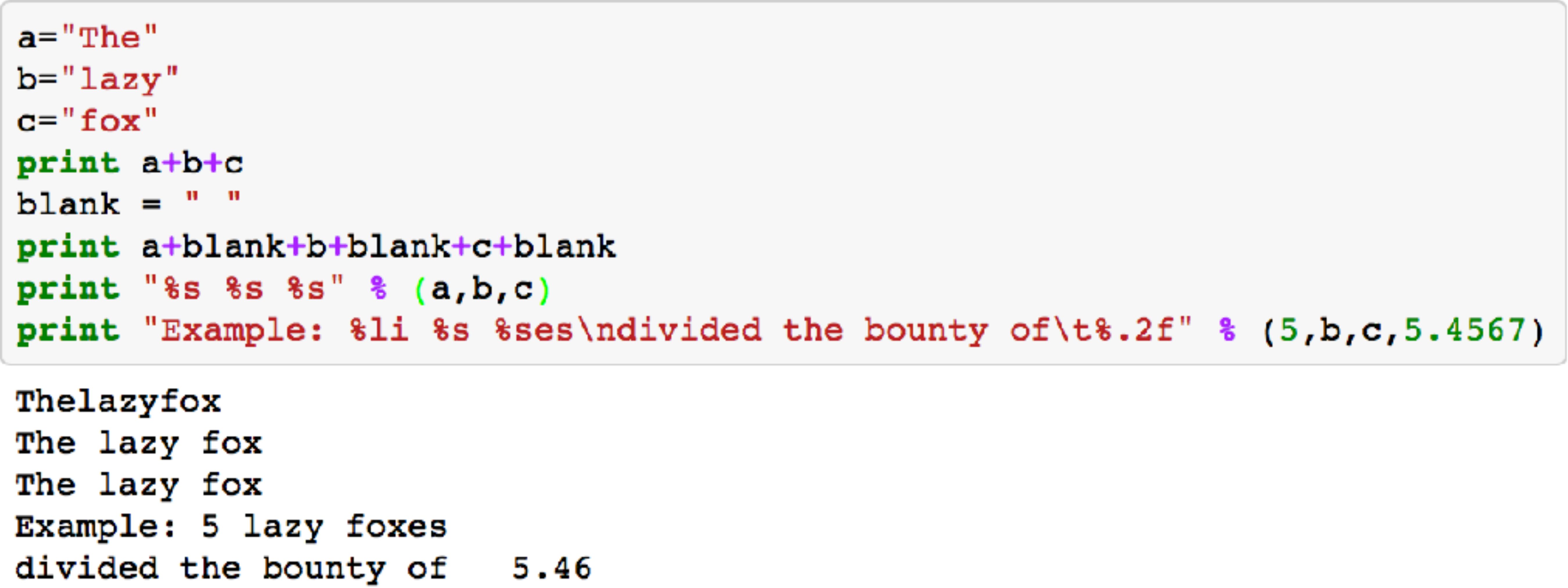
```
['The', 'quick', 'fox', 'jumps', 'over', 'the', 'lazy', 'dog']
```



```
s = """The quick fox
jumps over
the lazy dog"""
slist = s.split()
print slist
slist2 = s.split(' ')
print slist2
```

```
['The', 'quick', 'fox', 'jumps', 'over', 'the', 'lazy', 'dog']
['The', 'quick', 'fox', '\njumps', 'over', '\nthe', 'lazy', 'dog']
```

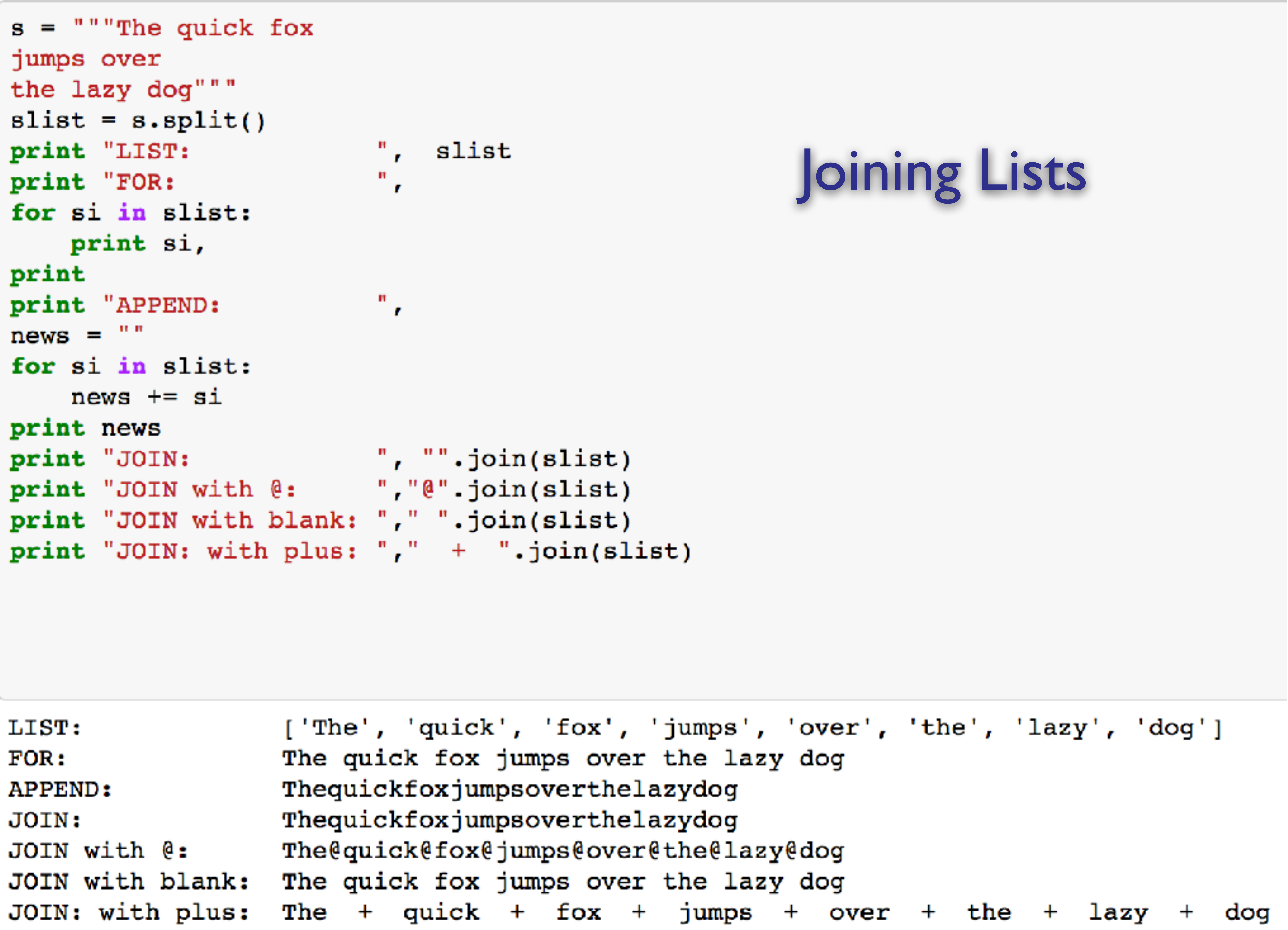
Joining and printing



```python
a="The"
b="lazy"
c="fox"
print a+b+c
blank = " "
print a+blank+b+blank+c+blank
print "%s %s %s" % (a,b,c)
print "Example: %li %s %ses\ndivided the bounty of\t%.2f" % (5,b,c,5.4567)
```

```
Thelazyfox
The lazy fox
The lazy fox
Example: 5 lazy foxes
divided the bounty of   5.46
```

Joining Lists



```
s = """The quick fox
jumps over
the lazy dog"""
slist = s.split()
print "LIST:            ",  slist
print "FOR:             ",
for si in slist:
    print si,
print
print "APPEND:          ",
news = ""
for si in slist:
    news += si
print news
print "JOIN:            ", "".join(slist)
print "JOIN with @:     ","@".join(slist)
print "JOIN with blank: "," ".join(slist)
print "JOIN: with plus: ","  +  ".join(slist)
```

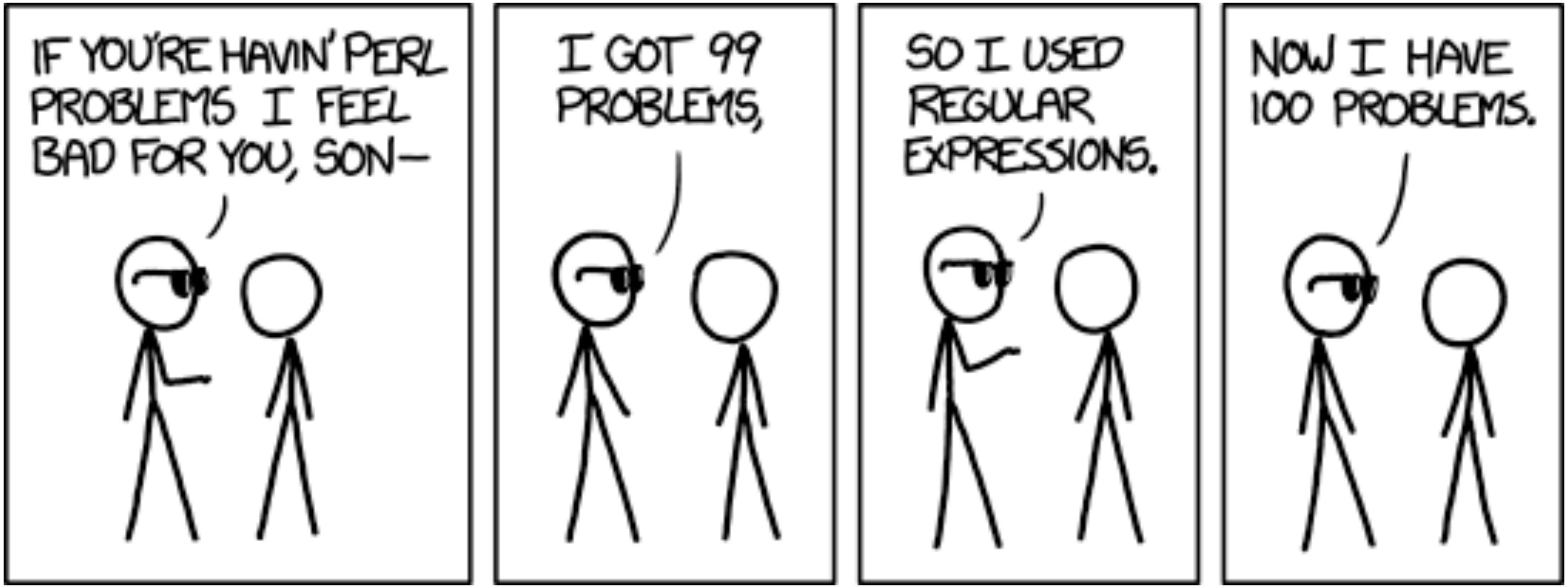
```
LIST:             ['The', 'quick', 'fox', 'jumps', 'over', 'the', 'lazy', 'dog']
FOR:              The quick fox jumps over the lazy dog
APPEND:           Thequickfoxjumpsoverthelazydog
JOIN:             Thequickfoxjumpsoverthelazydog
JOIN with @:      The@quick@fox@jumps@over@the@lazy@dog
JOIN with blank:  The quick fox jumps over the lazy dog
JOIN: with plus:  The  +  quick  +  fox  +  jumps  +  over  +  the  +  lazy  +  dog
```

Regular Expressions

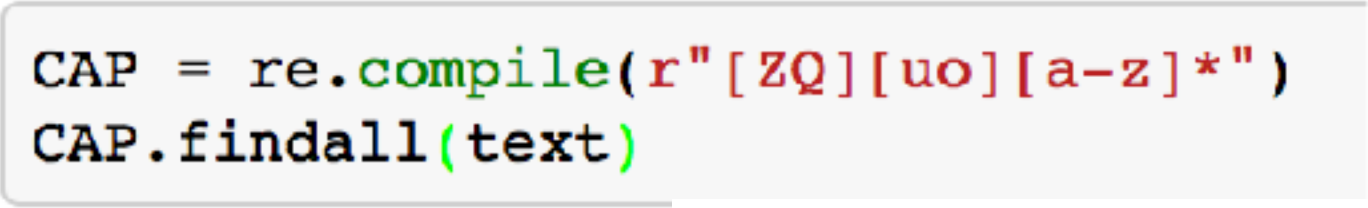
- Documentation
  - http://docs.python.org/library/re.html
- Tutorial
  - http://docs.python.org/howto/regex.html

- regular expression module
  - import re

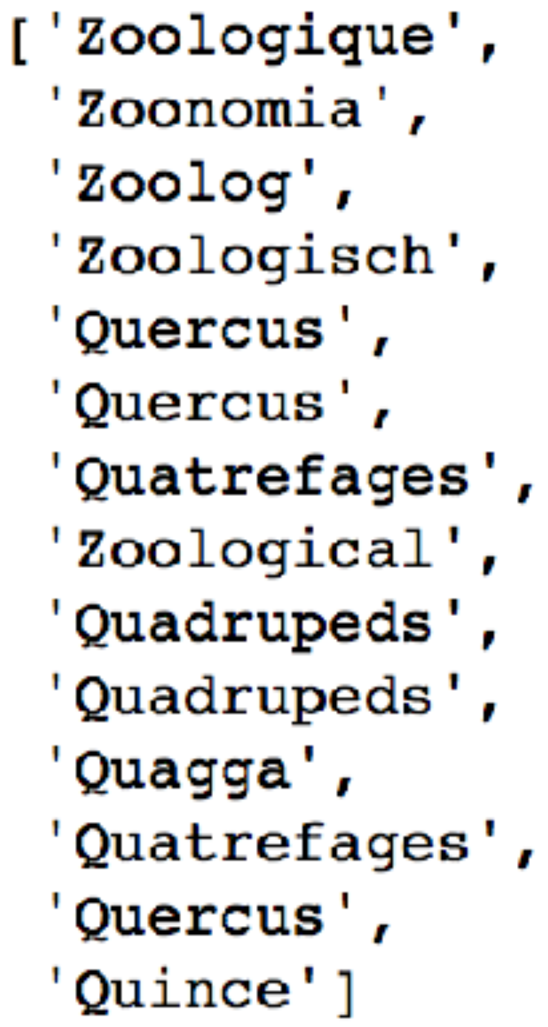
```
'compile','copy_reg','error','escape','findall','finditer','
match','purge','search','split','sub',
```


IF YOU'RE HAVIN' PERL PROBLEMS I FEEL BAD FOR YOU, SON–
I GOT 99 PROBLEMS,
SO I USED REGULAR EXPRESSIONS.
NOW I HAVE 100 PROBLEMS.

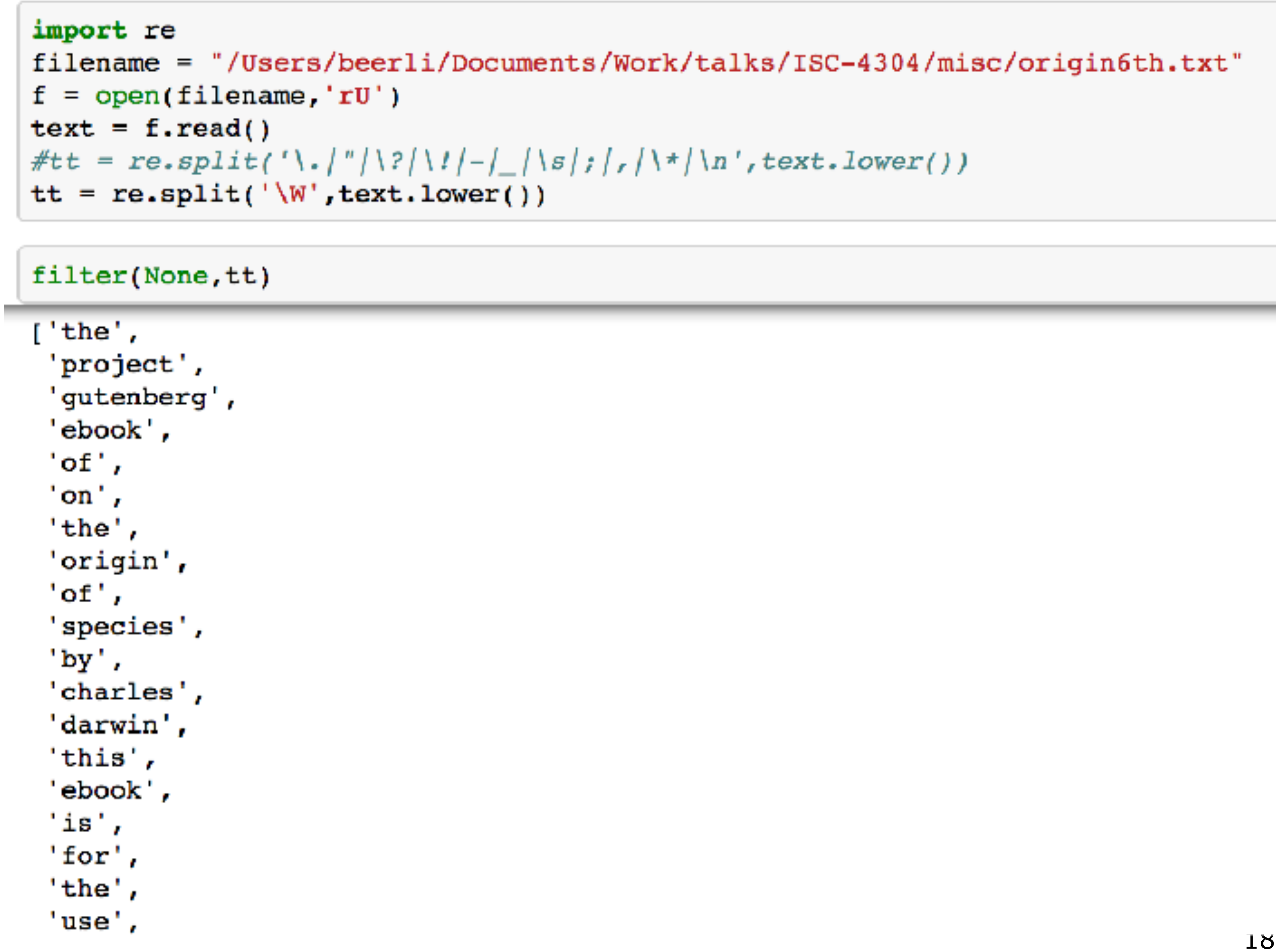
Simple Searches



```
CAP = re.compile(r"[ZQ][uo][a-z]*")
CAP.findall(text)
```



```
['Zoologique',
 'Zoonomia',
 'Zoolog',
 'Zoologisch',
 'Quercus',
 'Quercus',
 'Quatrefages',
 'Zoological',
 'Quadrupeds',
 'Quadrupeds',
 'Quagga',
 'Quatrefages',
 'Quercus',
 'Quince']
```



```
import re
filename = "/Users/beerli/Documents/Work/talks/ISC-4304/misc/origin6th.txt"
f = open(filename,'rU')
text = f.read()
#tt = re.split('\.|"|\?|\!|-|_|\s|;|,|\*|\n',text.lower())
tt = re.split('\W',text.lower())
```

```
filter(None,tt)
```

```
['the',
 'project',
 'gutenberg',
 'ebook',
 'of',
 'on',
 'the',
 'origin',
 'of',
 'species',
 'by',
 'charles',
 'darwin',
 'this',
 'ebook',
 'is',
 'for',
 'the',
 'use',
```

Repetition

- pattern = “a*b” # 0 or more
- pattern = “a+b” # 1 or more
- pattern = “a?b”   # 0 or 1
- pattern = “a{2}” # 2 copies of a

Special Forms

- . : any character
- $: end of string
- ^ : beginning of string
- a-z : "-" range of characters
- [a-c3-5] : any of the characters within [ ]
- [^a-c] : all characters *except* [abc]

Further abbreviations

- \d : digit character class: [0-9]
- \D : non-digit character
- \w : alphanumeric char
- \W: non-alphanumeric char
- \A : beginning of string
- etc.

Greediness

- By default, matching generates the longest possible match: *greedy* (*,+,?)
- It is possible to reverse the behavior to non-greedy (*?, +?, ??)

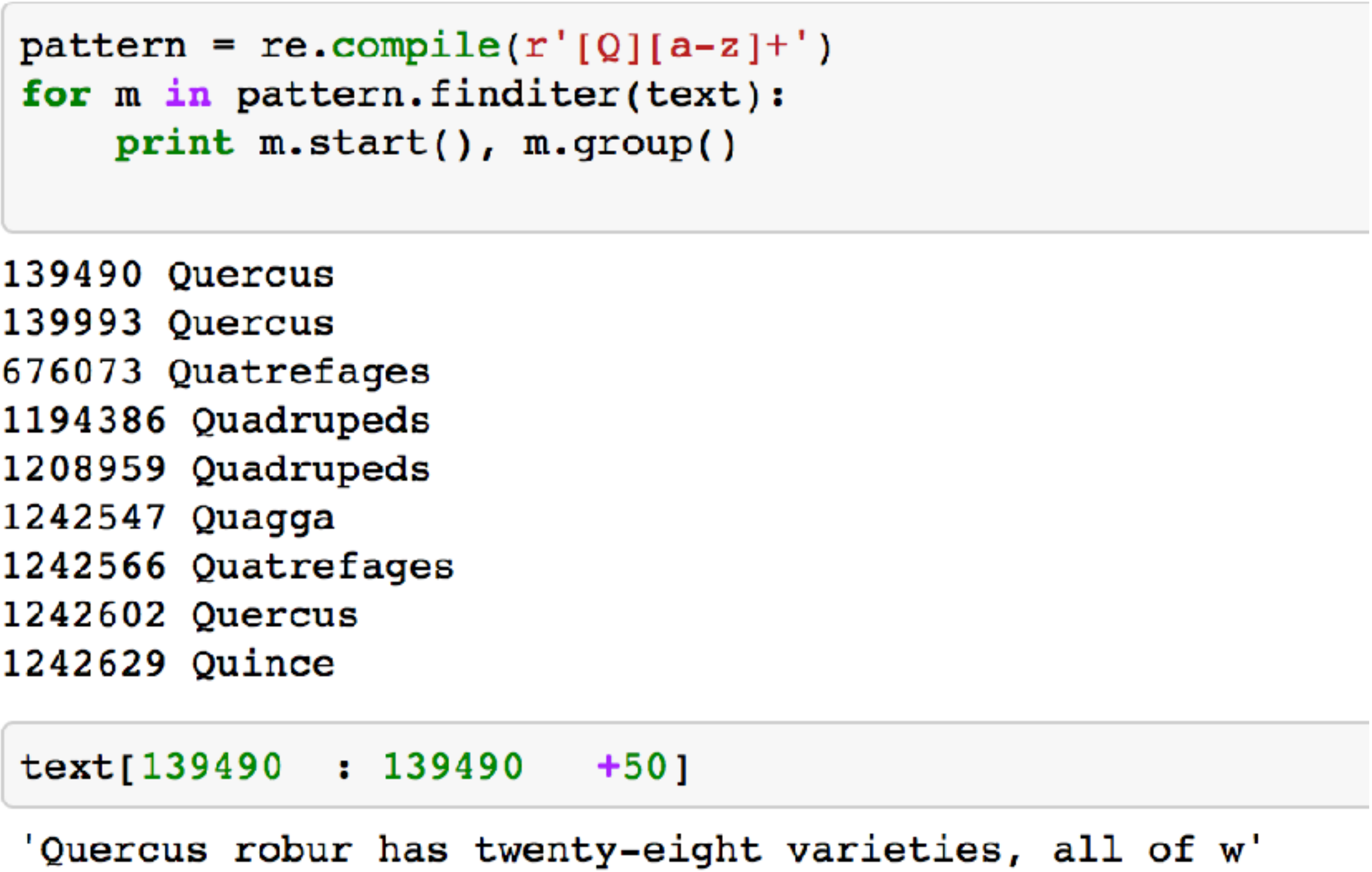
Greedy vs non-greedy

- tx = “abab c4 ab”
- re.search(“a.*b”, tx) # ==> “ab c4 ab”
- re.search(“a.?*b”, tx) # ==> “ab”
- re.search(“[ab]{2}.*[ab]”) ==> “abab c4 ab”

A few methods from re module

- re.search(pattern, str)
  - returns a *matchObject* for the *leftmost* substring
- re.sub(pattern, replace, str)
  - return string with pattern replaced by replace
- re.findall(pattern, str)
  - return a list of nonoverlapping patterns in string
- re.compile(pattern, flags)
  - compile the pattern for efficiency

Locating matches



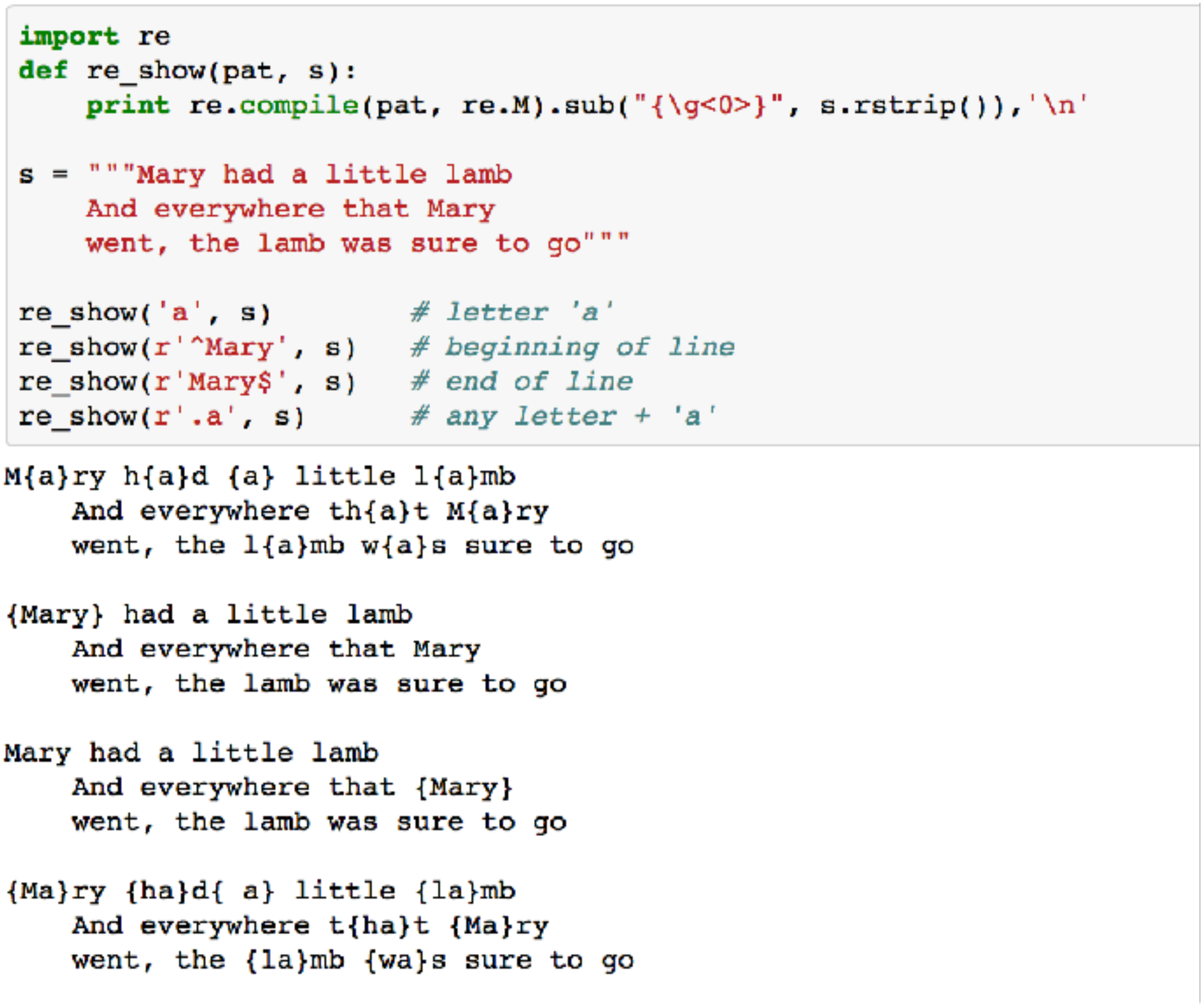
```
pattern = re.compile(r'[Q][a-z]+')
for m in pattern.finditer(text):
    print m.start(), m.group()
```

```
139490 Quercus
139993 Quercus
676073 Quatrefages
1194386 Quadrupeds
1208959 Quadrupeds
1242547 Quagga
1242566 Quatrefages
1242602 Quercus
1242629 Quince
```

```
text[139490  : 139490   +50]
```

```
'Quercus robur has twenty-eight varieties, all of w'
```

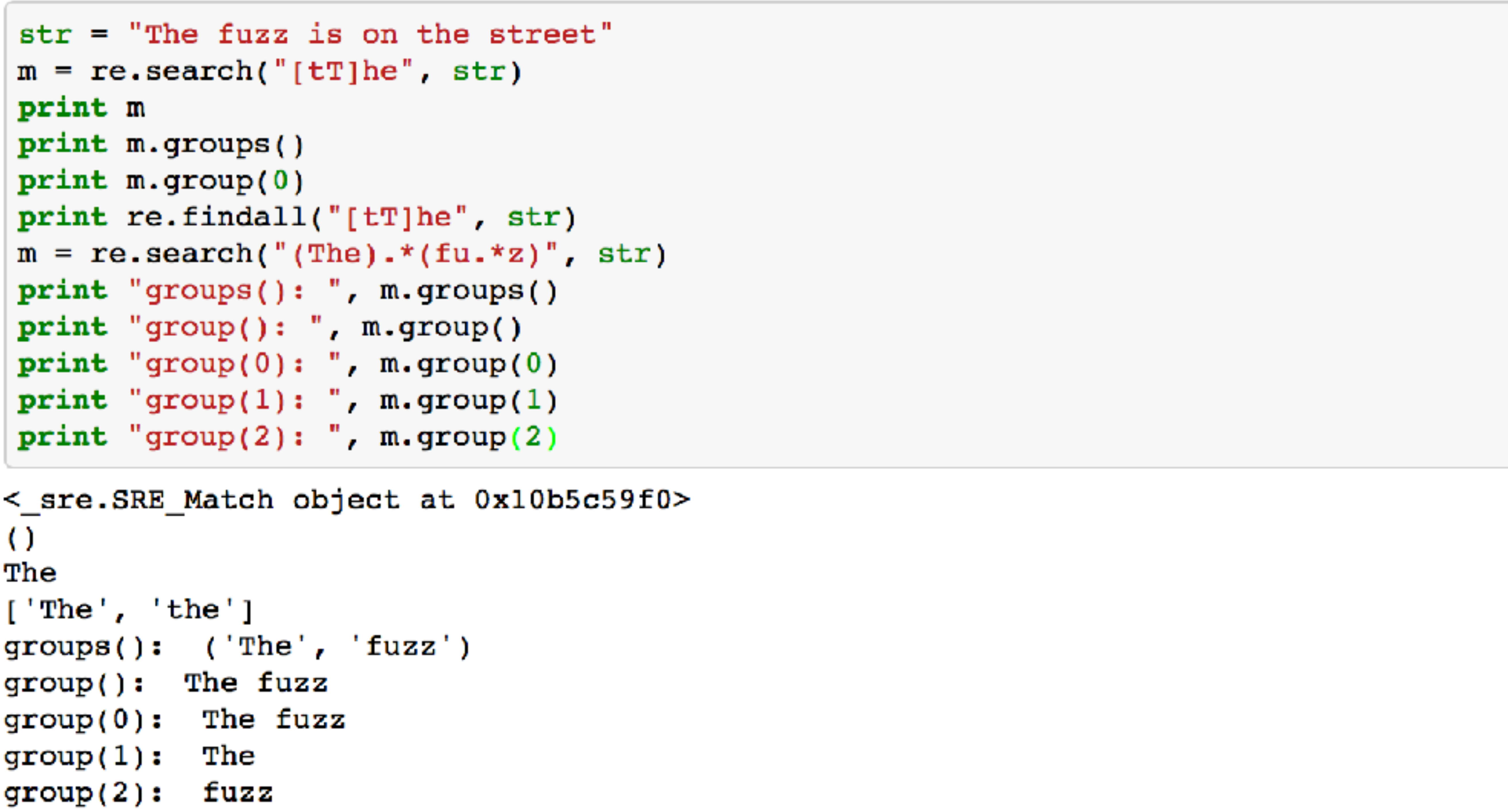
re_show()



```
import re
def re_show(pat, s):
    print re.compile(pat, re.M).sub("{\g<0>}", s.rstrip()),'\n'

s = """Mary had a little lamb
    And everywhere that Mary
    went, the lamb was sure to go"""

re_show('a', s)        # letter 'a'
re_show(r'^Mary', s)   # beginning of line
re_show(r'Mary$', s)   # end of line
re_show(r'.a', s)      # any letter + 'a'
```

```
M{a}ry h{a}d {a} little l{a}mb
    And everywhere th{a}t M{a}ry
    went, the l{a}mb w{a}s sure to go

{Mary} had a little lamb
    And everywhere that Mary
    went, the lamb was sure to go

Mary had a little lamb
    And everywhere that {Mary}
    went, the lamb was sure to go

{Ma}ry {ha}d{ a} little {la}mb
    And everywhere t{ha}t {Ma}ry
    went, the {la}mb {wa}s sure to go
```

re.search



```
str = "The fuzz is on the street"
m = re.search("[tT]he", str)
print m
print m.groups()
print m.group(0)
print re.findall("[tT]he", str)
m = re.search("(The).*(fu.*z)", str)
print "groups(): ", m.groups()
print "group(): ", m.group()
print "group(0): ", m.group(0)
print "group(1): ", m.group(1)
print "group(2): ", m.group(2)
```

```
<_sre.SRE_Match object at 0x10b5c59f0>
()
The
['The', 'the']
groups():  ('The', 'fuzz')
group():  The fuzz
group(0):  The fuzz
group(1):  The
group(2):  fuzz
```

Some Constant flags

- re.I : re.IGNORECASE :
- re.L : locale
- re.M : multiline
    - pattern match do not cross ‘\n’ boundaries)

Explorations

- Regular expressions offer much more than discussed