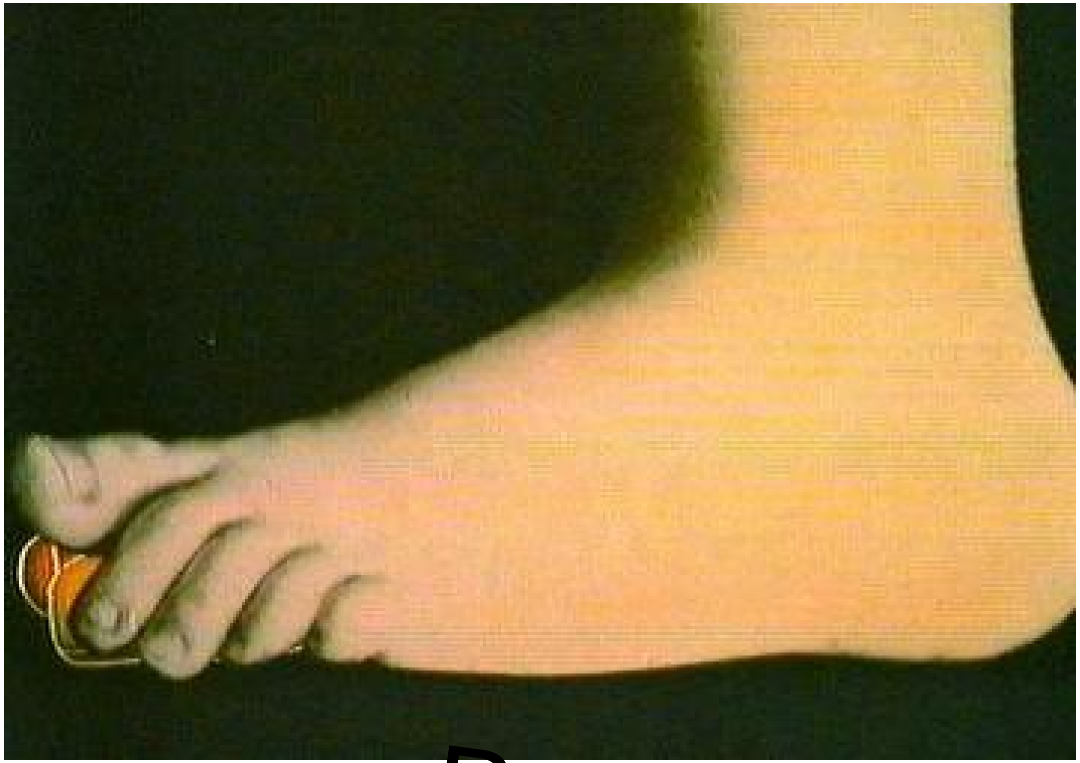
Python: Introduction for Programmers

Bruce Beckles

Bob Dowling

University Computing Service

Scientific Computing Support e-mail address:
escience-support@ucs.cam.ac.uk



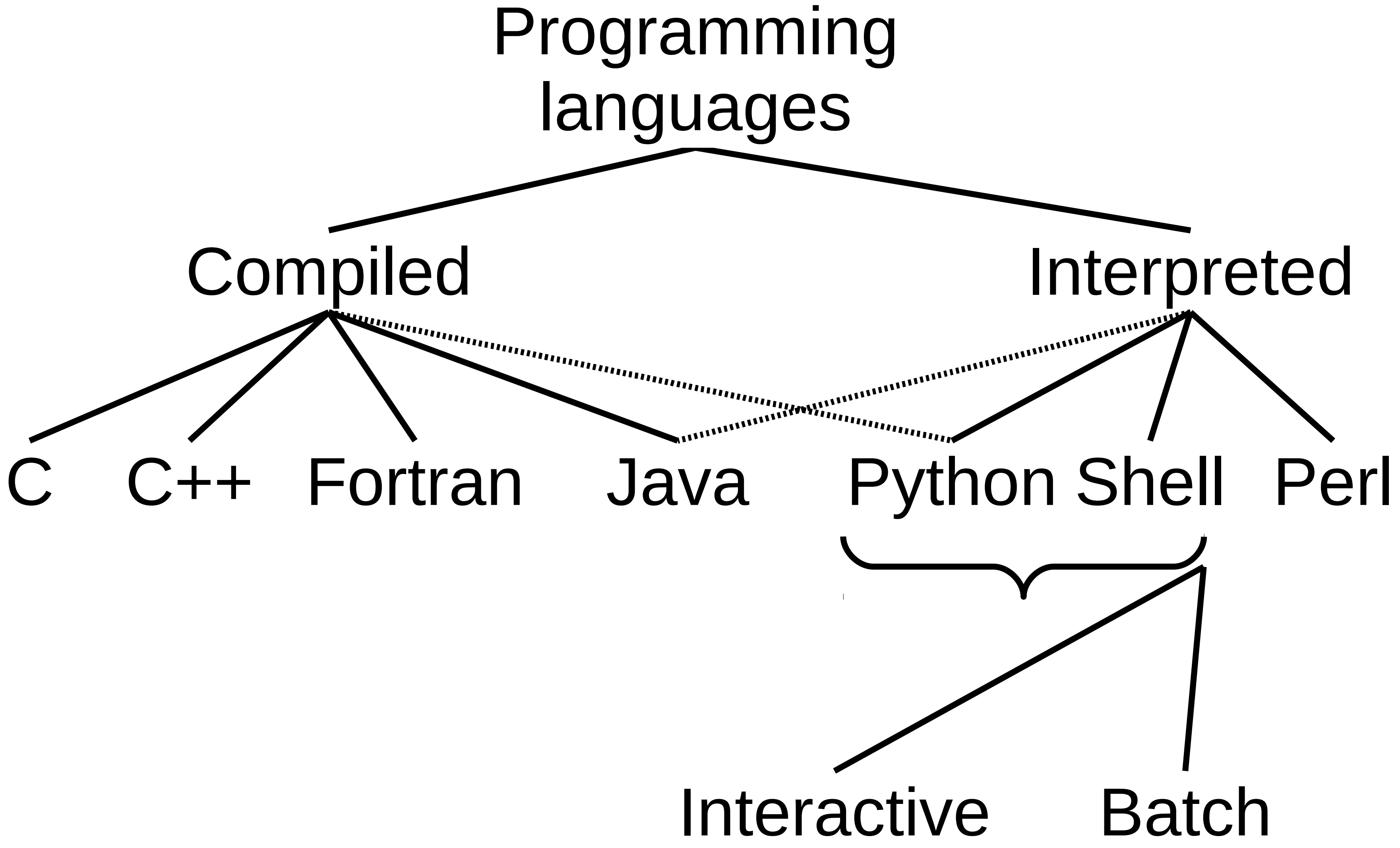
Python: Introduction for Programmers

Bruce Beckles Bob Dowling

University Computing Service

Scientific Computing Support e-mail address:

escience-support@ucs.cam.ac.uk



Programming languages

Compiled

Interpreted

C

C++

Fortran

Java

Python

Shell

Perl

Interactive

Batch

Interactive use

Unix prompt

```
$ python
Python 2.6 (r26:66714, Feb  3 2009, 20:52:03)
[GCC 4.3.2 [gcc-4_3-branch revision 141291]] on …
Type "help", "copyright", "credits" or "license" …
```

Python prompt

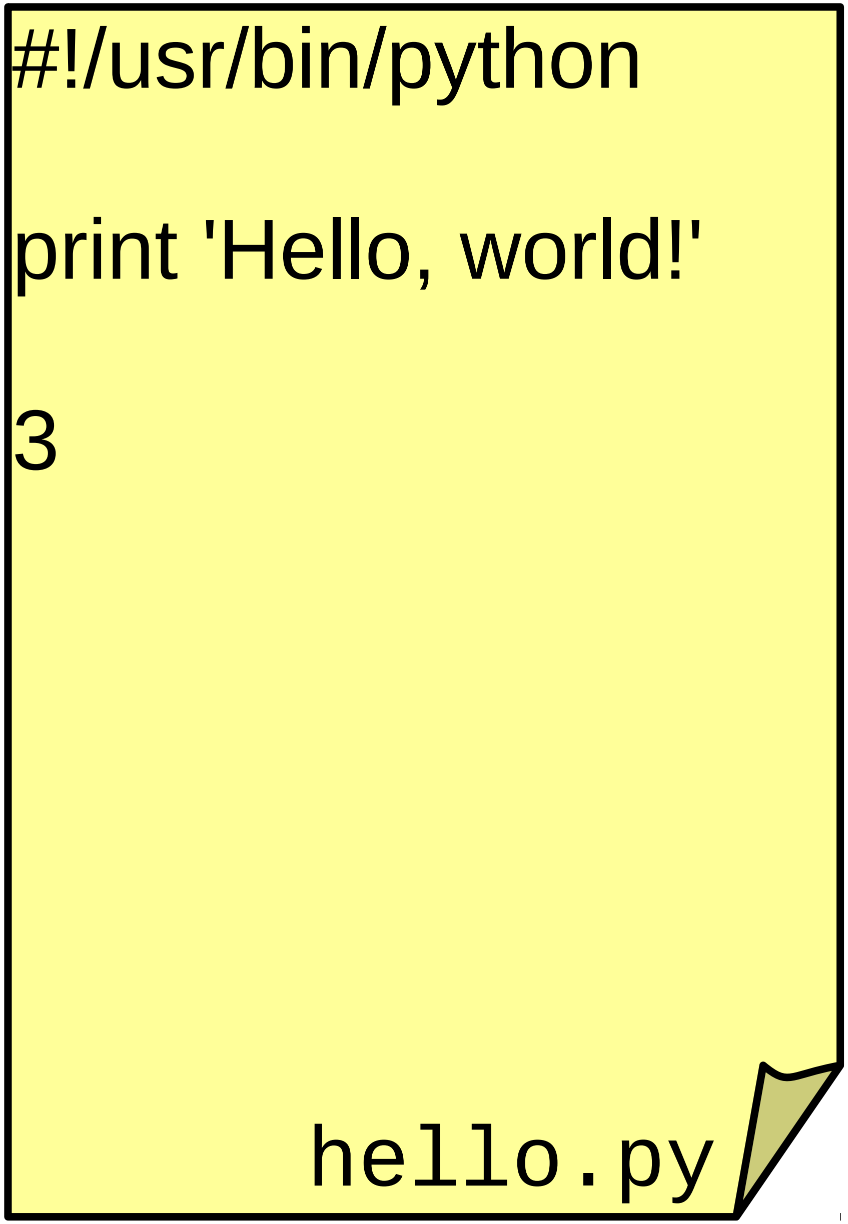
```
>>> print 'Hello, world!'
Hello, world!

>>> 3
3
```

```
$ python
Python 2.6 (r26:66714, Feb  3 2009, 20:52:03)
[GCC 4.3.2 [gcc-4_3-branch revision 141291]] on …
Type "help", "copyright", "credits" or "license" …

>>> print 'Hello, world!'
Hello, world!

>>> 3
3

>>>
$
```

To quit the Python interpreter:
Press *control+d*

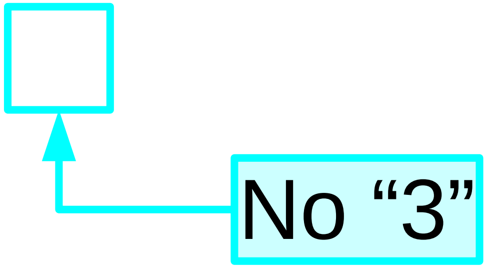
Unix prompt

Batch use



```
#!/usr/bin/python

print 'Hello, world!'

3
```

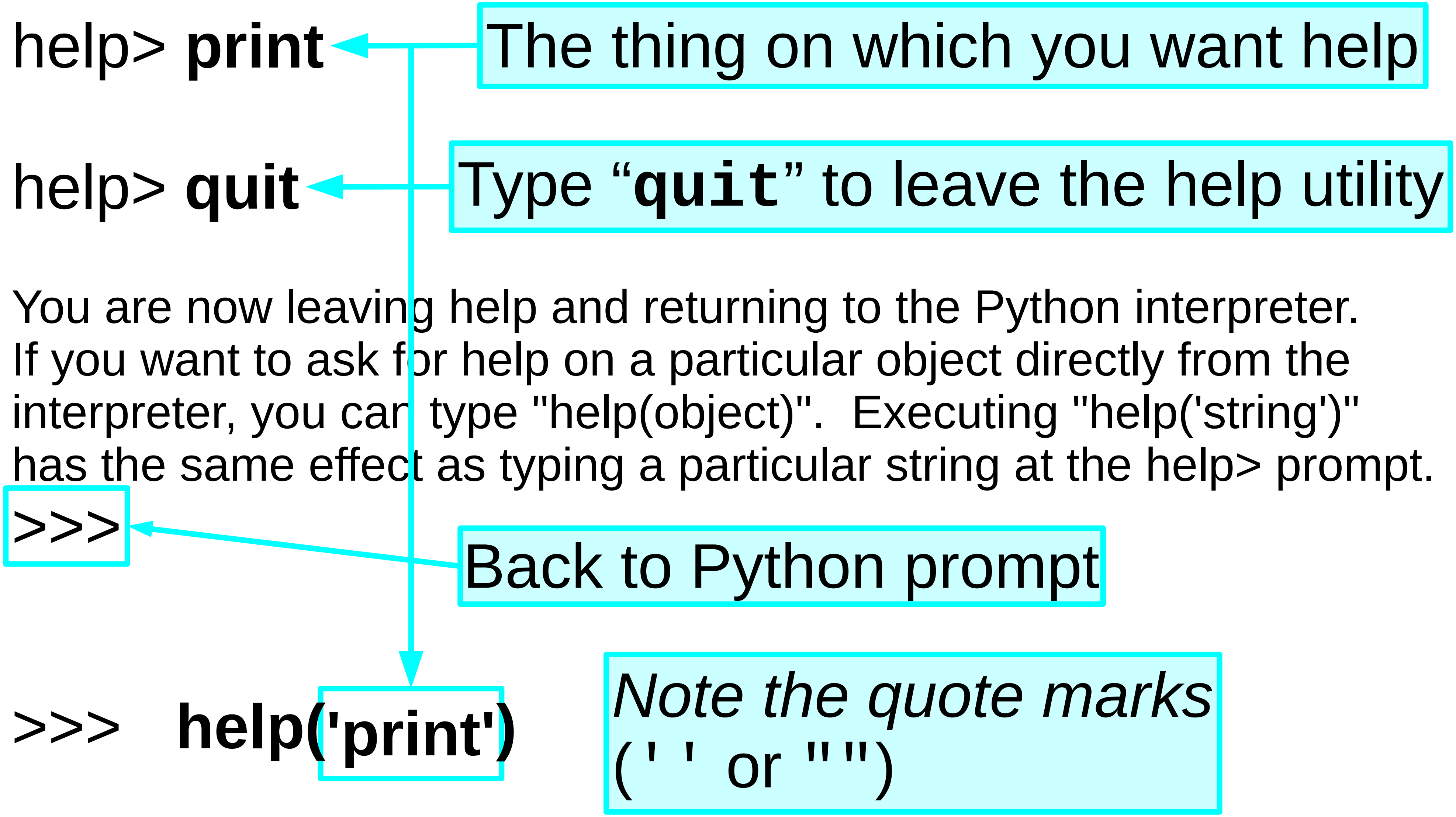
```
$ python hello.py

Hello, world!
```



No “3”

```
$ python
Python 2.6 (r26:66714, Feb  3 2009, 20:52:03)
[GCC 4.3.2 [gcc-4_3-branch revision 141291]] on …
Type "help", "copyright", "credits" or "license" …
>>> help
Type help() for interactive help, or help(object) for
help about object.
>>> help()

Welcome to Python 2.6!  This is the online help utility.

If this is your first time using Python, …

help>
```

help utility prompt



```
help> print

help> quit

You are now leaving help and returning to the Python interpreter.
If you want to ask for help on a particular object directly from the
interpreter, you can type "help(object)".  Executing "help('string')"
has the same effect as typing a particular string at the help> prompt.
>>>

>>>   help('print')

>>>
```

The thing on which you want help

Type “**quit**” to leave the help utility

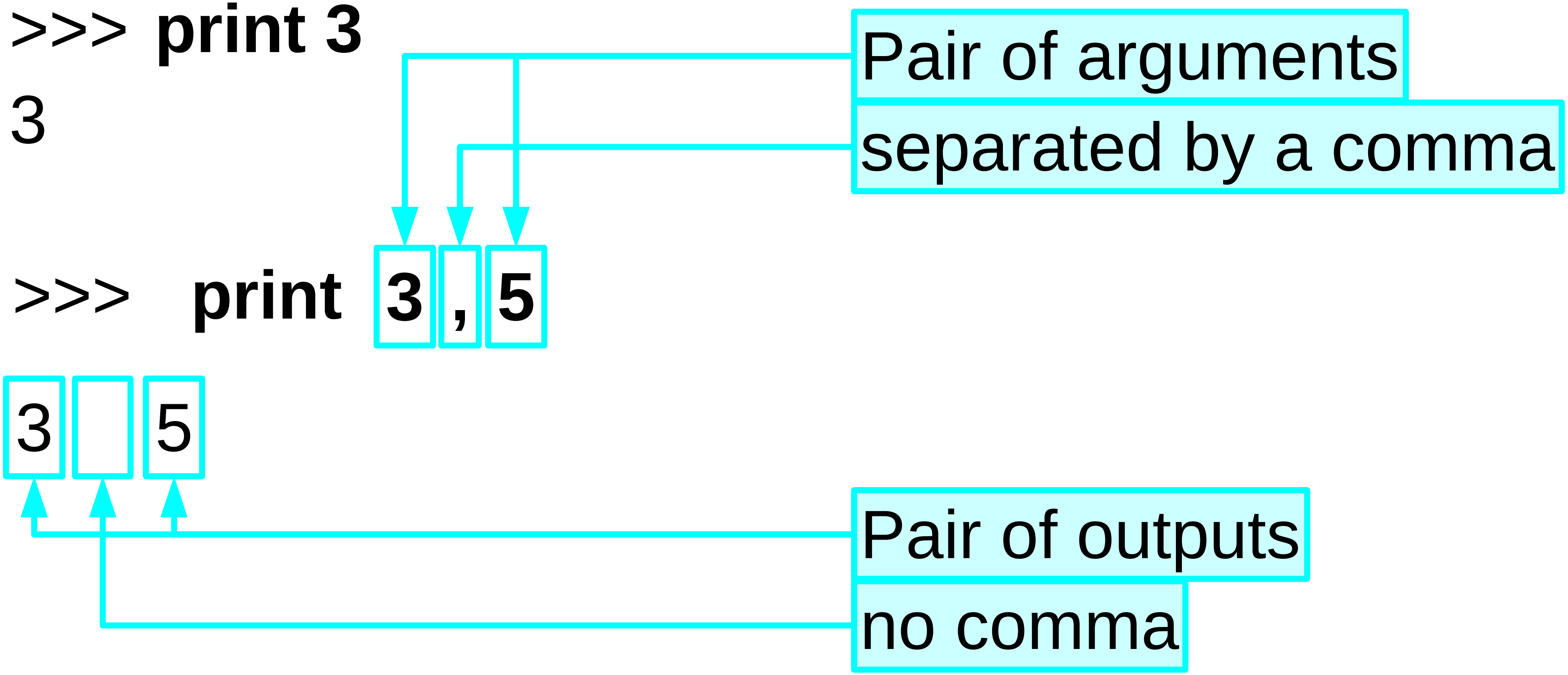
Back to Python prompt

Note the quote marks
(' ' or "")

Official Python documentation (includes tutorial):

http://docs.python.org/

```
$ python
Python 2.6 (r26:66714, Feb  3 2009, 20:52:03)
[GCC 4.3.2 [gcc-4_3-branch revision 141291]] on …
Type "help", "copyright", "credits" or "license" …

>>> print 3
3

>>>  print 3 , 5
3 5
```



Pair of arguments
separated by a comma

Pair of outputs
no comma

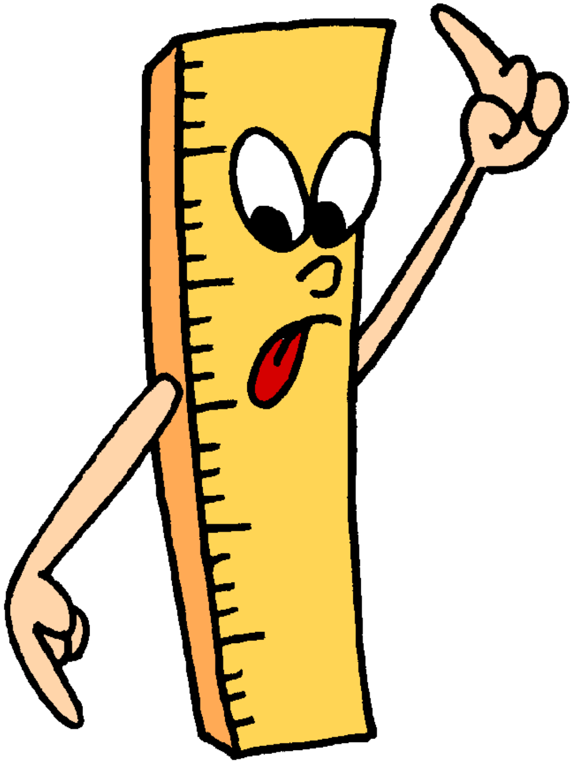
Using Python for science

Science

Quantitative

Numbers

Using numbers in Python



$\mathbb{Z}$ Integers

$\{ \ldots -2, -1, 0, 1, 2, 3, \ldots \}$

```
>>> 7+3              >>> 7-3
10                   4
```

```
>>> 7*3              >>> 7**3
21                   343
```

7^3: use “**” for exponentiation

```
>>> 7/3              >>> -7/3
2                    -3
```

integer division rounds down

```
>>> 7%3              >>> -7%3
1                    2
```

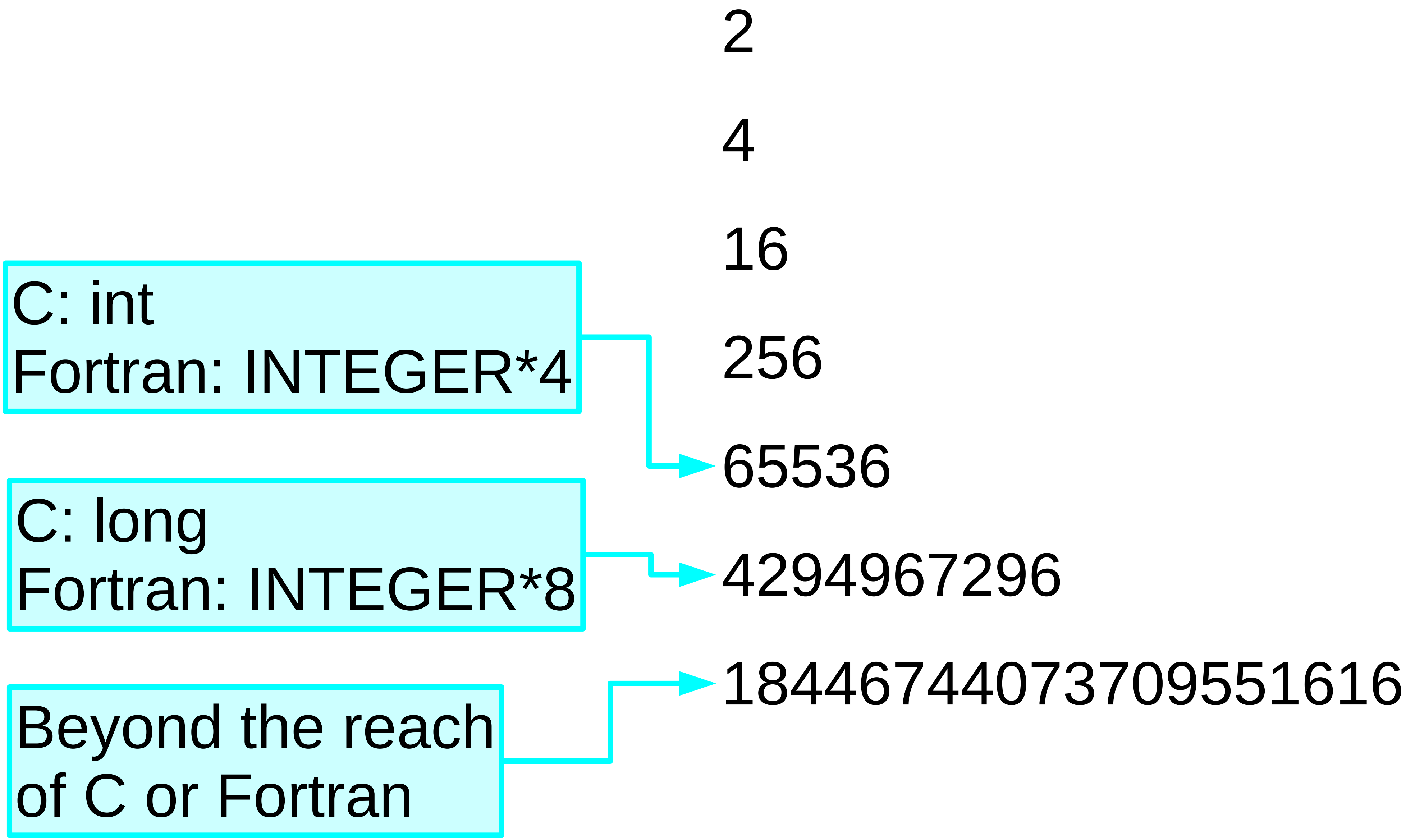
remainder (mod) returns 0 or positive integer

```
>>> 2*2
4

>>> 4*4
16

>>> 16*16
256

>>> 256*256
65536

>>> 65536*65536
4294967296L
```

“large” integer

```
>>>  4294967296*4294967296
18446744073709551616L

>>>  18446744073709551616 *
18446744073709551616
340282366920938463463374607431768211456L

>>>  2**521 - 1
6864797660130609714981900799081393217269
4353001433054093944634591855431833976560
5212255964066145455497729631139148085803
7121987999716643812574028291115057151L
```

No inherent limit to Python's integer arithmetic:
can keep going until we run out of memory



2

4

16

256

65536

4294967296

18446744073709551616

C: int
Fortran: INTEGER*4

C: long
Fortran: INTEGER*8

Beyond the reach
of C or Fortran

$\xcancel{\mathbb{R}}$

Floating point numbers

```
>>> 1.0
1.0

>>> 0.5
0.5

>>> 0.25
0.25

>>> 0.1
0.10000000000000001
```

Floating point number

½ is OK

¼ is OK

Powers of two

1/10 is *not*

Usual issues with representation in base 2

```
>>> 2.0*2.0
4.0
>>> 4.0*4.0
16.0

…

>>> 65536.0*65536.0
4294967296.0
>>> 4294967296.0*4294967296.0
1.8446744073709552e+19
```

17 significant figures

```
>>>  4294967296.0*4294967296.0
1.8446744073709552e+19

>>>  1.8446744073709552e+19*1.8446744073709552e+19
3.4028236692093846e+38

>>>  3.4028236692093846e+38*3.4028236692093846e+38
1.157920892373162e+77

>>>  1.157920892373162e+77*1.157920892373162e+77
1.3407807929942597e+154

>>>  1.3407807929942597e+154*1.3407807929942597e+154
inf
```

overflow

Limit at 2**1023

Machine epsilon

```
>>> 1.0 + 1.0e-16
1.0
>>> 1.0 + 2.0e-16
1.0000000000000002
```

too small to make a difference

large enough

```
>>> 1.0 + 1.1e-16
1.0
>>> 1.0 + 1.9e-16
1.0000000000000002
```

Spend the next few minutes using Python interactively to estimate machine epsilon – we'll write a Python program to do this for us a little later

Strings

```
'Hello, world!'
```

```
"Hello, world!"
```

```
'''Hello,
world!'''
```

```
"""Hello,
world!"""
```

Single quotes

'Hello, world!'

Single quotes around the string

Double quotes

"Hello, world!"

Double quotes around the string

Exactly equivalent

'He said "Hello, world!" to her.'

```
>>>  print 'He said "Hello, world!" to her.'
He said "Hello, world!" to her.
```

"He said 'Hello, world!' to her."

```
>>>  print "He said 'Hello, world!' to her."
He said 'Hello, world!' to her.
```

String concatenation

Two separate strings

```
>>> 'He said' 'something to her.'
'He saidsomething to her.'
```

Optional space(s)

```
>>> 'He said''something to her.'
'He saidsomething to her.'
```

```
>>> 'He said'+'something to her.'
'He saidsomething to her.'
```

Can also use + operator

Special characters

`\n` → ↵

`\t` → ⇥

`\a` → ♫

`\\` → \

`\'` → '

`\"` → "

```
>>> print 'Hello,\nworld!'
Hello,
world!
```

“\n” converted to “new line”

Long strings

Triple double quotes

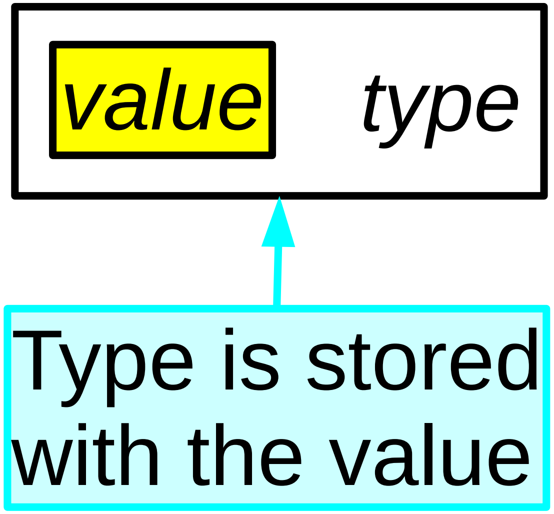
"""Long pieces of
text are easier to
handle if literal new
lines can be
embedded in them."""

Long strings

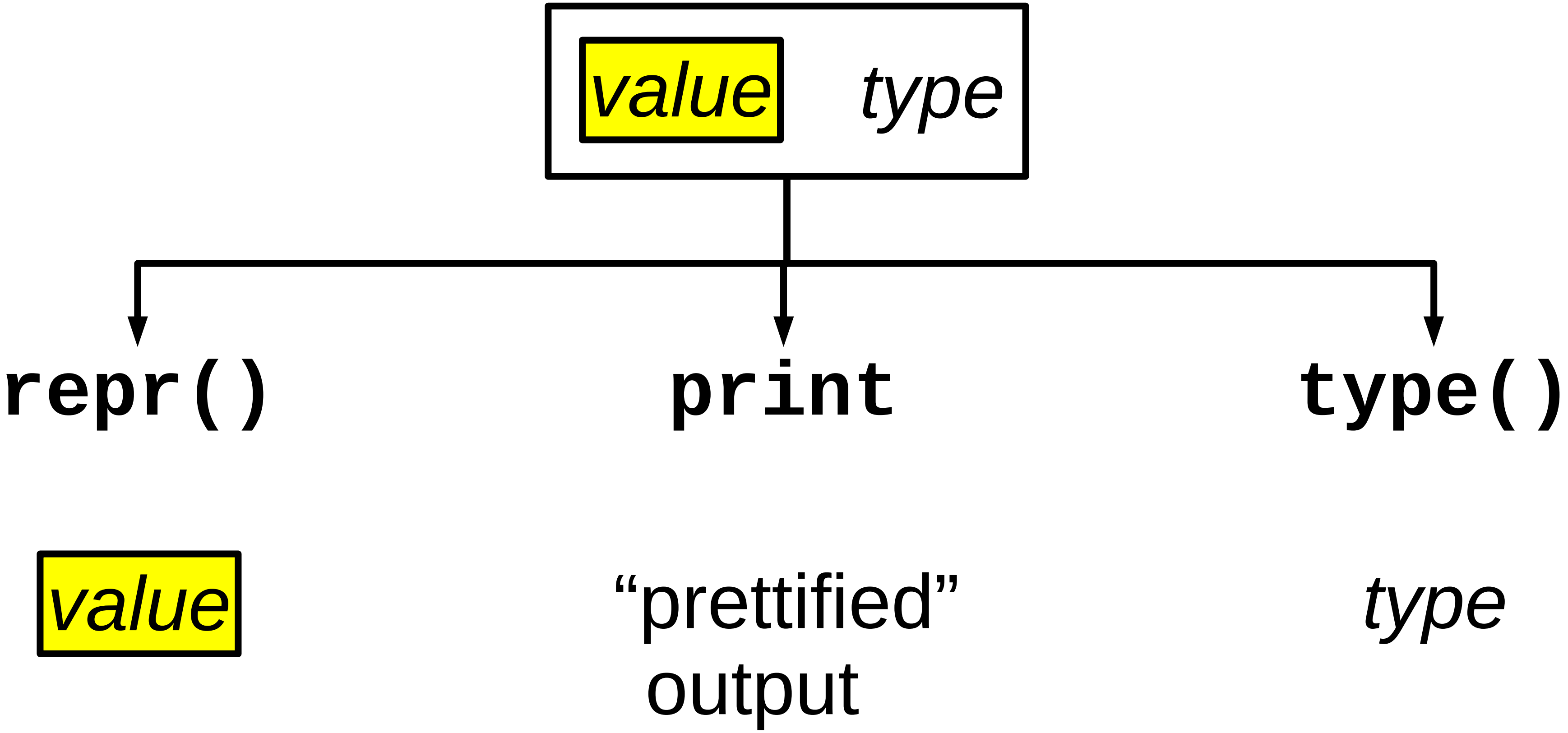
Triple single quotes

```
'''Long pieces of
text are easier to
handle if literal new
lines can be
embedded in them.'''
```

How Python stores values



Type is stored with the value



`repr()` **`print`** **`type()`**

value "prettified" output *type*

```
>>>  print  1.2345678901234567
1.23456789012

>>>  type( 1.2345678901234567 )
<type 'float'>

>>>  repr(  1.2345678901234567 )
'1.2345678901234567'
```

Two other useful types

Complex

```
>>>   (1.0 + 2.0j) * (1.5 + 2.5j)
(-3.5+5.5j)
```

Boolean

```
>>>   True  and  False
False

>>>   123   ==    234
False
```

Comparisons

```
>>>   1 == 2
False

>>>   1 <  2
True

>>>   1 >= 2
False

>>>   1 == 1.0
True
```

```
>>>   'abc'  ==  'ABC'
False

>>>   'abc'  <  'ABC'
False

>>>   'abc'  >=  'ABC'
True
```

… not equal to …

```
>>>   not 1 == 2
True
```

```
>>>   1 != 2
True
```

```
>>>   not 'abc' == 'ABC'
True
```

```
>>>   'abc' != 'ABC'
True
```

Conjunctions

```
>>>   1 == 2 and 3 == 3
False

>>>   1 == 2 or 3 == 3
True
```

Evaluate the following Python expressions in your head:

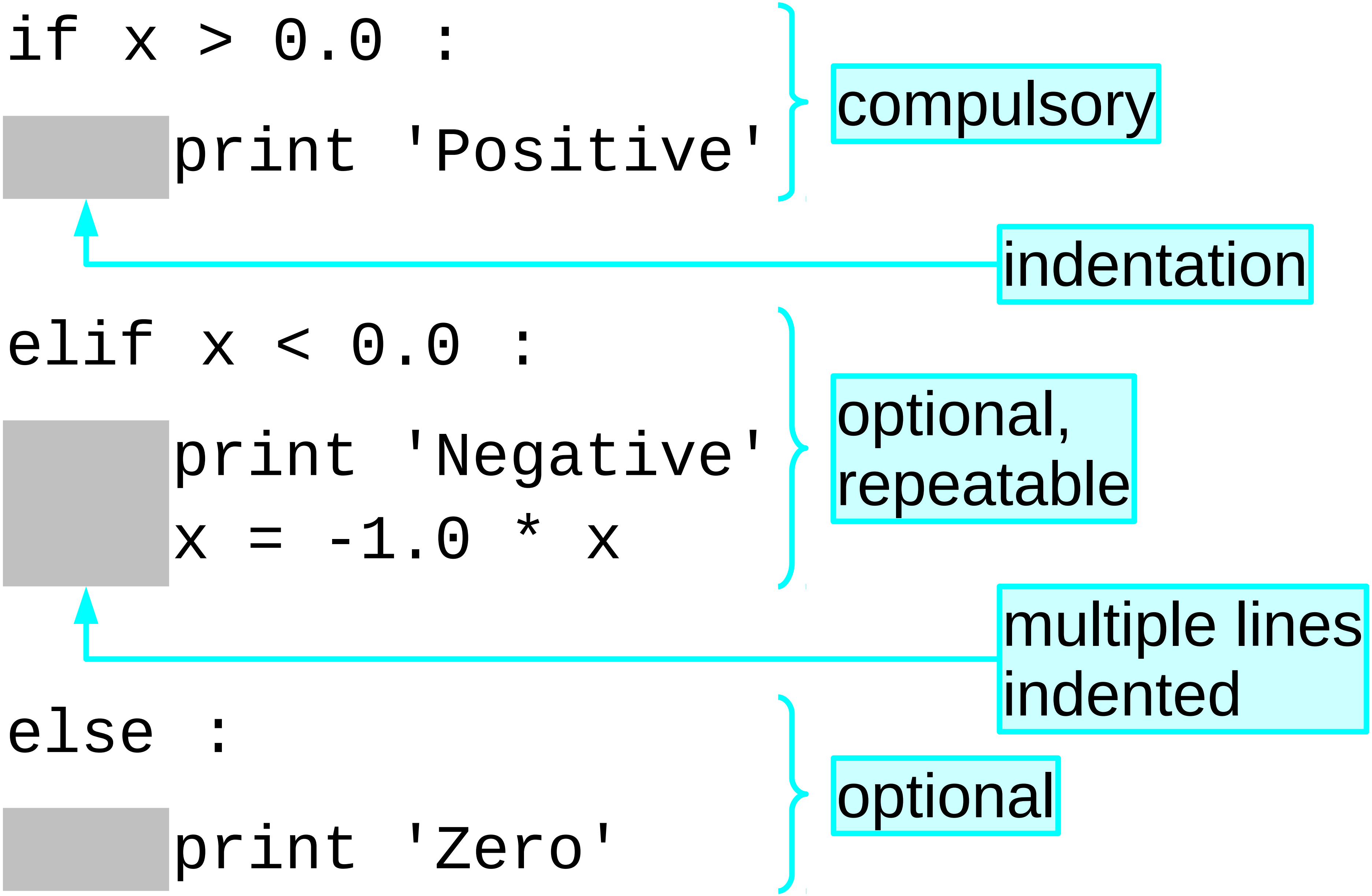
```
 >>>   2 - 2 == 1 / 2

 >>>   True and False or True

 >>>   1 + 1.0e-16 > 1

 >>>   5 == 6 or 2 * 8 == 16

 >>>   7 == 7 / 2 * 2

 >>>   'AbC' > 'ABC'
```

Now try them interactively in Python and see if you were correct.

Precedence

First	x**y	Arithmetic operations
	-6, +6	Arithmetic operations
	x/y, x*y, x%y	Arithmetic operations
	x+y, x-y	Arithmetic operations
	x<y, x<=y, …	
	x in y, x not in y	Logical operations
	not x	Logical operations
	x and y	Logical operations
Last	x or y	Logical operations

Flow control in Python: **if**



```
if x > 0.0 :
    print 'Positive'
elif x < 0.0 :
    print 'Negative'
    x = -1.0 * x
else :
    print 'Zero'
```

compulsory

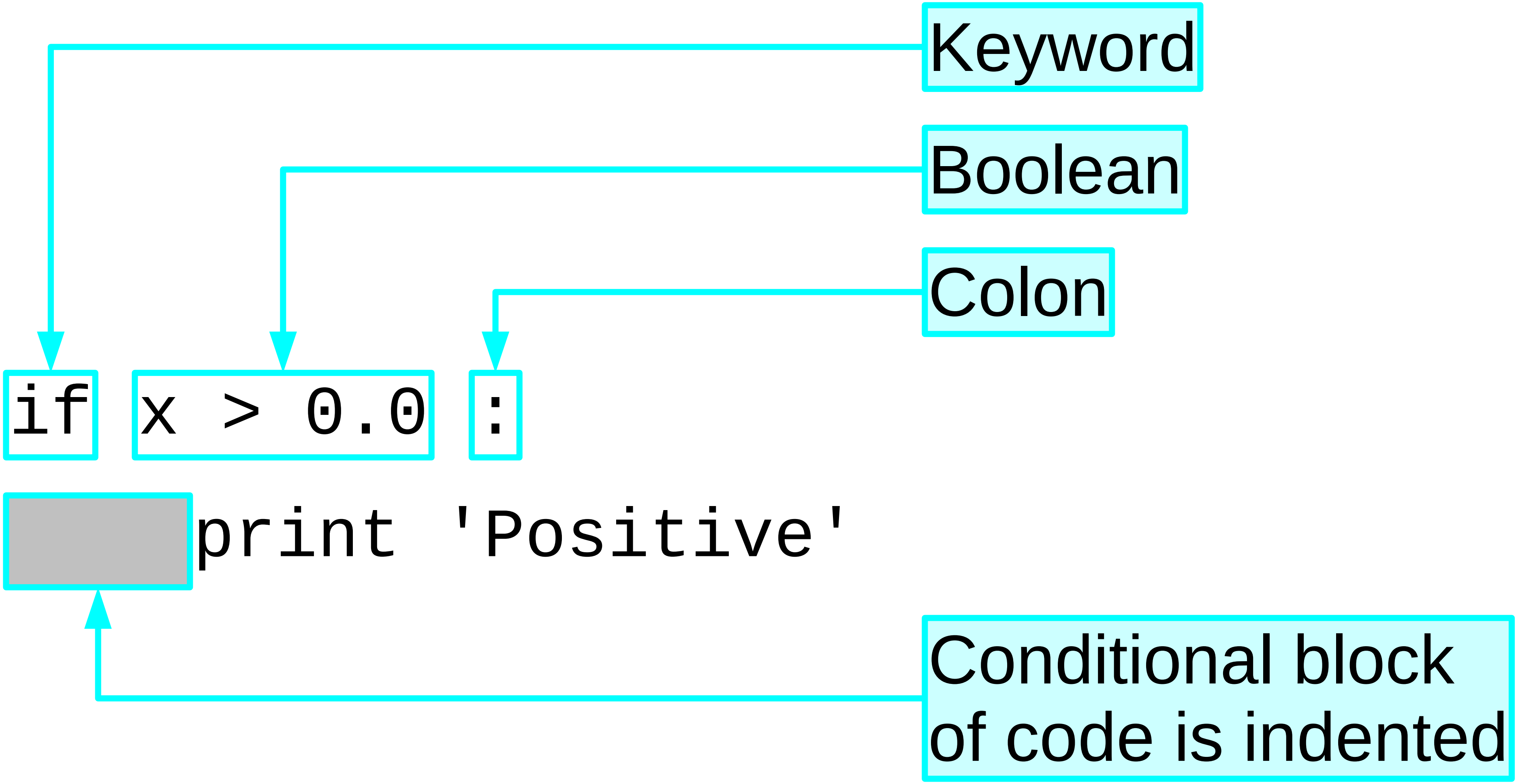
indentation

optional, repeatable

multiple lines indented

optional

Flow control in Python: **`if`**



```
if x > 0.0 :
    print 'Positive'
```

Keyword

Boolean

Colon

Conditional block of code is indented

Nested indentation

```
if x > 0.0 :
    print 'Positive'
else :
    if x < 0.0 :
        print 'Negative'
        x = -1.0 * x
    else :
        print 'Zero'
```

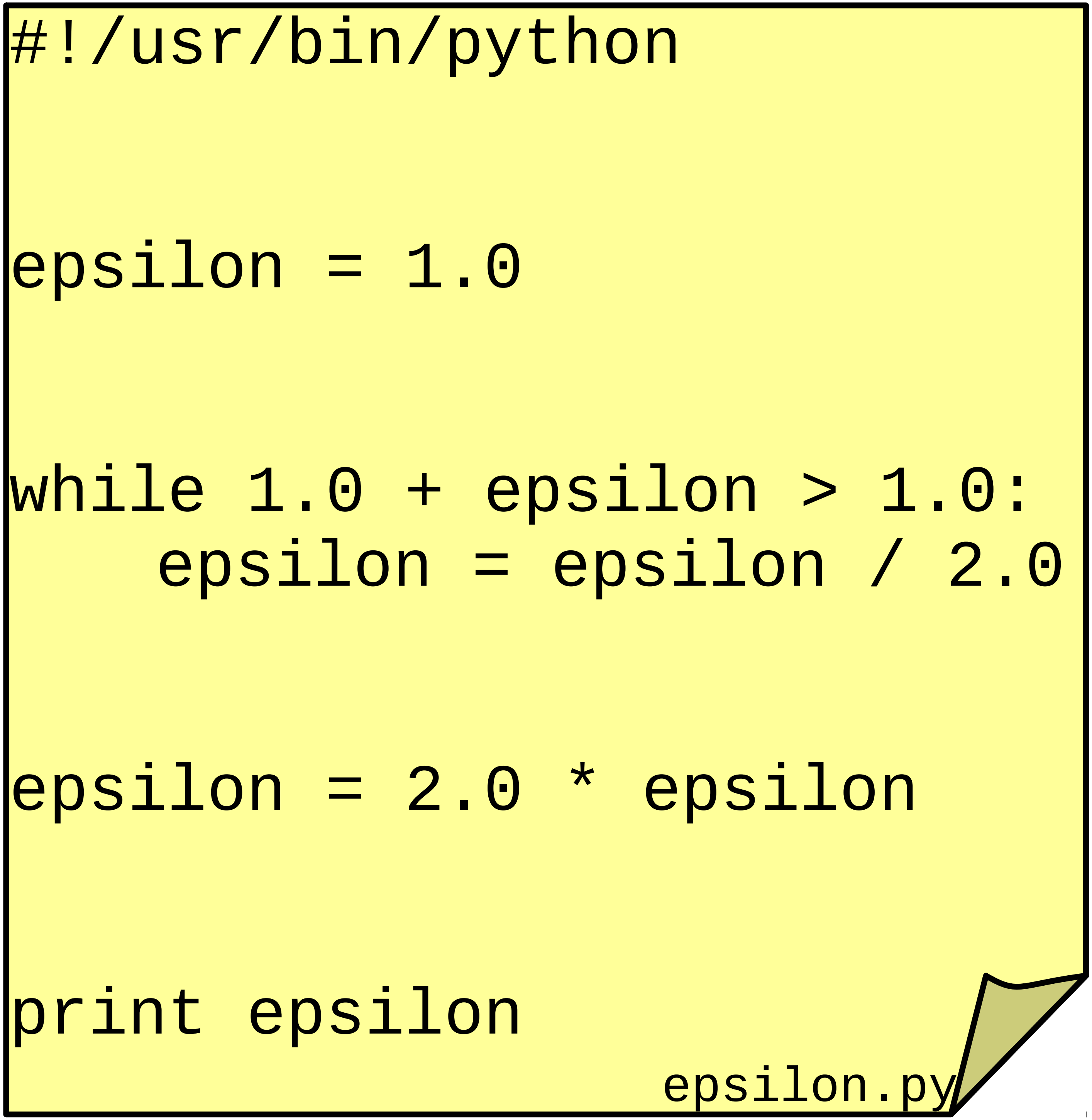
Flow control in Python: **`while`**

```
while x % 2 == 0 :
    print x, 'still even'
    x = x/2
```

compulsory

```
else :
    print x, 'is odd'
```

optional



```
#!/usr/bin/python

epsilon = 1.0

while 1.0 + epsilon > 1.0:
    epsilon = epsilon / 2.0

epsilon = 2.0 * epsilon

print epsilon
```

epsilon.py

Approximate machine epsilon

$1.0 + \varepsilon > 1.0$

$1.0 + \varepsilon/2 == 1.0$

```
#!/usr/bin/python

# Start with too big a value
epsilon = 1.0

# Halve it until it gets too small
while 1.0 + epsilon > 1.0:
    epsilon = epsilon / 2.0

# It's one step too small now,
# so double it again.
epsilon = 2.0 * epsilon

# And output the result
print epsilon
```

epsilon.py

Time for a break…

Have a look at the script `epsilon2.py` in your home directory.

This script gives a better estimate of machine than the script we just wrote.

See if you can figure out what it does – if there is anything you don’t understand, tell the course giver or a demonstrator.

A better estimate for machine epsilon

```
#!/usr/bin/python

too_large = 1.0
too_small = 0.0
tolerance = 1.0e-27

while  too_large - too_small  >  tolerance:

   mid_point = (too_large + too_small)/2.0

   if 1.0 + mid_point > 1.0:
      too_large = mid_point
   else:
      too_small = mid_point

print  too_small,  '< epsilon <',  too_large
```

epsilon2.py

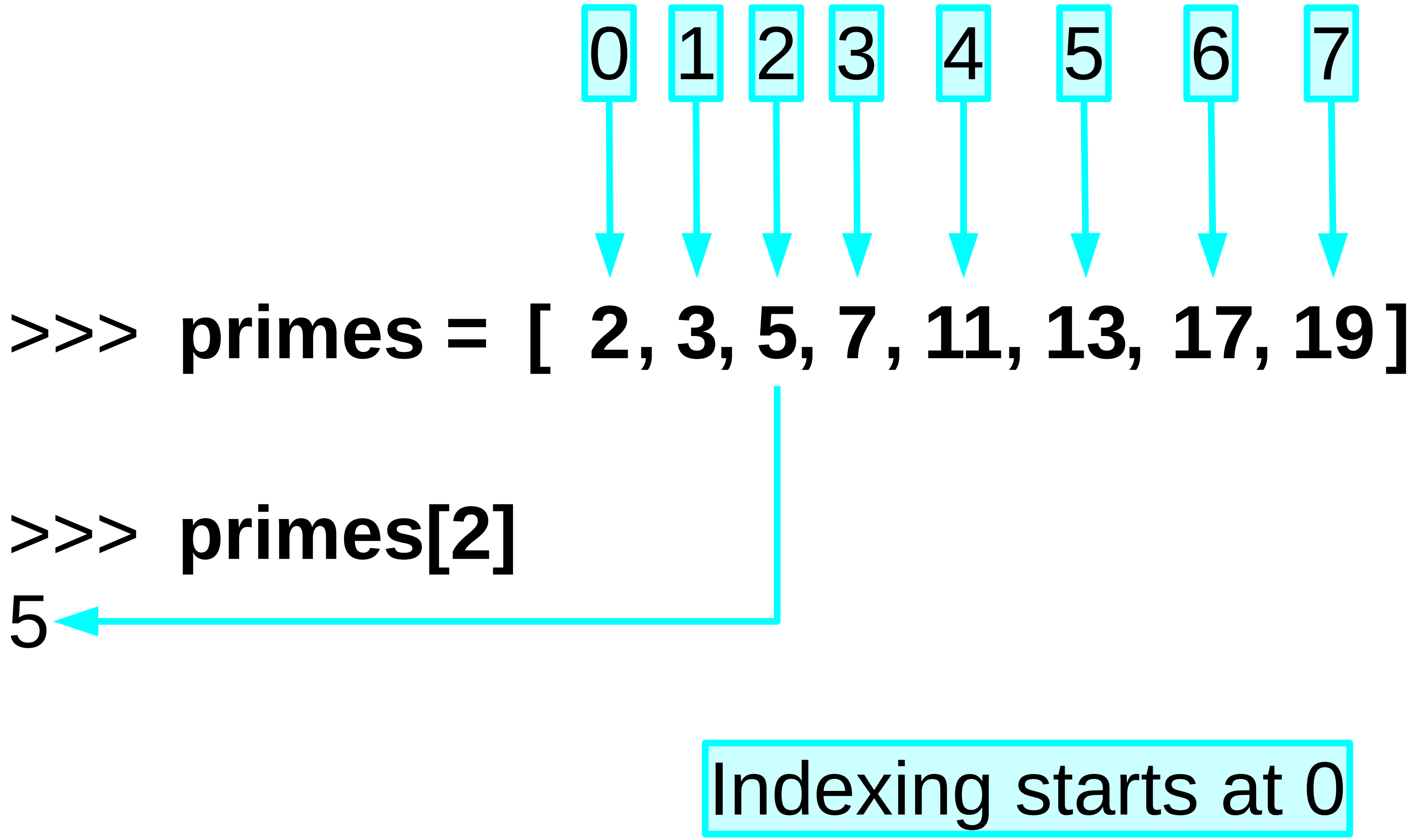
January February March April May June July
August September October November December

Lists

H He Li Be B C N O F Ne Na Mg Al Si P S Cl Ar

Red Orange Yellow Blue Indigo Violet

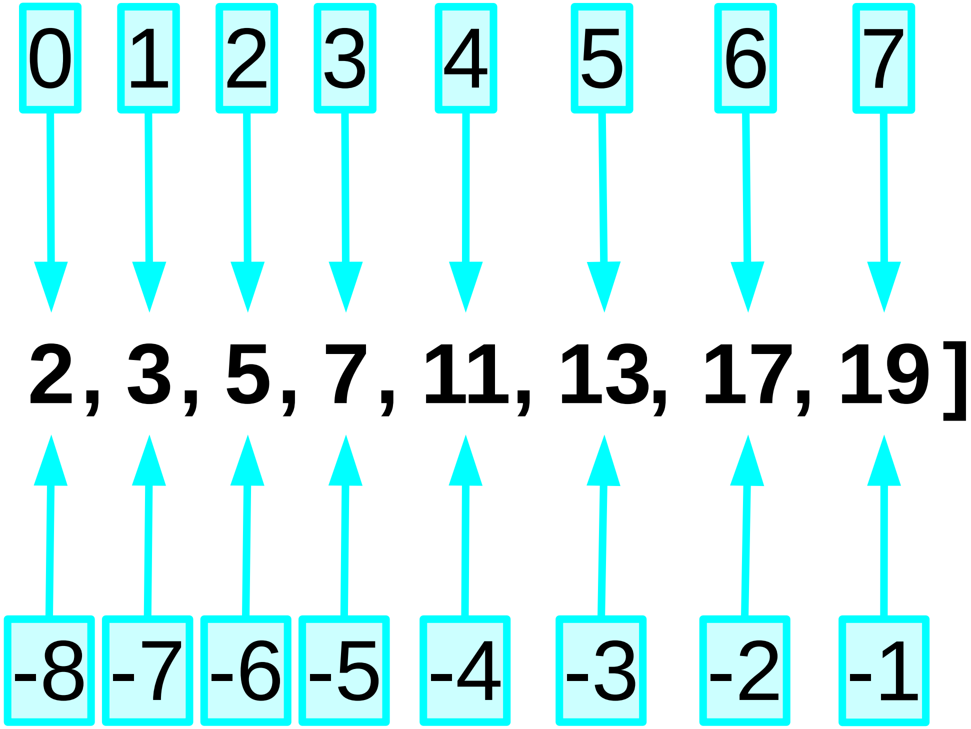
```
>>>  [ 2, 3, 5, 7, 11, 13, 17, 19]
[ 2, 3, 5, 7, 11, 13, 17, 19]

>>>  type([ 2, 3, 5, 7, 11, 13, 17, 19])
<type 'list'>

>>>  primes = [ 2, 3, 5, 7, 11, 13, 17, 19]
```

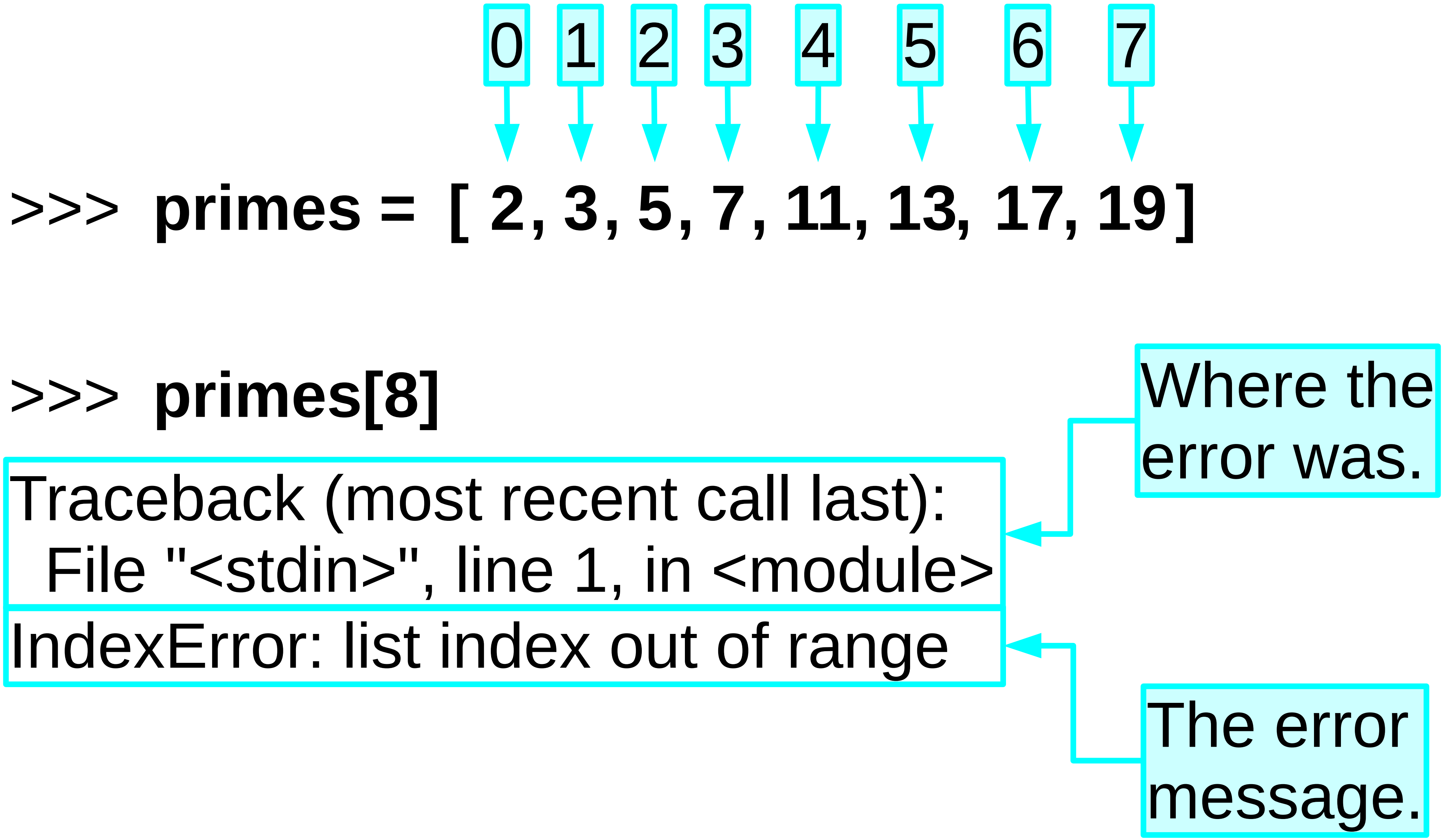


```
>>> primes = [ 2, 3, 5, 7, 11, 13, 17, 19 ]

>>> primes[2]
5
```

Indexing starts at 0

```
>>>  primes = [ 2, 3, 5, 7, 11, 13, 17, 19 ]

>>>  primes[-1]
19
```





0 1 2 3 4 5 6 7

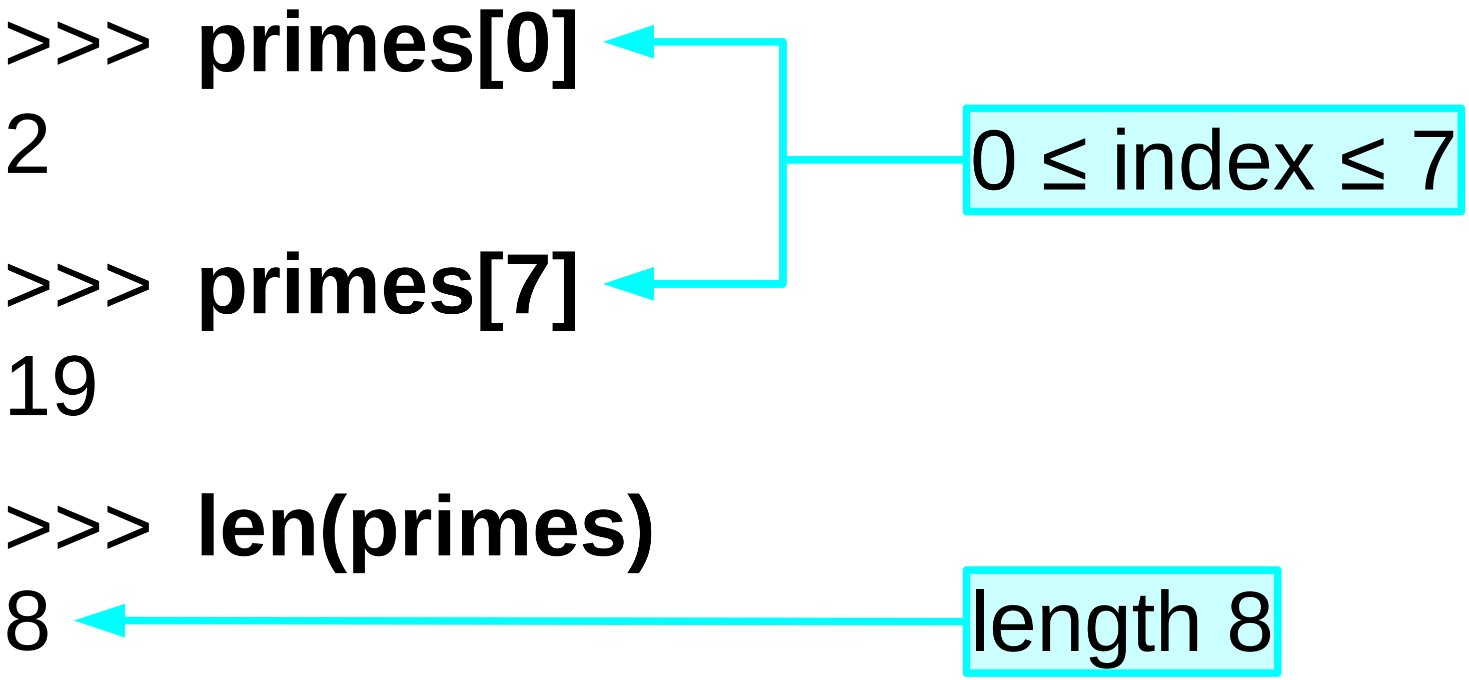
```
>>>  primes =  [ 2, 3, 5, 7, 11, 13, 17, 19 ]

>>>  primes[8]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

Where the error was.

The error message.

Counting from zero and the **`len()`** function

```
>>>  primes = [2, 3, 5, 7, 11, 13, 17, 19]
```

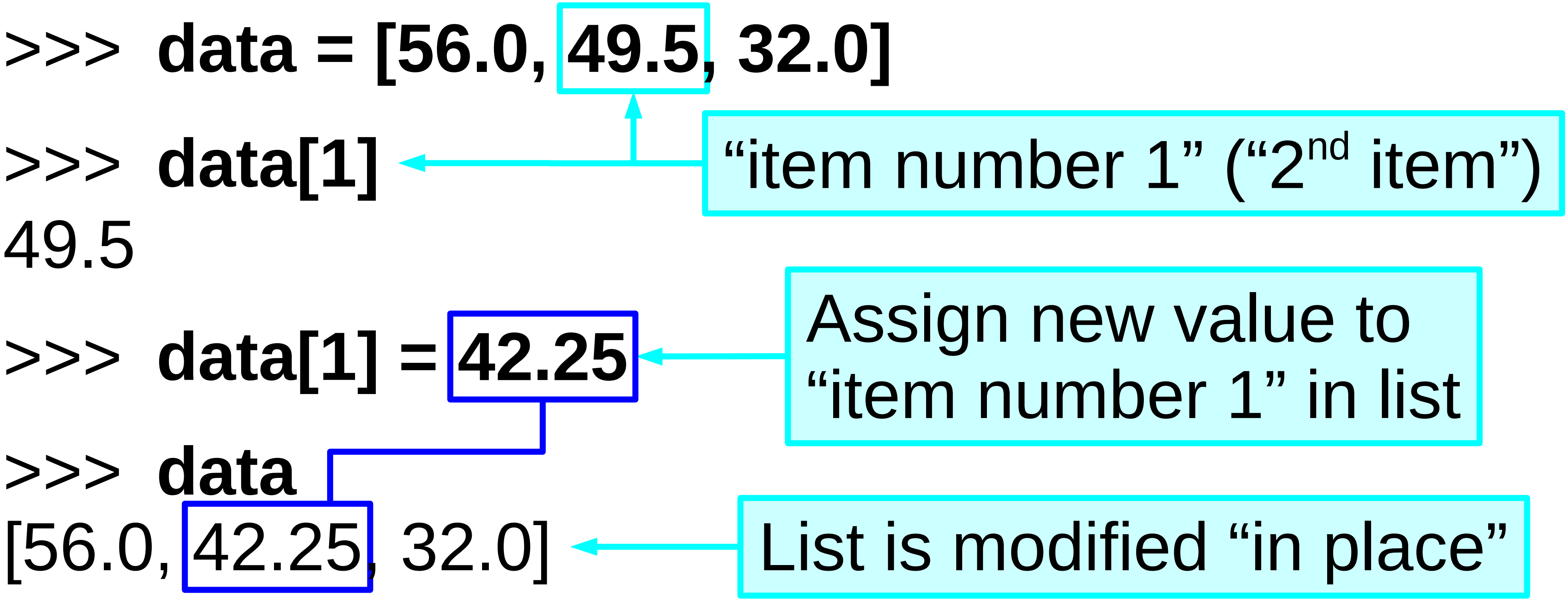


```
>>>  primes[0]
2

>>>  primes[7]
19

>>>  len(primes)
8
```

0 ≤ index ≤ 7

length 8

Changing an item in a list



```
>>> data = [56.0, 49.5, 32.0]
>>> data[1]
49.5
>>> data[1] = 42.25
>>> data
[56.0, 42.25, 32.0]
```

“item number 1” (“2nd item”)

Assign new value to “item number 1” in list

List is modified “in place”

Empty lists

```
>>>  empty = [ ]

>>>  len(empty)
0

>>>  len([ ])
0
```

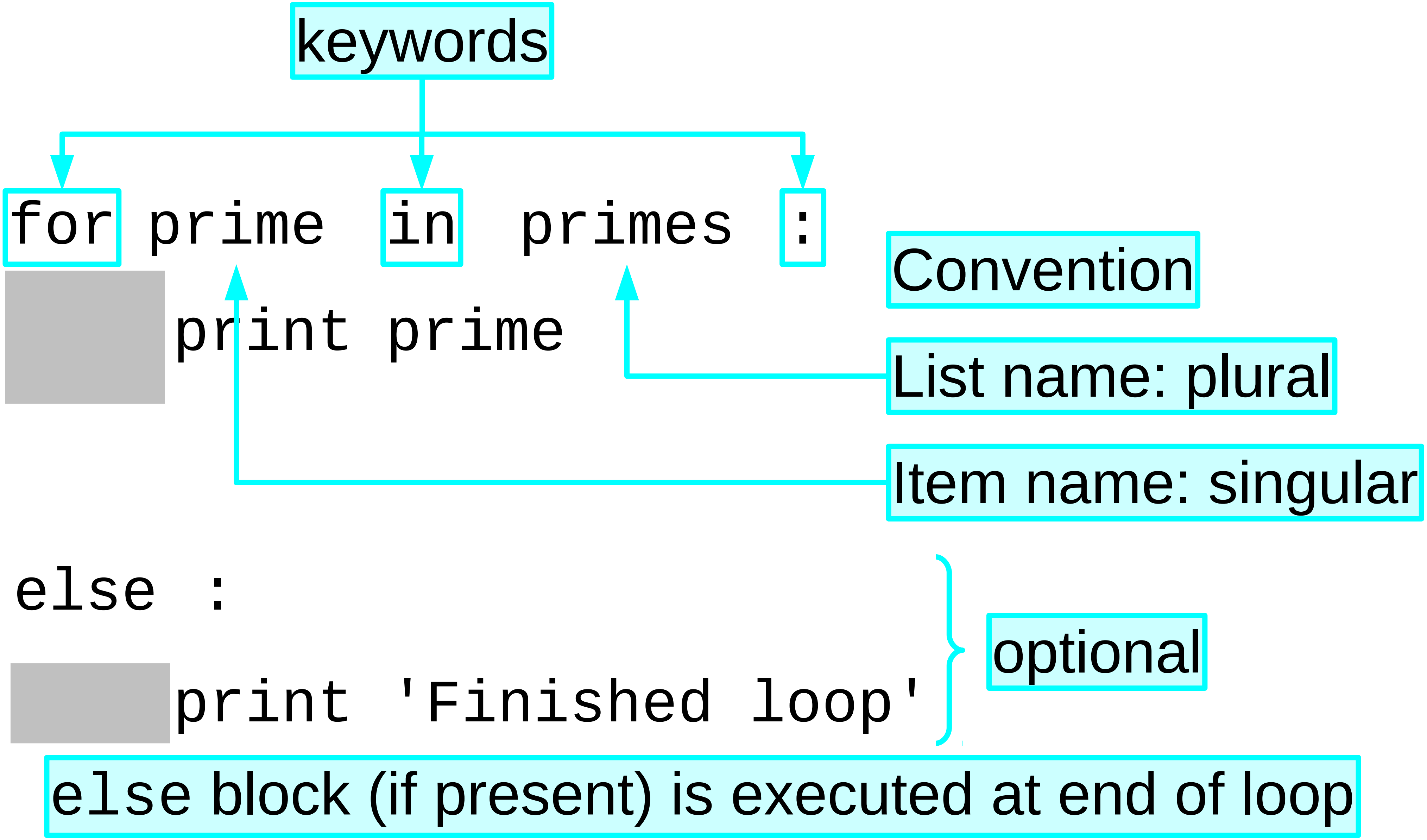
Single item lists

A list with one item is not the same as the item itself!

```
>>> [1234] == 1234
False
```

```
>>> type([1234])
<type 'list'>

>>> type(1234)
<type 'int'>
```

Flow control in Python: **for**



```
for prime in primes :
    print prime

else :
    print 'Finished loop'
```

keywords

Convention

List name: plural

Item name: singular

optional

else block (if present) is executed at end of loop

Warning: loop variable persists

```
for prime in primes :
    print prime
print 'Done!'
print prime
```

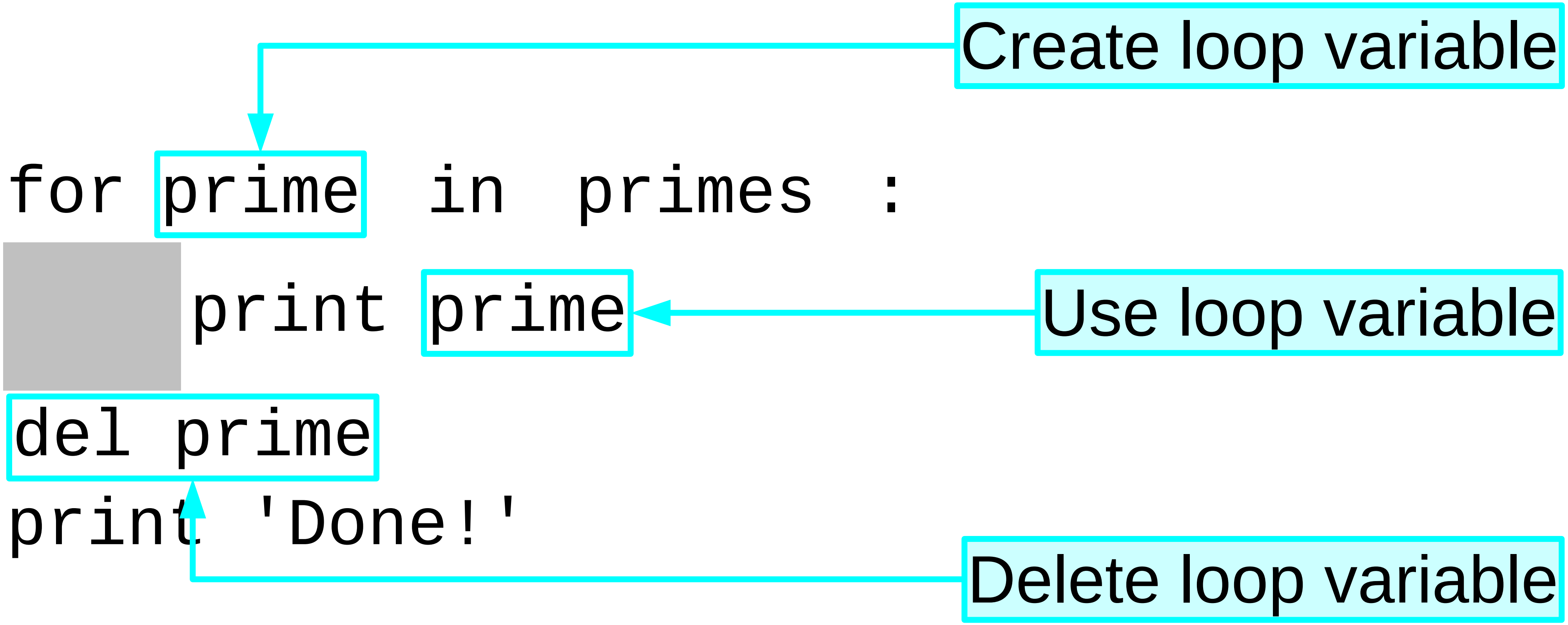
Definition of loop variable

Correct use of loop variable

Improper use of loop variable

But legal!

Loop variable “hygiene”



```
for prime  in  primes  :
    print prime
del prime
print 'Done!'
```

Create loop variable

Use loop variable

Delete loop variable

```
#!/usr/bin/python

# This is a list of numbers we want
#  to add up.
weights = [ 0.1, 0.5, 2.6, 7.0, 5.3 ]

# Add all the numbers in the list
#  together.



# Print the result.
print
```

What goes here?

addition.py

Answer

```
#!/usr/bin/python

# This is a list of numbers we want
#  to add up.
weights = [ 0.1, 0.5, 2.6, 7.0, 5.3 ]

# Add all the numbers in the list
#  together.
total = 0.0
for weight in weights:
    total = total + weight
del weight

# Print the result.
print total
```

addition.py

Lists of anything

```
primes = [ 2, 3, 5, 7, 11, 13, 17, 19 ]
```

List of integers

```
names = [ 'Alice', 'Bob', 'Cathy', 'Dave' ]
```

List of strings

```
roots = [ 0.0,  1.57079632679,   3.14159265359 ]
```

List of floats

```
lists = [ [ 1, 2, 3 ], [5], [9, 1] ]
```

List of *lists*

Mixed lists

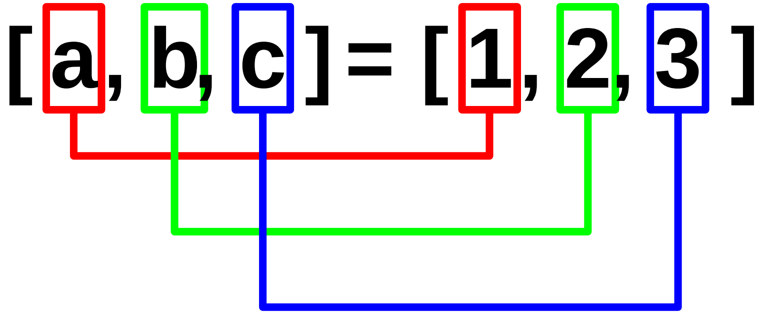
```
stuff = [ 2, 'Bob', 3.14159265359, 'Dave' ]
```

⚠

Legal, but not a good idea.

See “tuples” later.

Lists of variables



```
>>>  [a, b, c ] = [ 1, 2, 3 ]
```

```
>>>  a
1

>>>  b
2

>>>  c
3
```

All or nothing

Traceback: where the error happened

```
>>>   [ d, e, f ] = [ 1, 2, 3, 4 ]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: too many values to unpack
```

Error message

```
>>>  d
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'd' is not defined
```

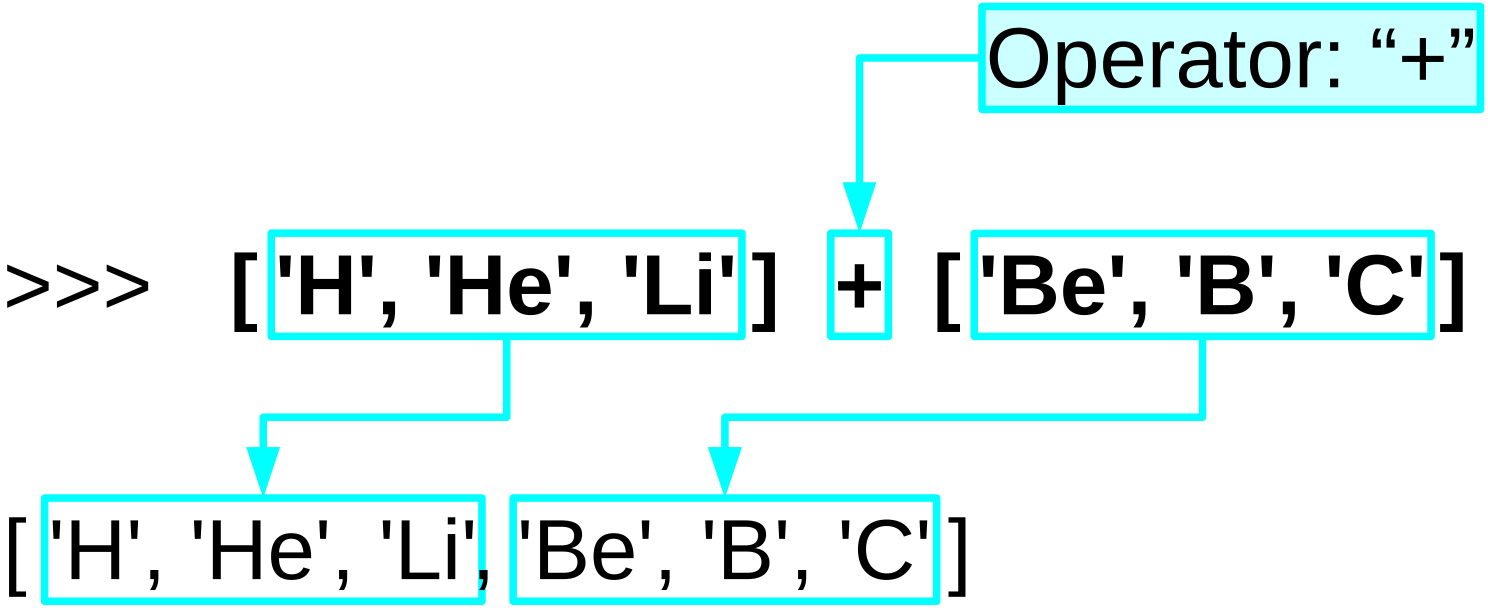
All or nothing

```
>>>   [ g, h, i , j  ]= [ 1, 2, 3 ]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: need more than 3 values to unpack
```

Error message

```
>>>  g
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'g' is not defined
```

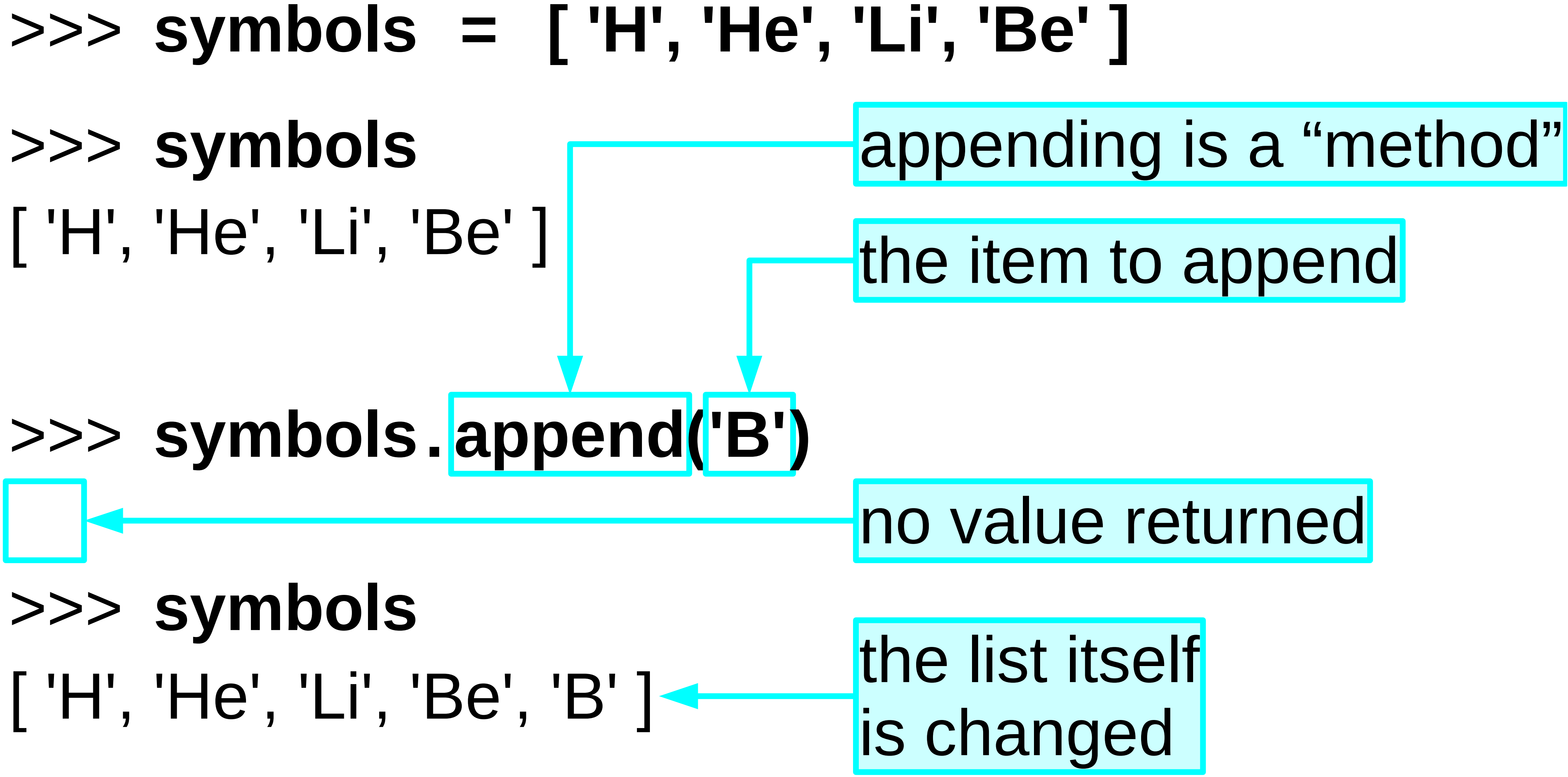
Concatenating lists



Operator: “+”

```
>>>   ['H', 'He', 'Li'] + ['Be', 'B', 'C']
['H', 'He', 'Li', 'Be', 'B', 'C']
```

Appending an item: **append( )**



```
>>> symbols = [ 'H', 'He', 'Li', 'Be' ]

>>> symbols
[ 'H', 'He', 'Li', 'Be' ]

>>> symbols.append('B')

>>> symbols
[ 'H', 'He', 'Li', 'Be', 'B' ]
```

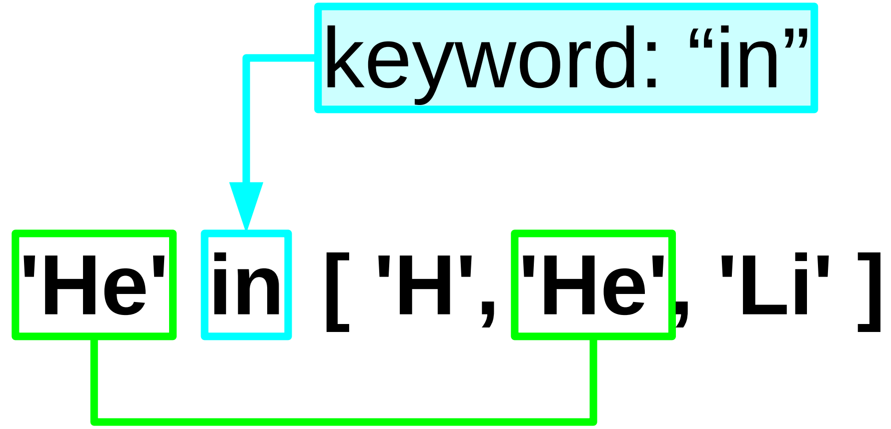
appending is a “method”

the item to append

no value returned

the list itself
is changed

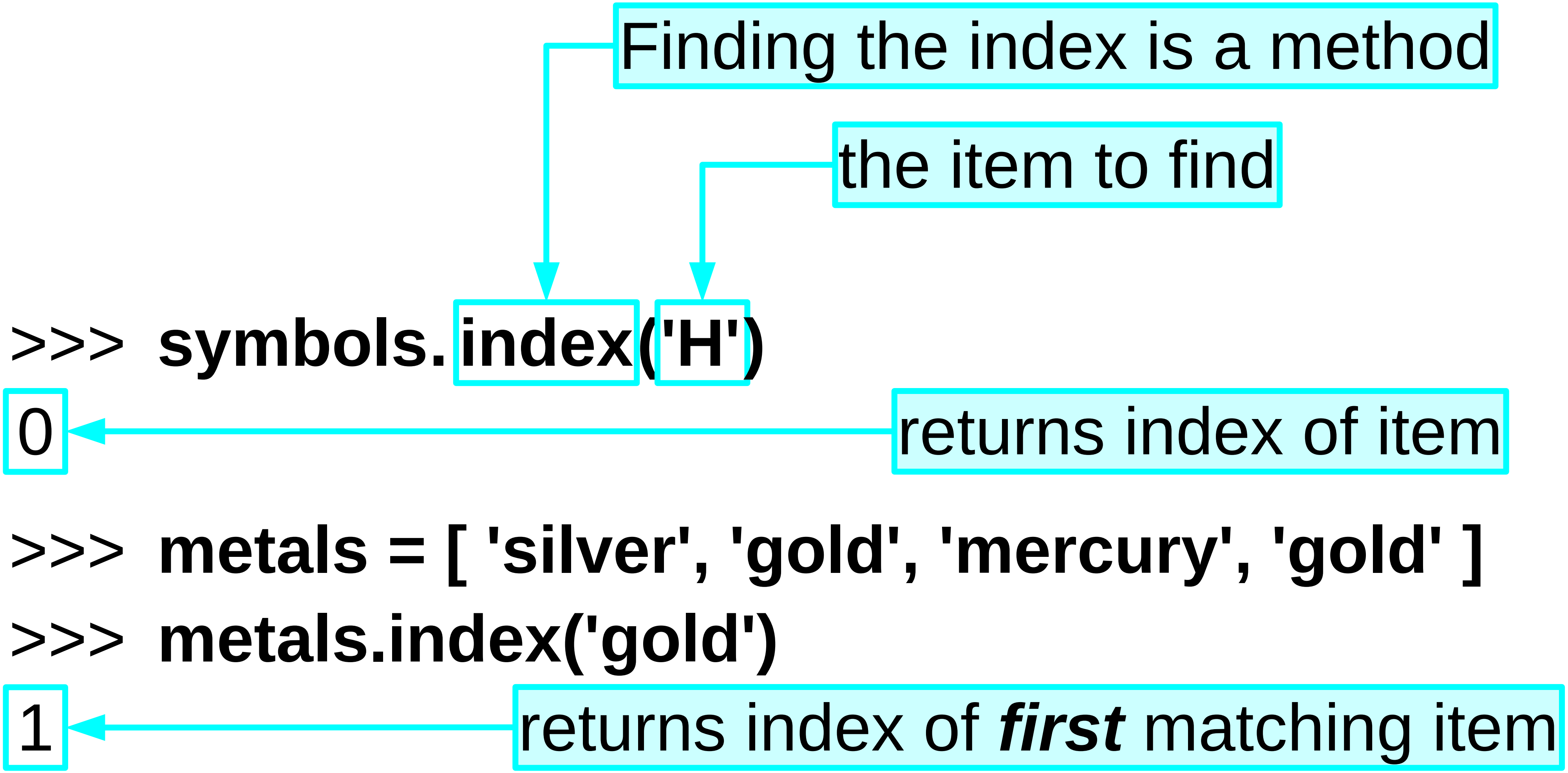
Membership of lists



keyword: “in”

```
>>>  'He' in [ 'H', 'He', 'Li' ]

True


>>>  'He' in [ 'Be', 'B', 'C' ]

False
```

Finding the index of an item



```
>>> symbols = [ 'H', 'He', 'Li', 'Be' ]
>>> symbols.index('H')
0
>>> metals = [ 'silver', 'gold', 'mercury', 'gold' ]
>>> metals.index('gold')
1
```

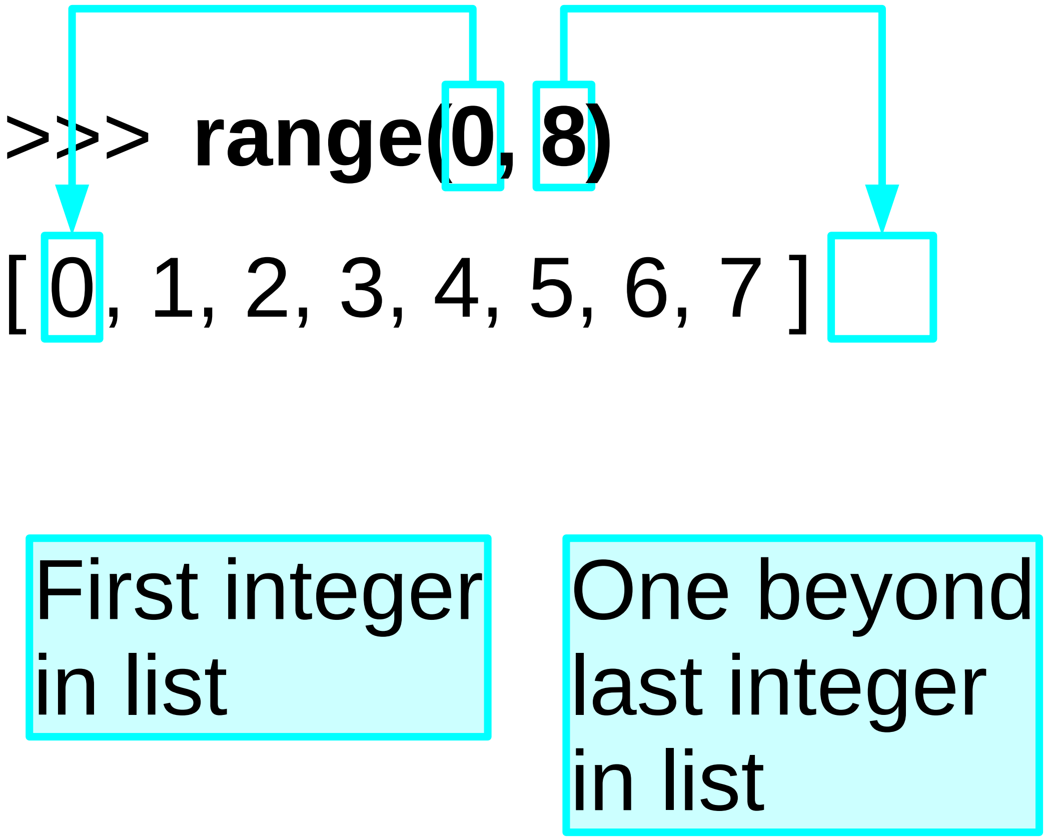
Finding the index is a method

the item to find

returns index of item

returns index of *first* matching item

Functions that give lists: **`range()`**

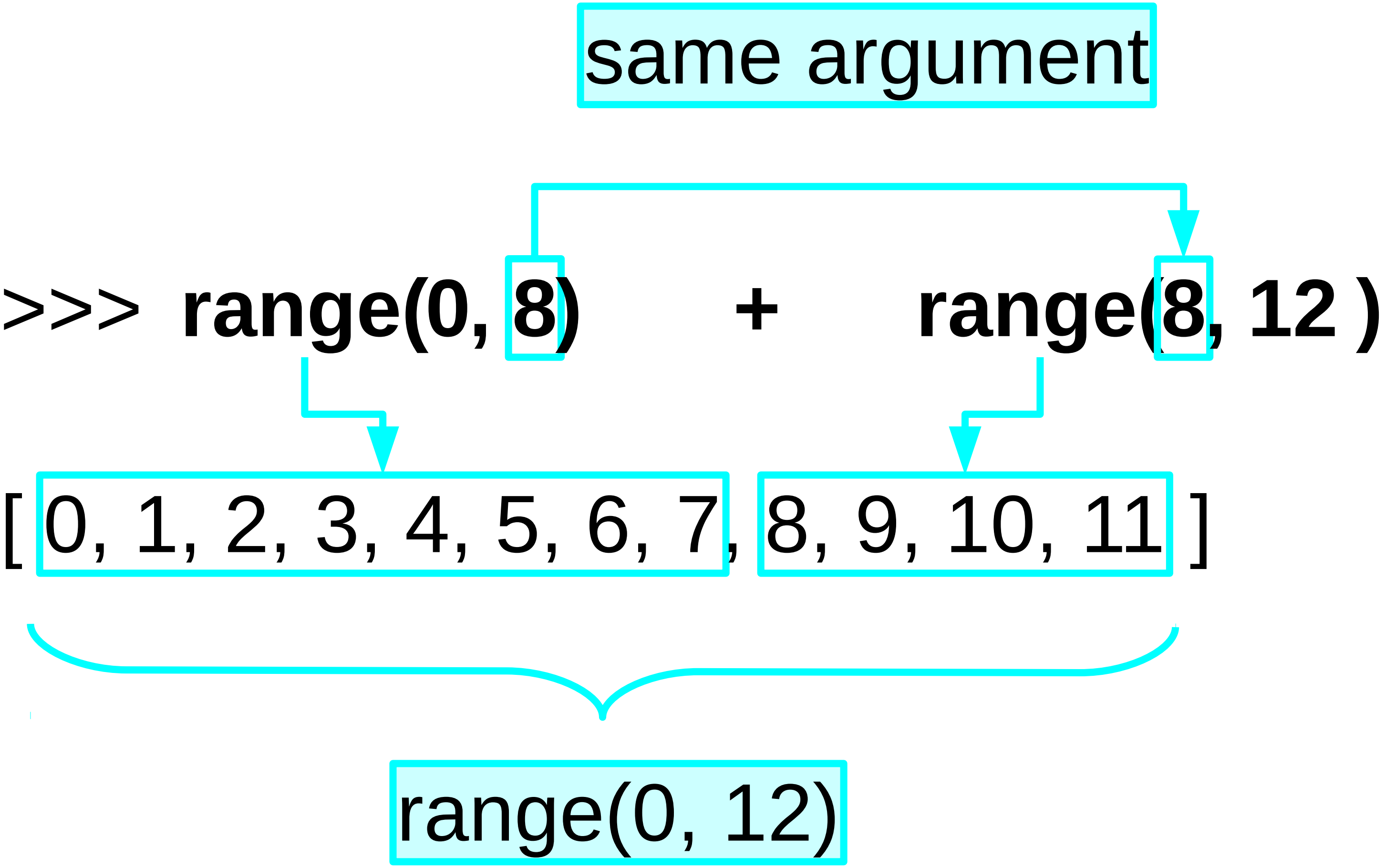


```
>>> range(0, 8)
[ 0, 1, 2, 3, 4, 5, 6, 7 ]
```

First integer in list

One beyond last integer in list

`range()`: Why miss the last number?

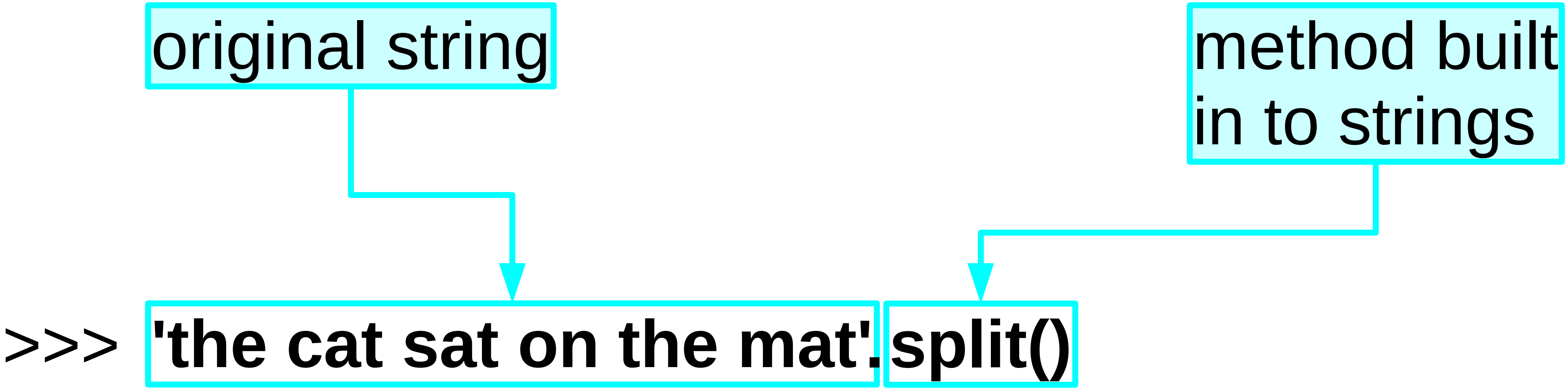


same argument

```
>>> range(0, 8) + range(8, 12 )
[ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11 ]
```

range(0, 12)

Functions that give lists: **`split()`**



original string

method built in to strings

```
>>> 'the cat sat on the mat'.split()
[ 'the', 'cat', 'sat', 'on', 'the', 'mat' ]
```

Split on white space

Spaces discarded

`split()`: Only good for trivial splitting

```
>>> 'the cat sat on the mat'.split()
[ 'the', 'cat', 'sat', 'on', 'the', 'mat' ]
```

Split on white space

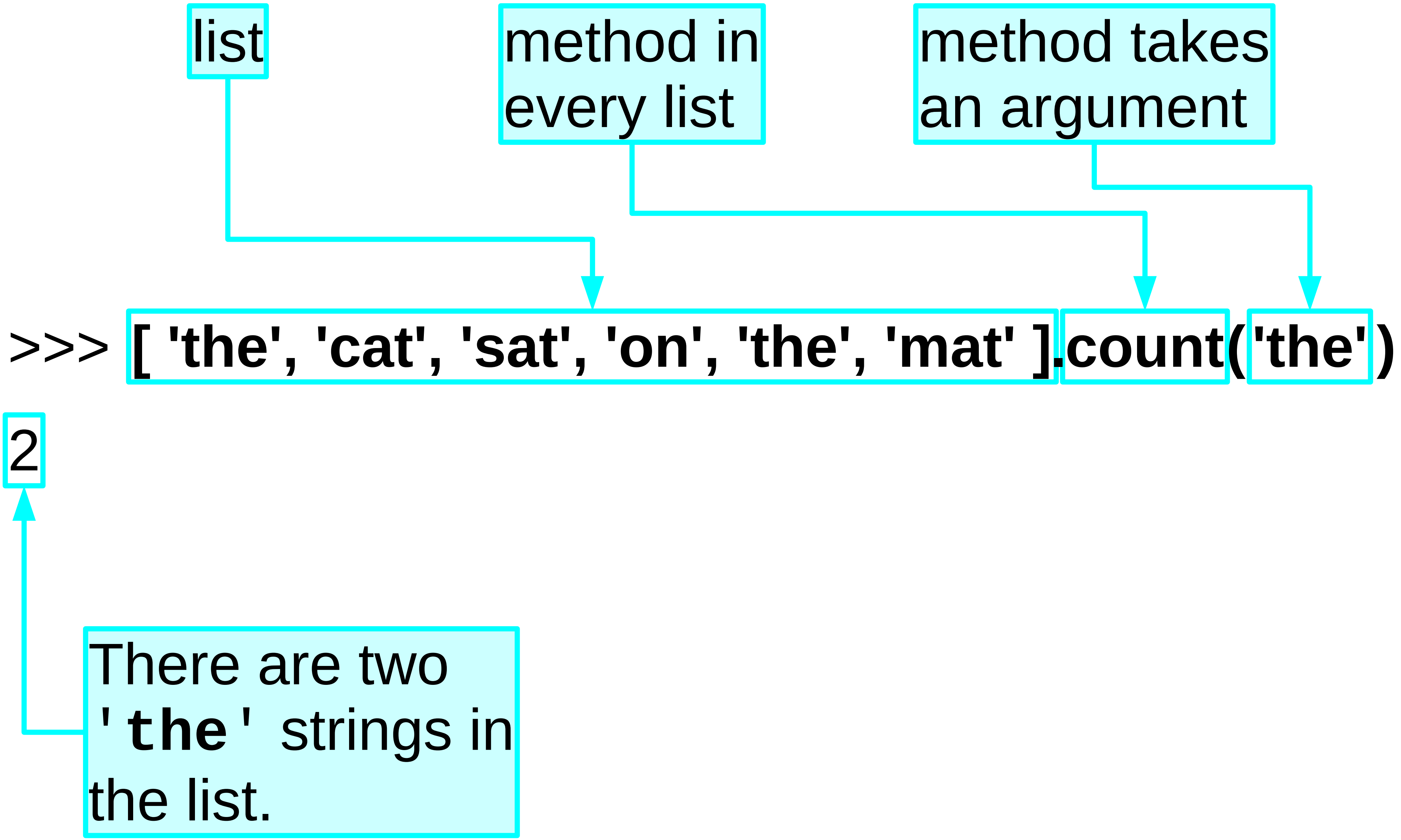
Spaces discarded

Trivial operation

Regular expressions

Comma separated values

Use the specialist Python support for these.



list

method in every list

method takes an argument

```
>>> [ 'the', 'cat', 'sat', 'on', 'the', 'mat' ].count('the')
2
```

There are two **'the'** strings in the list.

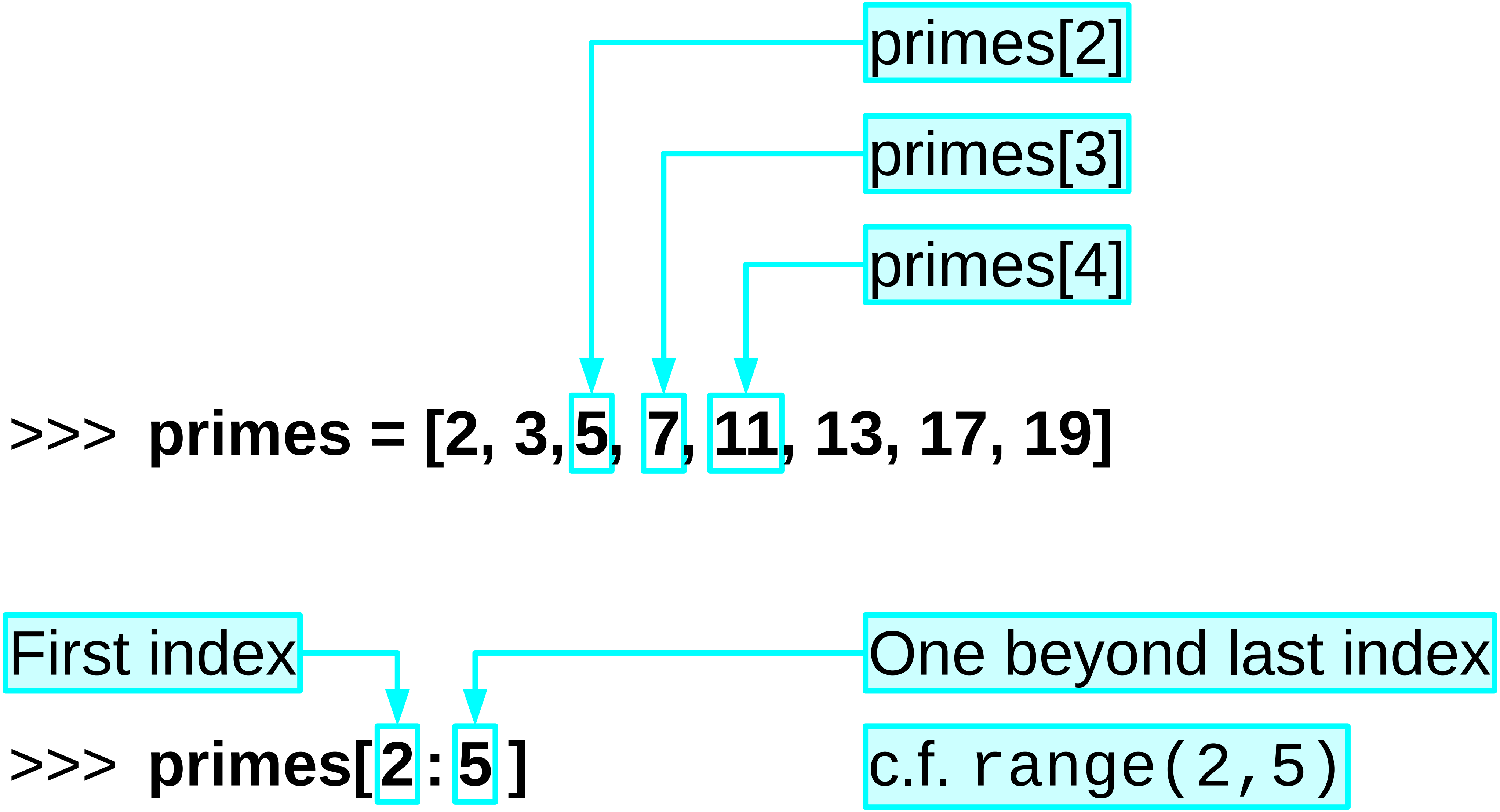
Combining methods

```
>>>  'the cat sat on the mat'.split().count('the')
2
```

First run **`split()`** to get a list

Second run `count(`**`'the'`**`)` on that list

Extracts from lists: “slices”



primes[2]

primes[3]

primes[4]

```
>>> primes = [2, 3, 5, 7, 11, 13, 17, 19]
```

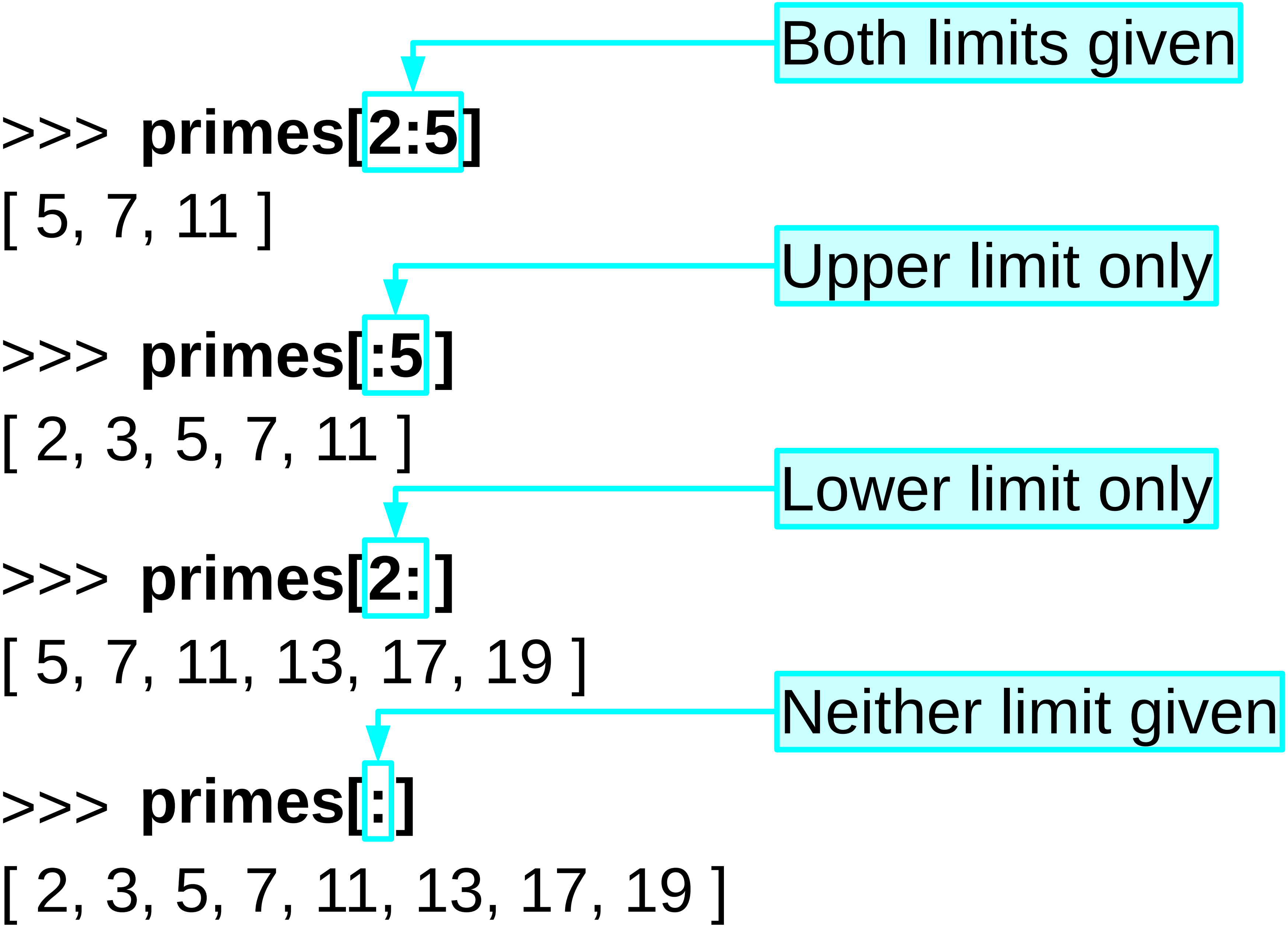
First index

One beyond last index

```
>>> primes[2:5]
```

c.f. `range(2,5)`

```
[ 5, 7, 11 ]
```



Both limits given

```
>>> primes[2:5]
[ 5, 7, 11 ]
```

Upper limit only

```
>>> primes[:5]
[ 2, 3, 5, 7, 11 ]
```

Lower limit only

```
>>> primes[2:]
[ 5, 7, 11, 13, 17, 19 ]
```

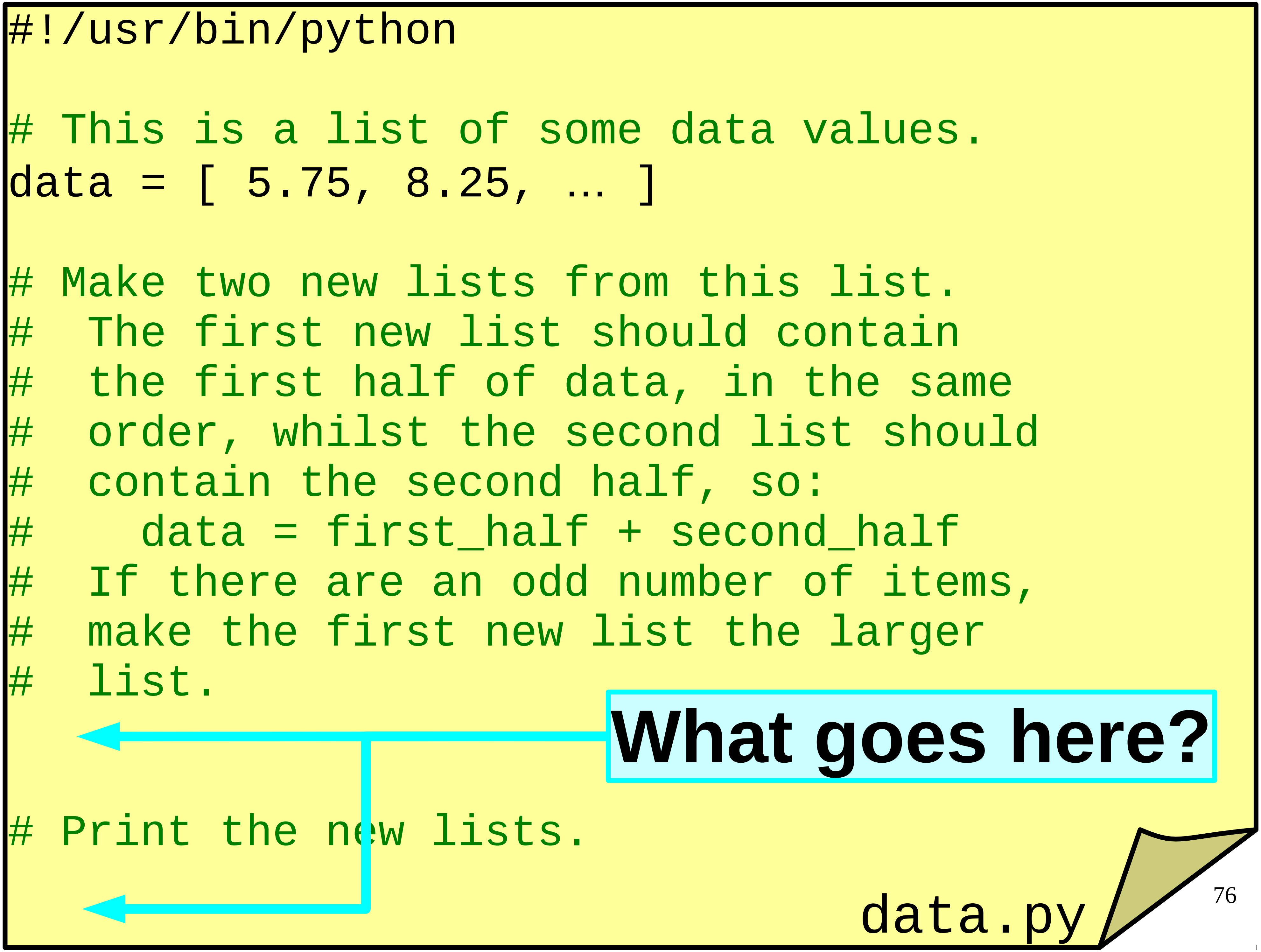
Neither limit given

```
>>> primes[:]
[ 2, 3, 5, 7, 11, 13, 17, 19 ]
```

```
#!/usr/bin/python

# This is a list of some metallic
#  elements.
metals = [ 'silver', 'gold', … ]

# Make a new list that is almost
#  identical to the metals list: the new
#  contains the same items, in the same
#  order, except that it does *NOT*
#  contain the item 'copper'.



# Print the new list.
```

What goes here?

metals.py



```
#!/usr/bin/python

# This is a list of some data values.
data = [ 5.75, 8.25, … ]

# Make two new lists from this list.
#  The first new list should contain
#  the first half of data, in the same
#  order, whilst the second list should
#  contain the second half, so:
#    data = first_half + second_half
#  If there are an odd number of items,
#  make the first new list the larger
#  list.


# Print the new lists.

```

What goes here?

data.py

An answer

```
#!/usr/bin/python

# This is a list of some metallic
#  elements.
metals = [ 'silver', 'gold', … ]

# Make a new list that is almost
#  identical to the metals list: the new
#  contains the same items, in the same
#  order, except that it does *NOT*
#  contain the item 'copper'.
new_metals = []
for metal in metals:
    if metal != 'copper':
      new_metals.append(metal)

# Print the new list.
print new_metals
```

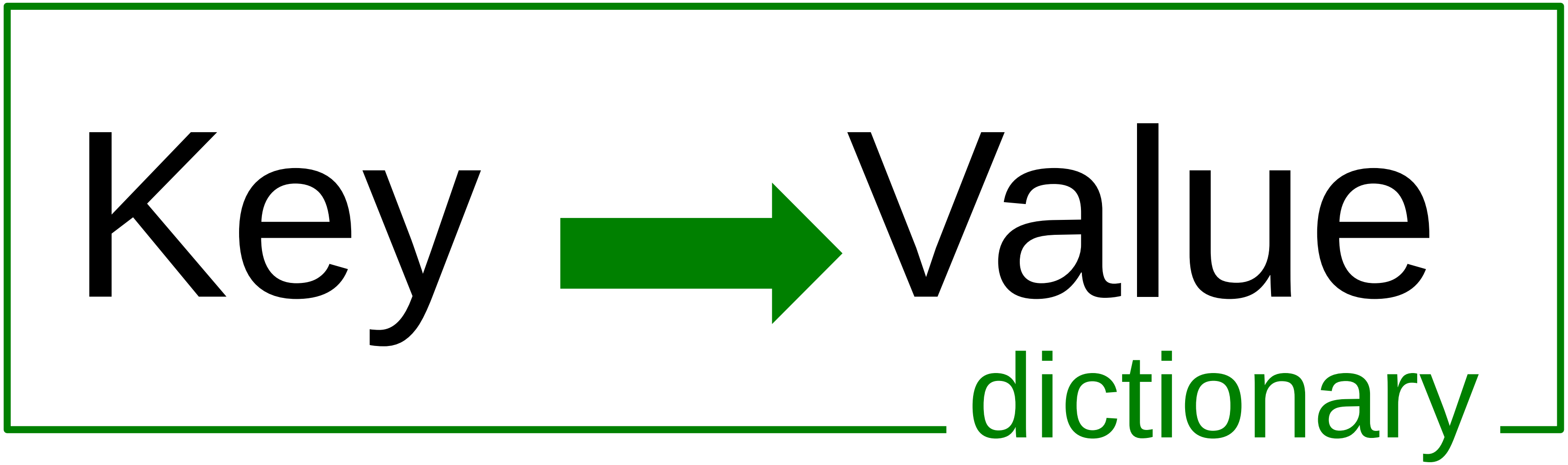
metals.py

Answer

```
#!/usr/bin/python

# This is a list of some data values.
data = [ 5.75, 8.25, … ]

# Make two new lists from this list.
#  The first new list should contain
#  the first half of data, in the same
#  order, whilst the second list should
#  contain the second half, so:
#    data = first_half + second_half
#  If there are an odd number of items,
#  make the first new list the larger
#  list.
if len(data) % 2 == 0:
    index = len(data) / 2
else:
    index = (len(data) + 1) / 2

first_half = data[:index]
second_half = data[index:]

# Print the new lists.
print first_half
print second_half
```

data.py

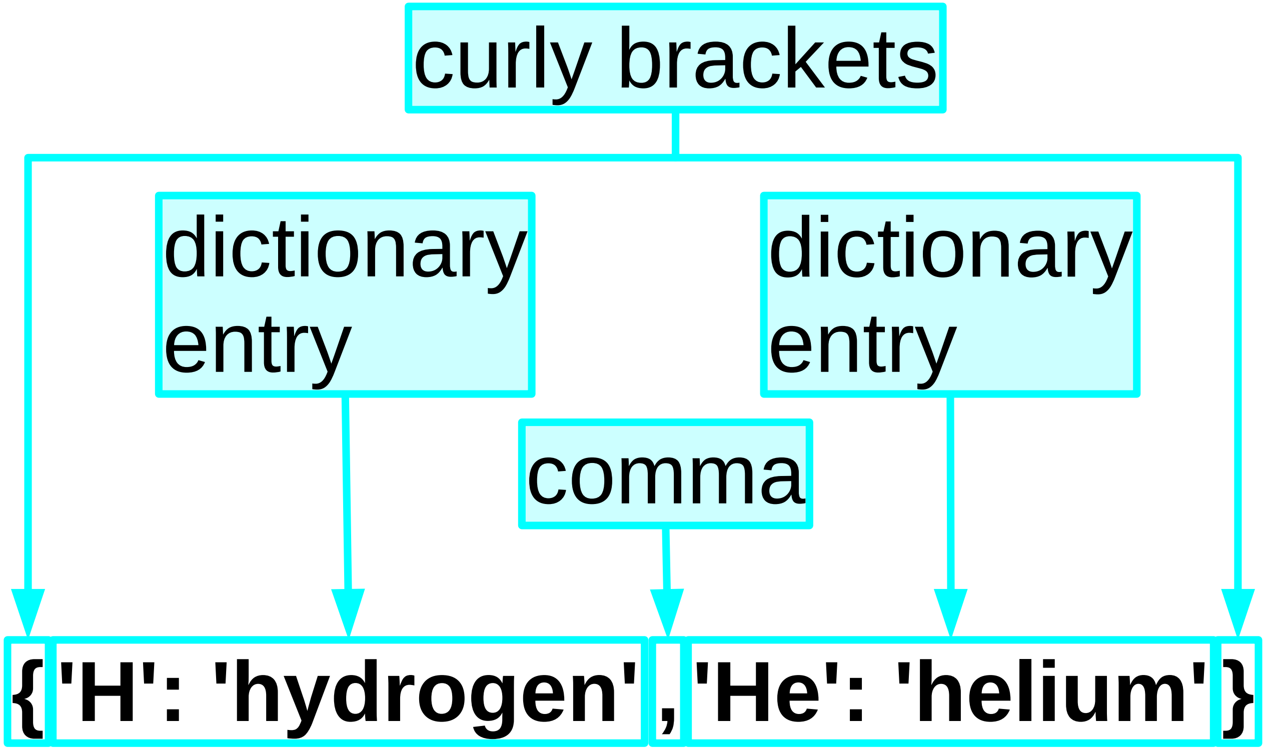
Dictionaries



Key → Value

dictionary

Creating a dictionary — 1



curly brackets

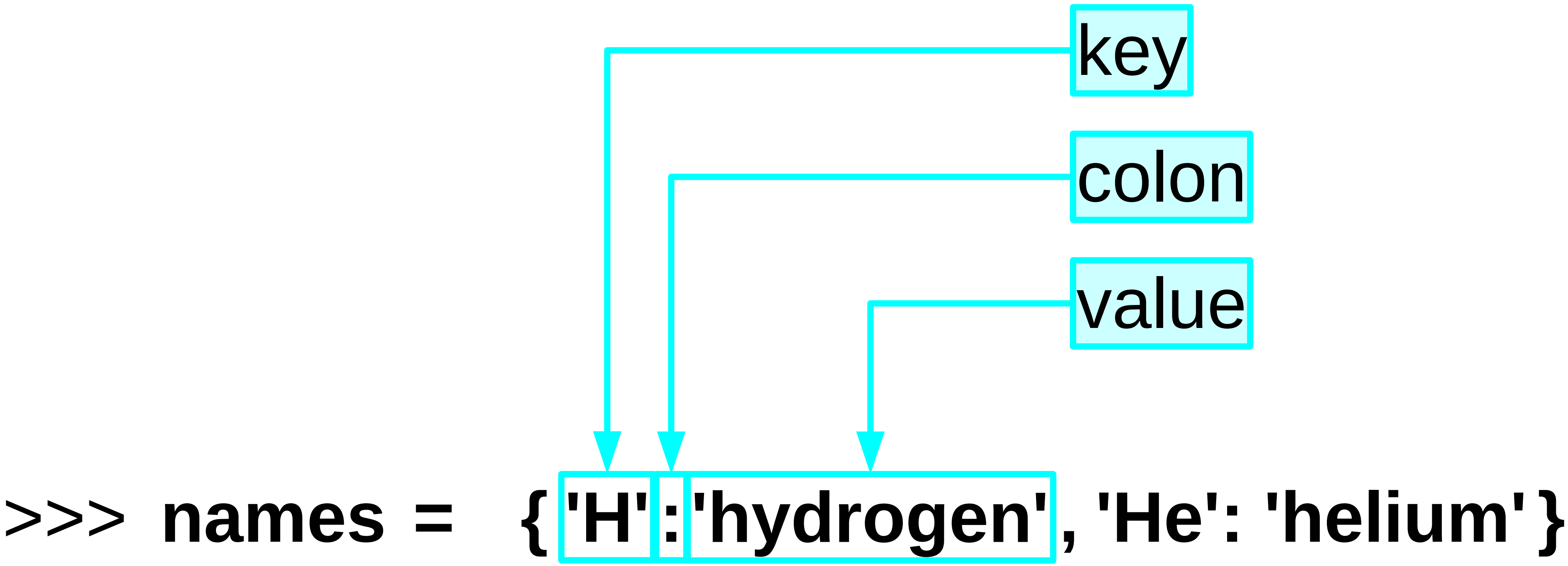
dictionary entry

comma

dictionary entry

```
>>> names = {'H': 'hydrogen','He': 'helium'}
```

Creating a dictionary — 2



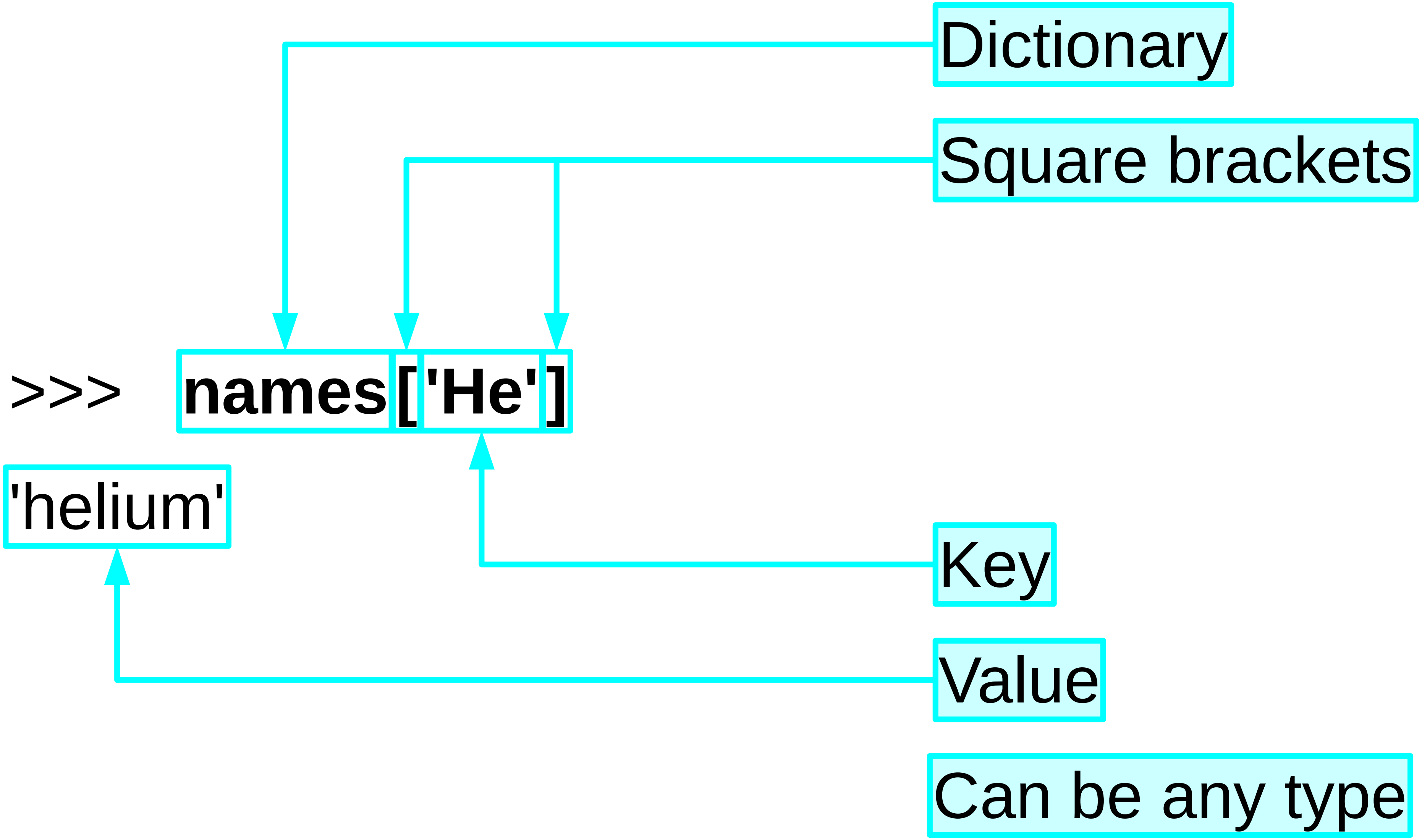
key

colon

value

```
>>>  names =   {'H':'hydrogen', 'He': 'helium'}
```

Accessing a dictionary



```
>>> names['He']
'helium'
```

Dictionary

Square brackets

Key

Value

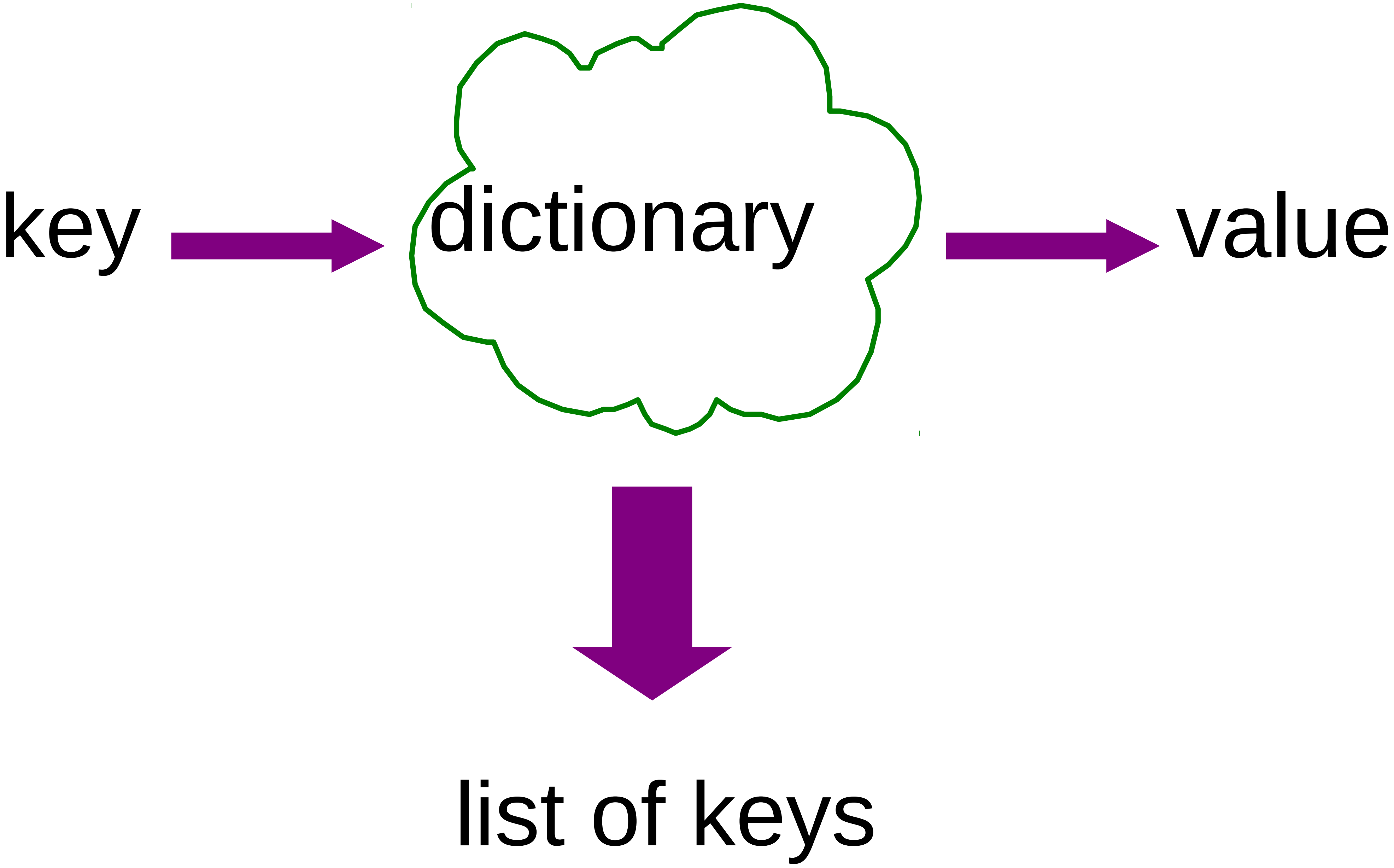
Can be any type

Creating a dictionary — 3

```
>>>  names = {}
>>>  names[ 'H' ]   =  'hydrogen'
>>>  names[ 'He' ] =  'helium'
>>>  names[ 'Li' ]  =  'lithium'
>>>  names[ 'Be' ] =  'beryllium'
```

Start with an empty dictionary

Add entries



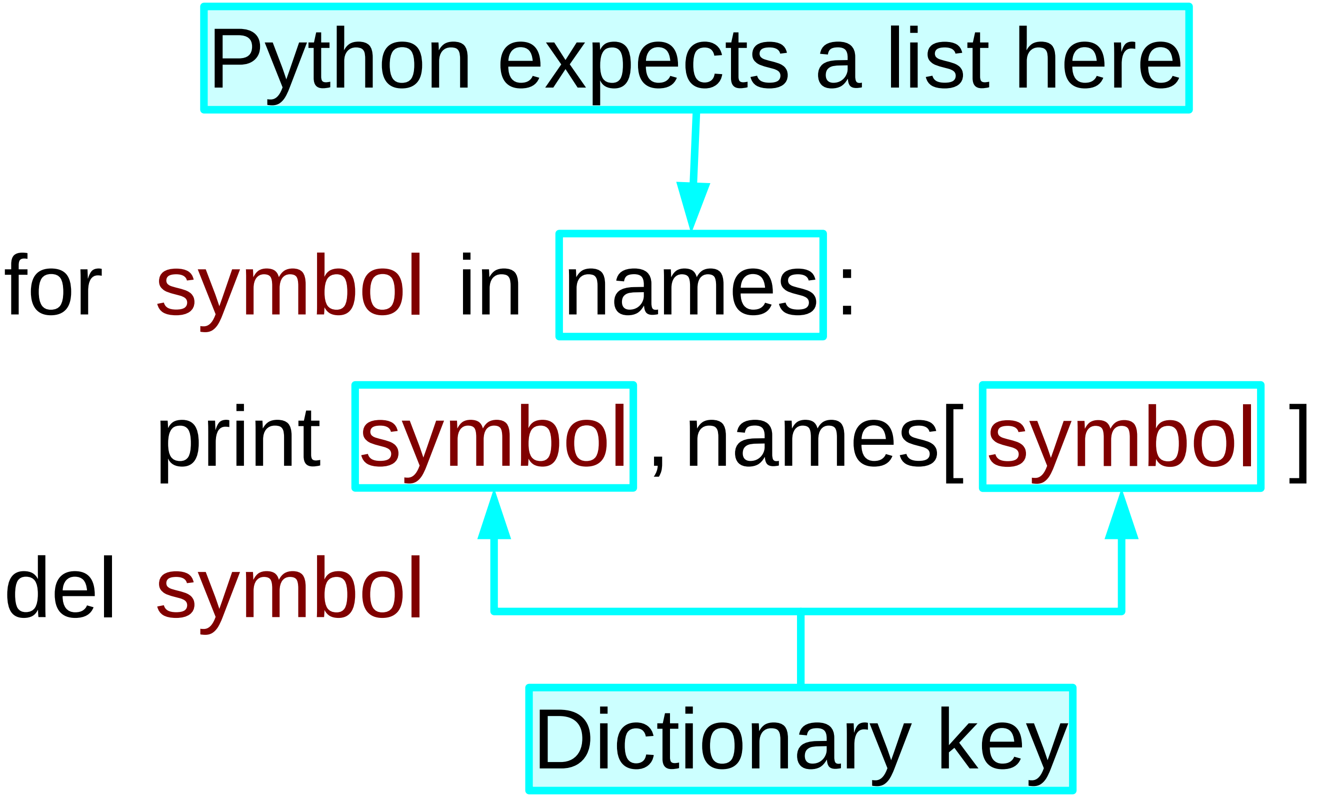
key

dictionary

value

list of keys

Treat a dictionary like a list…



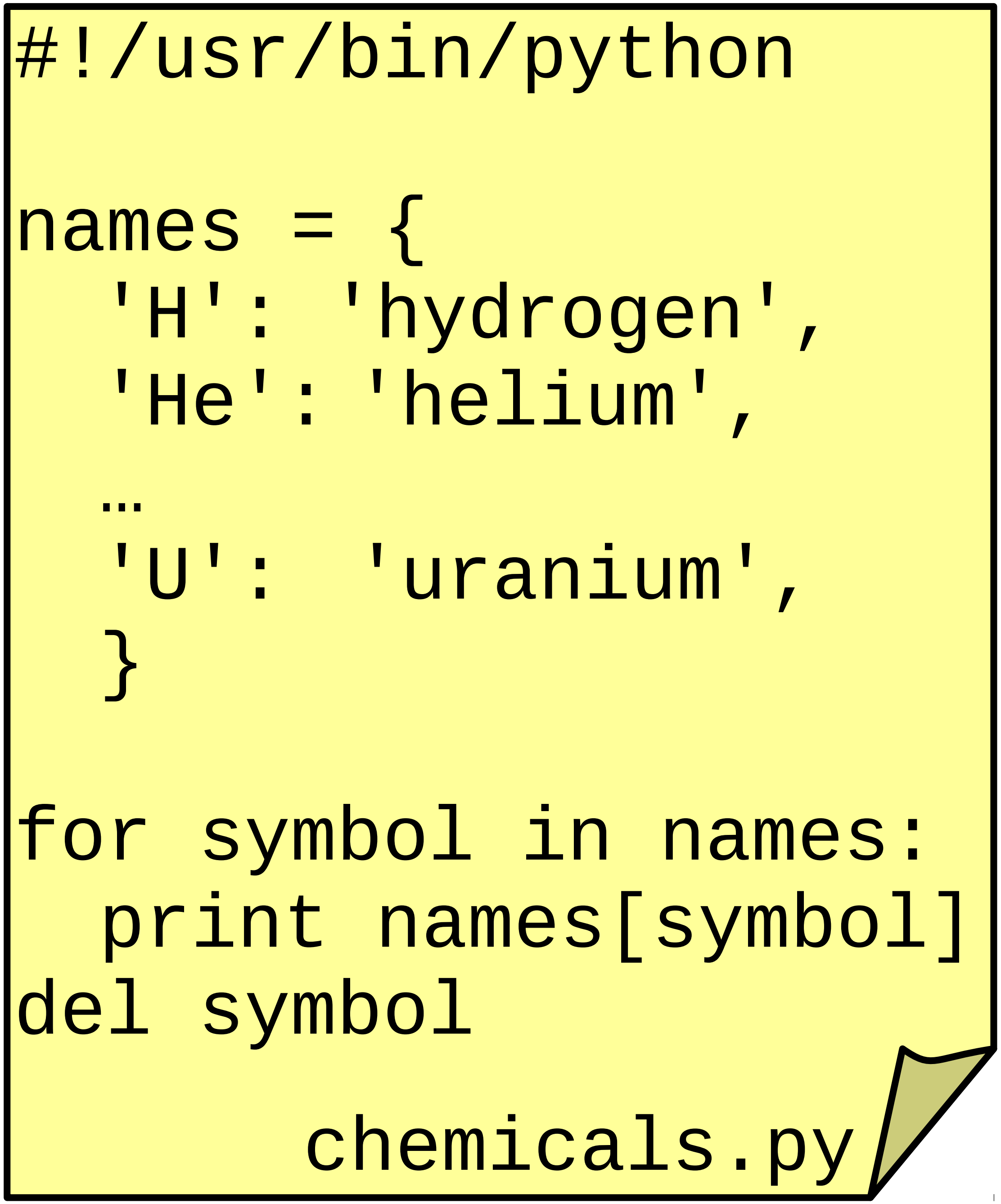
Python expects a list here

```
for  symbol in names:
     print symbol, names[ symbol ]
del symbol
```

Dictionary key

…and it behaves like a list of keys

Example



```
#!/usr/bin/python

names = {
  'H': 'hydrogen',
  'He': 'helium',
  …
  'U':  'uranium',
  }

for symbol in names:
  print names[symbol]
del symbol
```

chemicals.py

```
$ python chemicals.py

ruthenium
rhenium
…
astatine
indium
```

No relation between order in file and output!

Missing keys

missing key

```
>>>  names['Np']

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'Np'
```

Type of error

Missing key

Treat a dictionary like a list…

Python expects a list here

```
if symbol in names:
    print symbol, names[ symbol ]
```

…and it behaves like a list of keys

Missing keys

```
>>>   names['Np']

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'Np'


>>>  'Np' in names

False
```

Test for membership of a *list*

'Np' is not a key in the dictionary

And now for something completely…

Obviously when you create a dictionary you need to be clear about which items are the keys and which are the values. But what if you are given a dictionary that is the “wrong way round”?

Have a look at the script `chemicals_reversed.py` in your home directory.

See if you can figure out what it does – if there is anything you don’t understand, tell the course giver or demonstrator.

Defining functions

define a function

Values in: a_to_b

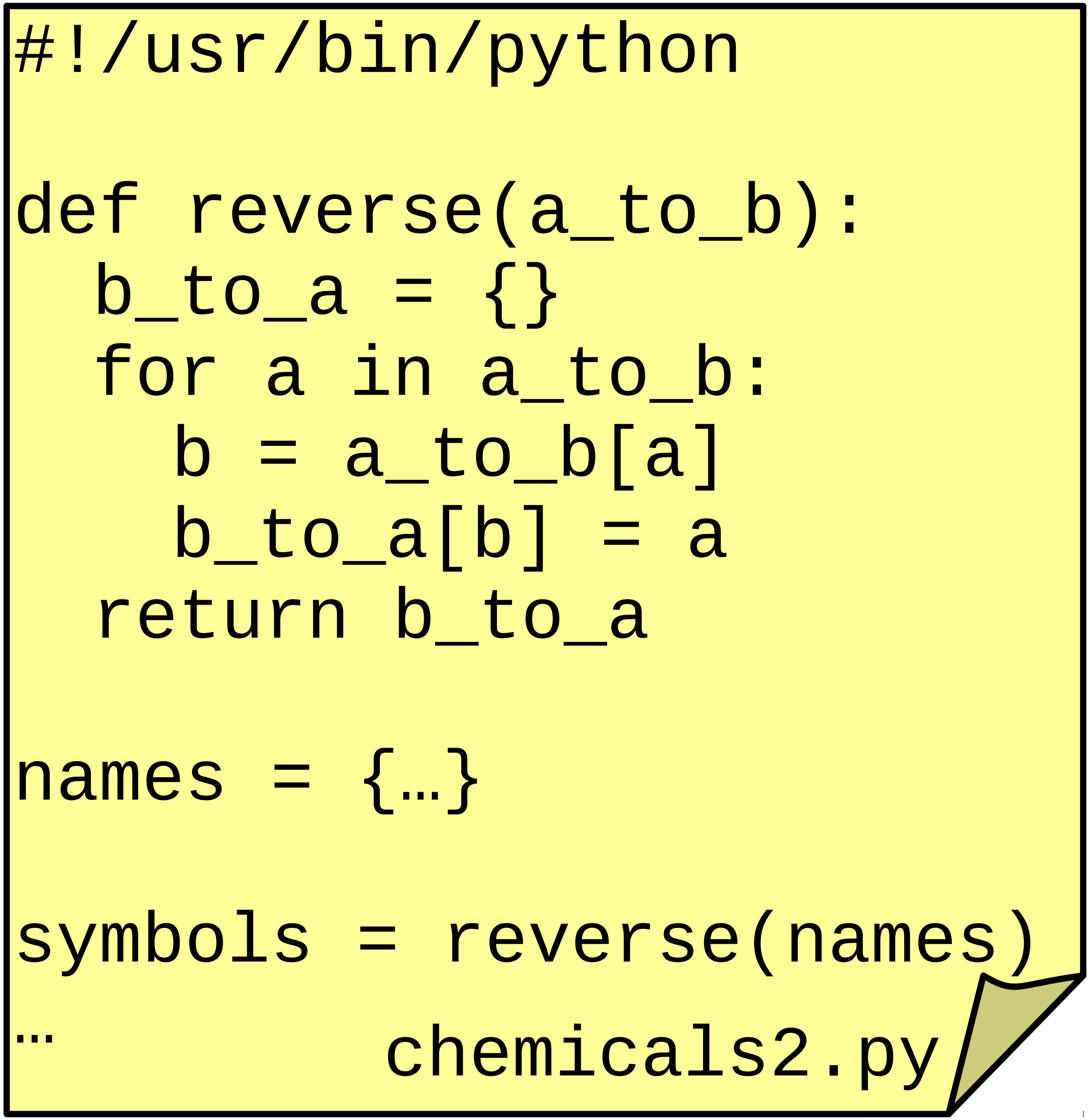
Values out: b_to_a

Internal values: a b

Internal values are automatically cleaned up on exit.

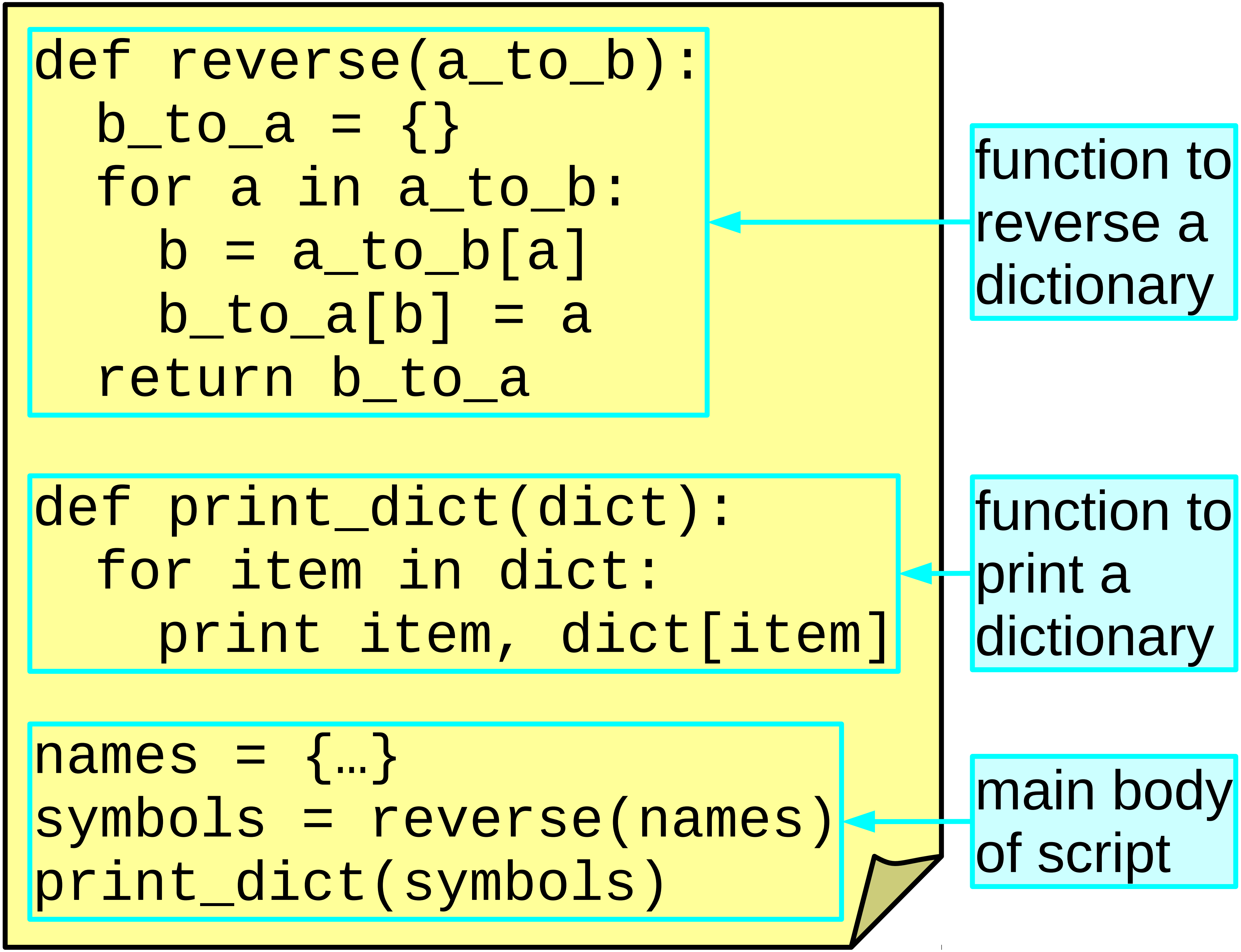
```
def reverse ( a_to_b ):
    b_to_a = {}
    for a in a_to_b :
        b = a_to_b[ a ]
        b_to_a [ b ] = a
    return b_to_a
```

Example



```
#!/usr/bin/python

def reverse(a_to_b):
  b_to_a = {}
  for a in a_to_b:
    b = a_to_b[a]
    b_to_a[b] = a
  return b_to_a

names = {…}

symbols = reverse(names)
…
```

chemicals2.py



```
def reverse(a_to_b):
  b_to_a = {}
  for a in a_to_b:
    b = a_to_b[a]
    b_to_a[b] = a
  return b_to_a
```

function to reverse a dictionary

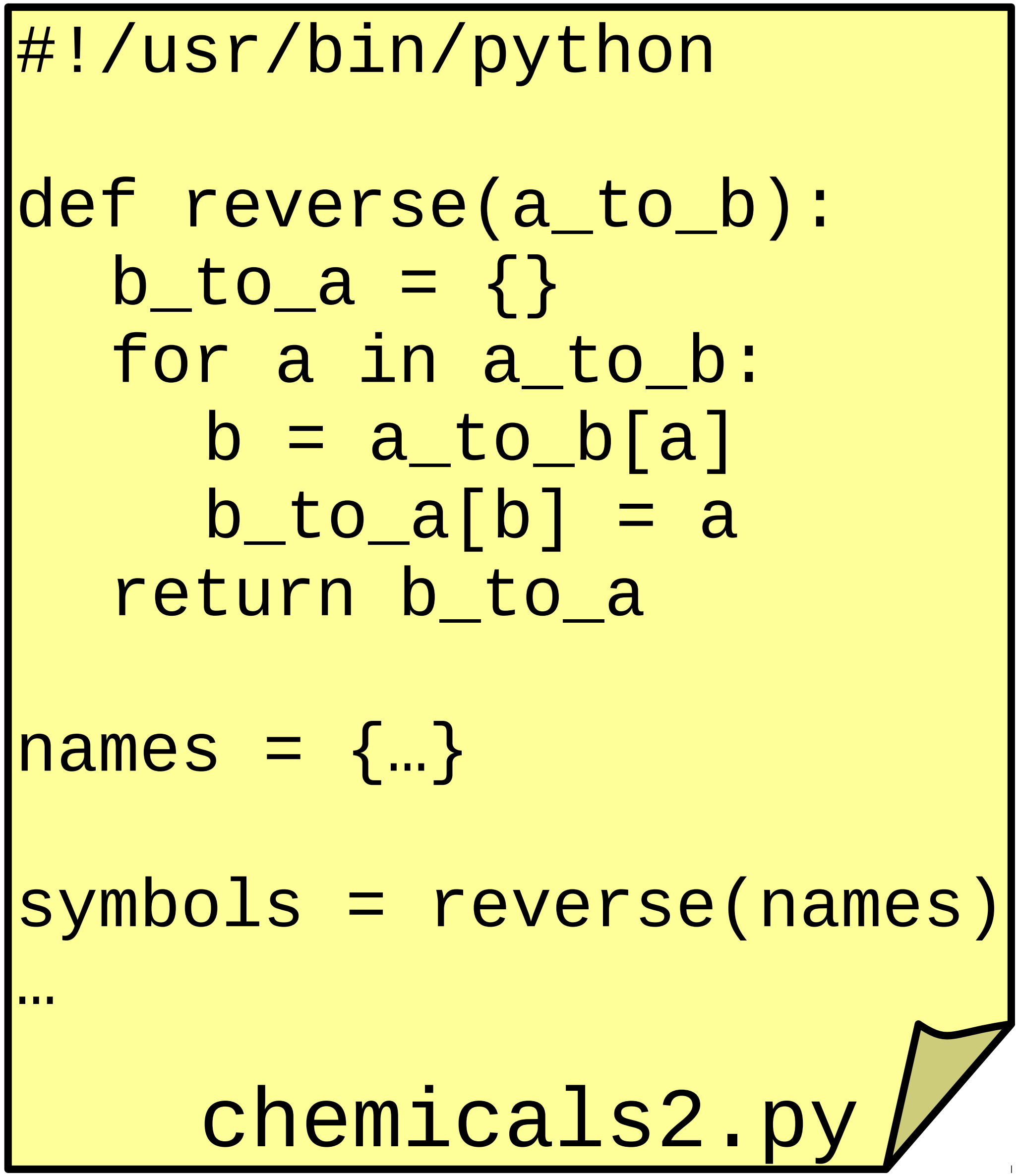
```
def print_dict(dict):
  for item in dict:
    print item, dict[item]
```

function to print a dictionary

```
names = {…}
symbols = reverse(names)
print_dict(symbols)
```

main body of script

Let’s try it out…



```
#!/usr/bin/python

def reverse(a_to_b):
  b_to_a = {}
  for a in a_to_b:
    b = a_to_b[a]
    b_to_a[b] = a
  return b_to_a

names = {…}

symbols = reverse(names)
…
```

chemicals2.py

```
$ python chemicals2.py
gold Au
neon Ne
cobalt Co
germanium Ge
…
tellurium Te
xenon Xe
```

Re-using functions: “modules”

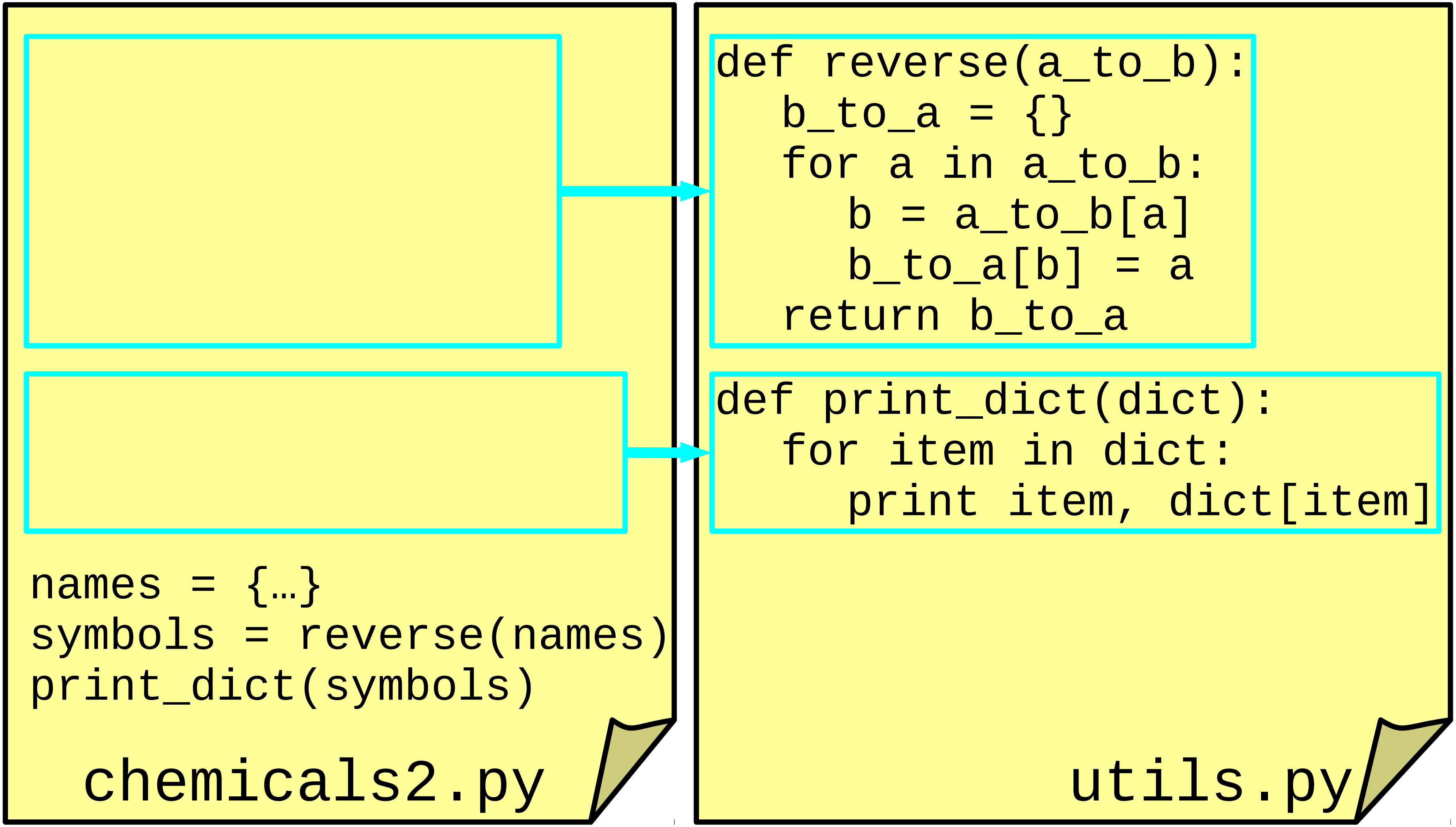
Two functions:

```
reverse(a_to_b)
print_dict(dict)
```

currently in `chemicals2.py`

Modules — 1

Put function definitions to new file `utils.py`



```
names = {…}
symbols = reverse(names)
print_dict(symbols)
```

chemicals2.py

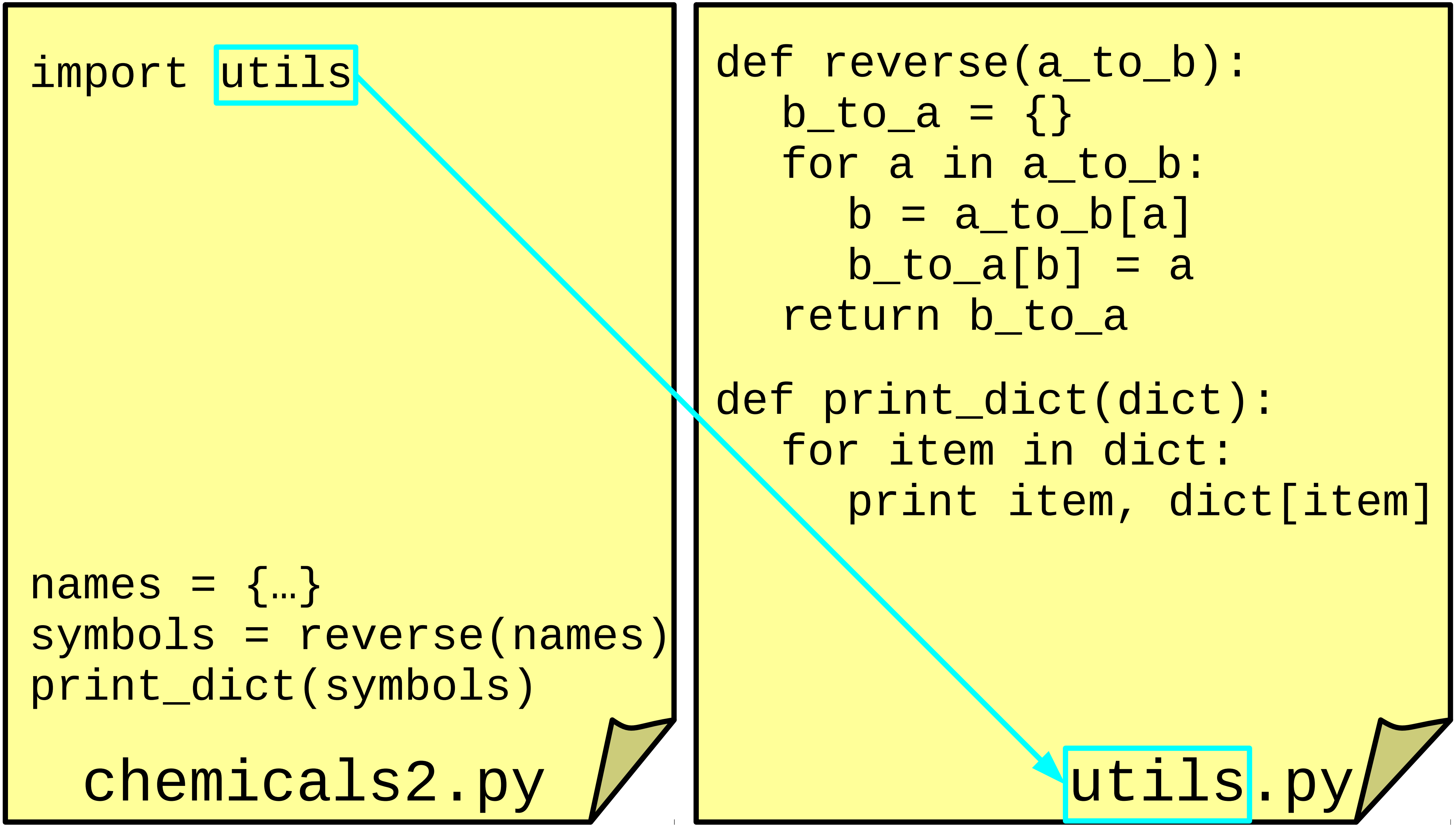
```
def reverse(a_to_b):
  b_to_a = {}
  for a in a_to_b:
    b = a_to_b[a]
    b_to_a[b] = a
  return b_to_a
```

```
def print_dict(dict):
  for item in dict:
    print item, dict[item]
```

utils.py

Modules — 2

“import” the module



```
import utils






names = {…}
symbols = reverse(names)
print_dict(symbols)
```

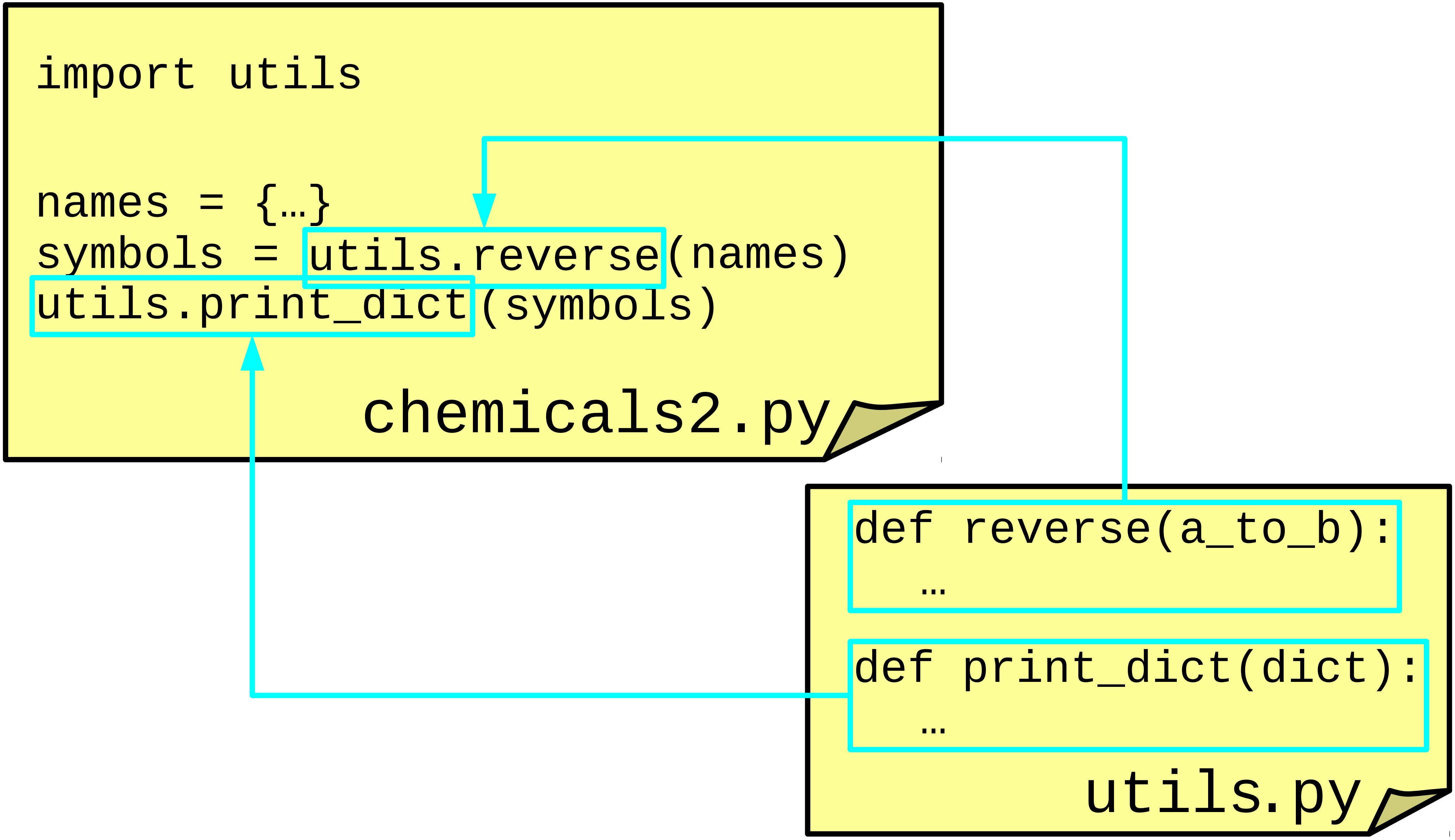
chemicals2.py

```
def reverse(a_to_b):
   b_to_a = {}
   for a in a_to_b:
     b = a_to_b[a]
     b_to_a[b] = a
   return b_to_a

def print_dict(dict):
   for item in dict:
     print item, dict[item]
```

utils.py

Modules — 3

Use functions from the module



```
import utils


names = {…}
symbols = utils.reverse(names)
utils.print_dict(symbols)
```

chemicals2.py

```
def reverse(a_to_b):
    …

def print_dict(dict):
    …
```

utils.py

Let’s check it still works…

```
#!/usr/bin/python

import utils

names = {…}

symbols = utils.reverse(names)
utils.print_dict(symbols)
```

chemicals2.py

```
$ python chemicals2.py
gold Au
neon Ne
cobalt Co
germanium Ge
…
tellurium Te
xenon Xe
```

System modules

profile getpass re bisect os gzip pickle time

anydbm bz2

unittest calendar

cgi csv asyncore atexit shelve datetime

optparse asynchat webbrowser mmap hmac

sched

BaseHTTPServer math

cmath heapq

SimpleHTTPServer

imageop

CGIHTTPServer email audioop

Cookie logging sys unicodedata base64 sets

codecs stringprep tempfile hashlib mutex

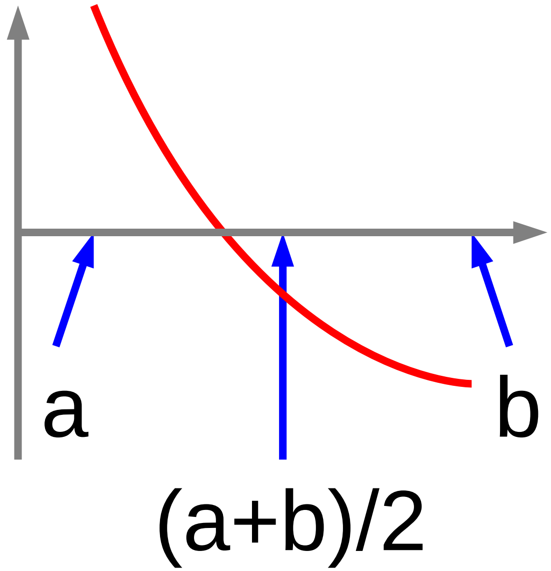
select string chunk

locale cmd code ConfigParser

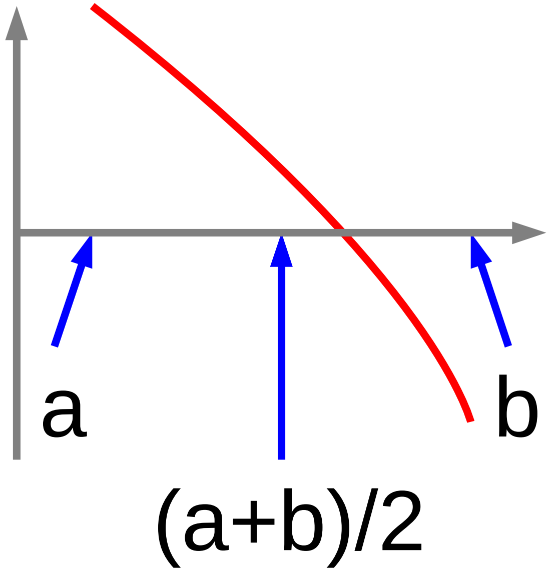
linecache

colorsys gettext collections glob

Root-finding by bisection



y=f(x)



a, b

↓

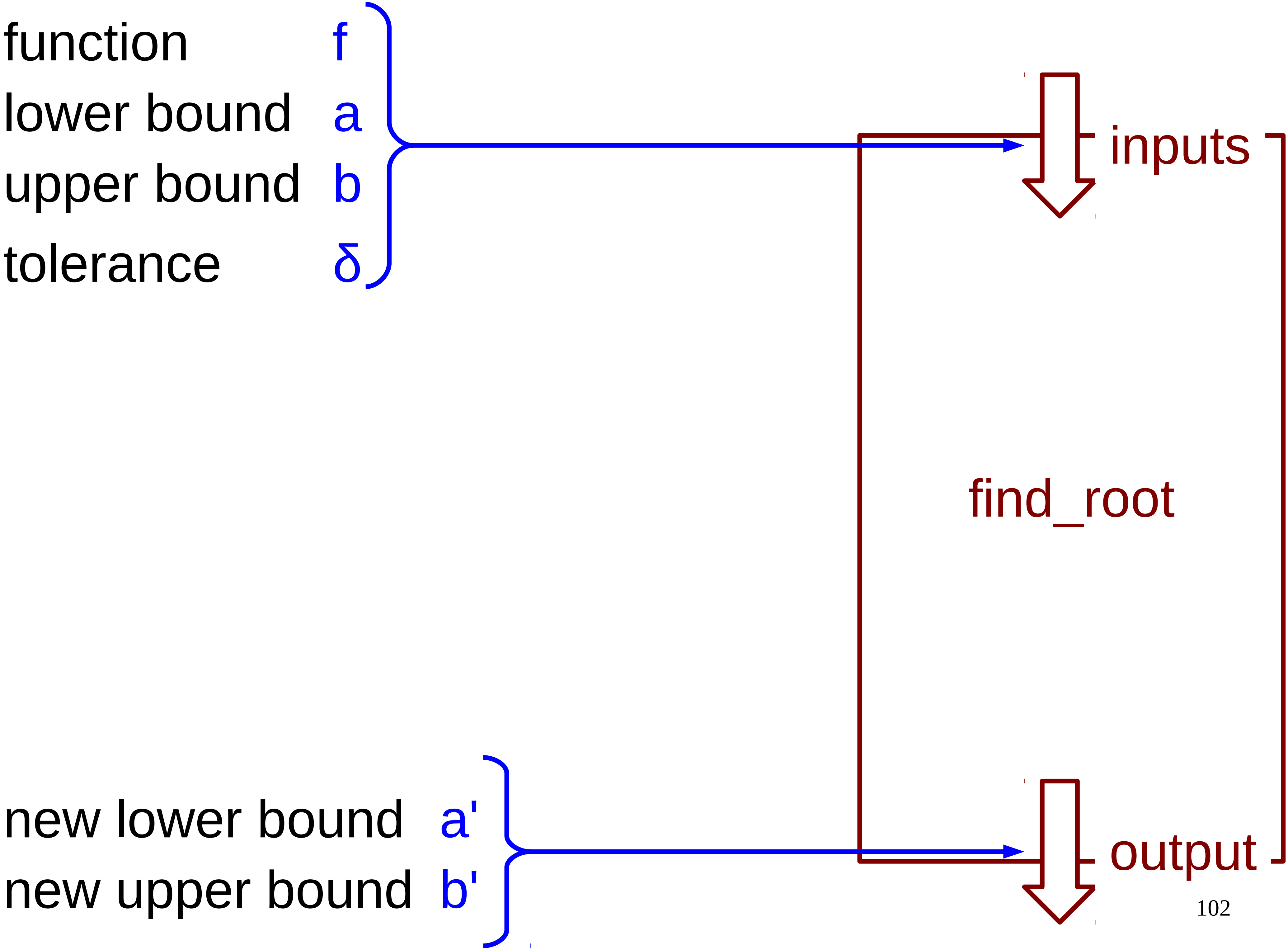
a, (a+b)/2

until b-a<δ

a, b

↓

(a+b)/2, b



function f

lower bound a

upper bound b

tolerance δ

inputs

find_root

output

new lower bound a'

new upper bound b'

Multiple values for input

```
def find_root(
    function,
    lower,
    upper,
    tolerance
    ):
    function body
```

pass in functions simply

comma separated

meaningful parameter names

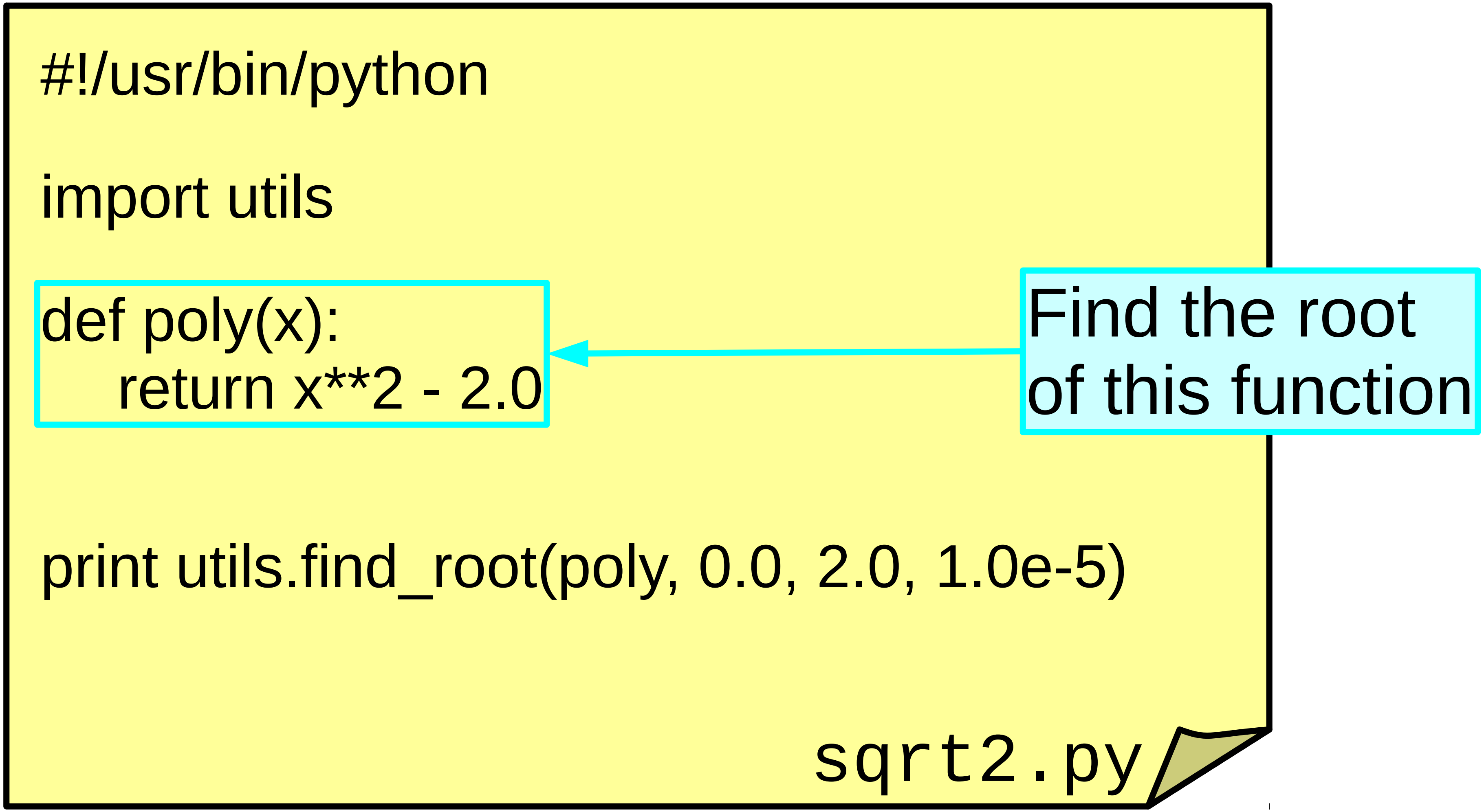
Multiple values for output

```
def find_root(
        …
        ):
        function body
        return (
            lower ,
            upper
            )
```

typically on a single line

```
def find_root(
    function,
    lower,
    upper,
    tolerance
    ):

    while upper - lower > tolerance:
        middle = (lower + upper) / 2.0
        if function(middle)*function(upper) > 0.0:
            upper = middle
        else:
            lower = middle

    return (lower, upper)
```

`utils.py`

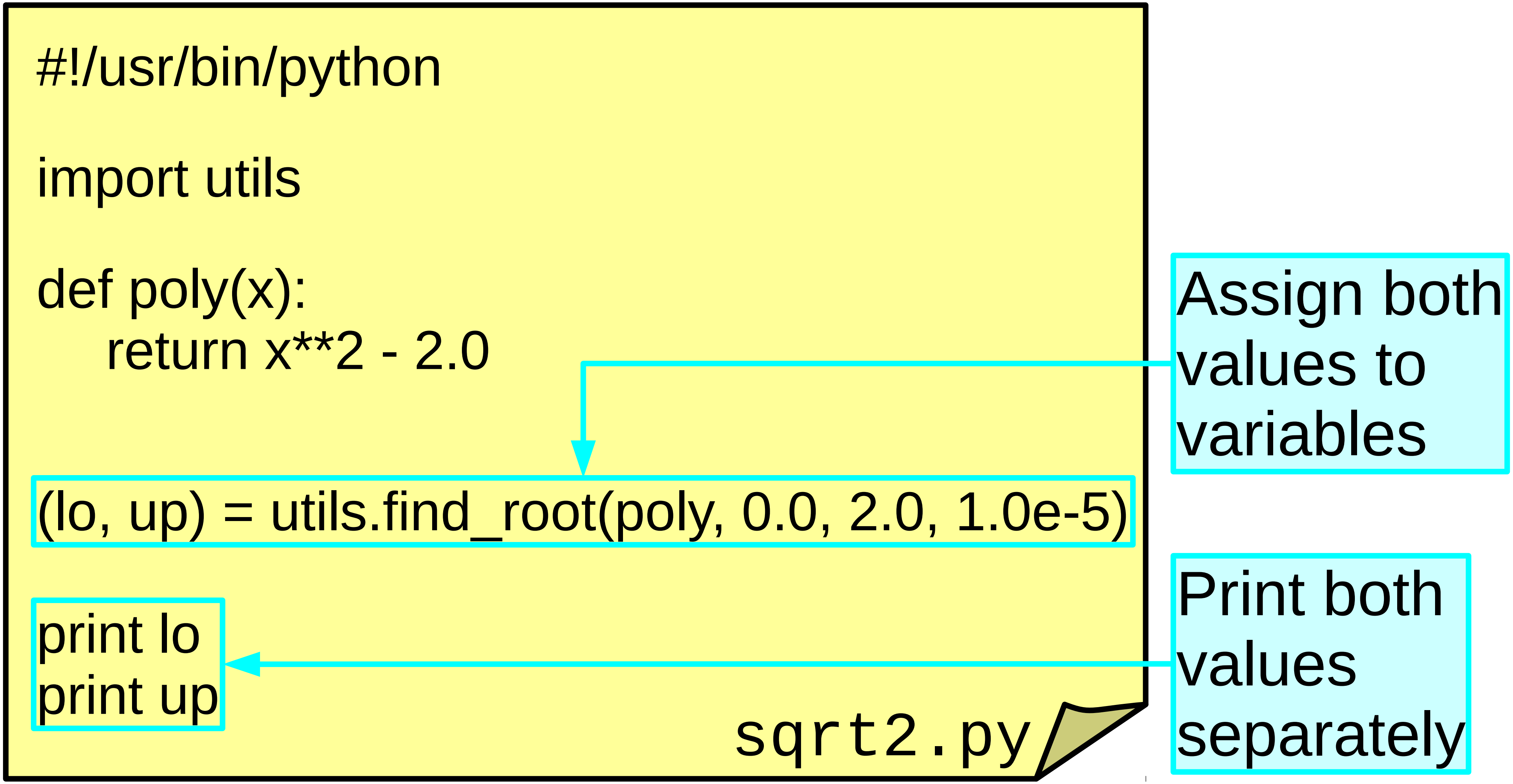


```
#!/usr/bin/python

import utils

def poly(x):
    return x**2 - 2.0


print utils.find_root(poly, 0.0, 2.0, 1.0e-5)
```

Find the root of this function

sqrt2.py

```
$ python sqrt2.py
(1.4142074584960938, 1.414215087890625)
```



```
#!/usr/bin/python

import utils

def poly(x):
    return x**2 - 2.0


(lo, up) = utils.find_root(poly, 0.0, 2.0, 1.0e-5)

print lo
print up
```

sqrt2.py

Assign both values to variables

Print both values separately

```
$ python sqrt2.py
1.4142074585
1.41421508789
```

Let’s break for an exercise…

Write a function that takes a list of numbers as input, and returns the following:

- smallest number in list
- arithmetic mean of list
- largest number in list

If you run into problems with this exercise, ask the course giver or a demonstrator for help.

Answer

```
def stats(numbers):

  min = numbers[0]
  max = numbers[0]
  total = 0

  for number in numbers:
     if number < min:
        min = number
     if number > max:
        max = number
     total = total + number

  return (min,
          total/(len(numbers)+0.0),
          max)
```

utils.py

n.b. Function *fails* if the list is empty.

Tuples

Singles

Doubles

Triples

Quadruples

Quintets

(42 , 1.95 , 'Bob')

(-1 , +1)

('Intro. to Python', 25, 'TTR1')

“not the same as lists”

Tuples are not the same as lists

(minimum, maximum)

(age, name, height)

(age, height, name)

(age, height, name, weight)

Independent, grouped items

Related, sequential items

[2, 3, 5, 7]

[2, 3, 5, 7, 11]

[2, 3, 5, 7, 11, 13]

Access to components

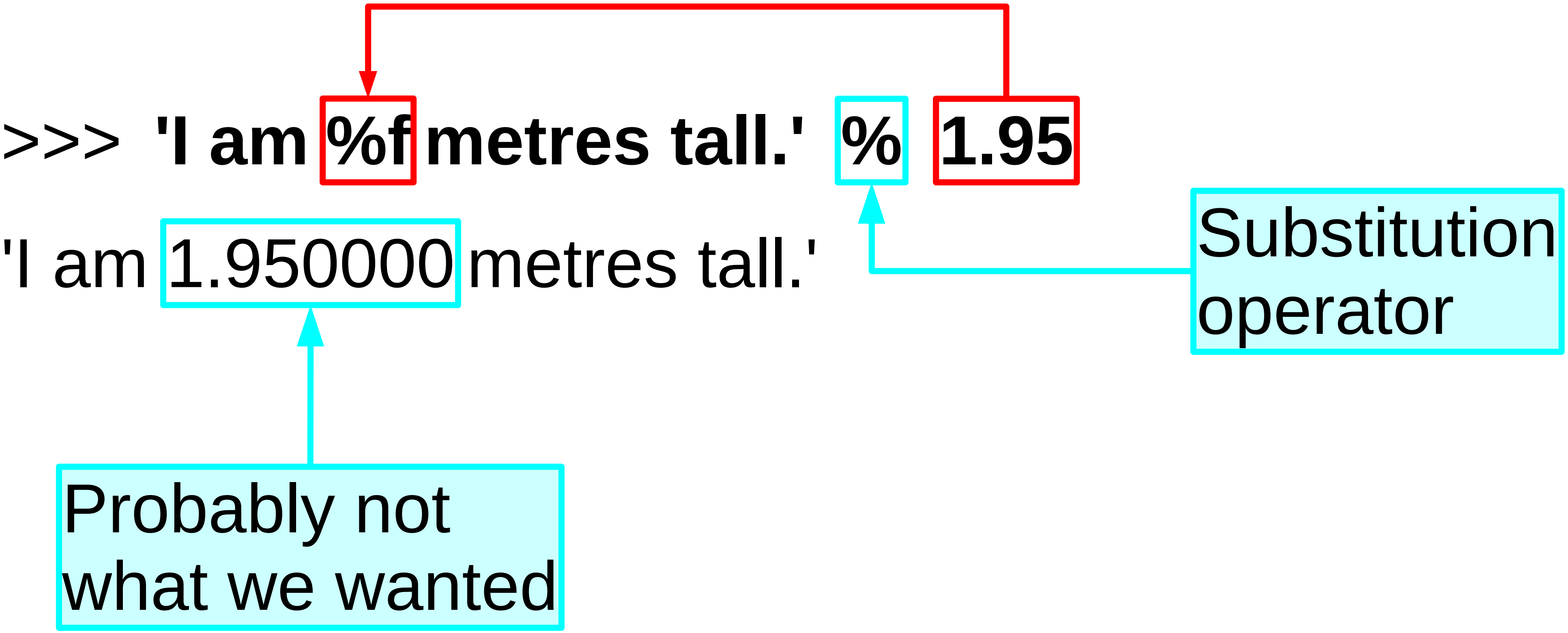
Same access syntax as for lists:

```
>>> ('Bob', 42, 1.95)[0]
'Bob'
```

But tuples are *immutable*:

```
>>> ('Bob', 42, 1.95)[1]   =   43
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support
item assignment
```

String substitution

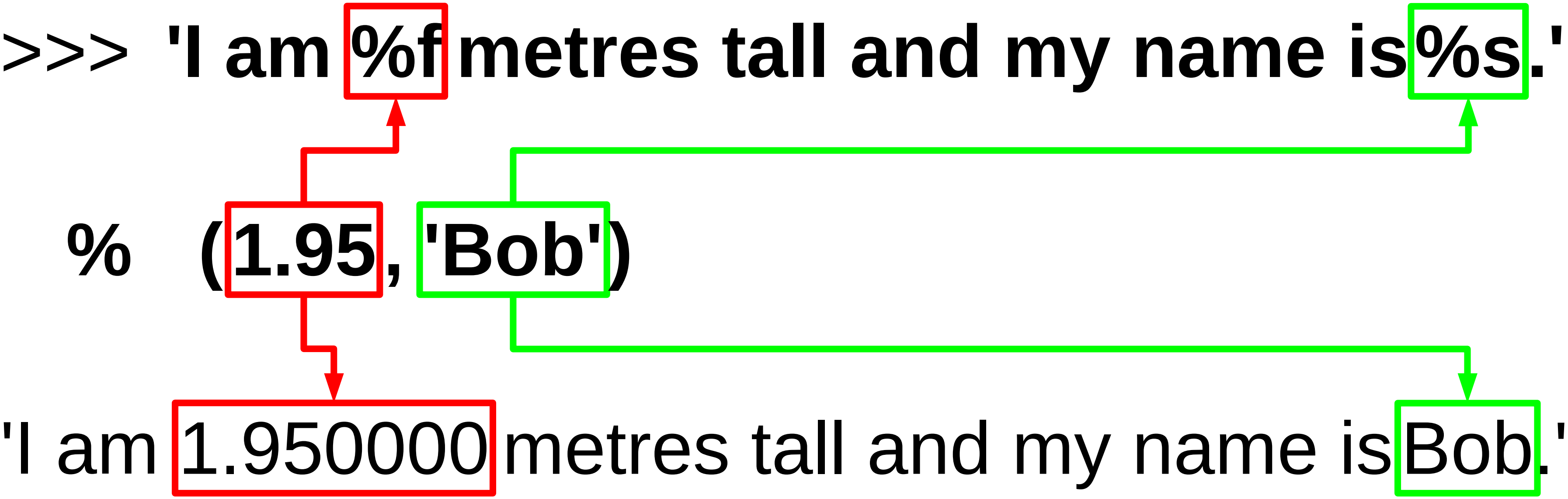


```
>>> 'I am %f metres tall.' % 1.95
'I am 1.950000 metres tall.'
```

Substitution operator

Probably not what we wanted

Substituting multiple values



```
>>> 'I am %f metres tall and my name is %s.'
  %  (1.95, 'Bob')
'I am 1.950000 metres tall and my name is Bob.'
```

Formatted substitution

standard float marker

```
>>> '%f'    %    0.23
'0.230000'
```

six decimal places

modified float marker: “.3”

```
>>> '%.3f'  %    0.23
'0.230'
```

three decimal places

More complex formatting possible

```
'23'  '23.4567'  '  23.46'
'  23' '23.456700''23.46  '
'0023' '23.46'    ' +23.46'
' +23' '+23.4567' '+23.46 '
'+023' '+23.456700'
'23  ' '+23.46'   'Bob'
'+23 ' '0023.46'  'Bob  '
      '+023.46'  '  Bob'
```

Uses of tuples

1. Functions

2. Related data

3. String substitution

```
#!/usr/bin/python

# The keys of this dictionary are the
#  symbols for the atomic elements.
# The values are tuples:
#  (name, atomic number, boiling point).
chemicals = {…}

# For each key in the chemicals
#  dictionary, print the name and
#  boiling point (to 1 decimal place),
#  e.g. for the key 'H', print:
#          hydrogen: 20.3K
```

What goes here?

chemicals3.py

Answer

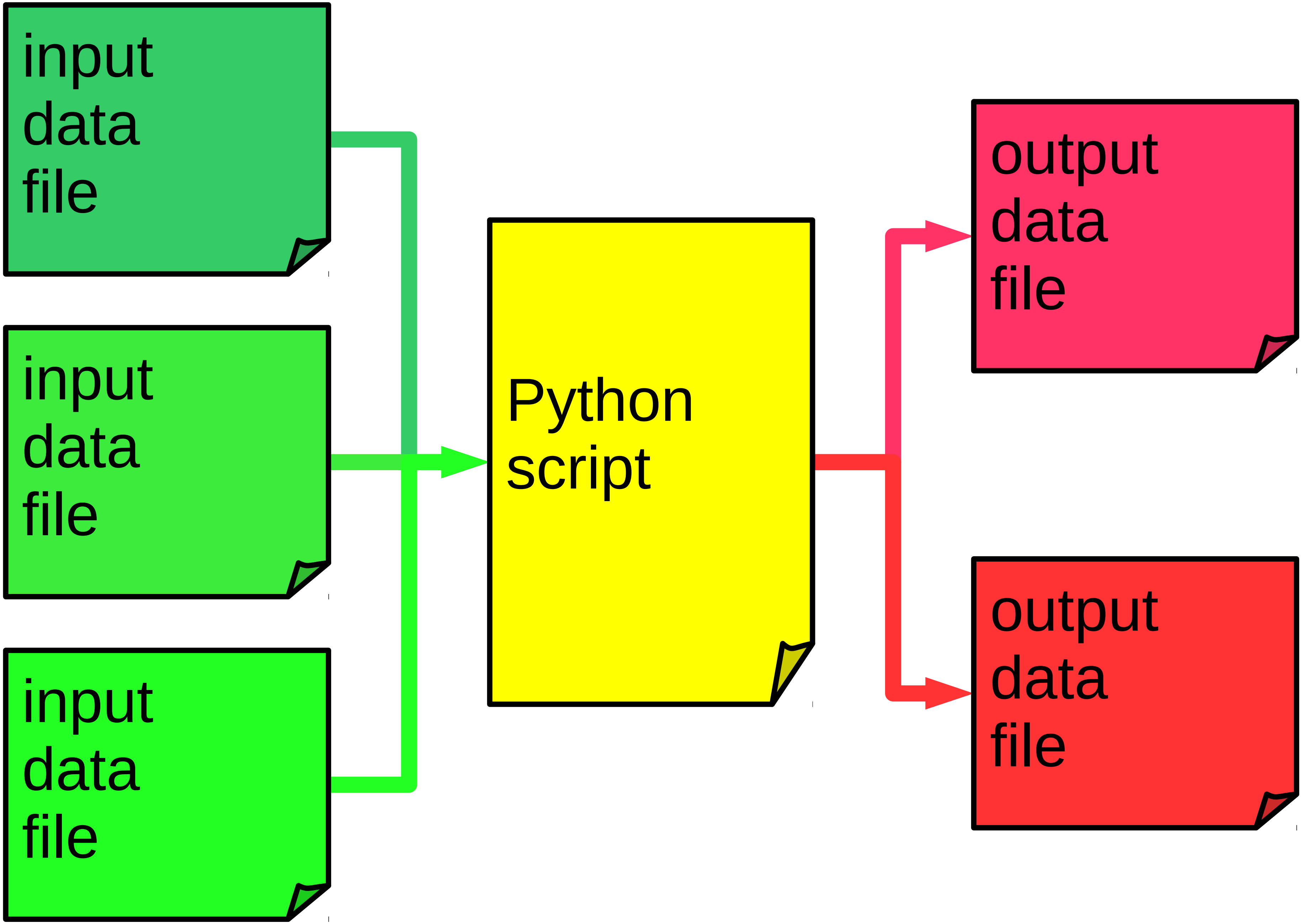
```
#!/usr/bin/python

# The keys of this dictionary are the
#  symbols for the atomic elements.
# The values are tuples:
#  (name, atomic number, boiling point).
chemicals = {…}

# For each key in the chemicals
#  dictionary, print the name and
#  boiling point (to 1 decimal place),
#  e.g. for the key 'H', print:
#          hydrogen: 20.3K
for symbol in chemicals:
    (name, number, boil) = chemicals[symbol]
    print "%s: %.1fK" % (name, boil)
del name, number, boil
del symbol
```

chemicals3.py

Accessing the system

1. Files
2. Standard input & output
3. The command line

May want to access many files



input data file

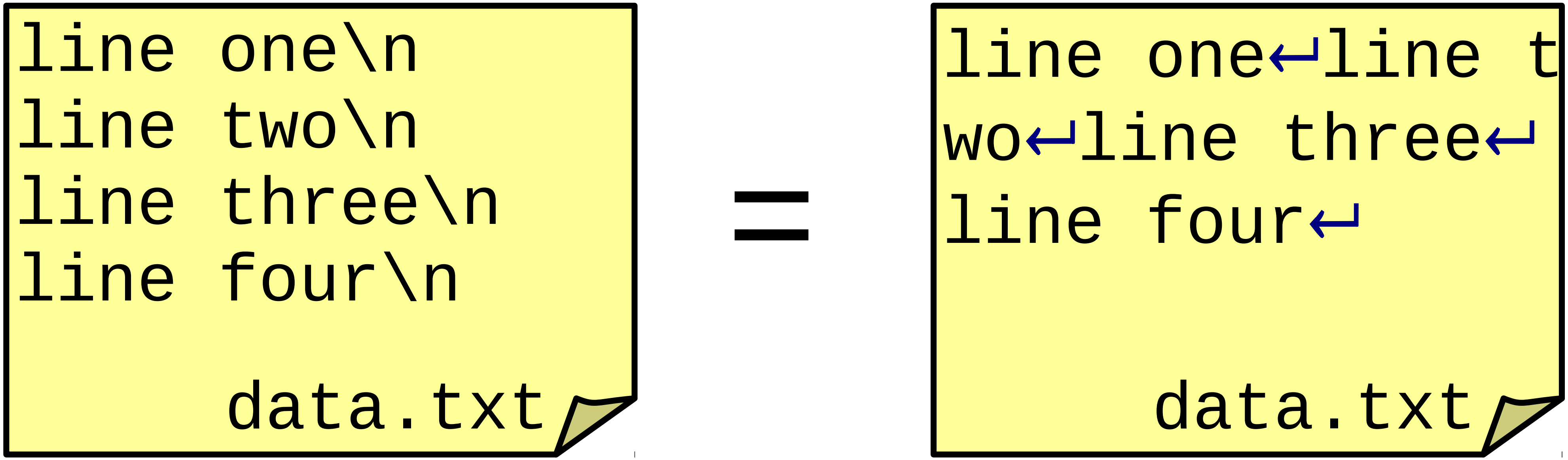
input data file

input data file

Python script

output data file

output data file



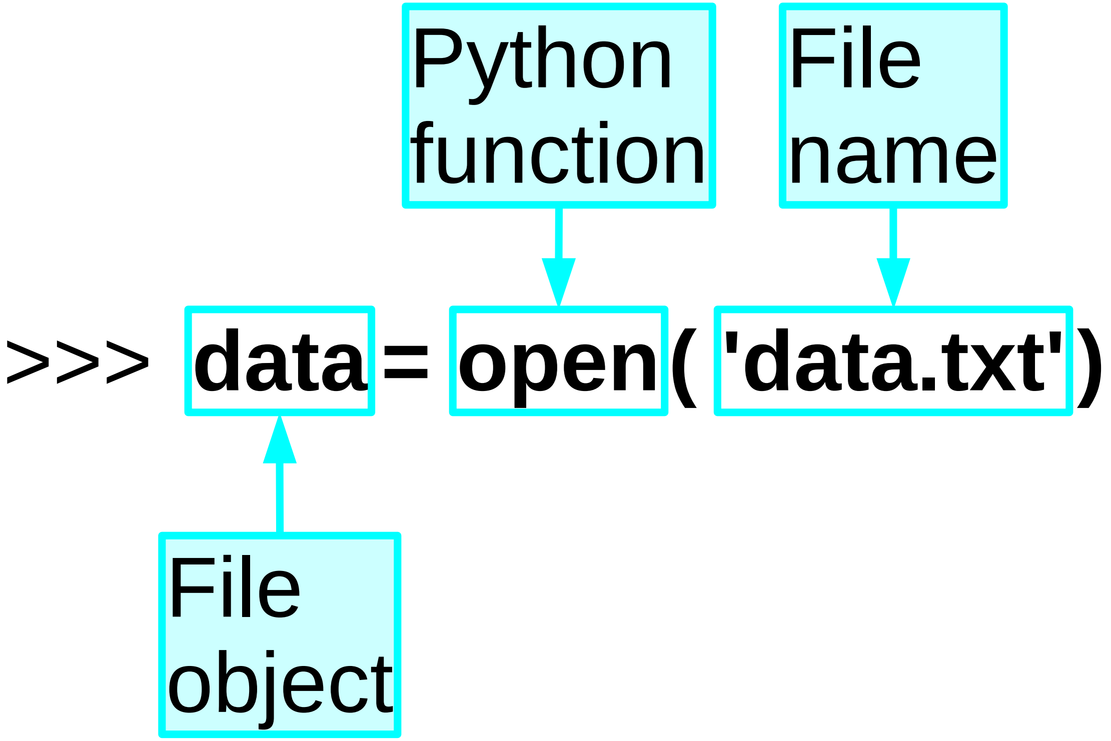
```
line one\n
line two\n
line three\n
line four\n

       data.txt
```

=

```
line one↵line t
wo↵line three↵
line four↵

       data.txt
```

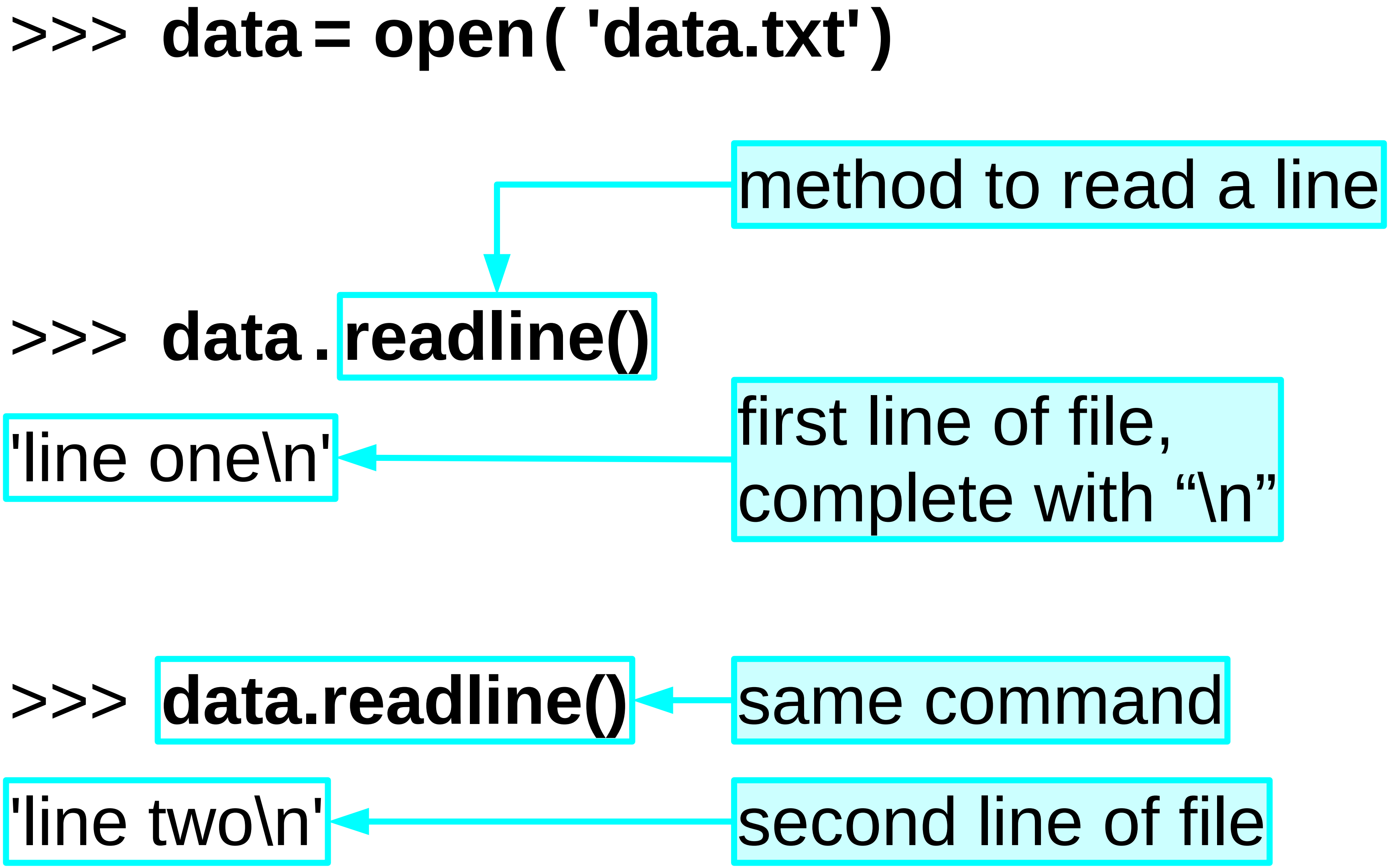
\n

↵



```
>>> data = open('data.txt')
```

All access to the file is via the file object



```
>>>  data = open( 'data.txt' )

>>>  data . readline()
'line one\n'

>>>  data.readline()
'line two\n'
```

method to read a line

first line of file, complete with “\n”

same command

second line of file

```
>>> data = open( 'data.txt' )
>>> data.readline()
'line one\n'
>>> data.readline()
'line two\n'

>>> data.readlines()
[ 'line three\n', 'line four\n' ]
```

remaining lines

```
>>> data = open( 'data.txt' )
>>> data.readline()
'line one\n'
>>> data.readline()
'line two\n'
>>> data.readlines()
[ 'line three\n', 'line four\n' ]


>>> data.close()
```

disconnect

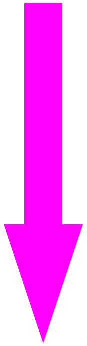
```
>>> del data
```

delete the variable

Treating file objects like lists:

for line in data.readlines():
    do stuff

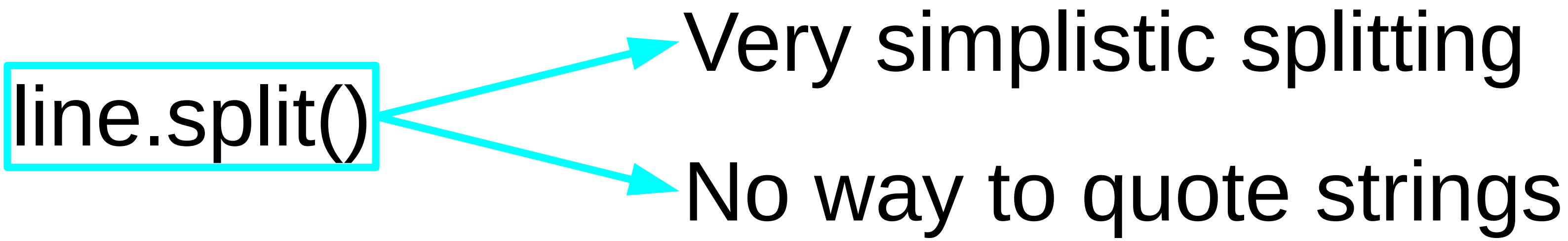
reads the lines all at once



for line in data:
    do stuff

reads the lines as needed

Very primitive input



line.split()

Very simplistic splitting

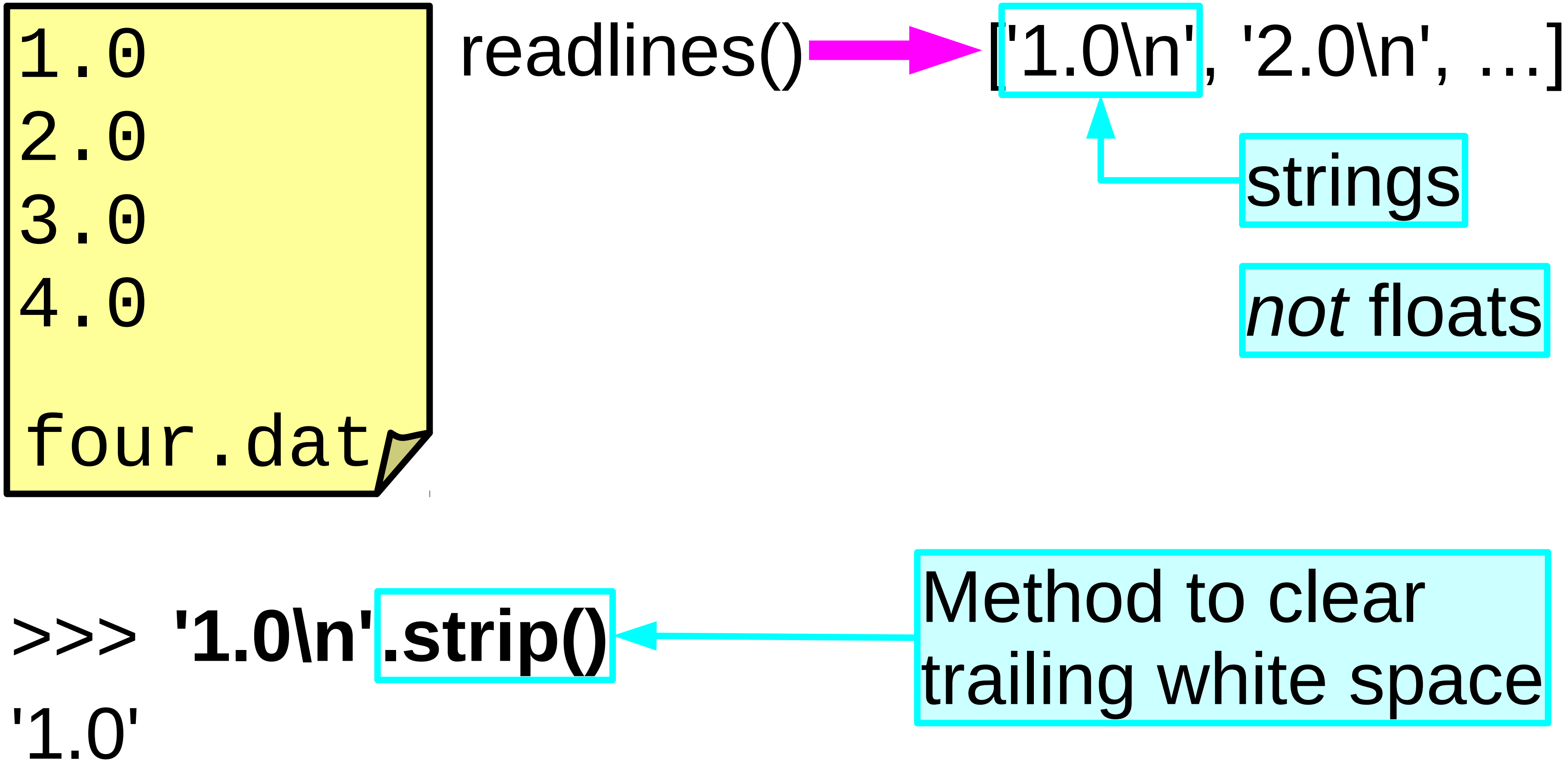
No way to quote strings

Comma separated values:	**csv**	module
Regular expressions:	**re**	module

“Python: Further Topics” course

“Python: Regular Expressions” course

Reading data gets you strings



```
1.0
2.0
3.0
4.0
```

four.dat

readlines() → ['1.0\n', '2.0\n', …]

strings

not floats

```
>>> '1.0\n'.strip()
'1.0'
```

Method to clear trailing white space

Still need to convert string to other types

Converting from one type to another

In and out of strings

```
>>> float('0.25')                 >>> str(0.25)
0.25                  ↔           '0.25'


>>> int('123')                    >>> str(123)
123                   ↔           '123'
```

Converting from one type to another

Between numeric types

```
>>> int(12.3)
12
```

loss of precision

```
>>> float(12)
12.0
```

Converting from one type to another

If you treat it like a list…

```
>>> list('abcd')
['a', 'b', 'c', 'd']

>>> list(data)
['line one\n', 'line two\n', 'line three\n', 'line four\n']

>>> list({'H':'hydrogen', 'He':'helium'})
['H', 'He']
```

```
#!/usr/bin/python

# This script reads in some
#  numbers from the file 'numbers.txt'.
# It then prints out the smallest
#  number, the arithmetic mean of
#  the numbers, and the largest
#  number.
```

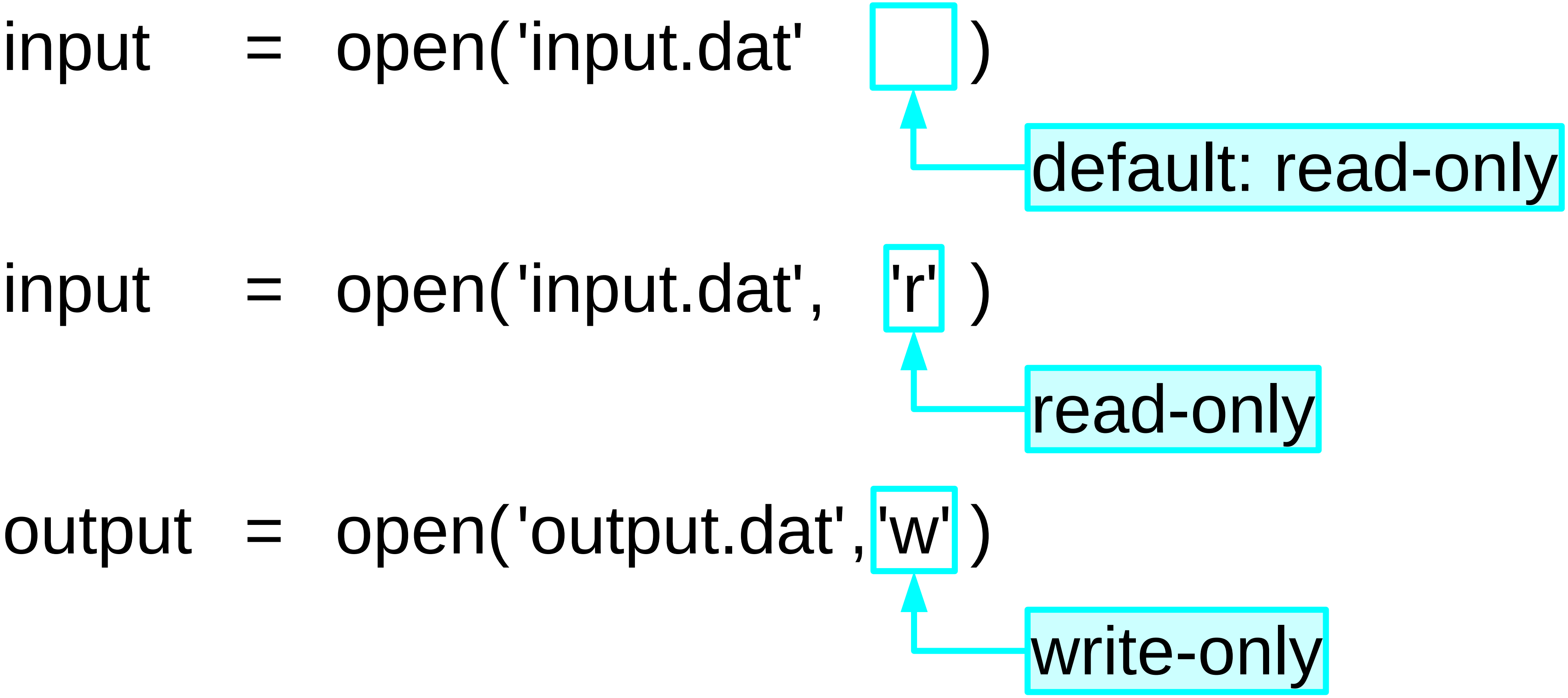
What goes here? (Use the function you wrote in an earlier exercise.)

Answer

```
#!/usr/bin/python

# This script reads in some
#  numbers from the file 'numbers.txt'.
# It then prints out the smallest
#  number, the arithmetic mean of
#  the numbers, and the largest
#  number.

import utils

data = open('numbers.txt')

numbers = []
for line in data:
        numbers.append(float(line))
del line

data.close()
del data
print utils.stats(numbers)
```

function you wrote in earlier exercise

Output to files



```
input   =  open('input.dat'      )
input   =  open('input.dat',  'r' )
output  =  open('output.dat','w' )
```

default: read-only

read-only

write-only

Output to files

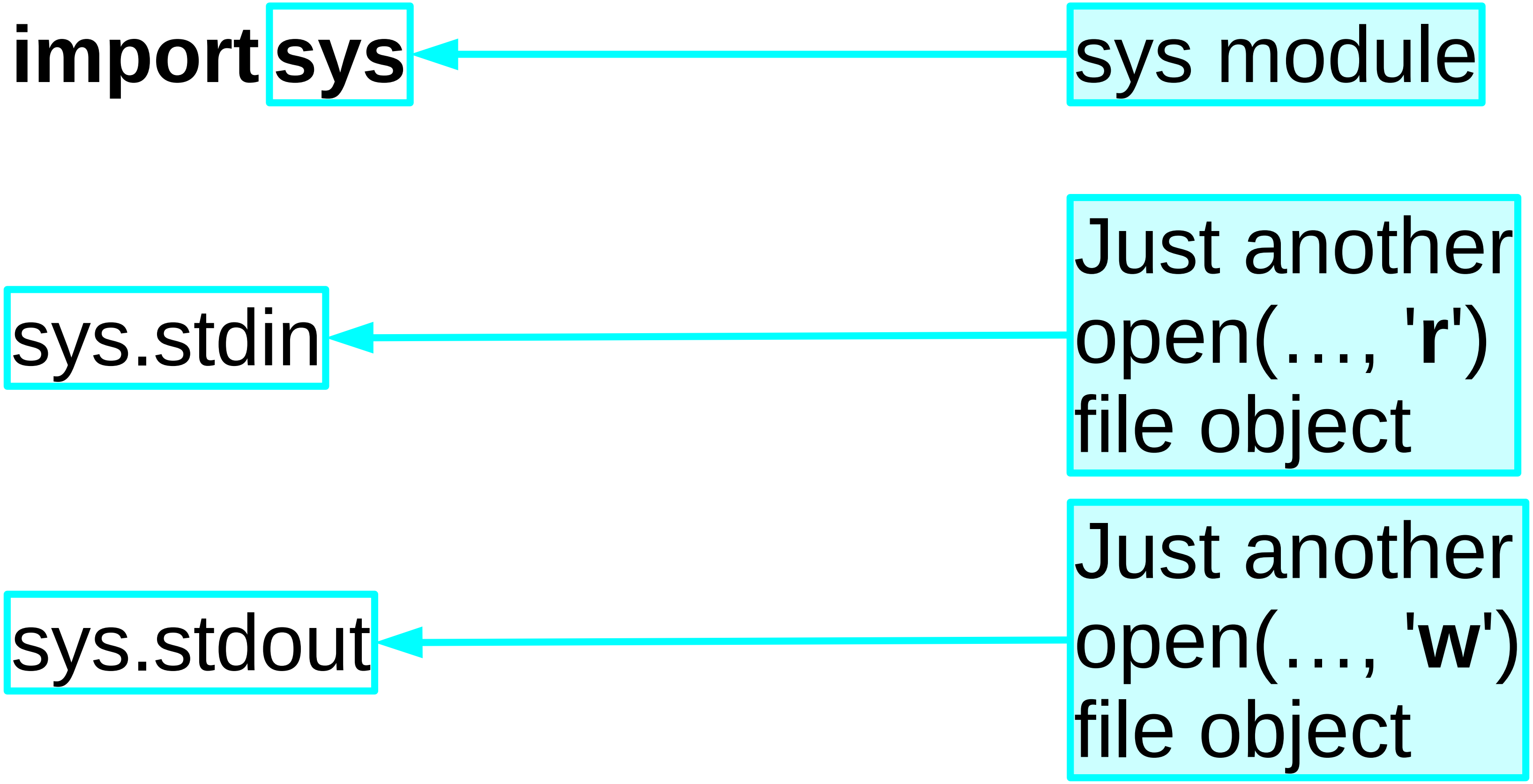
```
>>>  output = open('output.dat', 'w')
>>>  output.write('alpha\n')
>>>  output.write('bet')
>>>  output.write('a\n' )
>>>  output.writelines([ 'gamma\n', 'delta\n'])
>>>  output.close()
```

explicit “\n”

write(): writes *lumps* of data

Flushes to file system

Standard input and output



import sys

sys module

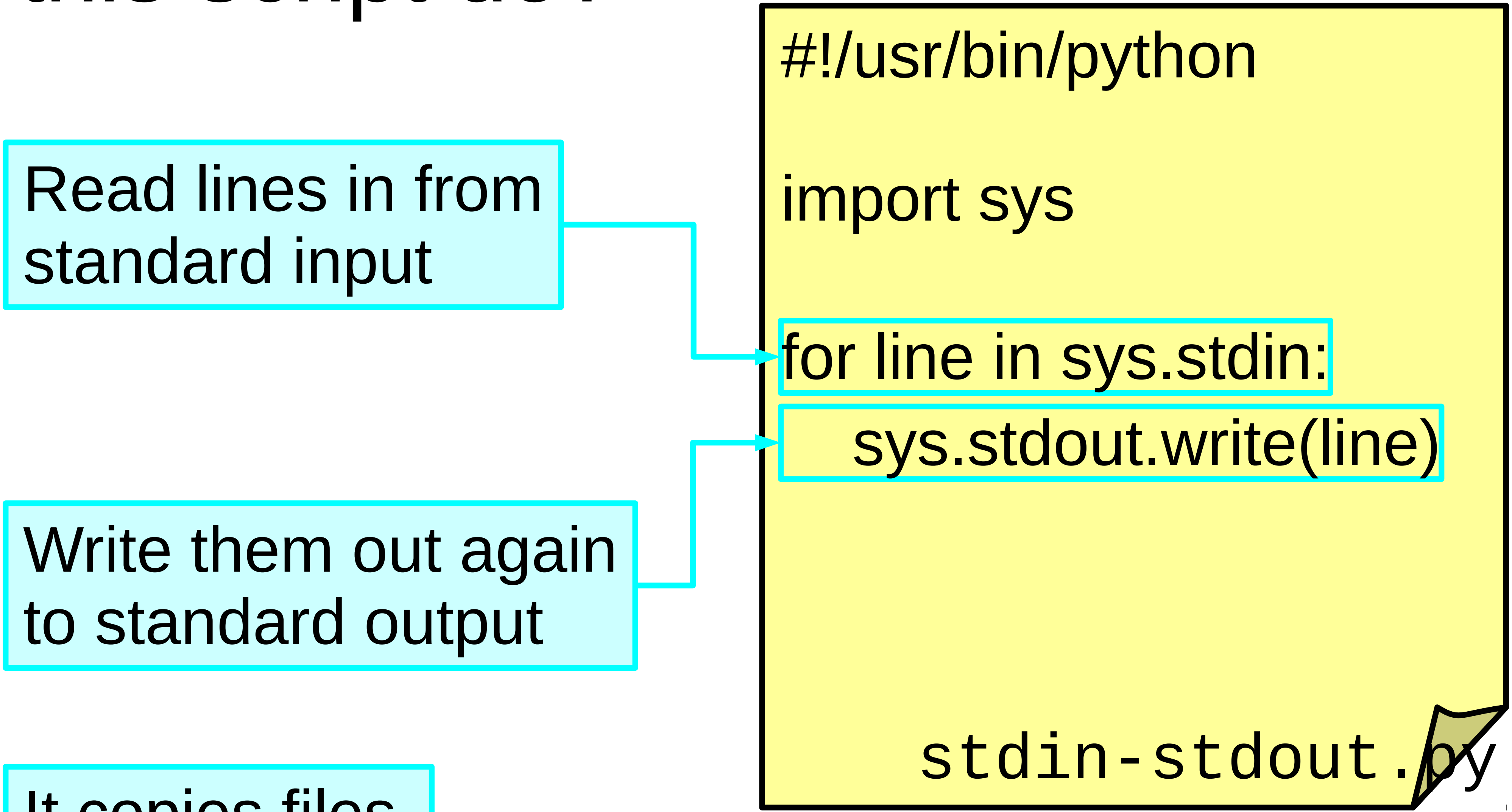
sys.stdin

Just another open(…, '**r**') file object

sys.stdout

Just another open(…, '**w**') file object

So, what does this script do?



Read lines in from standard input

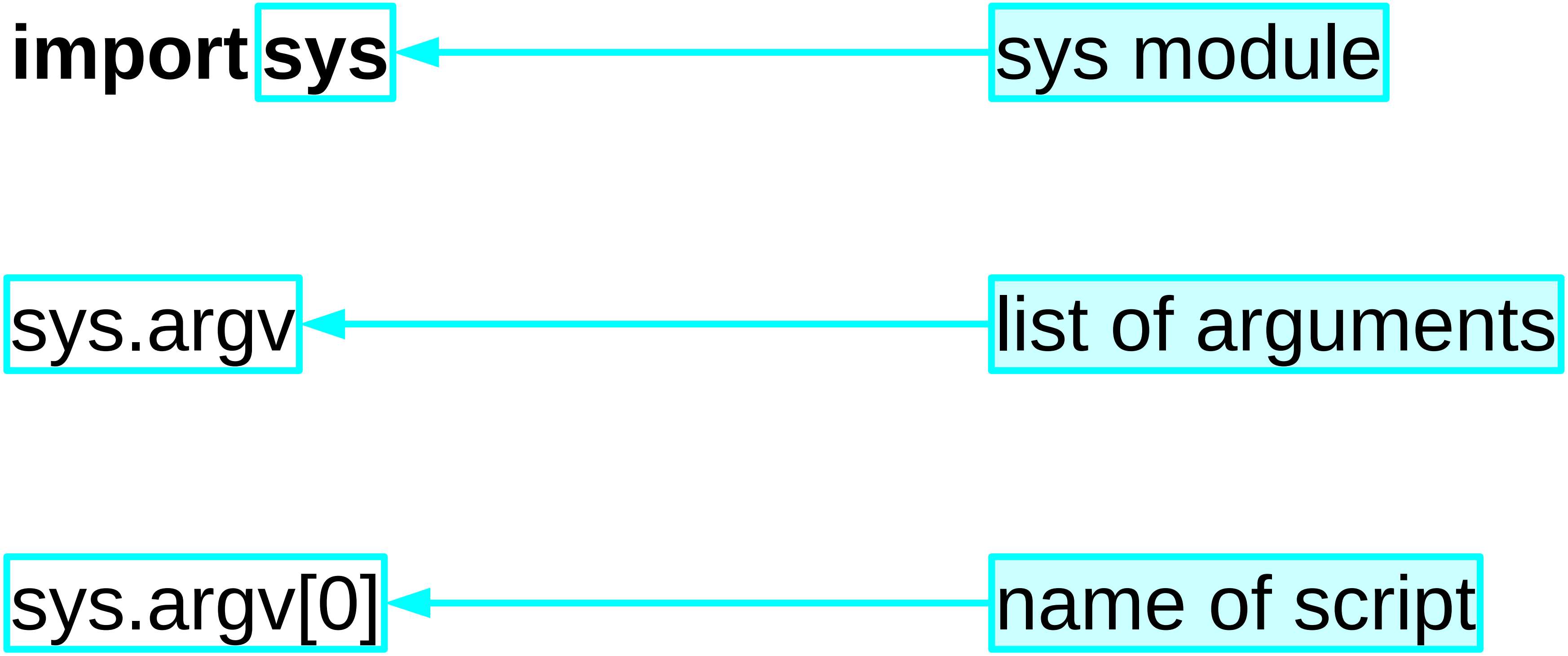
Write them out again to standard output

```
#!/usr/bin/python

import sys

for line in sys.stdin:
    sys.stdout.write(line)
```

stdin-stdout.py

It copies files, line by line

Command line



import sys

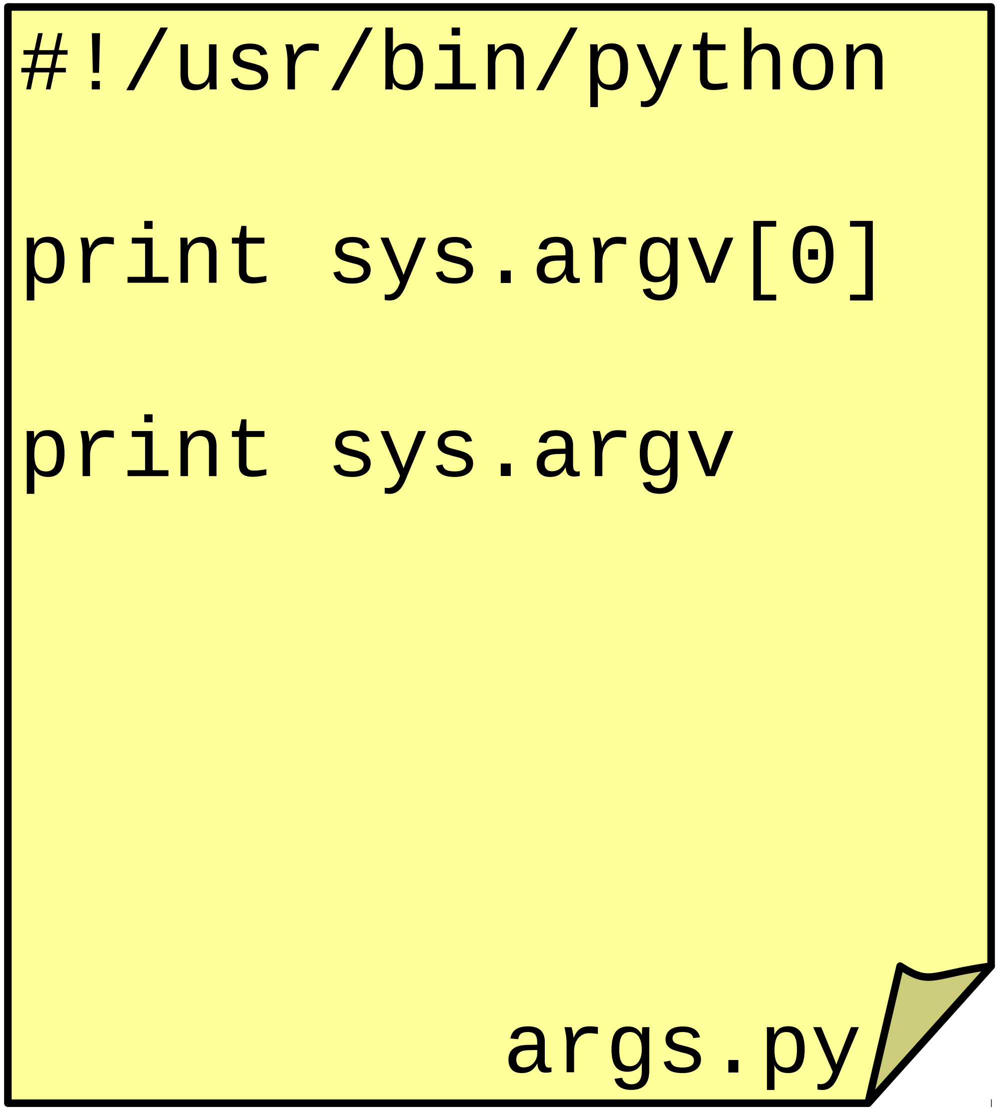
sys module

sys.argv

list of arguments

sys.argv[0]

name of script



```
#!/usr/bin/python

print sys.argv[0]

print sys.argv
```

args.py

```
$ python args.py 0.25 10

args.py

['args.py', '0.25', '10']
```

NB: list of *strings*

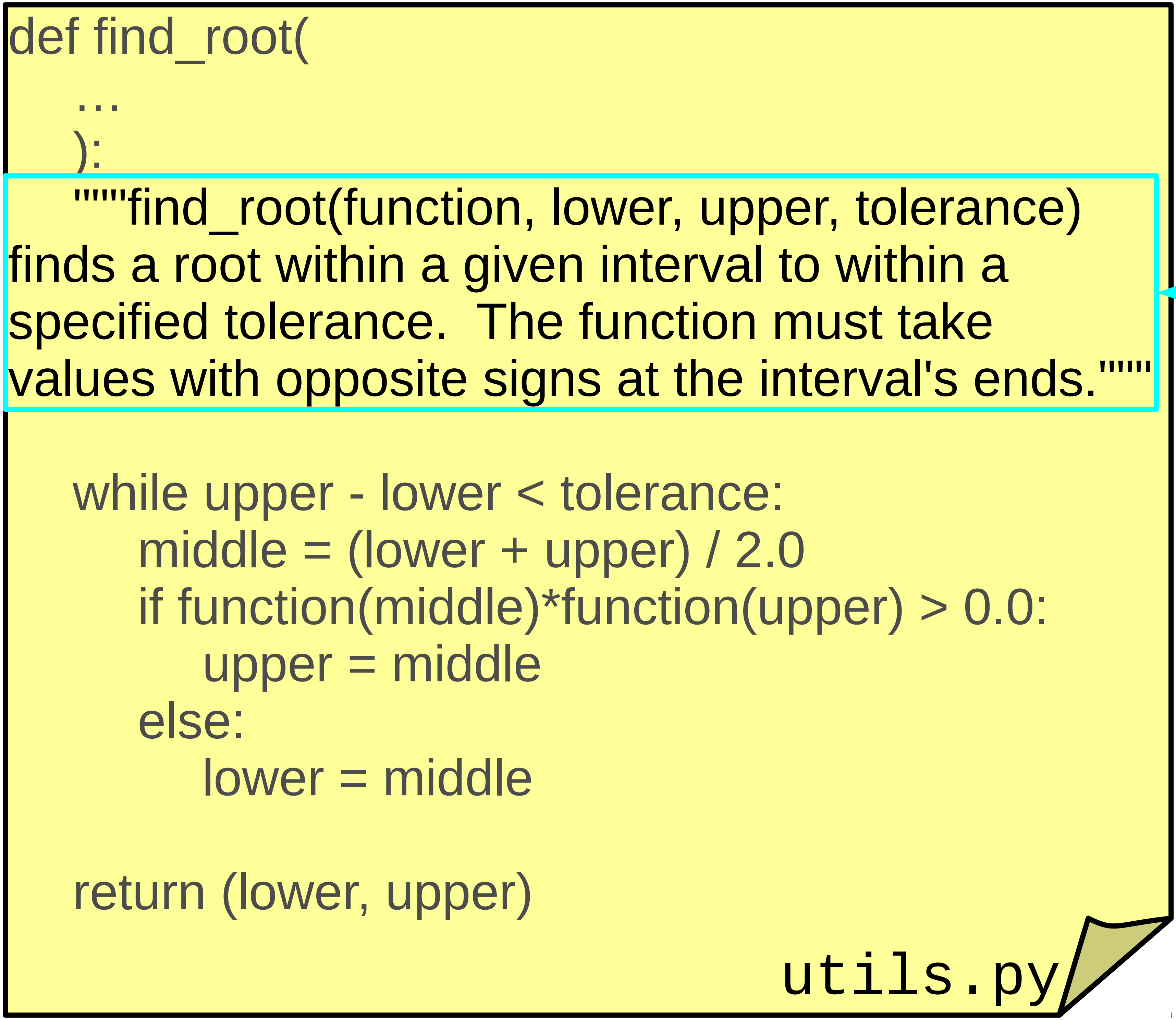
```
#!/usr/bin/python

# This script takes some numbers as
#  arguments on the command line.
# It then prints out the smallest
#  number, the arithmetic mean of
#  the numbers, and the largest
#  number.
```

What goes here?

Answer

```
#!/usr/bin/python

# This script takes some numbers as
#  arguments on the command line.
# It then prints out the smallest
#  number, the arithmetic mean of
#  the numbers, and the largest
#  number.

import sys
import utils

numbers=[]
for arg in sys.argv[1:]:
        numbers.append(float(arg))
del arg
print utils.stats(numbers)
```

function you wrote earlier



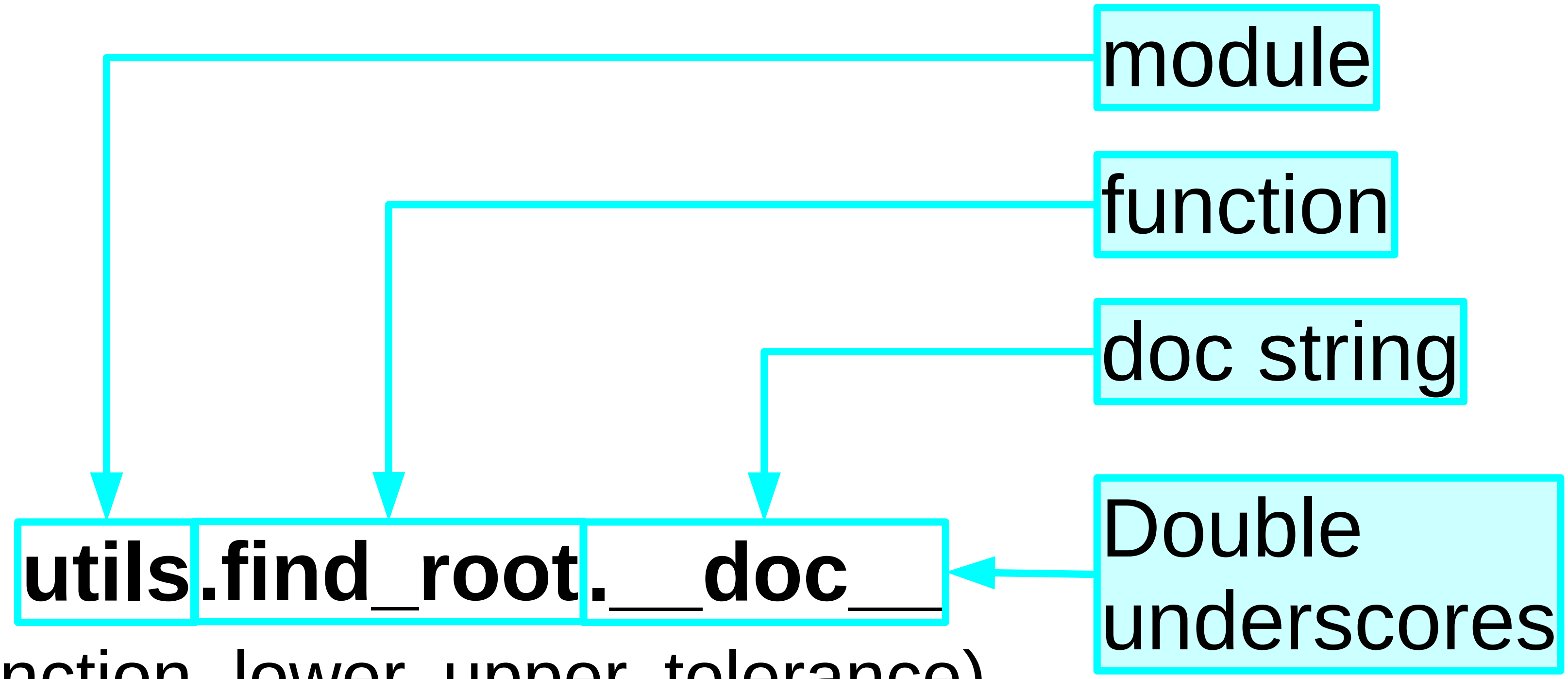
```
def find_root(
    …
    ):
    """find_root(function, lower, upper, tolerance)
finds a root within a given interval to within a
specified tolerance.  The function must take
values with opposite signs at the interval's ends."""

    while upper - lower < tolerance:
        middle = (lower + upper) / 2.0
        if function(middle)*function(upper) > 0.0:
            upper = middle
        else:
            lower = middle

    return (lower, upper)
```

utils.py

Inserted string

Doc strings for functions

```
>>>  import utils
```



module

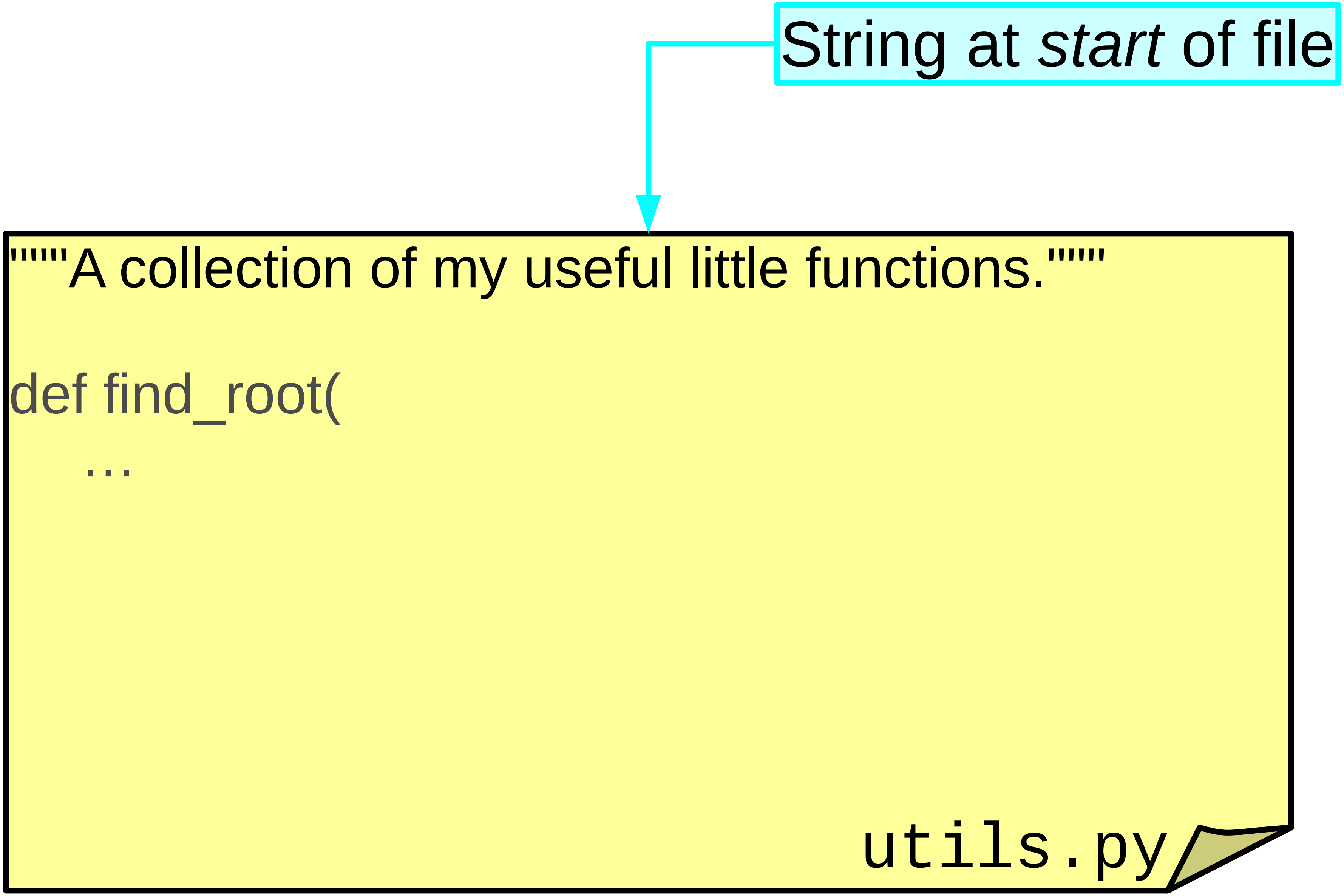
function

doc string

Double underscores

```
>>>  print utils.find_root.__doc__
find_root(function, lower, upper, tolerance)
finds a root within a given interval to within a
specified tolerance.  The function must take
values with opposite signs at the interval's ends.
```

Doc strings for modules



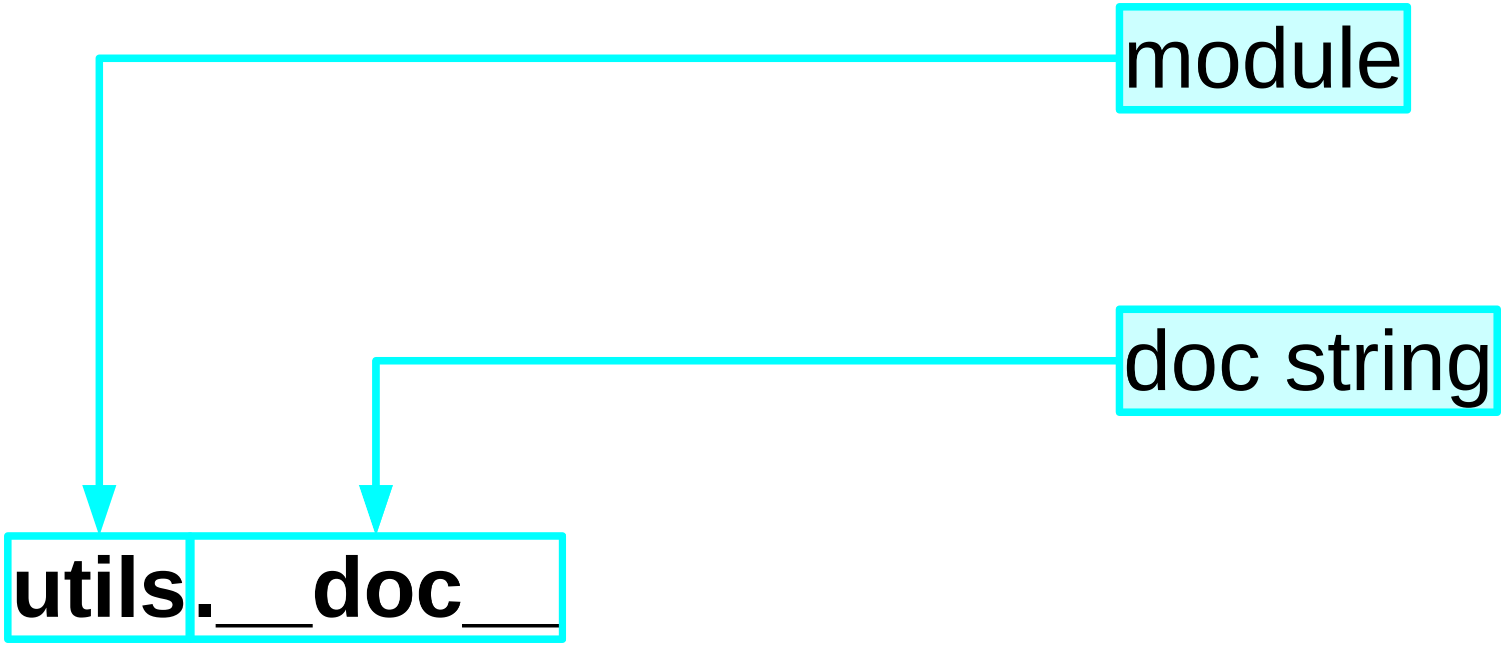
String at *start* of file

```
"""A collection of my useful little functions."""

def find_root(
    …
```

utils.py

Doc strings for modules

```
>>>  import utils
```



module

doc string

```
>>>  print utils.__doc__
A collection of my useful little functions.
```

Final exercise

Write the rest of this script.

```
#!/usr/bin/python

# This script takes some atomic symbols
#  on the command line.  For each symbol,
#  it prints the atomic element's name, and
#  boiling point (to 2 decimal places),
#  e.g. for the symbol 'H', print:
#  hydrogen has a boiling point of 20.3K
# Finally, it tells you which of the given
#  atomic elements has the lowest atomic
#  number.

# The keys of this dictionary are the
#  symbols for the atomic elements.
# The values are tuples:
#  (name, atomic number, boiling point).
chemicals = {…}
```

chemicals4.py

References and further courses

Dive Into Python
Mark Pilgrim
Apress
ISBN: 1-59059-356-1

http://diveintopython.org/

Best book on Python your course presenter has found. (It was written for Python 2.3, though. Luckily, Python 2.4, 2.5 and 2.6 are very similar to Python 2.3.)

Official Python documentation: http://docs.python.org/

“**Python: Further Topics**” is the follow-on course from this one. For details of this and other University Computing Service courses on Python, see:

http://training.csx.cam.ac.uk/theme/scicomp?scheduled=all