HEP Computing
Part II
Python

Marcella Bona

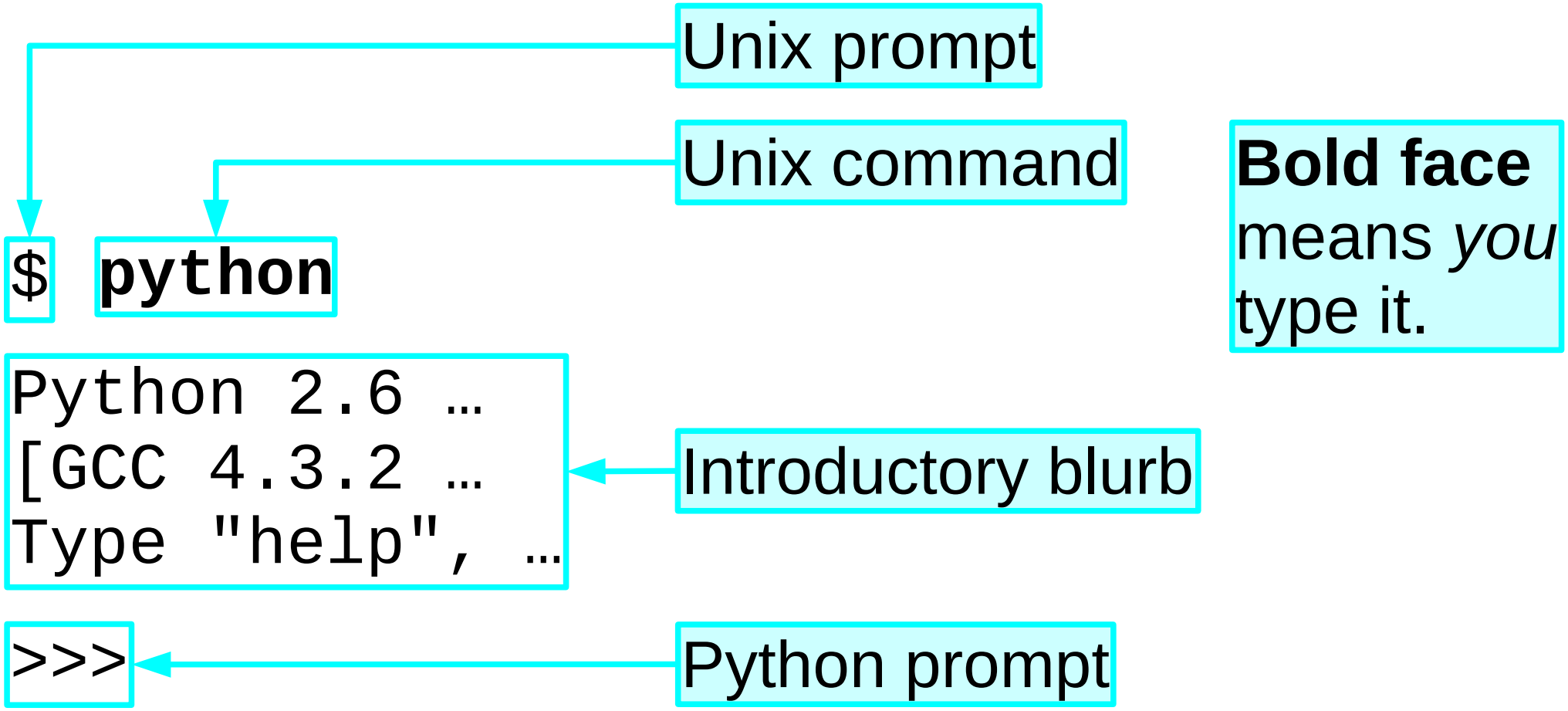
Lectures 3-4

© Bob Dowling

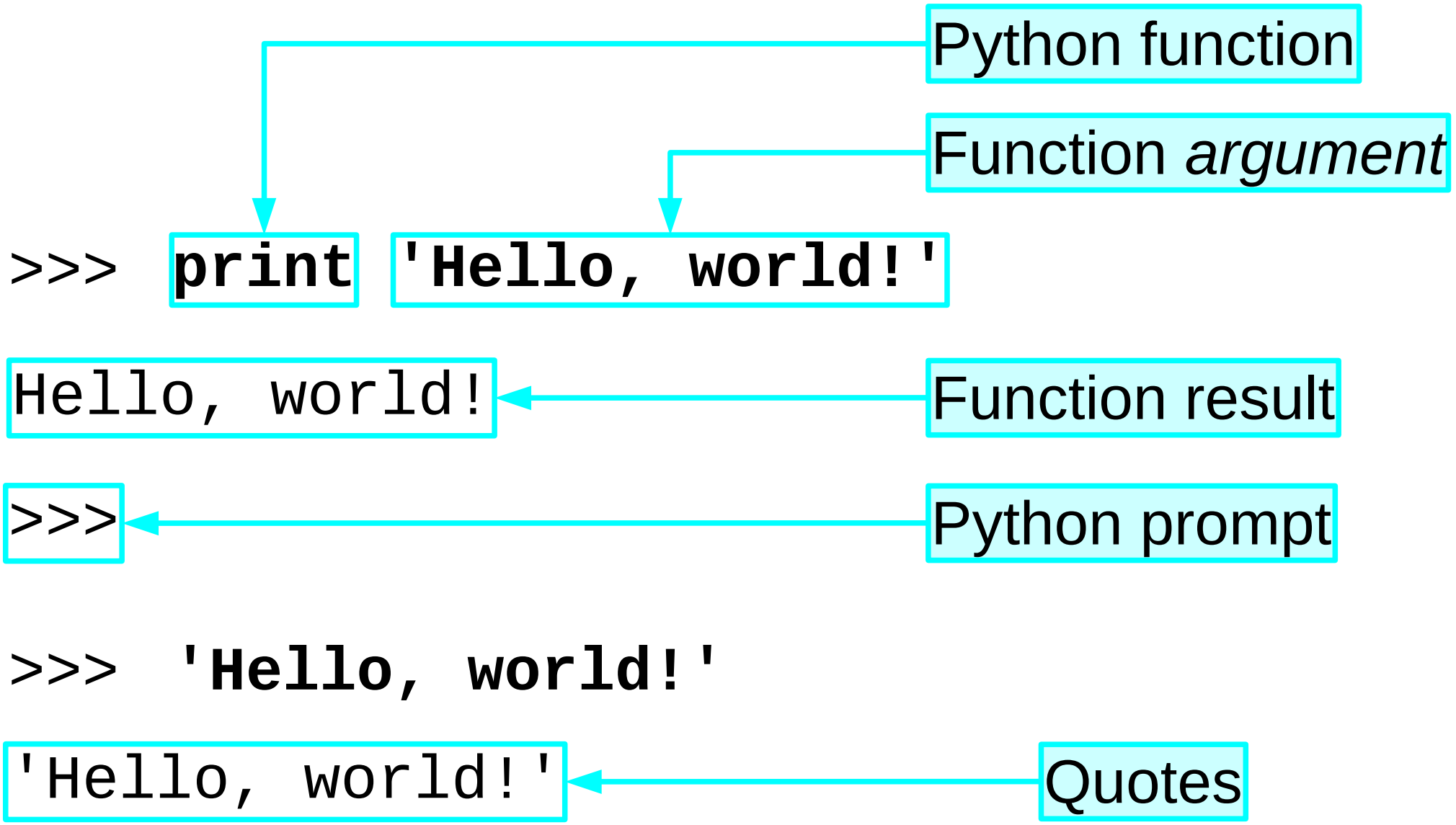
Queen Mary University of London

m.bona@qmul.ac.uk

Launching Python interactively



Using Python interactively



Using Python interactively

```
>>> print(3)
```
Instruct Python to print a 3

```
3
```
Python prints a 3

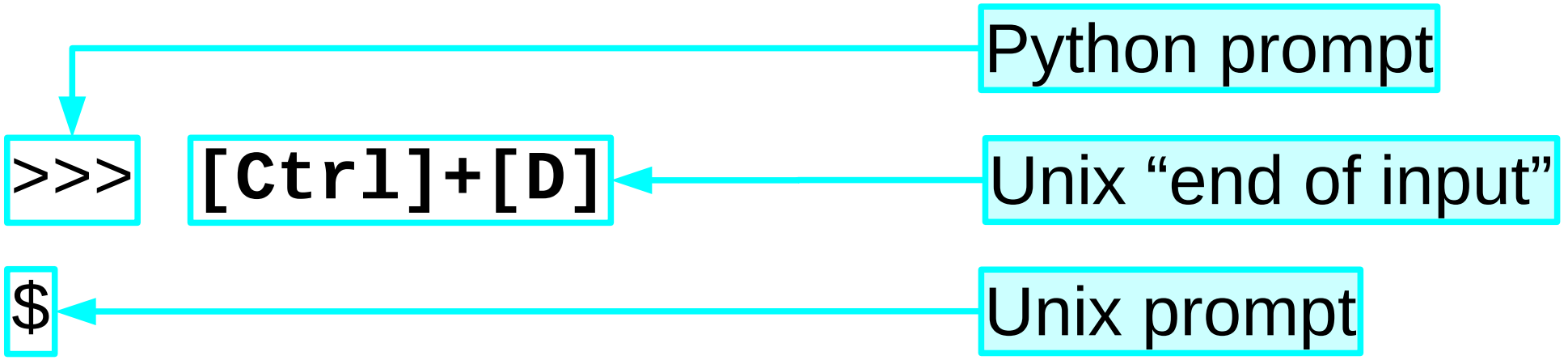
```
>>> 5
```
Give Python a literal 5

```
5
```
Python evaluates and displays a 5

```
>>> 2 + 3
```
Give Python an equivalent to 5

```
5
```
Python evaluates and displays a 5

Quitting Python interactively



1. Launch a terminal window.
2. Launch Python.
4. Run these Python expressions (one per line):
   (b) 26+18
   (c) 26<18
   (d) 26>18
5. Exit Python (but not the terminal window).

Launching Python scripts

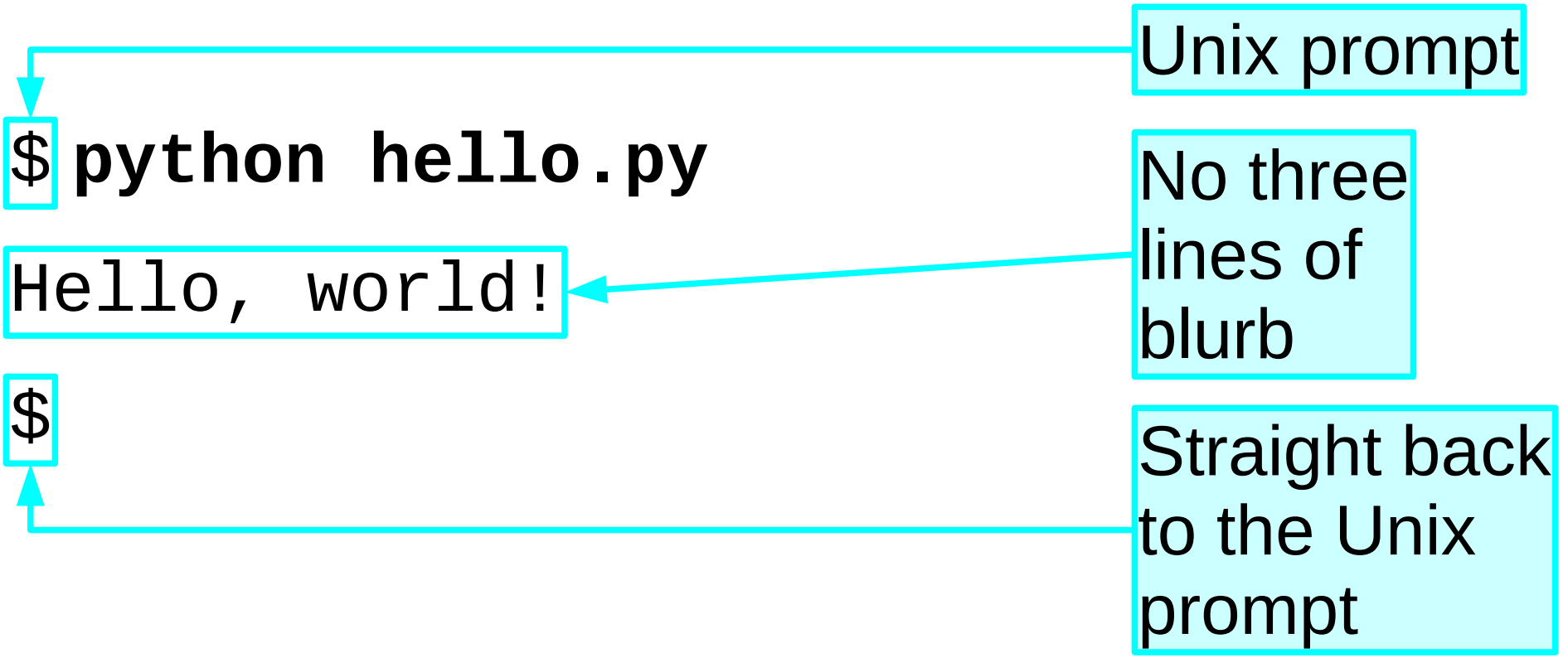
Read / edit the script

Run the script

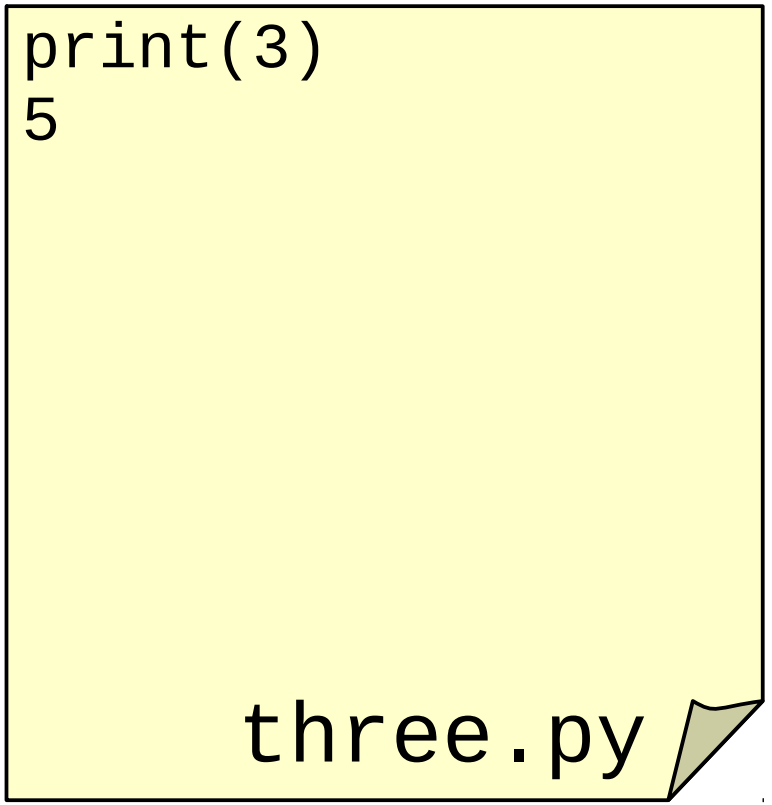
Launching Python scripts

Edit a file with one single line:

```
$ emacs -nw hello.py
        print('Hello, world!')
```



Launching Python scripts



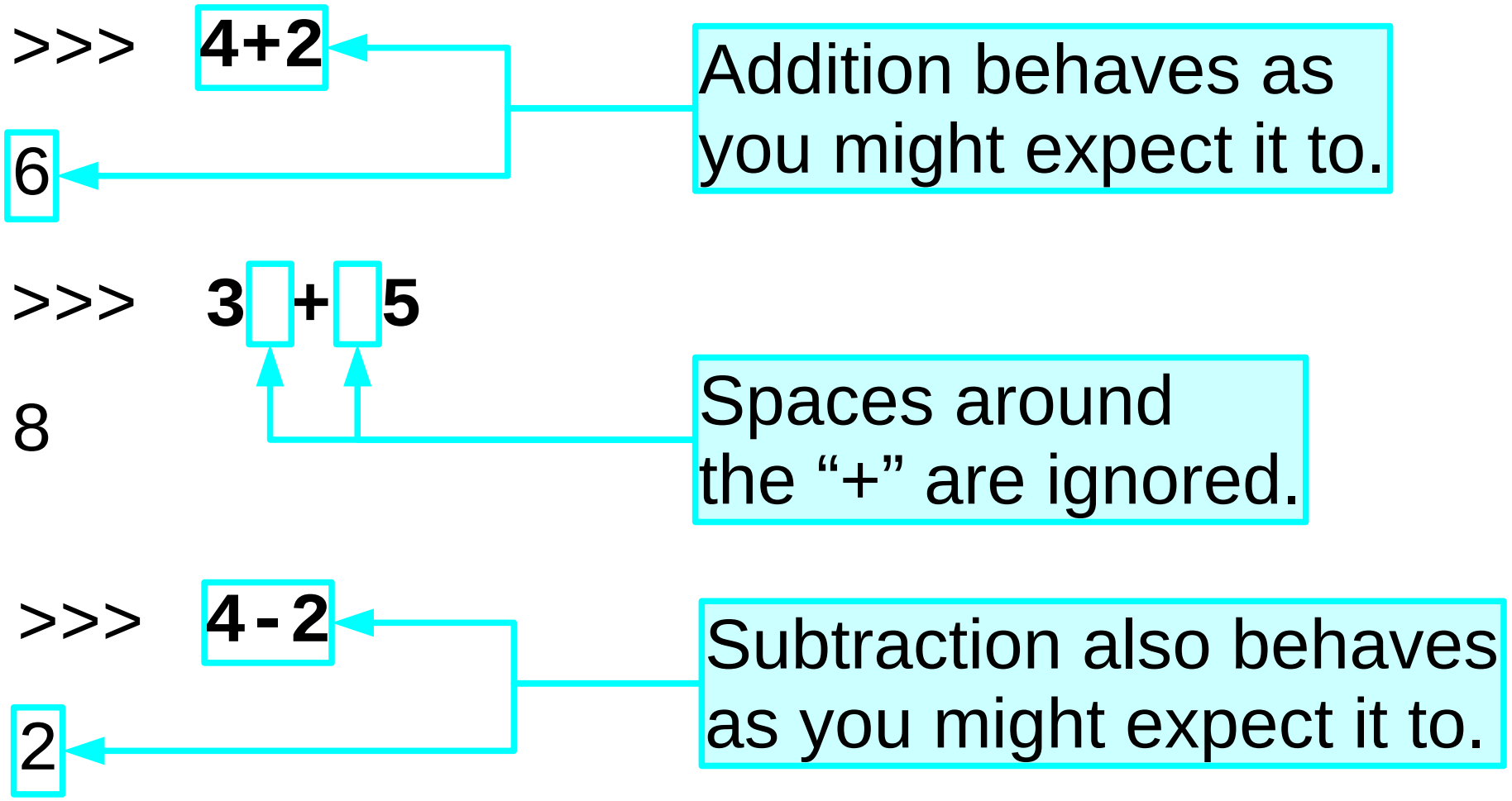

```
$ python three.py
3
$
```

No “5” !

Types of values

Numbers	Whole numbers
	Decimal numbers
Text	
“Boolean”	True
	False

Integers


>>> 4+2
6
Addition behaves as you might expect it to.
>>> 3 + 5
8
Spaces around the “+” are ignored.
>>> 4-2
2
Subtraction also behaves as you might expect it to.

Integers

```
>>>  3 * 5
15
```

Multiplication uses a "*" instead of a "×".

```
>>>  5 / 3
1
```

Division rounds down.

```
>>>  -5 / 3
-2
```

Strictly down.

```
>>>  4**2
16
```

Raising to powers uses "**4**2**" instead of "4^2".

Integers

```
>>> 65536 * 65536
4294967296L
>>> 4294967296 * 4294967296
18446744073709551616L
>>> 18446744073709551616 *
18446744073709551616

340282366920938463463374607431768211456L
```

Long integer

No limit to size of Python's integers!

Floating point numbers

1	1.0
1 ¼	1.25
1 ½	1.5

But

1 ⅓	1.3	
	1.33	
	1.333	
	1.3333	

```
>>> 1.0
1.0
>>> 0.5
0.5
>>> 0.25
0.25
```

1 is OK

½ is OK

¼ is OK

Powers of two.

```
>>> 0.1
0.10000000000000001
```

1/10 is not!

Floating point numbers are…

…printed in decimal

…stored in binary

17 significant figures

```
>>> 4294967296.0 ** 2
1.8446744073709552e+19
```

17 significant figures

$\times 10^{19}$

$1.8446744073709552 \times 10^{19} =$

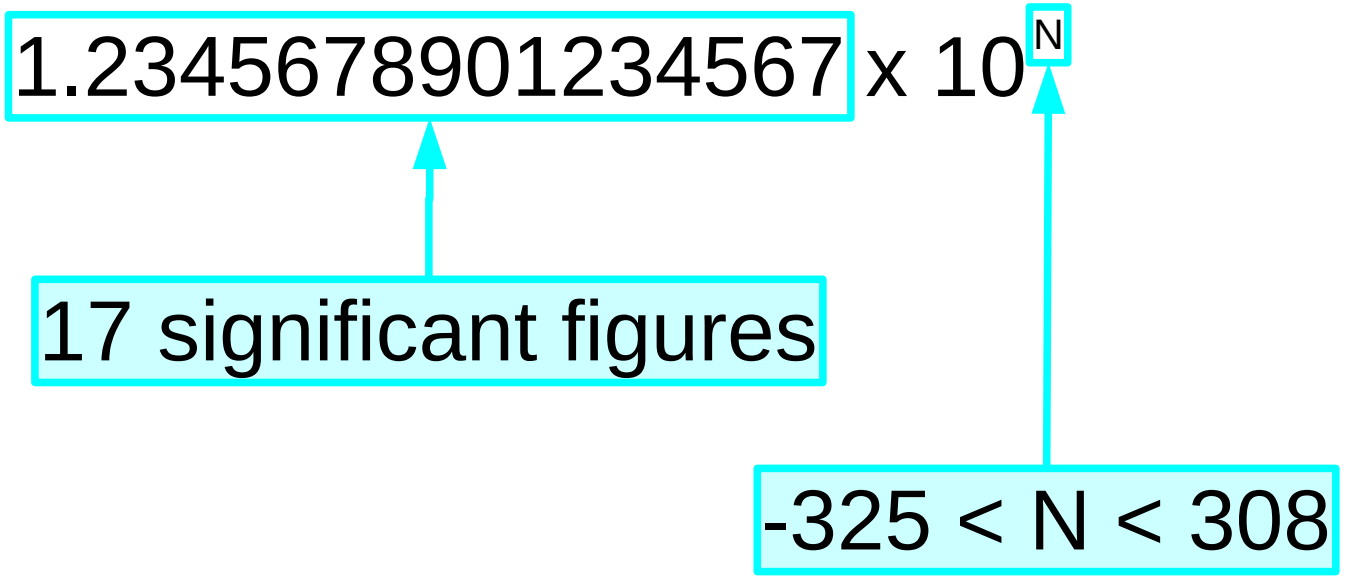
Approximate answer → 18,446,744,073,709,552,000

$4294967296 \times 4294967296 =$

Exact answer → 18,446,744,073,709,551,616

Difference → 384

Floating point limits



Positive values:

4.94065645841e-324 < x < 8.98846567431e+307

Summary on Numbers

Floating Point numbers

$1.25 \longrightarrow$ `1.25`

$1.25 \times 10^5 \longrightarrow$ `1.25e5`

Limited accuracy

Limited range of sizes

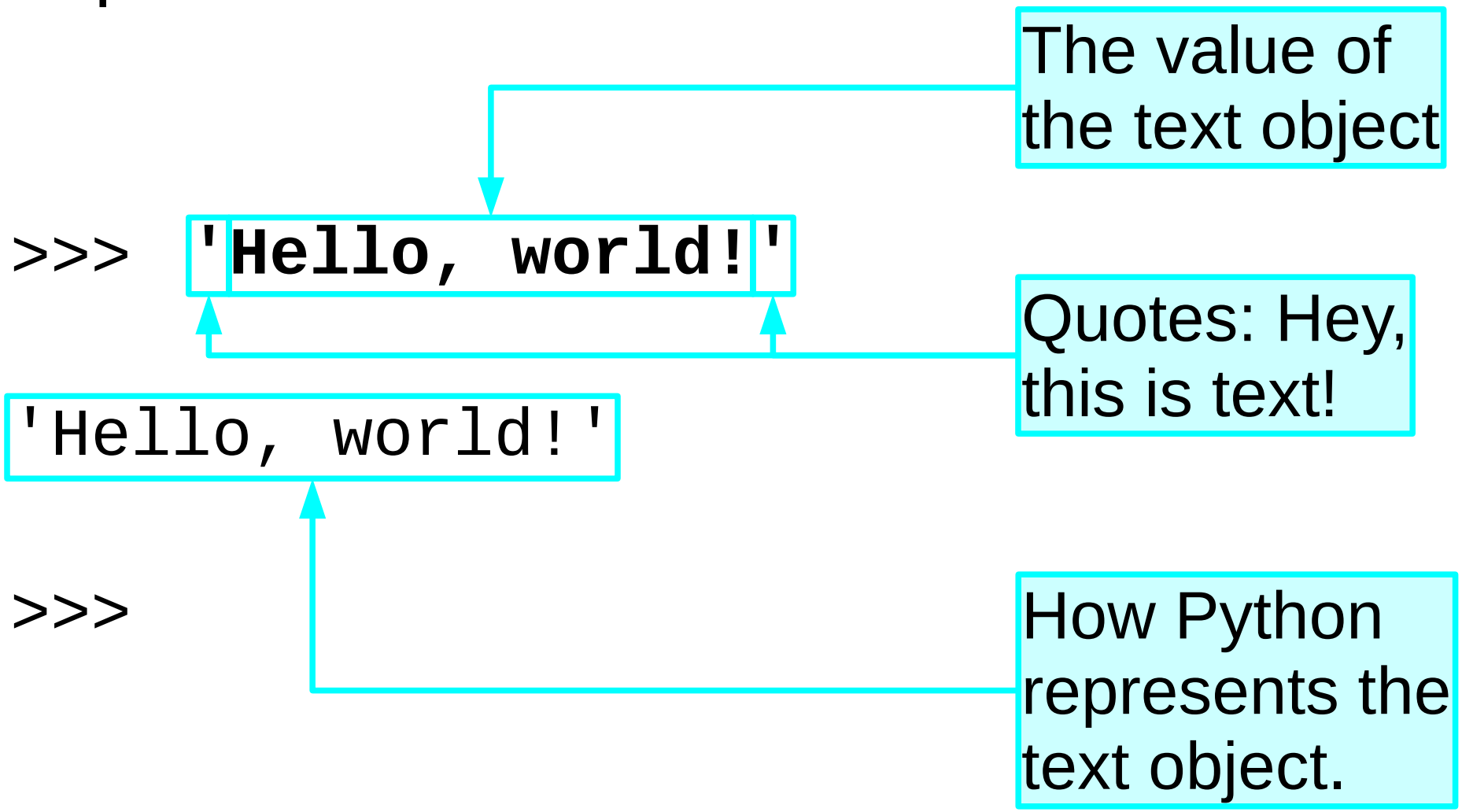
(but typically good enough)

Mathematical operations

$a+b$	$a-b$	$a \times b$	$a \div b$	a^b
`a+b`	`a-b`	`a*b`	`a/b`	`a**b`

Strings

quotes



Why do we need quotes?

3 → It's a number

print → ? → Is it a command?

print → ? → Is it a string?

'print' → It's a string

double quotes work the same

print → It's a command

Mixed quotes

```
>>> print 'He said "Hello" to her.'
He said "Hello" to her.

>>> print "He said 'Hello' to her."
He said 'Hello' to her.
```

Joining strings together

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

>>> 'He said ' + 'something.'
'He said something.'
```

Repeated text

```
>>> 'Bang! '  *  3
'Bang! Bang! Bang! '

>>> 3  *  'Bang! '
'Bang! Bang! Bang! '
```

Line breaks

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

\n → **n**ew line

Special characters

\a → ♪ bell

\n → ↵

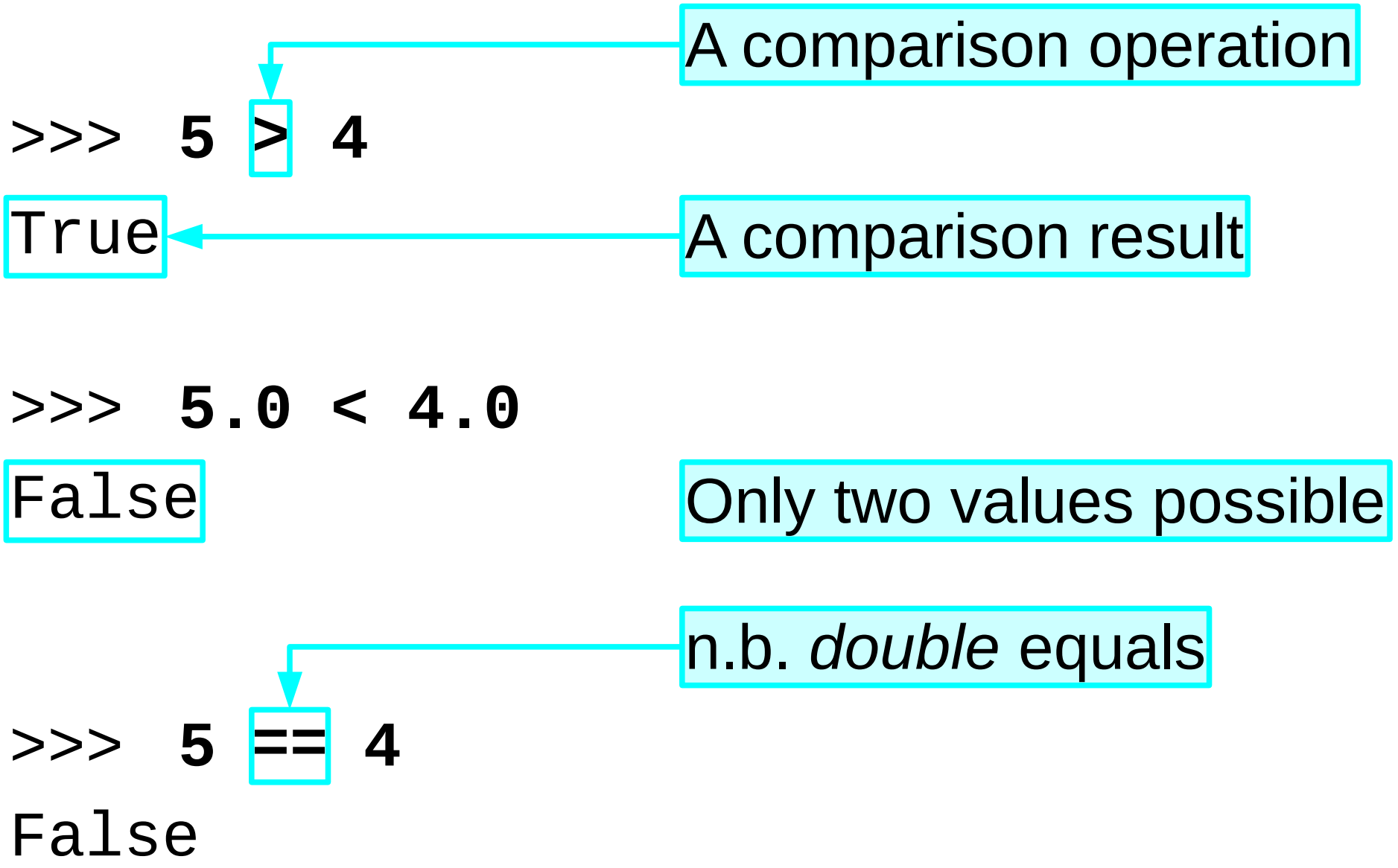
\t → →| tab

\' → '

\" → "

\\ → \

Comparisons



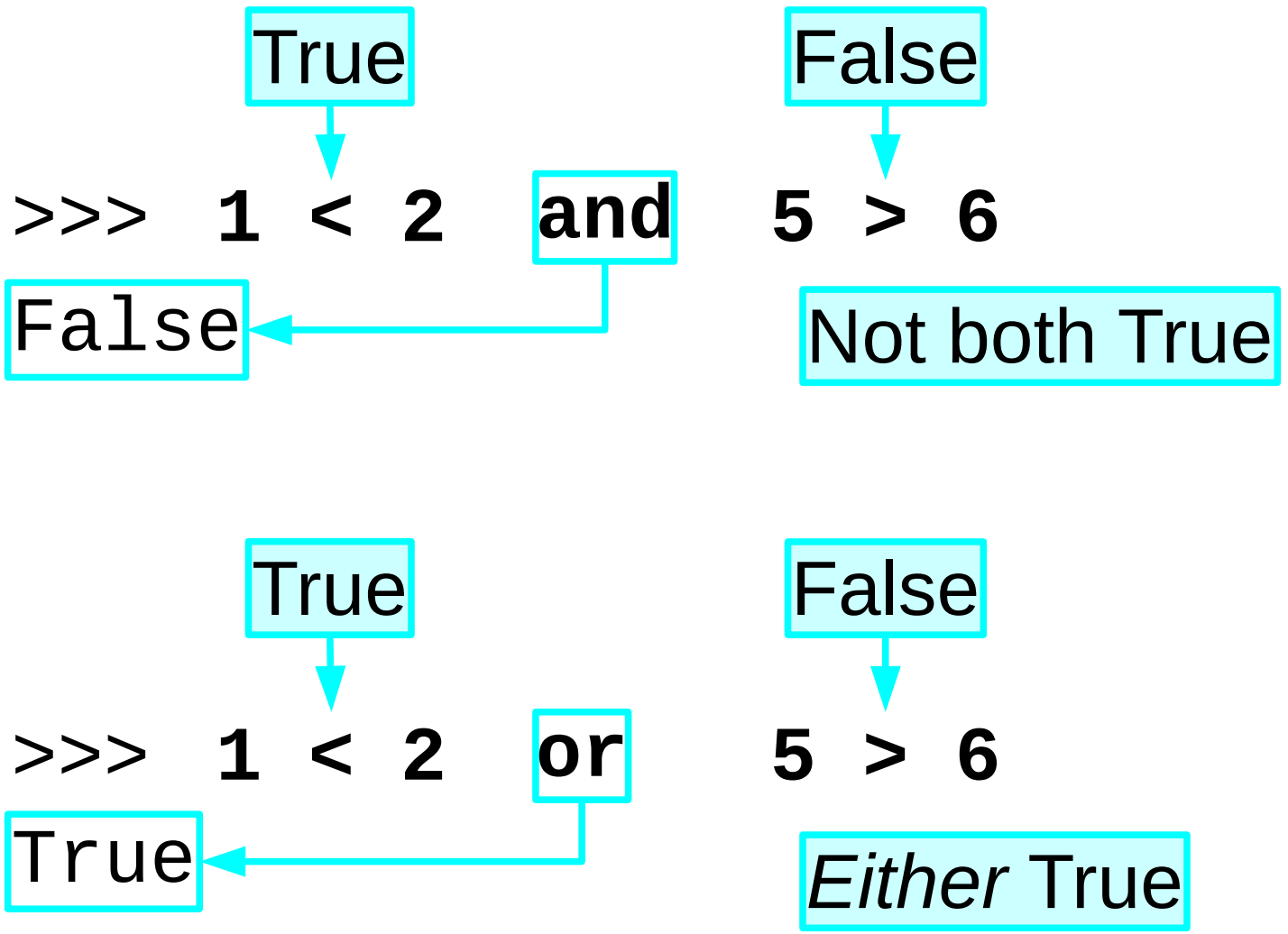
Comparing strings

```
>>> 'cat' < 'mat'
True

>>> 'bad' < 'bud'
True

>>> 'Cat' < 'cat'
True

>>> 'Fat' < 'cat'
True
```

Alphabetic order…

ABCDEFGHIJKLMNOPQRSTUVWXYZ…
abcdefghijklmnopqrstuvwxyz

Combining booleans



Negating booleans

```
>>> 1 > 2
False
```

```
>>> 1 == 2
False
```

True → False
False → True

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

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

```
>>> not False
True
```

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

Precedence

First 

```
**    %     /     *     -     +
```

Arithmetic

```
==    !=    >=    >     <=    <
```

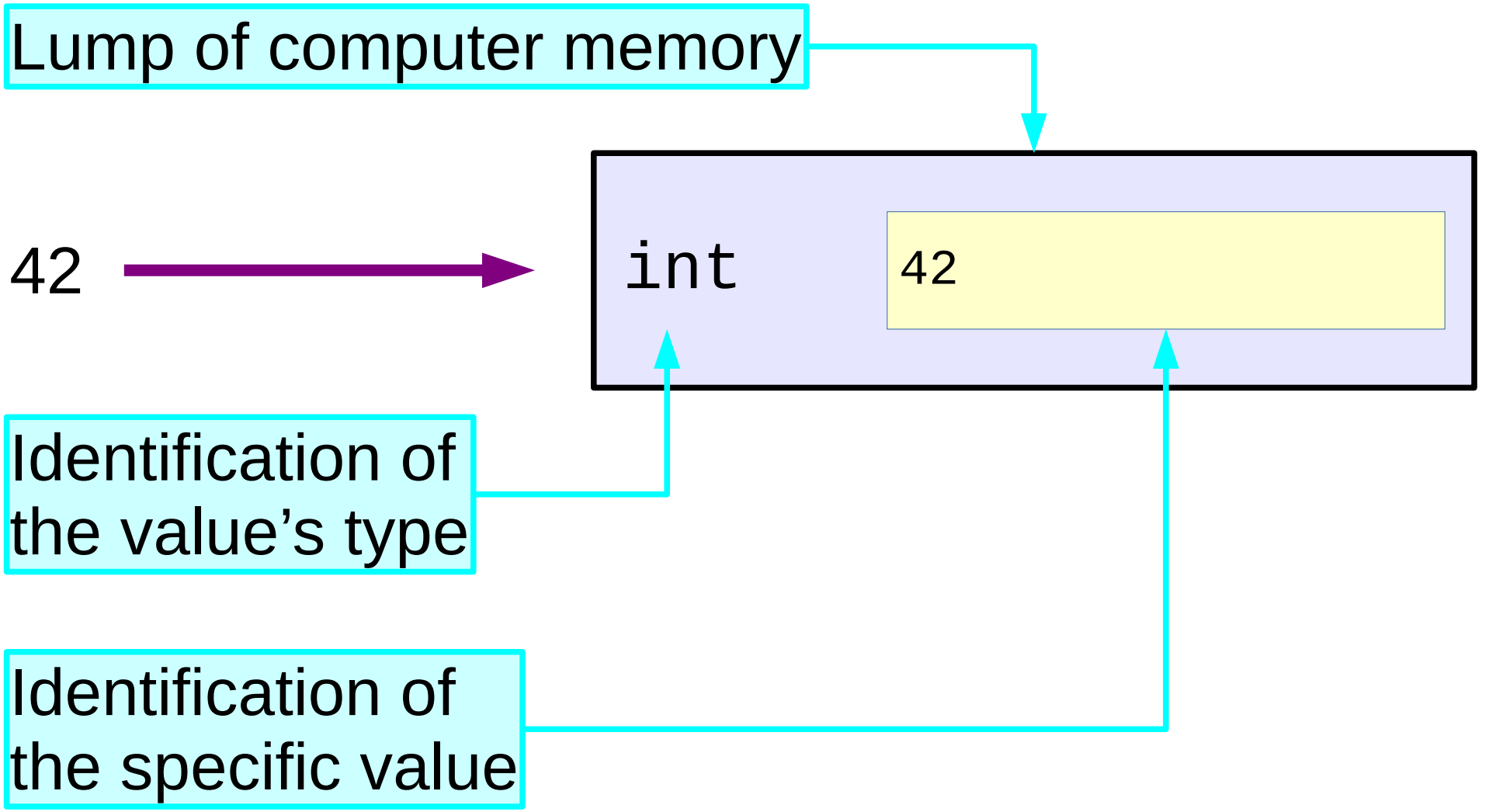
Comparison

```
not   and   or
```

Logical

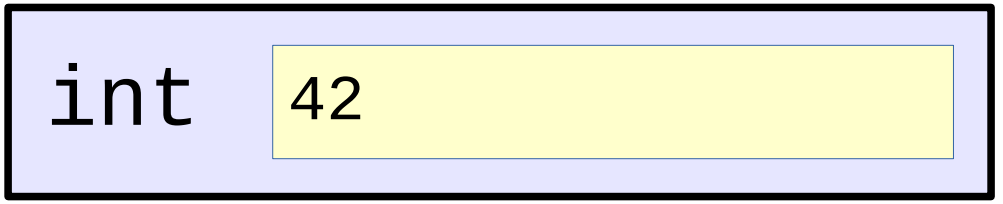
 Last

How Python stores values

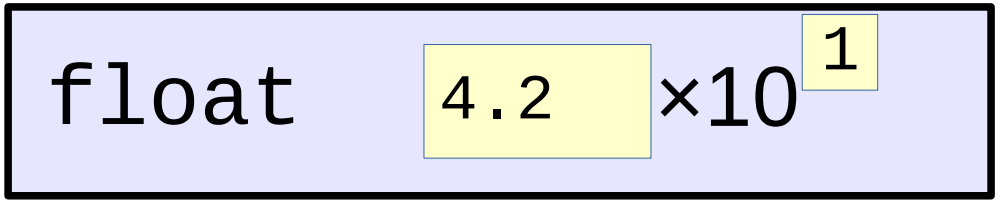


How Python stores values

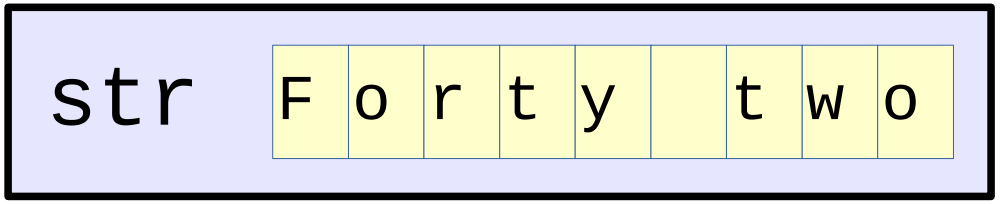
42



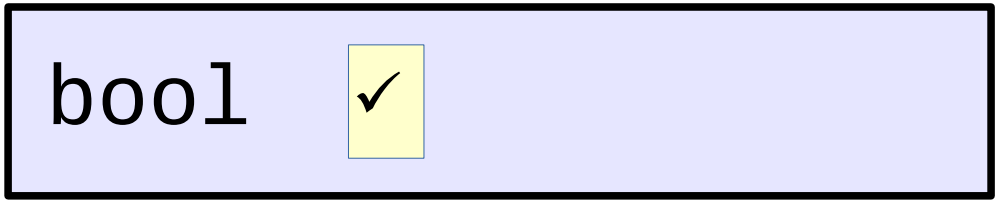
$4{\cdot}2\times10^{1}$



'Forty two'



True



Variables

Attaching a name to a value.

```
>>> 40 + 2
42
```

An expression

The expression's value

```
>>> answer = 42
```

Attaching the name `answer` to the value `42`.

```
>>> answer
42
```

The name given

The attached value returned

if … then … else …

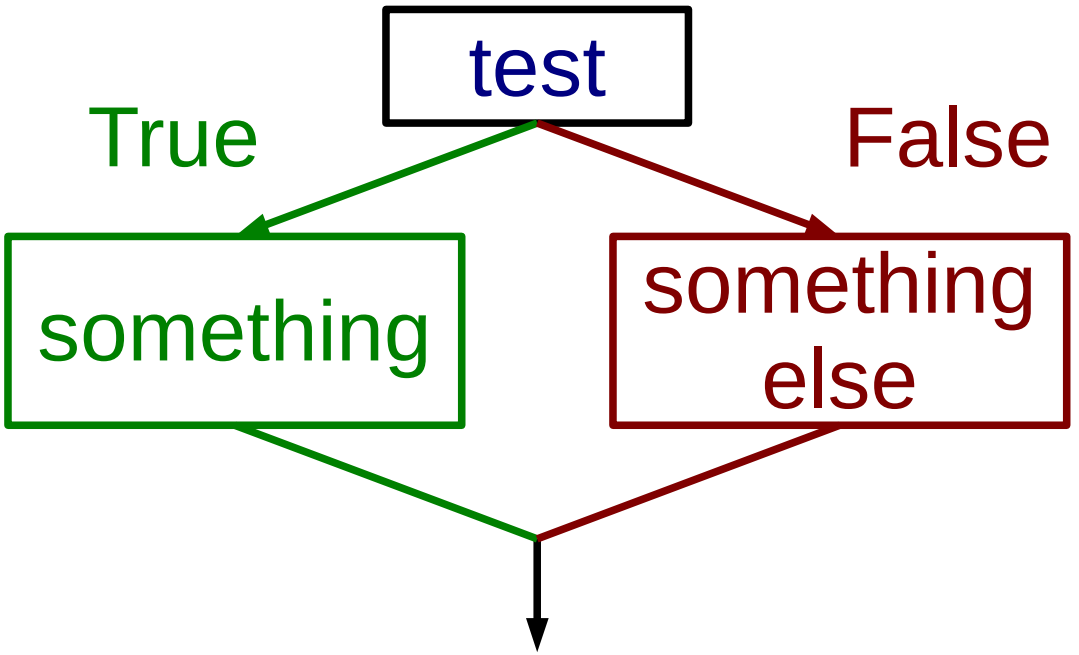
Run a test

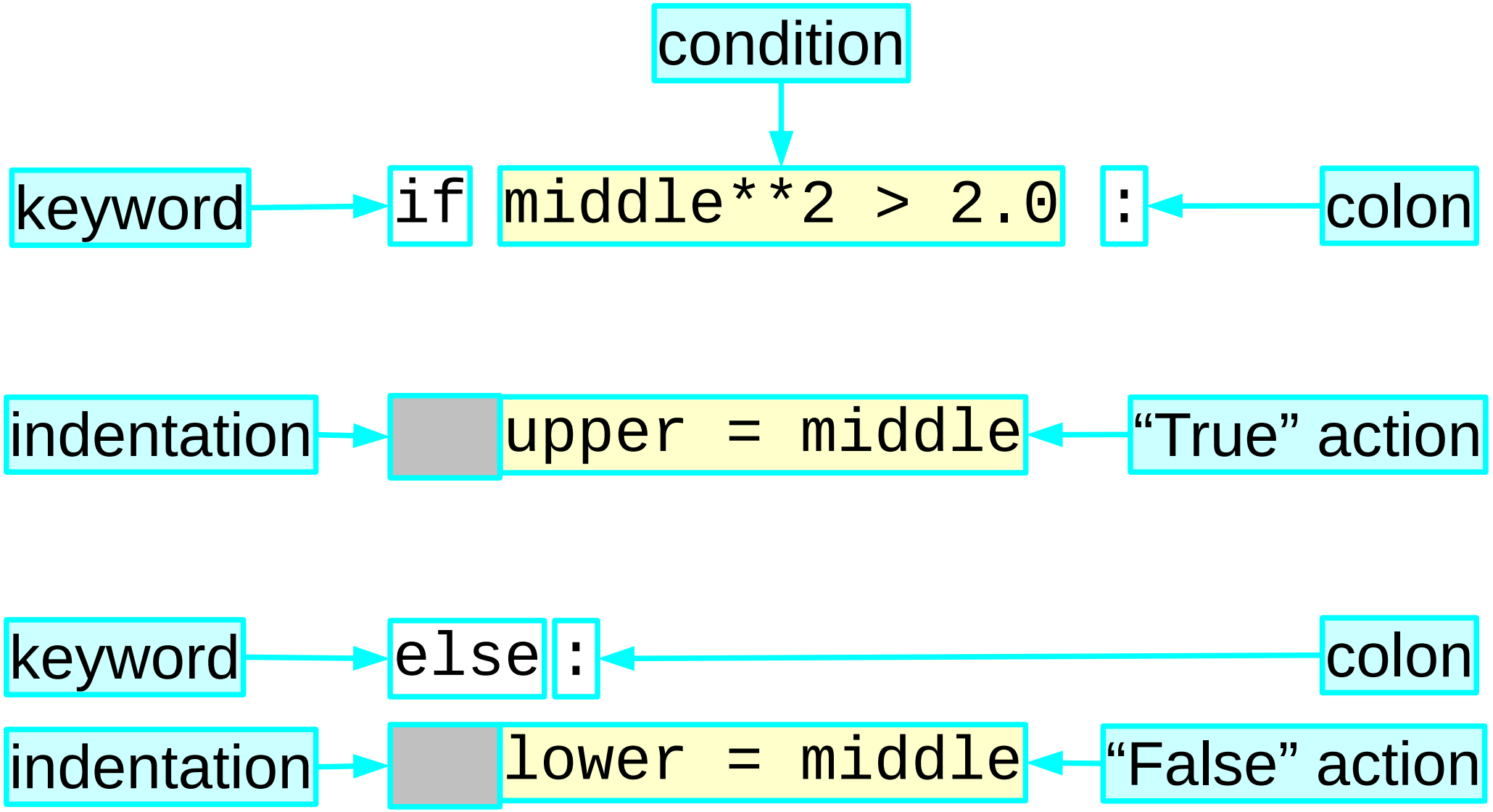
Do something
if it succeeds.

Do something
else if it fails.

Colon…Indentation

```
if test :
  something
else :
  something else
```




condition
keyword
if
middle**2 > 2.0
:
colon
indentation
upper = middle
“True” action
keyword
else
:
colon
indentation
lower = middle
“False” action

Example script:

middle1.py

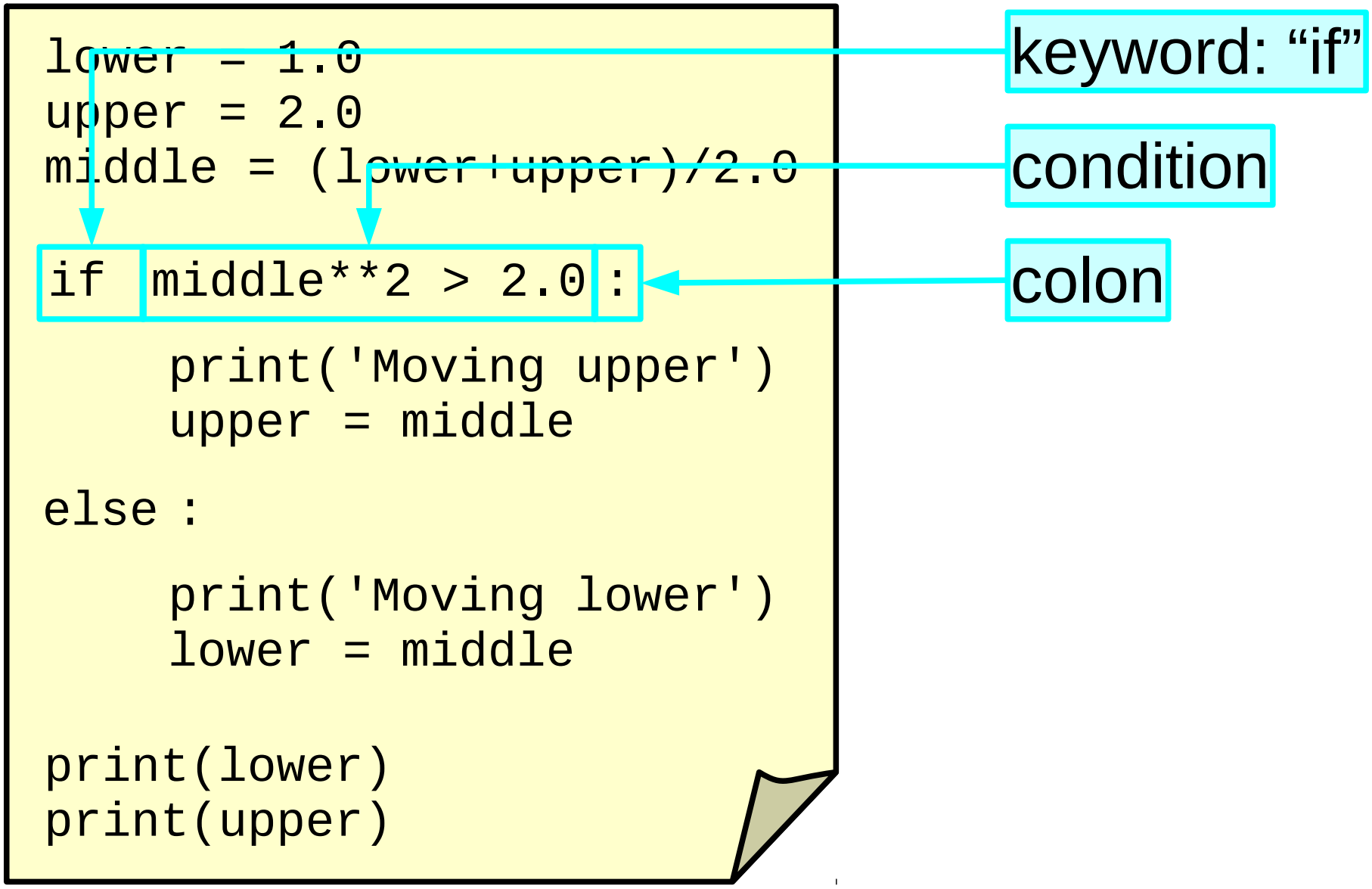
```
lower = 1.0
upper = 2.0
middle = (lower+upper)/2.0

if  middle**2 > 2.0 :

    print('Moving upper')
    upper = middle

else :

    print('Moving lower')
    lower = middle


print(lower)
print(upper)
```

Example script: before

```
lower = 1.0
upper = 2.0
middle = (lower+upper)/2.0

if  middle**2 > 2.0 :

    print('Moving upper')
    upper = middle

else :

    print('Moving lower')
    lower = middle


print(lower)
print(upper)
```

Set-up prior to the test.

Example script: if…



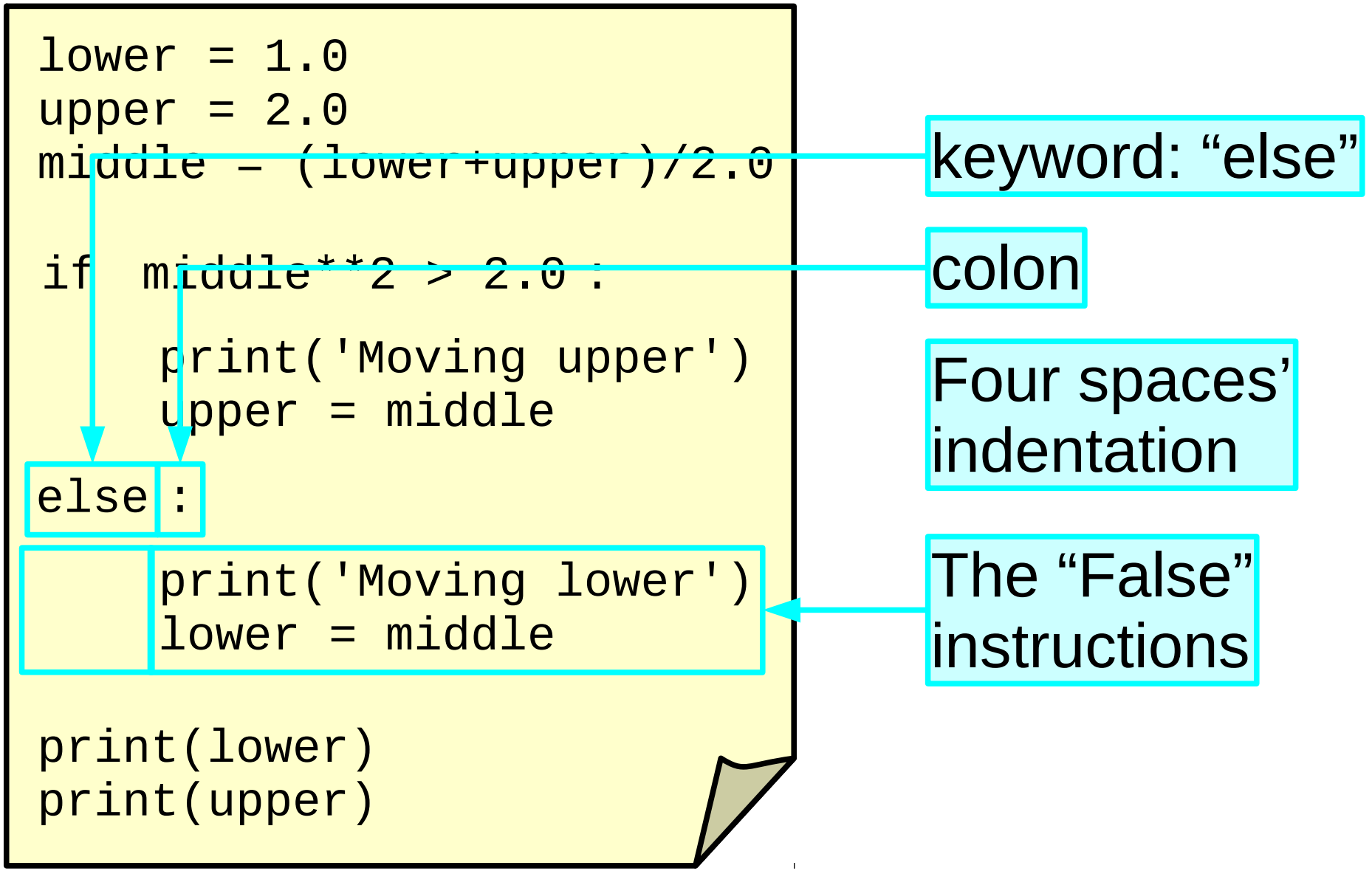
Example script: then…

```
lower = 1.0
upper = 2.0
middle = (lower+upper)/2.0

if  middle**2 > 2.0 :
    print('Moving upper')
    upper = middle

else :
    print('Moving lower')
    lower = middle

print(lower)
print(upper)
```

Four spaces' indentation

The "True" instructions

Example script: else…



Example script: after

```
lower = 1.0
upper = 2.0
middle = (lower+upper)/2.0

if  middle**2 > 2.0 :

    print('Moving upper')
    upper = middle

else :

    print('Moving lower')
    lower = middle

print(lower)
print(upper)
```

Not indented

Run regardless of the test result.

Example script: running it

```
lower = 1.0
upper = 2.0
middle = (lower+upper)/2.0

if  middle**2 > 2.0 :

    print('Moving upper')
    upper = middle

else :

    print('Moving lower')
    lower = middle


print(lower)
print(upper)
```

Unix prompt

```
$ python middle1.py
Moving upper
1.0
1.5
$
```

while

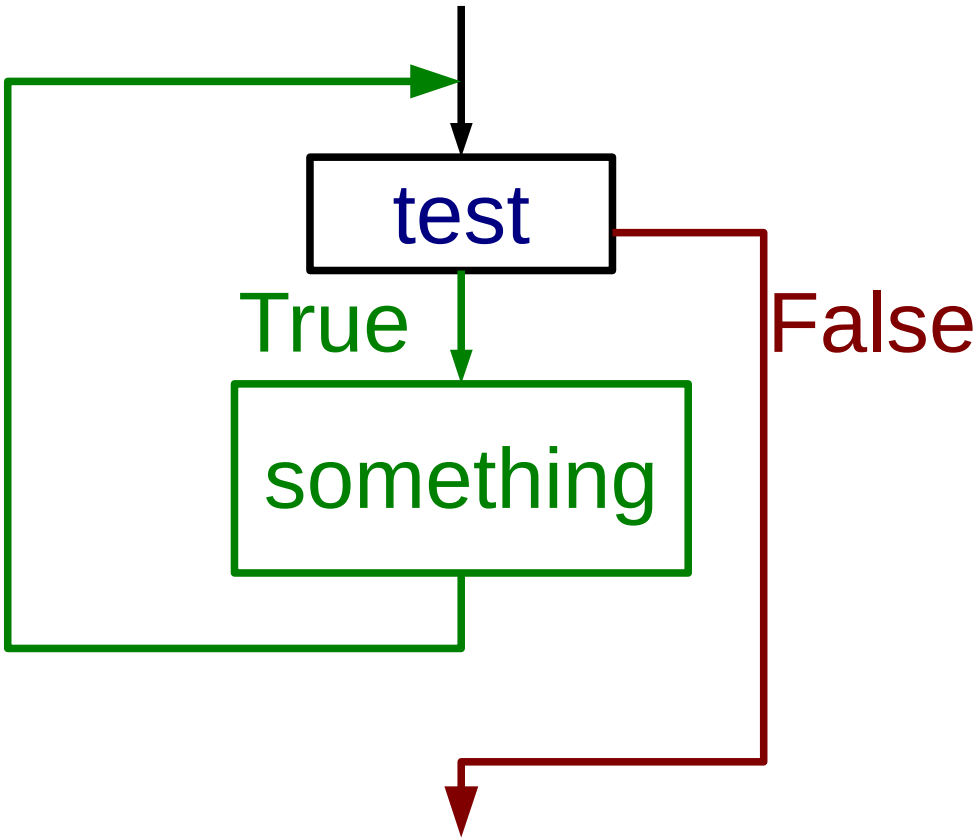
```
while test :
  something
```

Run a test

Do something
if it succeeds.

Finish if
it fails.

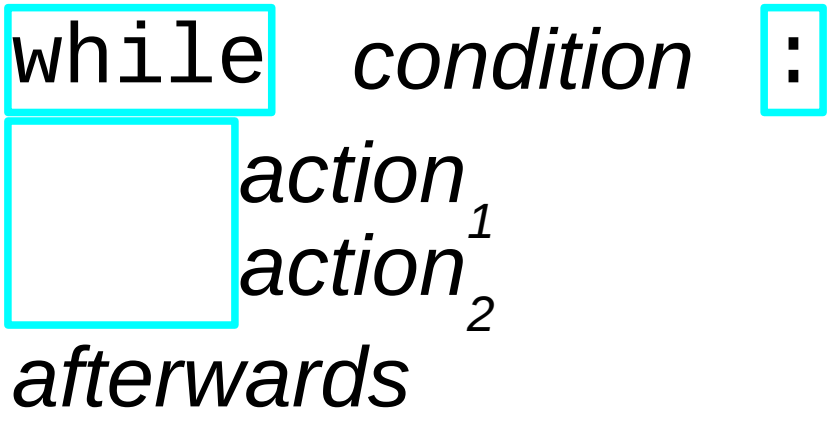
Go back to the test.



while

Keep going while…

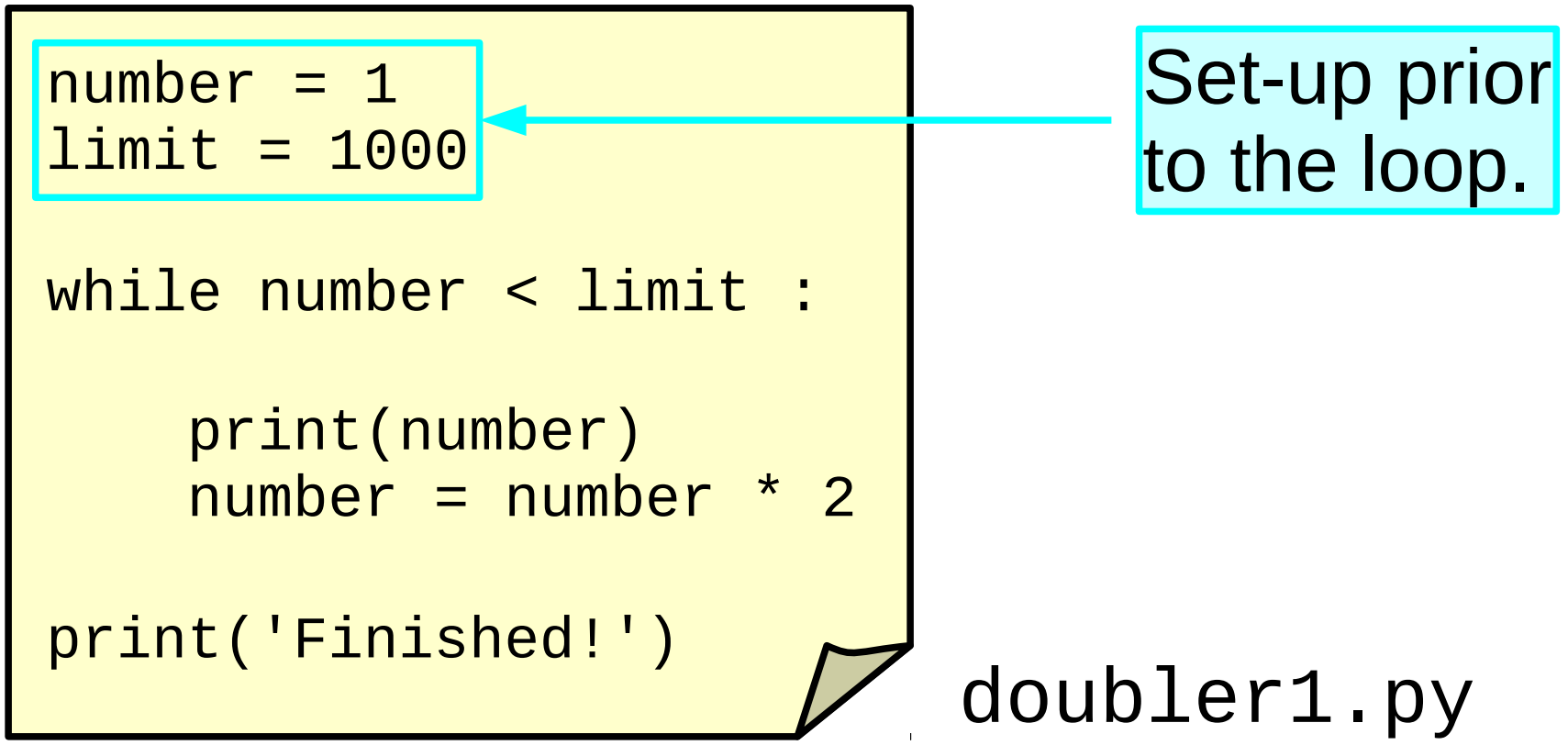
…then stop.



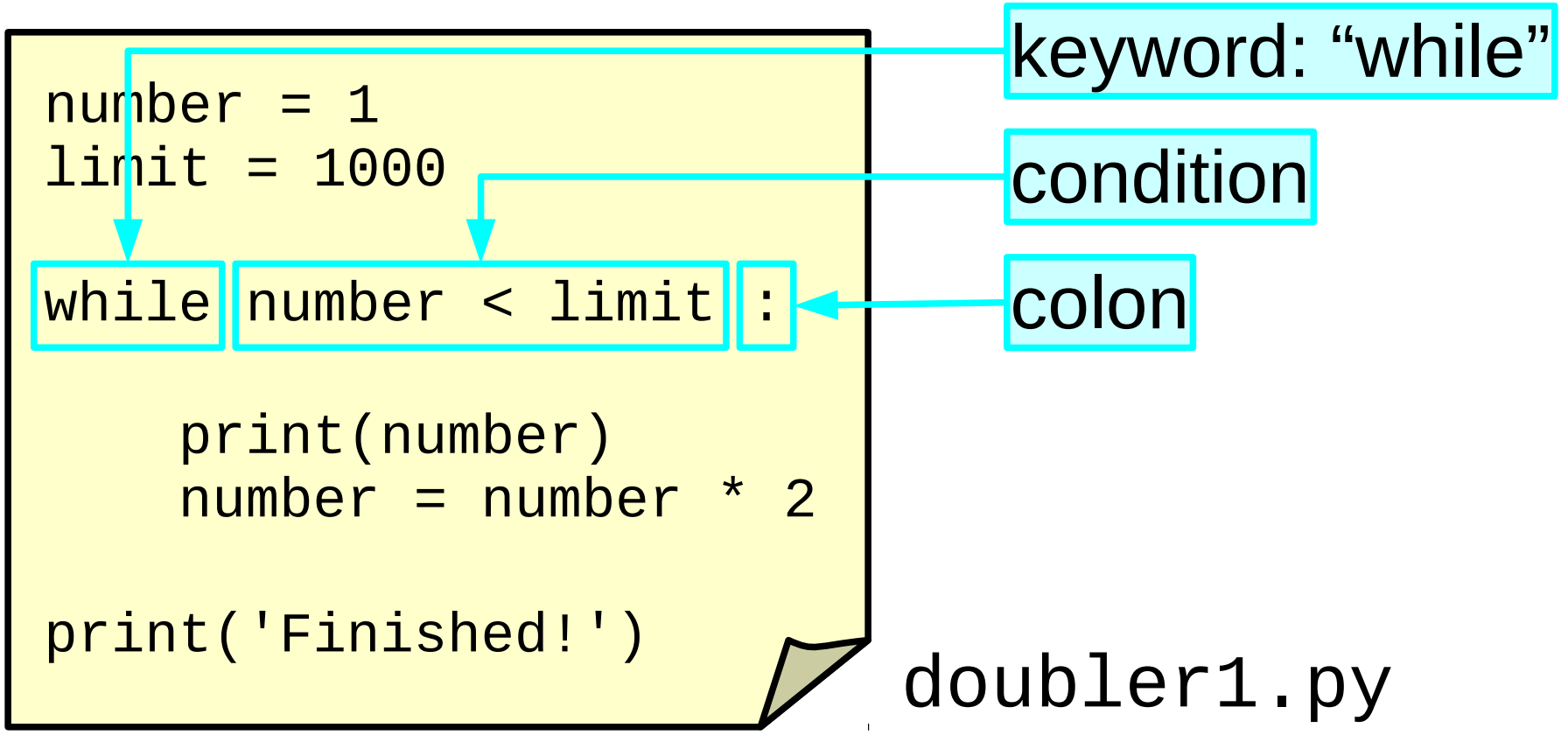

```
number = 1
limit = 1000

while number < limit :

    print(number)
    number = number * 2

print('Finished!')
```

doubler1.py

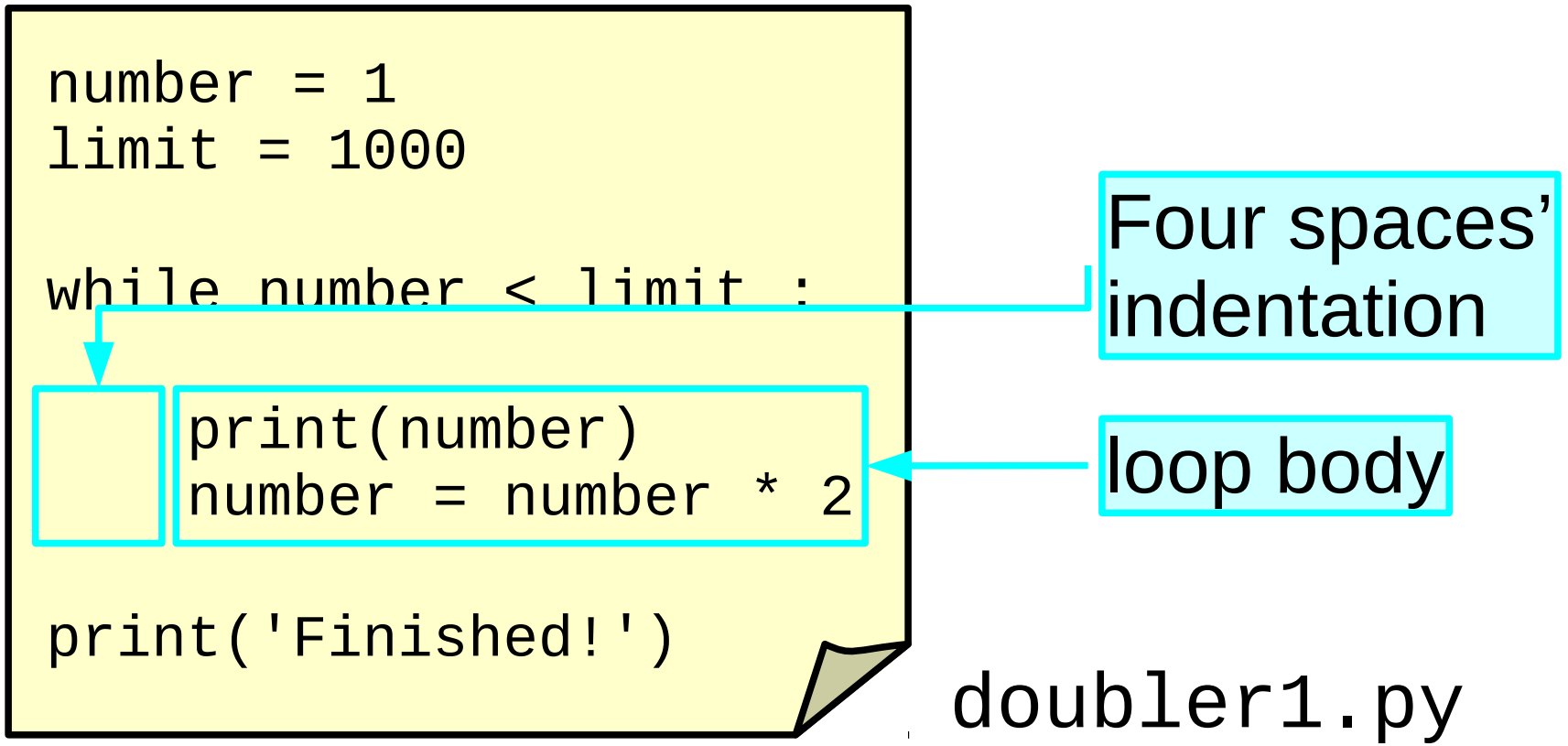
Example script: before



Example script: while…



Example script: loop body



Example script: after

```
number = 1
limit = 1000

while number < limit :

    print(number)
    number = number * 2

print('Finished!')
```

doubler1.py

Not indented

Run after the looping is finished.

Example script: running it

```
number = 1
limit = 1000
                        :
while number < limit

    print(number)
    number = number * 2

print('Finished!')
```

```
> python doubler1.py

1
2
4
8
16
32
64
128
256
512
Finished!
```

while... if … then … else …

```
lower = 1.0
upper = 2.0

while upper - lower > 1.0e-15 :

    middle = (upper+lower)/2.0

    if middle**2 > 2.0:
        print('Moving upper')
        upper = middle
    else:
        print('Moving lower')
        lower = middle

print(middle)
```

Double indentation

Double indentation

Running the script

```
lower = 1.0
upper = 2.0

while upper - lower > 1.0e-15:

    middle = (upper+lower)/2.0

    if middle**2 > 2.0 :
        print('Moving upper')
        upper = middle
    else :
        print('Moving lower')
        lower = middle

print(middle)
```

```
> python root2.py

Moving upper
Moving lower
Moving lower
Moving upper
…
Moving upper
Moving upper
Moving lower
Moving lower
1.41421356237
```

Indentation: level 1

```
lower = 1.0
upper = 2.0

while upper - lower > 1.0e-15 :

    middle = (upper+lower)/2.0

    if middle**2 > 2.0  :
        print('Moving upper')
        upper = middle
    else :
        print('Moving lower')
        lower = middle

print(middle)
```

Colon starts the block

Indentation marks the extent of the block.

Unindented line
End of block

Indentation: level 2

```
lower = 1.0
upper = 2.0

while upper - lower > 1.0e-15 :

    middle = (upper+lower)/2.0

    if middle**2 > 2.0 :
        print('Moving upper')
        upper = middle
    else :
        print('Moving lower')
        lower = middle

print(middle)
```

Colon…indentation

“else” unindented

Colon…indentation

Arbitrary nesting

Not just two levels deep

As deep as you want

Any combination

if… inside while…

while… inside if…

if… inside if…

while… inside while…

```
number = 20

if number % 2 == 0:
    if number % 3 == 0:
        print('Number divisible by six')
    else:
        print('Number divisible by two but not three')
else:
    if number % 3 == 0:
        print('Number divisible by three but not two')
    else:
        print('Number indivisible by two or three')
```

Comments

#

The “hash” character. a.k.a. “sharp”, “pound” “number”

Lines starting with “#” are ignored

Partial lines too.

```
# Set the initial bounds of the interval.  Then
# refine it by a factor of two each iteration by
# looking at the square of the value of the
# interval’s mid-point.

# Terminate when the interval is 1.0e-15 wide.

lower = 1.0 # Initial bounds.
upper = 2.0

while upper - lower < 1.0e-15 :
    …
```

Comments

Reading someone else's code.  Writing code for someone else.

Reading *your own* code six months later.  Writing code you can come back to.

Lists

```
['Jan', 'Feb', 'Mar', 'Apr',
 'May', 'Jun', 'Jul', 'Aug',
 'Sep', 'Oct', 'Nov', 'Dec']
```

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

```
[0.0, 1.5707963267948966, 3.141592653589793]
```

Lists — getting it wrong

A script that prints the names of the chemical elements in atomic number order.

```
print('hydrogen')
print('helium')
print('lithium')
print('beryllium')
print('boron')
print('carbon')
print('nitrogen')
print('oxygen')
…
```

Repetition of “print”

Lists — getting it right

A script that prints the names of the chemical elements in atomic number order.

1. Create a list of the element names

2. Print each entry in the list

Creating a list

```
>>> [ 1, 2, 3 ]
[1, 2, 3]

>>> numbers = [ 1, 2, 3 ]

>>> numbers
[1, 2, 3]

>>> []
[]
```

Here's a list

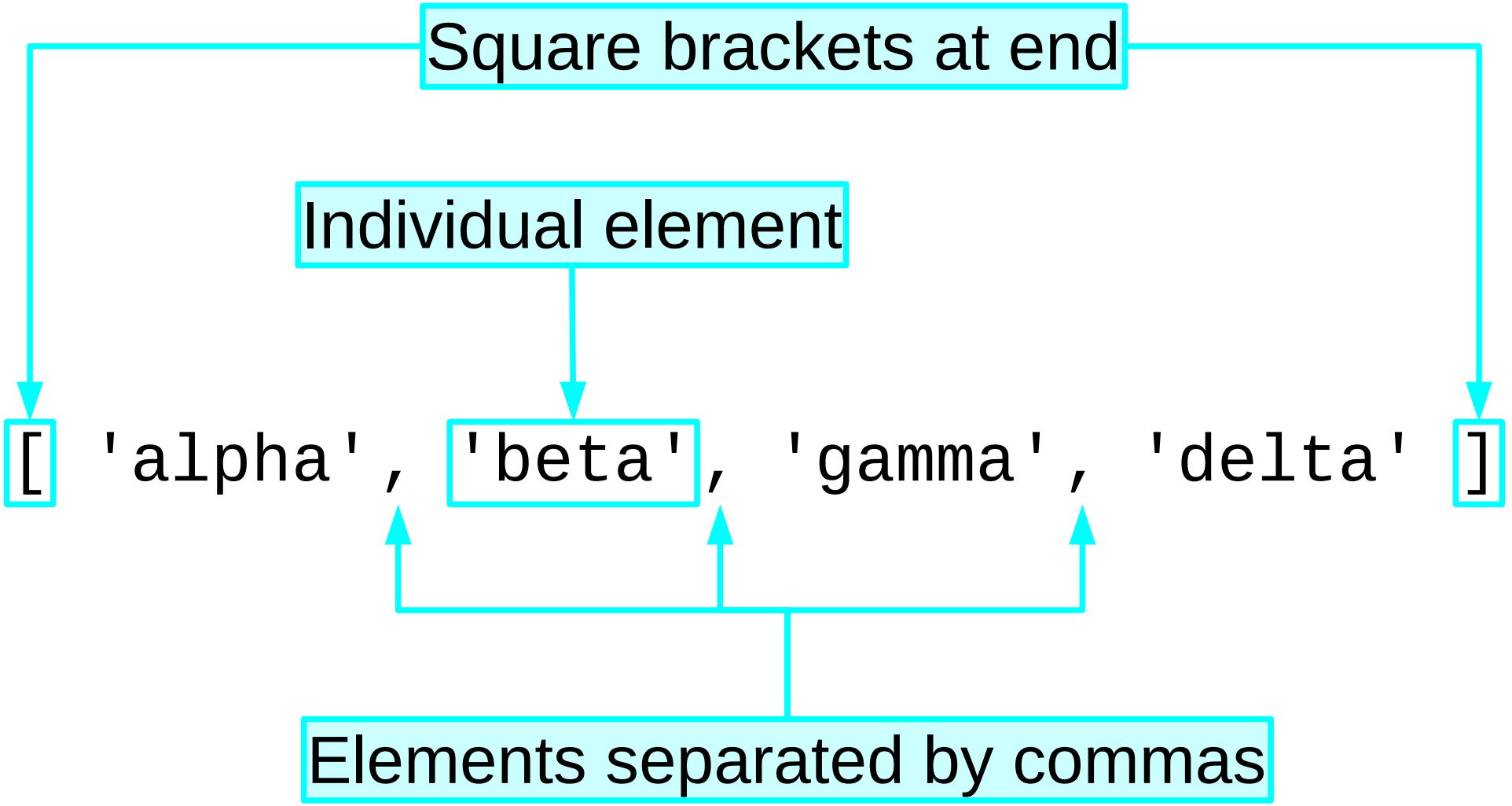
Yes, that's a list

Attaching a name to a variable.

Using the name

Empty list

Anatomy of a list



Order of elements

No “reordering”

```
>>> [ 1, 2, 3 ]
[1, 2, 3]
```

```
>>> [ 3, 2, 1 ]
[3, 2, 1]
```

```
>>> [ 'a', 'b' ]
['a', 'b']
```

```
>>> [ 'b', 'a' ]
['b', 'a']
```

Repetition

No “uniqueness”

```
>>> [ 1, 2, 3, 1, 2, 3 ]
[1, 2, 3, 1, 2, 3]

>>> [ 'a', 'b', 'b', 'c' ]
['a', 'b', 'b', 'c']
```

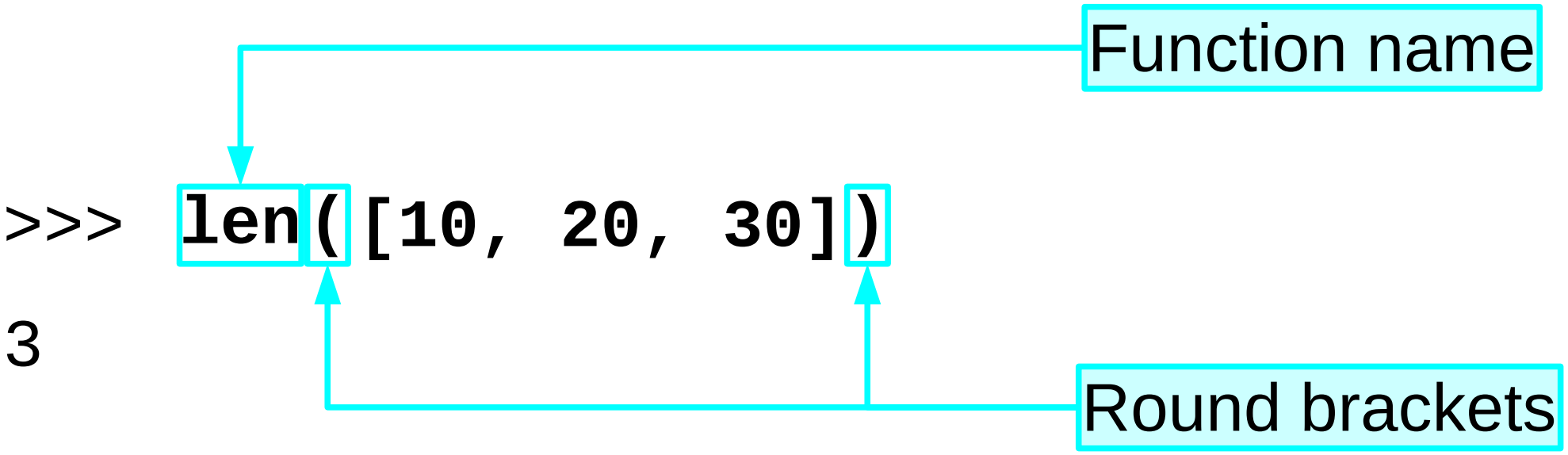
Concatenation

"+" used to join lists.

```
>>> [ 1, 2, 3 ] + [ 4, 5, 6, 7 ]
[1, 2, 3, 4, 5, 6, 7]

>>> ['alpha','beta'] + ['gamma']
['alpha', 'beta', 'gamma']

>>> [ 1, 2, 3 ] + [ 3, 4, 5, 6, 7 ]
[1, 2, 3, 3, 4, 5, 6, 7]
```

"3" appears twice

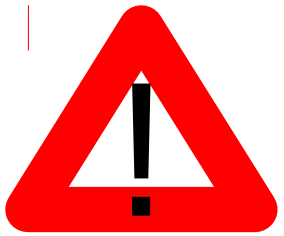
How long is the list?



How long is a string?

Same function

```
>>> len('Hello, world!')
13
```



Recall:
Quotes say “this is a string”.
They are not part of the string.

How long is a *number*?

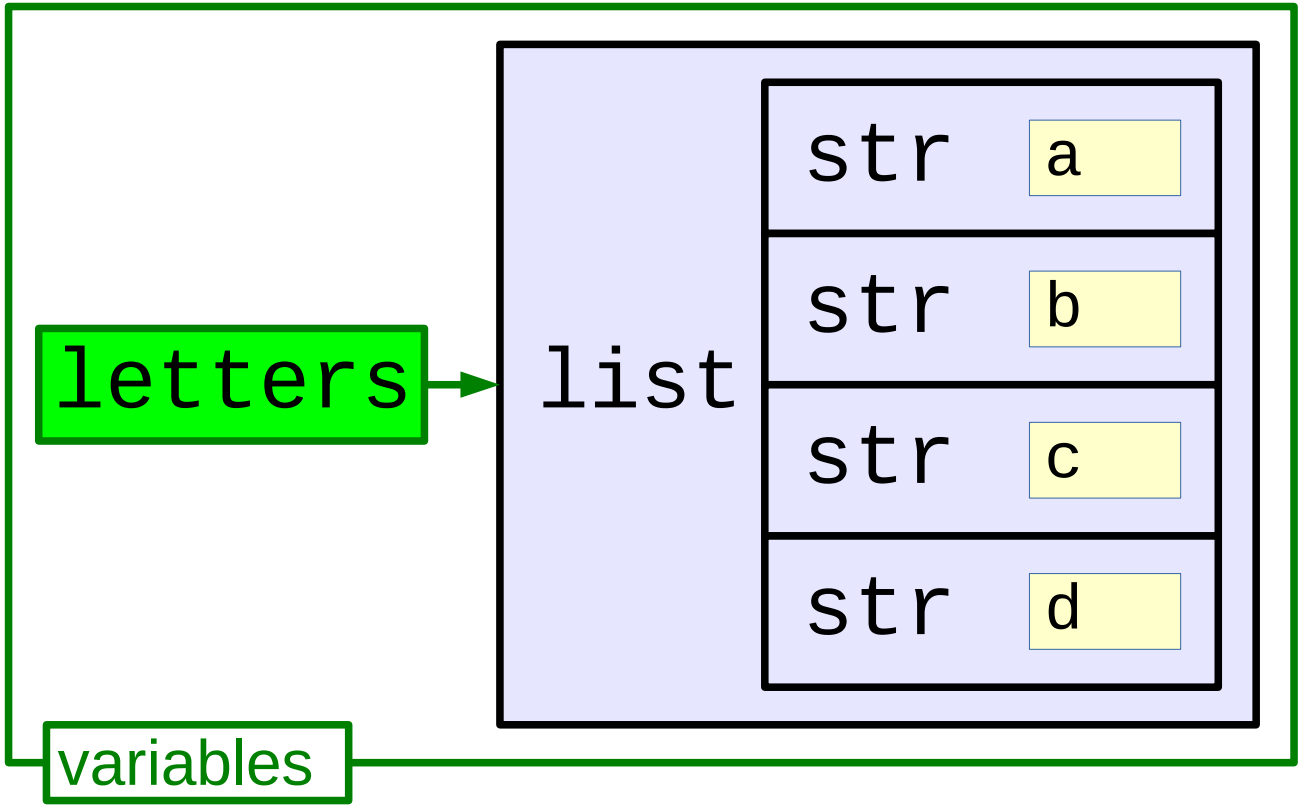
```
>>> len(42)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError:
object of type 'int' has no len()
```

Error message

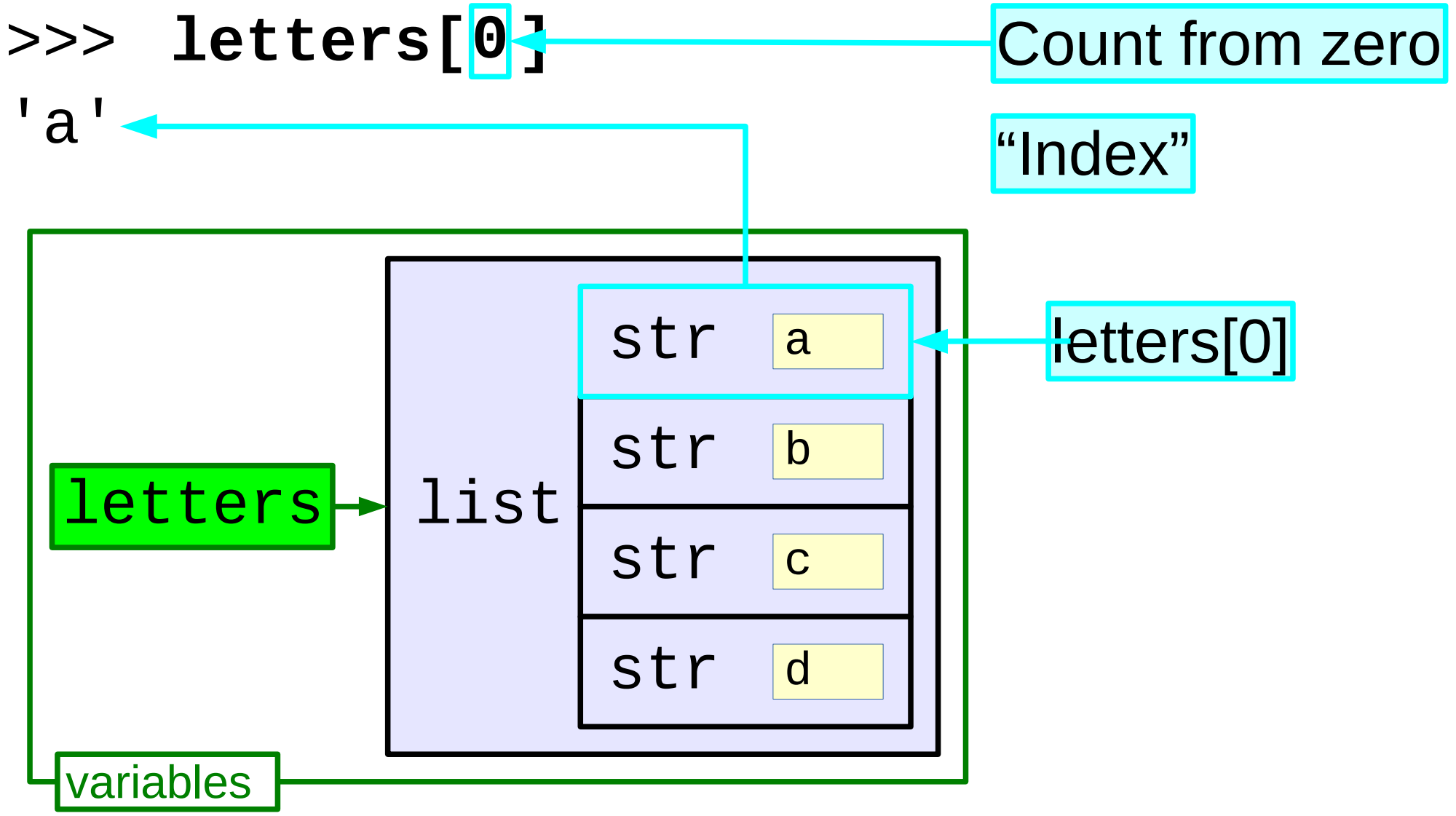
Numbers don't have a "length".

Picking elements from a list

```
>>> letters = ['a', 'b', 'c', 'd']
```



The first element in a list

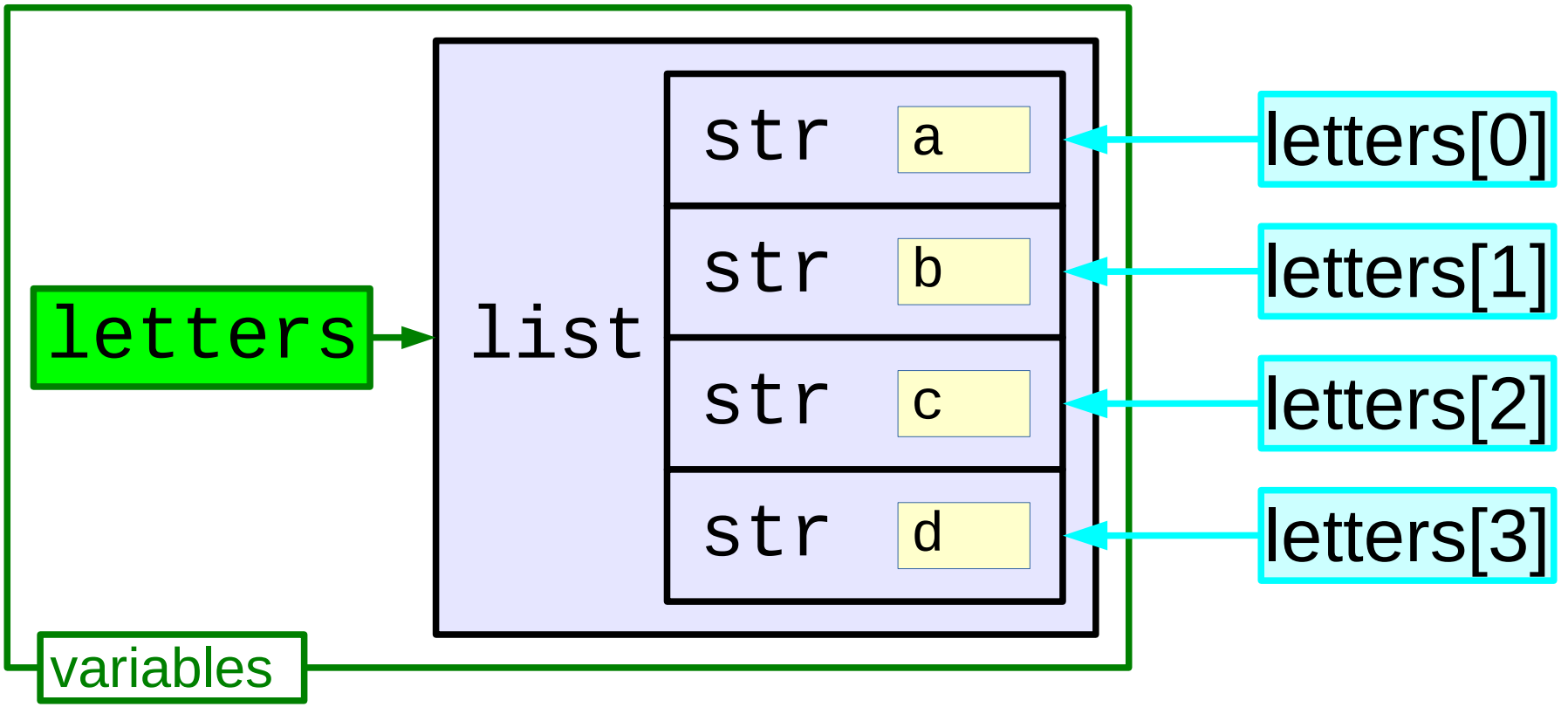


“Element number 2”

`numbers[N]` Indexing into a list

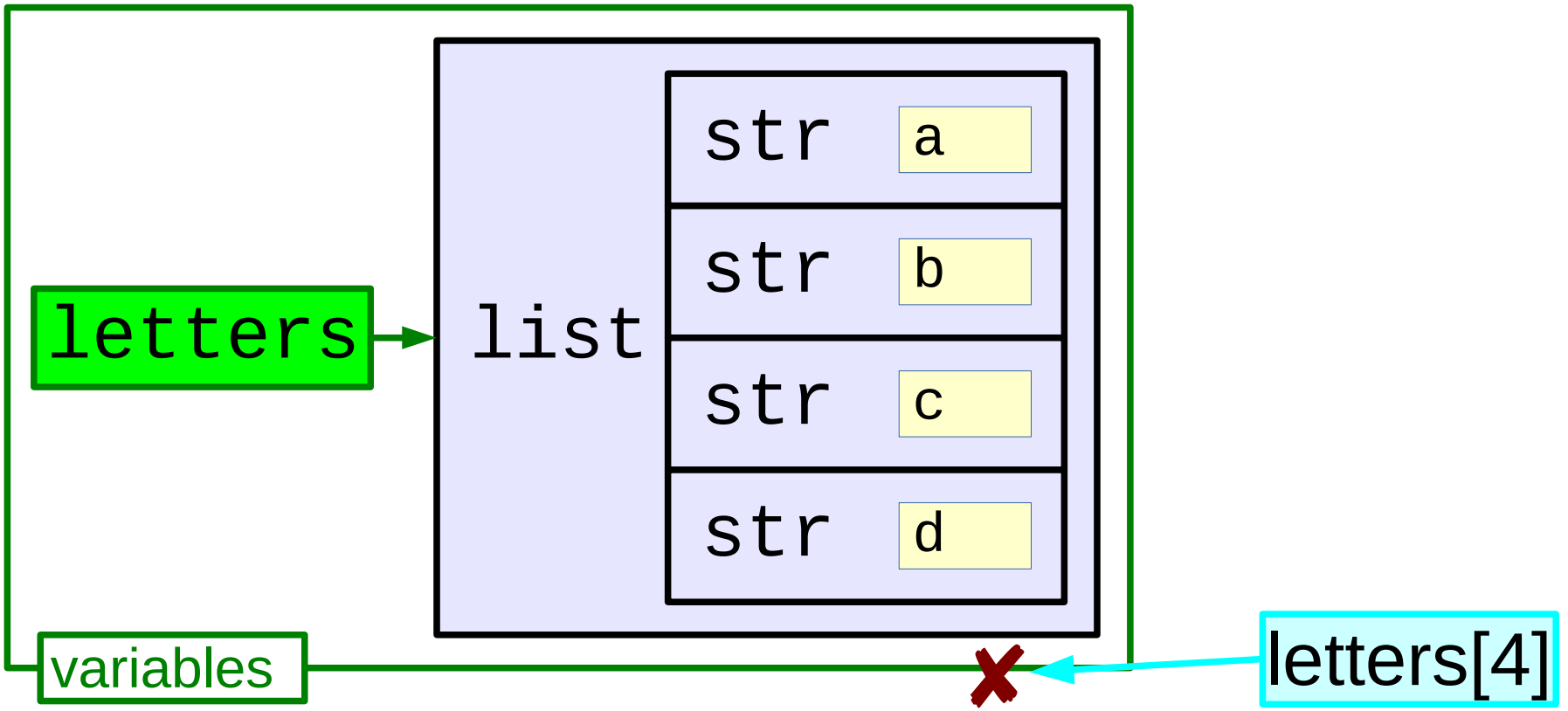
```
>>> letters[2]
'c'
```

The *third* element



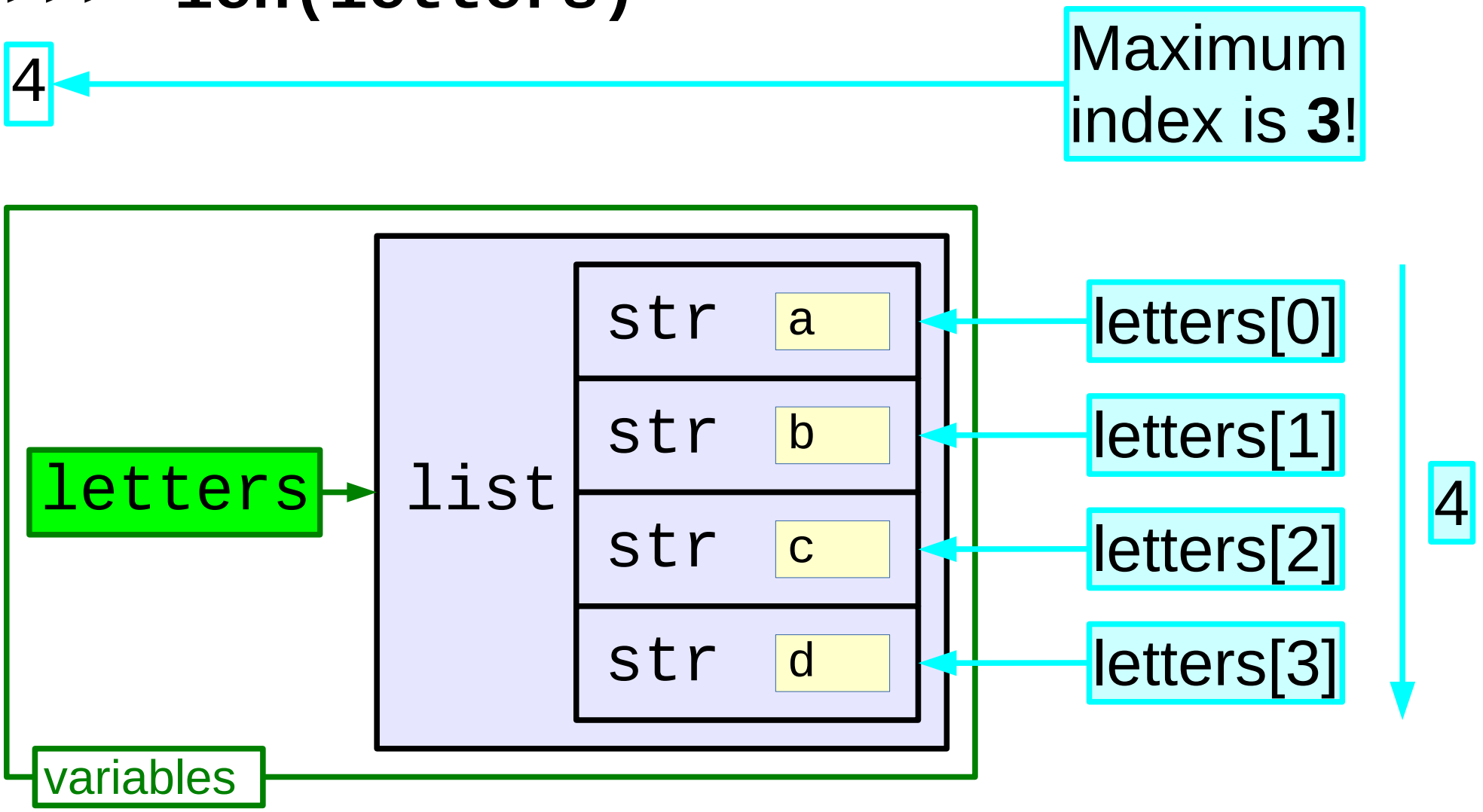
Going off the end

```
>>> letters[4]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```



Maximum index vs. length

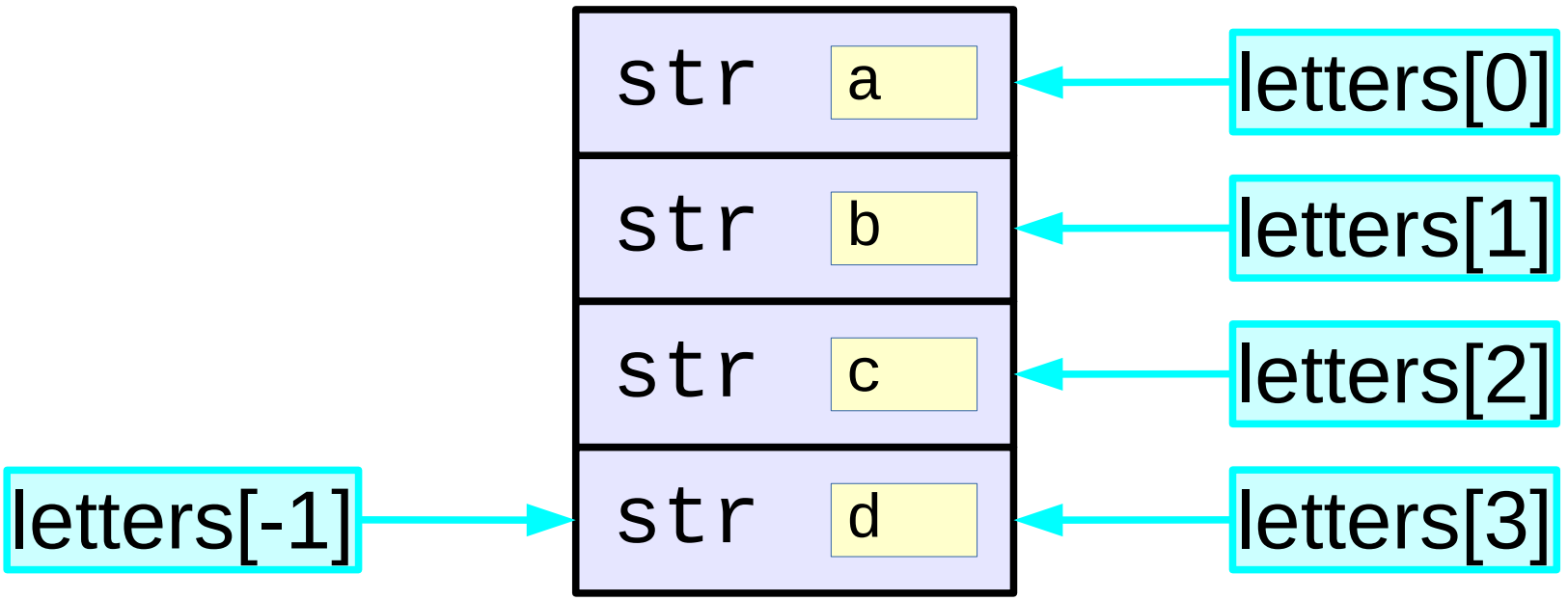
```
>>> len(letters)
```



“Element number -1 !”

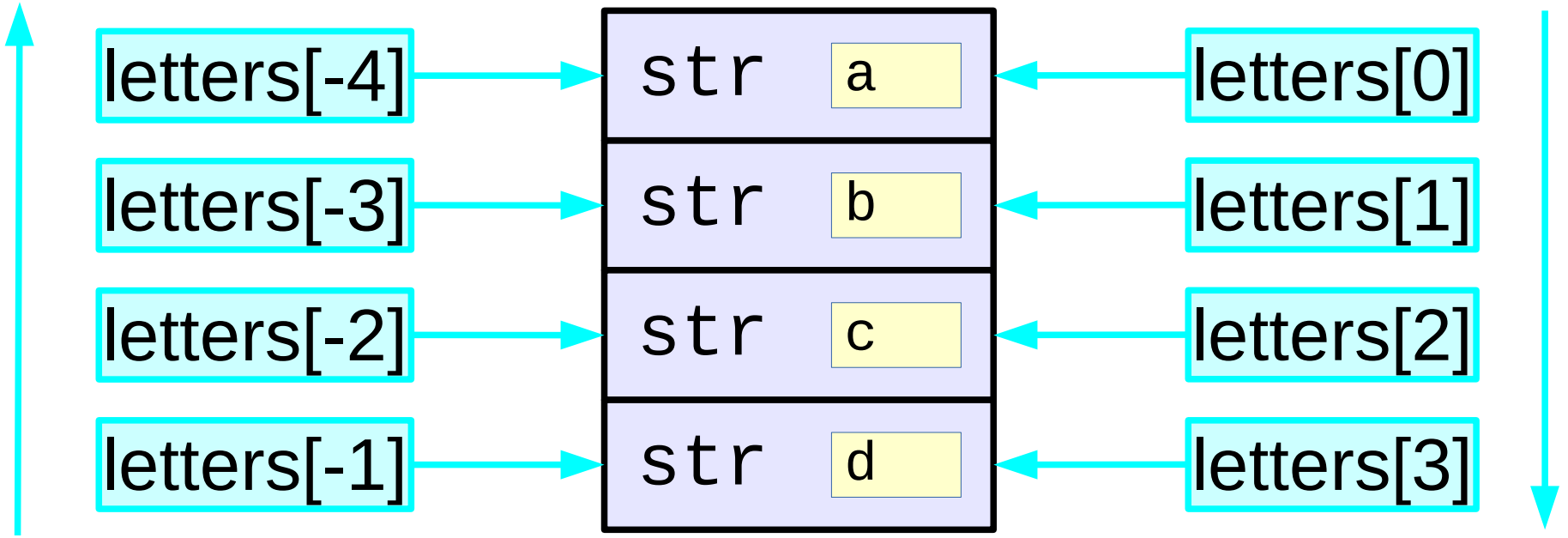
```
>>> letters[-1]
'd'
```

The *final* element



Negative indices

```
>>> letters[-3]
'b'
```

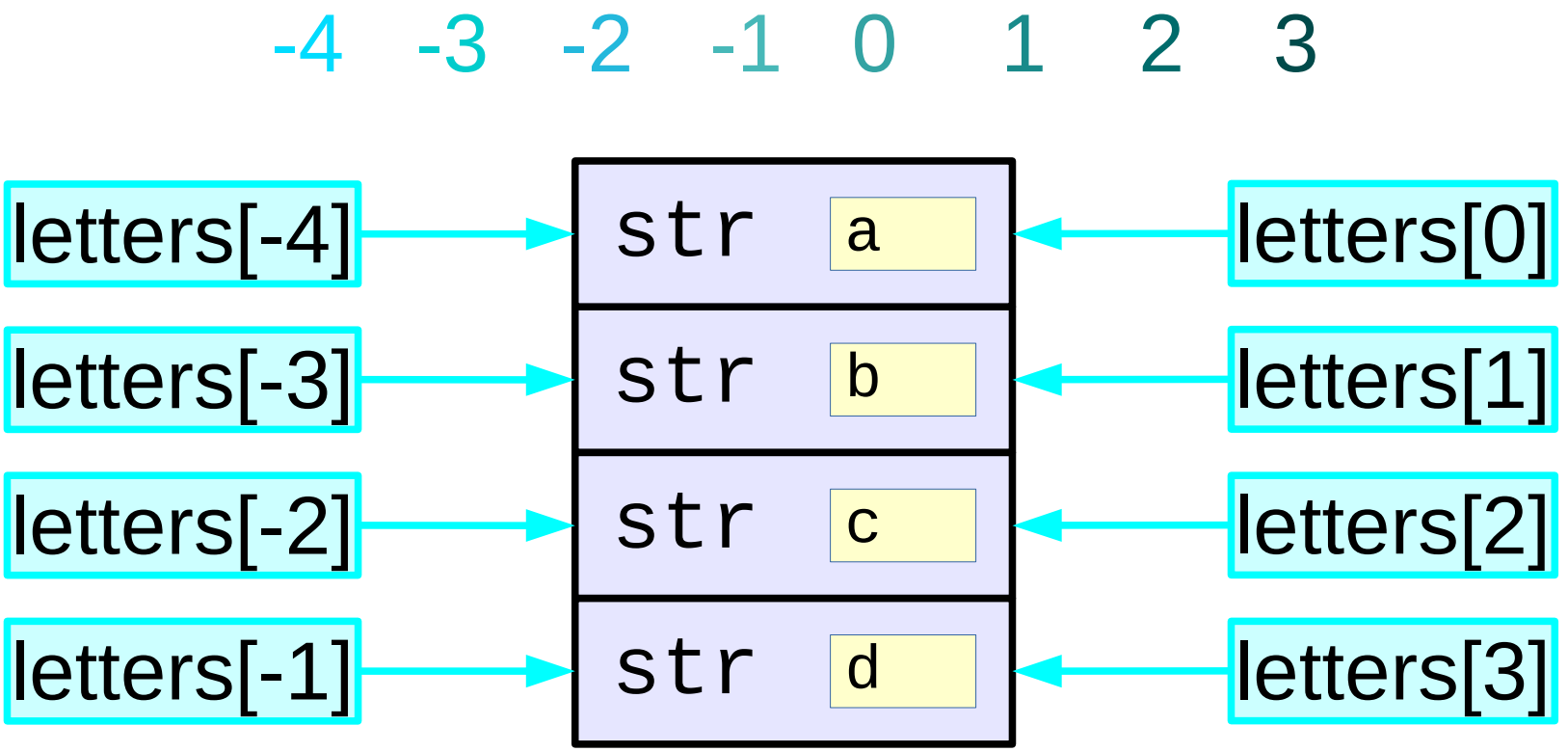


Going off the end

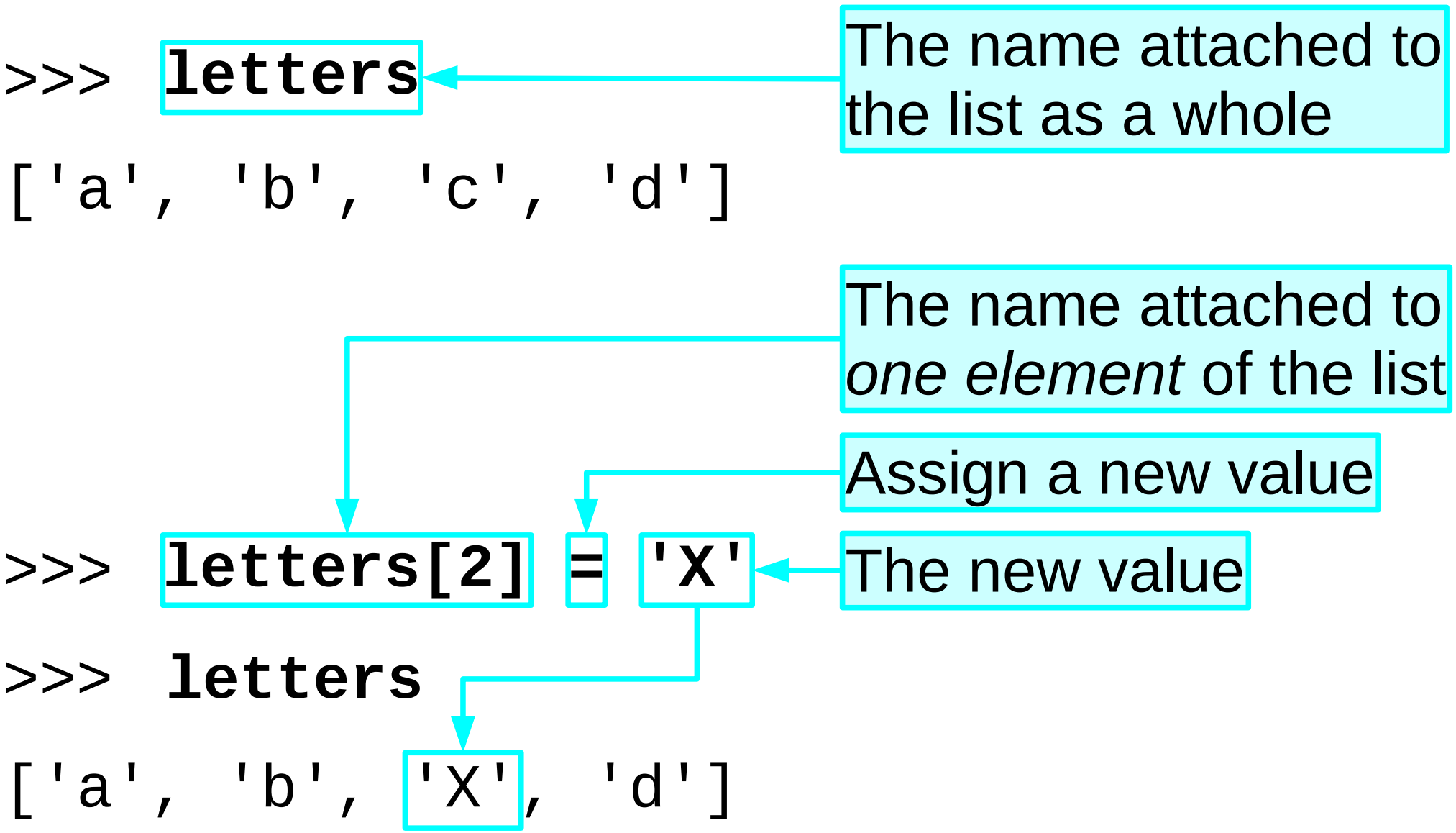
```
>>> letters[-5]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

Valid range of indices

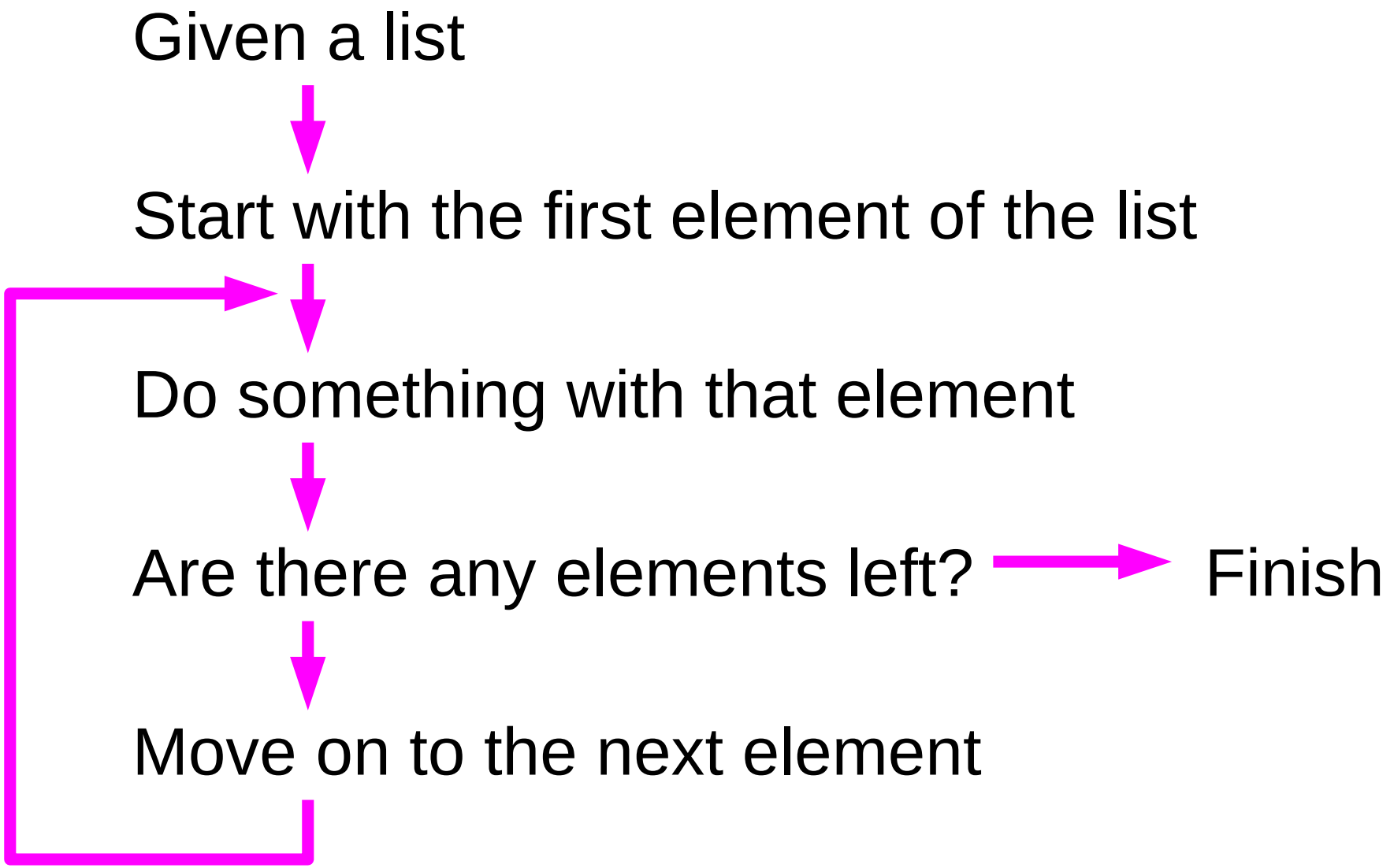
```
>>> len(letters)
4
```



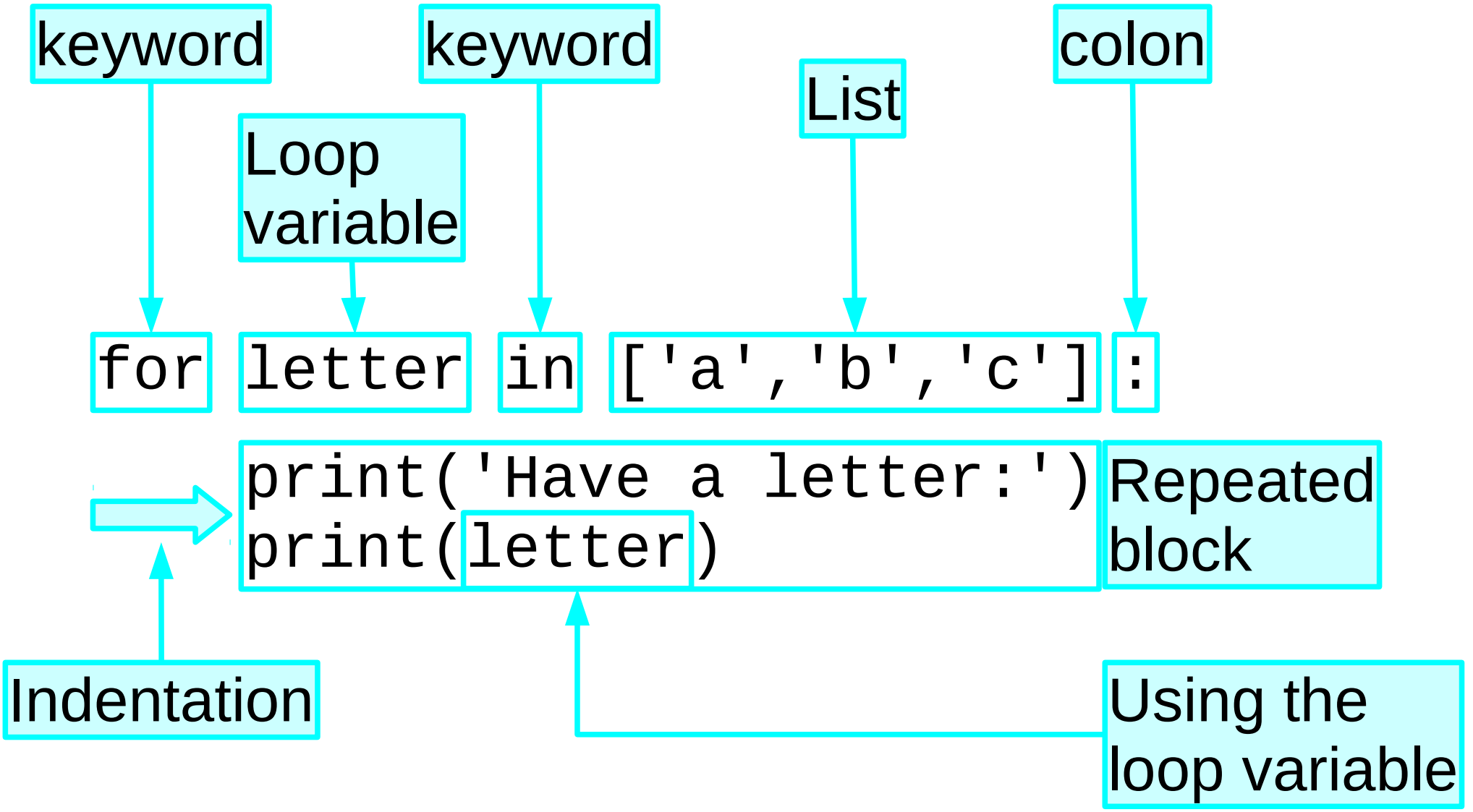
Assigning list elements



Doing something with a list



The “for loop”


keyword
keyword
List
colon
Loop variable
for letter in ['a','b','c'] :
print('Have a letter:')
print(letter)
Repeated block
Indentation
Using the loop variable

The “for loop”

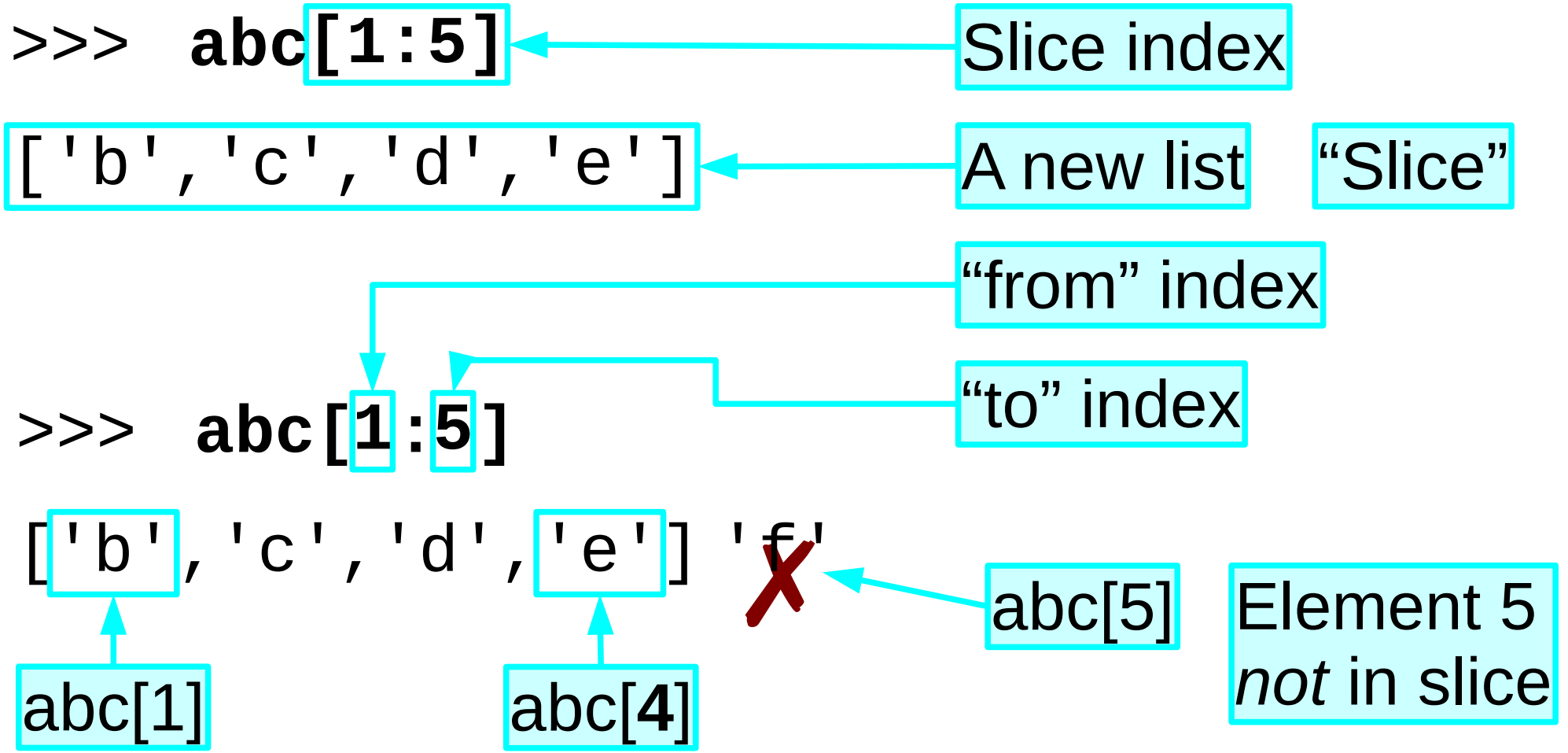
```
for letter in ['a','b','c']:
    print('Have a letter:')
    print(letter)
print('Finished!')
```

for1.py

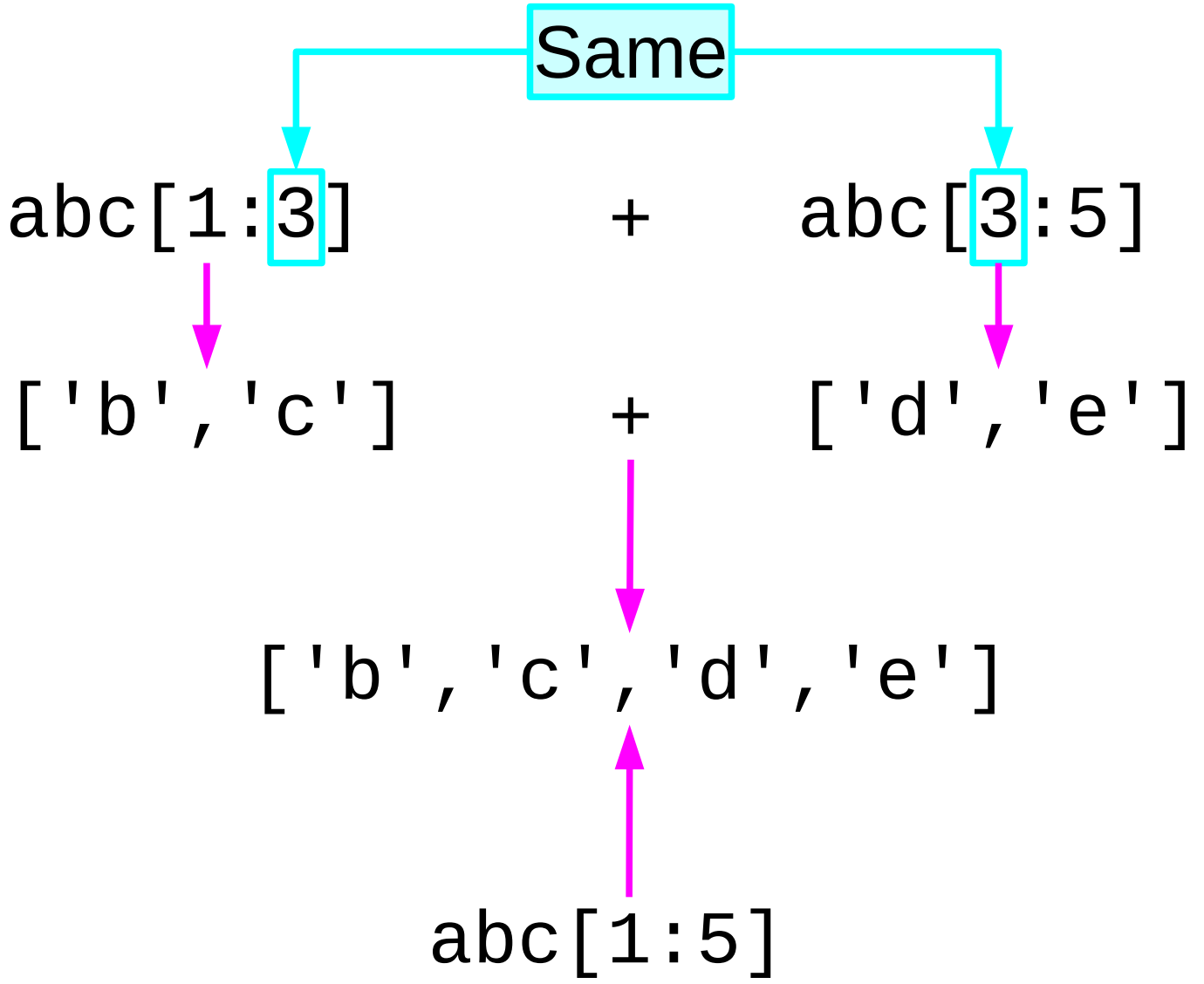
```
$  python for1.py
Have a letter:
a
Have a letter:
b
Have a letter:
c
Finished!
```

“Slices” of a list

```
>>> abc = ['a','b','c','d','e','f','g']

>>> abc[1:5]
['b','c','d','e']

>>> abc[1:5]
['b','c','d','e'] 'f'
```



Slice feature



Open-ended slices

```
>>> abc = ['a','b','c','d','e','f','g']
>>> abc[3:]
['d','e','f','g']
>>> abc[:5]
['a','b','c','d','e']
>>> abc[:]
['a','b','c','d','e','f','g']
```

Open ended at the end

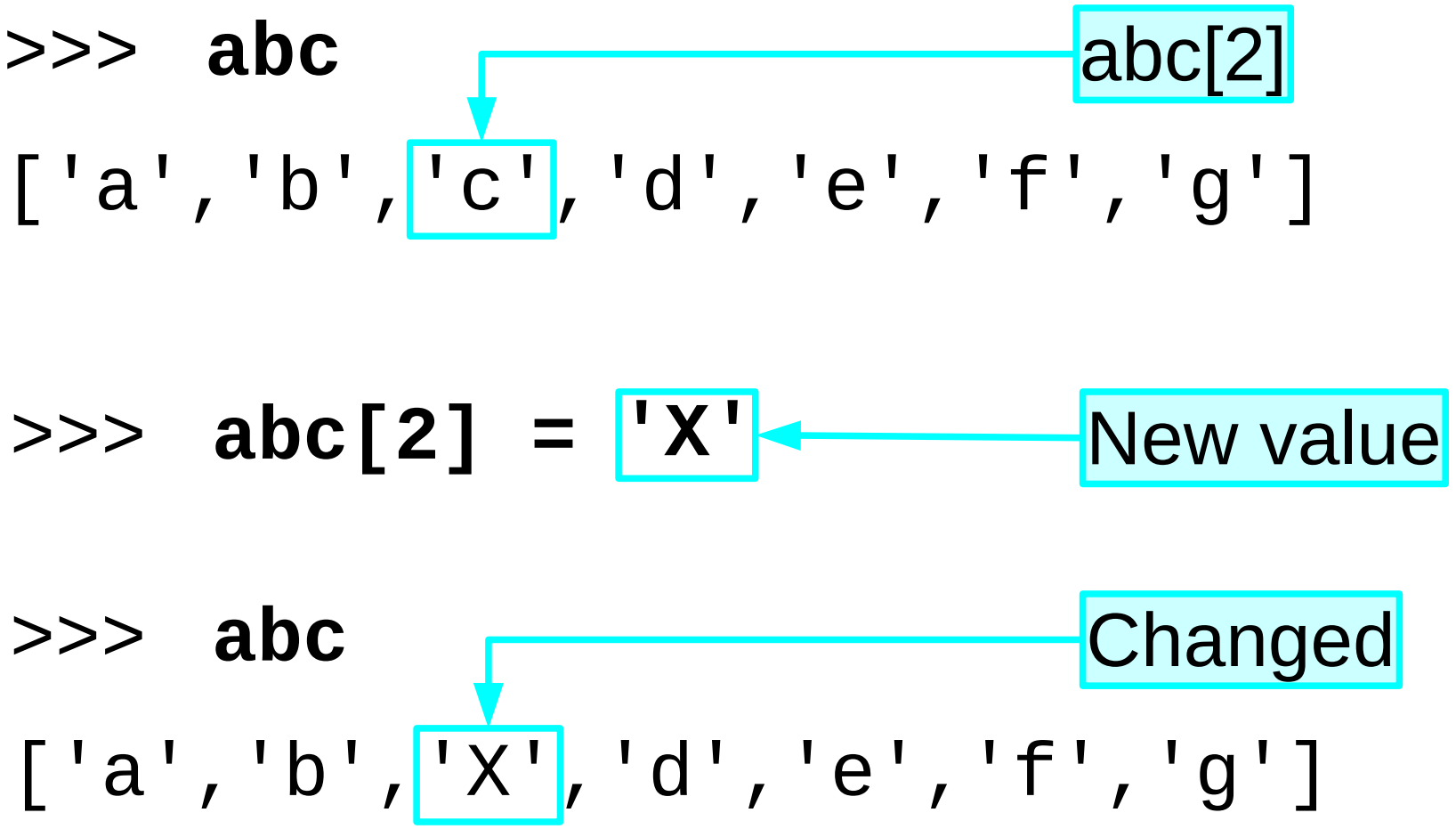
abc[3]

Open ended at the start

abc[4]

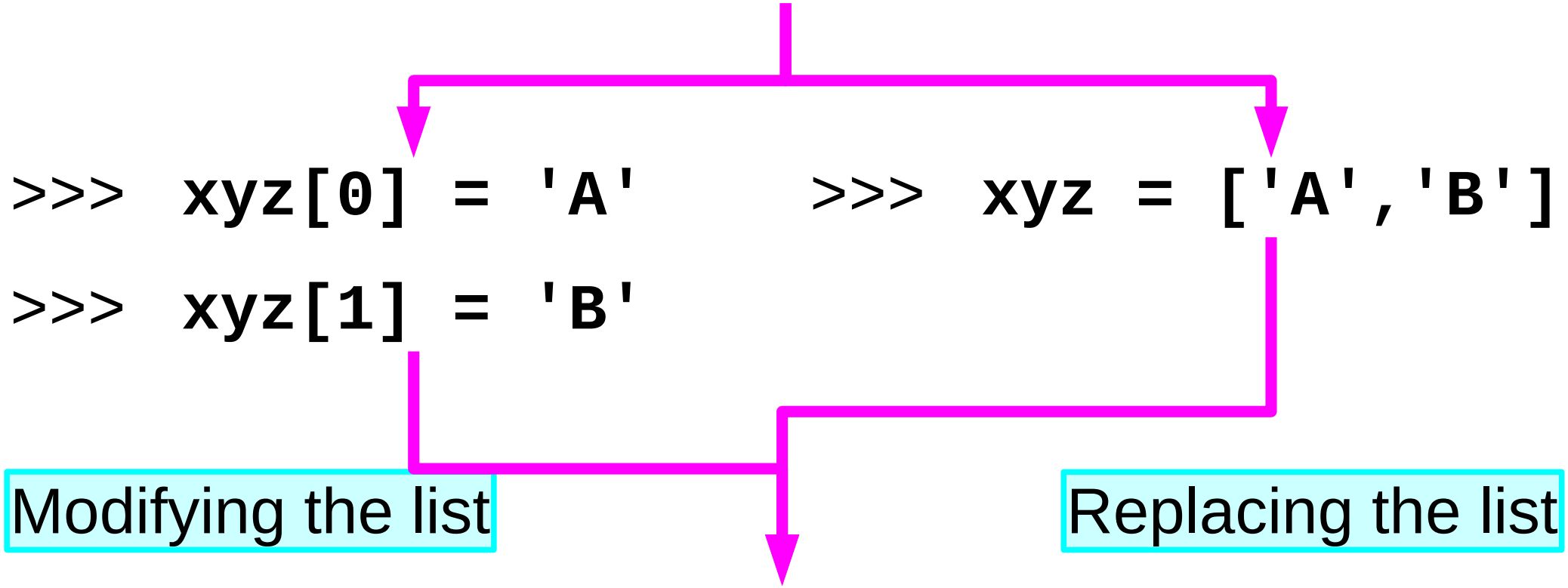
Open ended at *both* ends

Modifying lists — recap



Modifying *vs.* replacing ?

```
>>> xyz = ['x','y']
```



```
>>> xyz[0] = 'A'
>>> xyz[1] = 'B'
```

Modifying the list

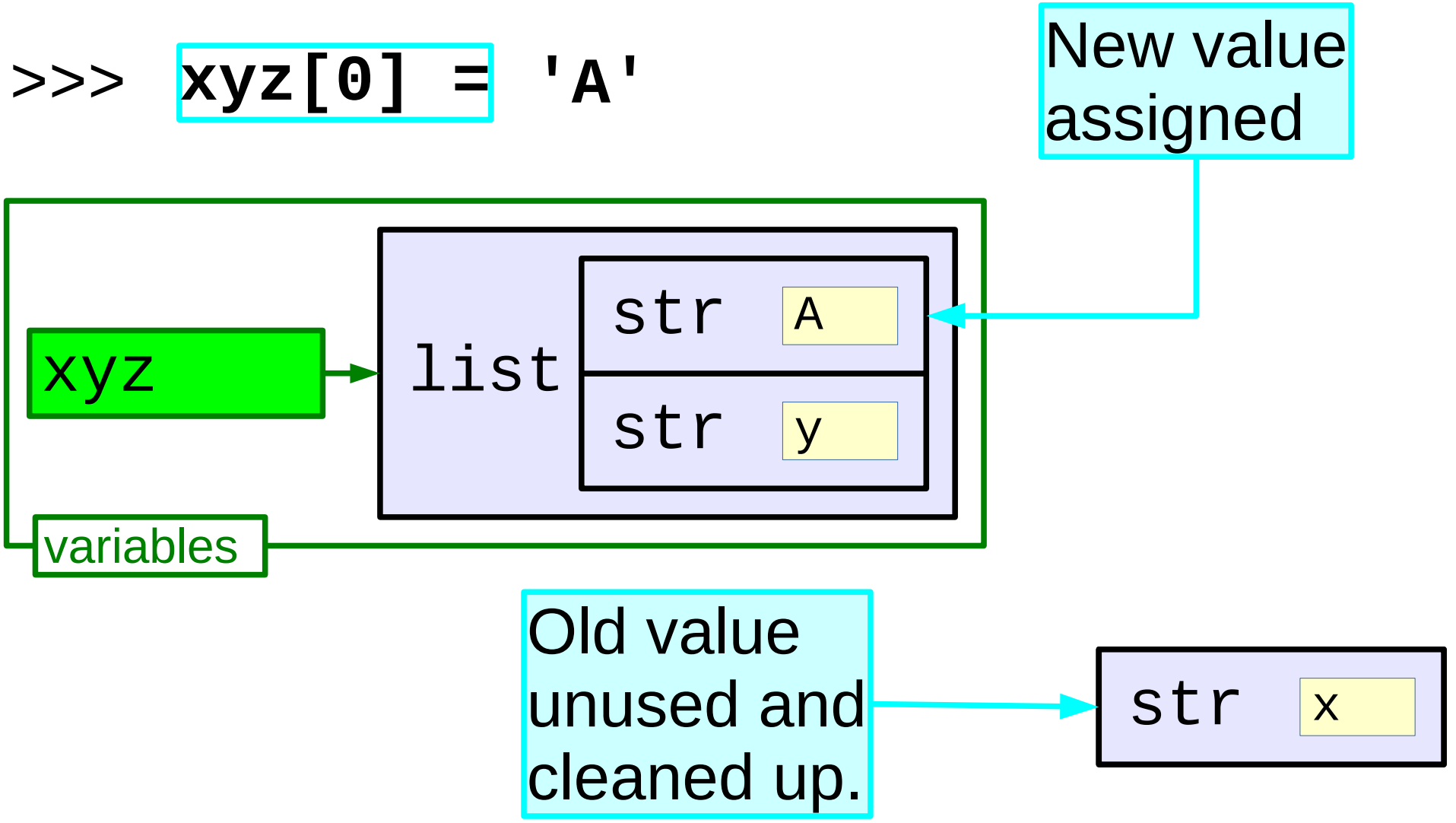
```
>>> xyz = ['A','B']
```

Replacing the list

```
>>> xyz
['A','B']
```

What's the difference? — 1

```
>>> xyz[0] = 'A'
```

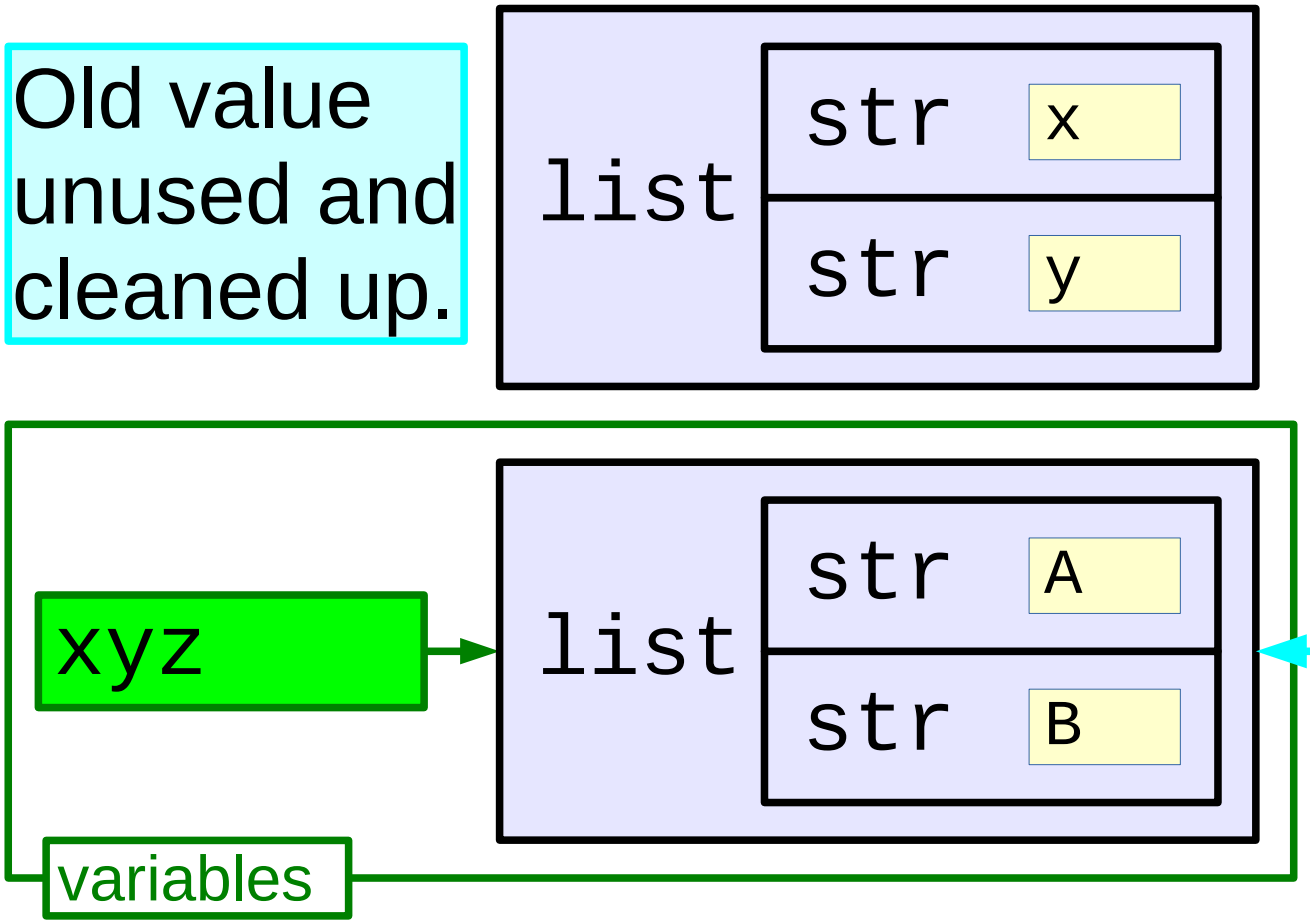


What's the difference? — 2

```
>>> xyz = ['A','B']
```

New value assigned

Old value unused and cleaned up.



What's the difference?

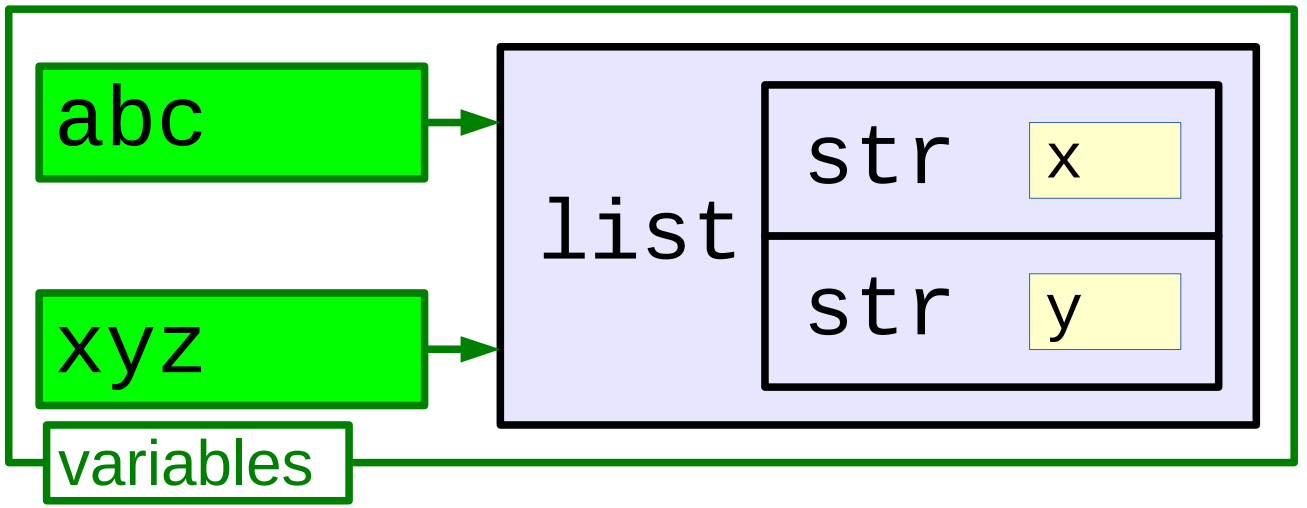
Modification: same list, different contents

Replacement: different list

? **Does it matter?**

Two names for the same list

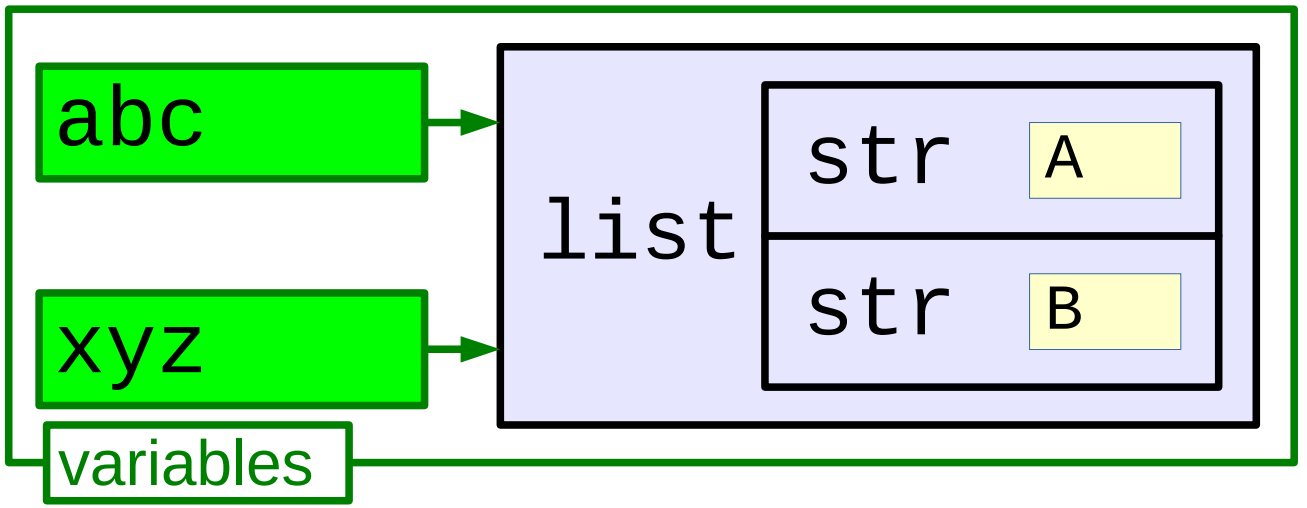
```
>>> xyz = ['x','y']
>>> abc = xyz
```



```
>>> abc[0] = 'A'
>>> abc[1] = 'B'
>>> xyz
['A', 'B']
```

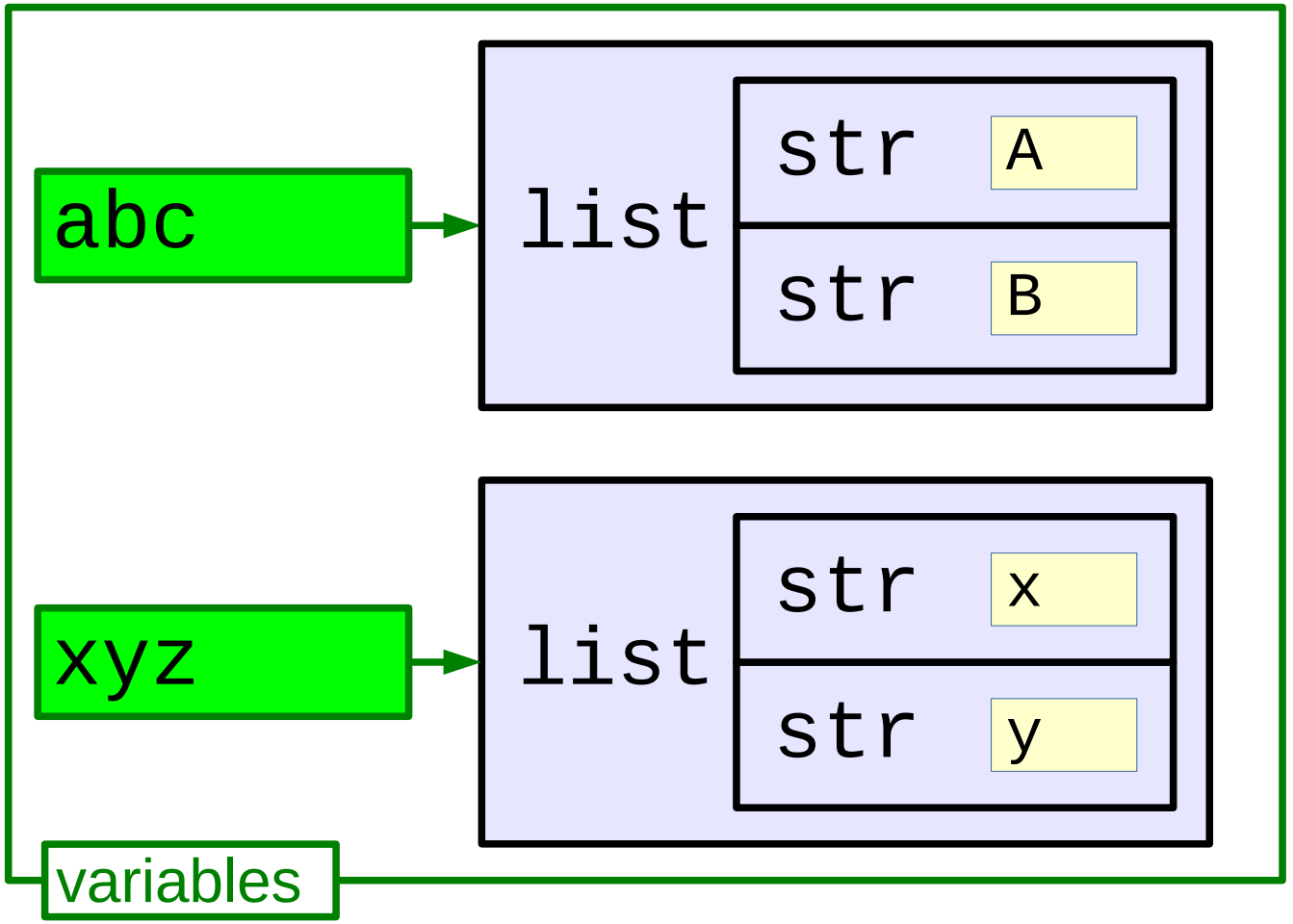
Modification

Modification



```
>>> abc = ['A','B']
>>> xyz
['x', 'y']
```

Replacement



One last trick with slices

```
>>> abc = ['a','b','c','d','e','f']
>>> abc[2:4]
['c','d']

>>> abc[2:4] = ['x',y','z']

>>> abc
['a','b','x',y','z','e','f']
```

Length 6

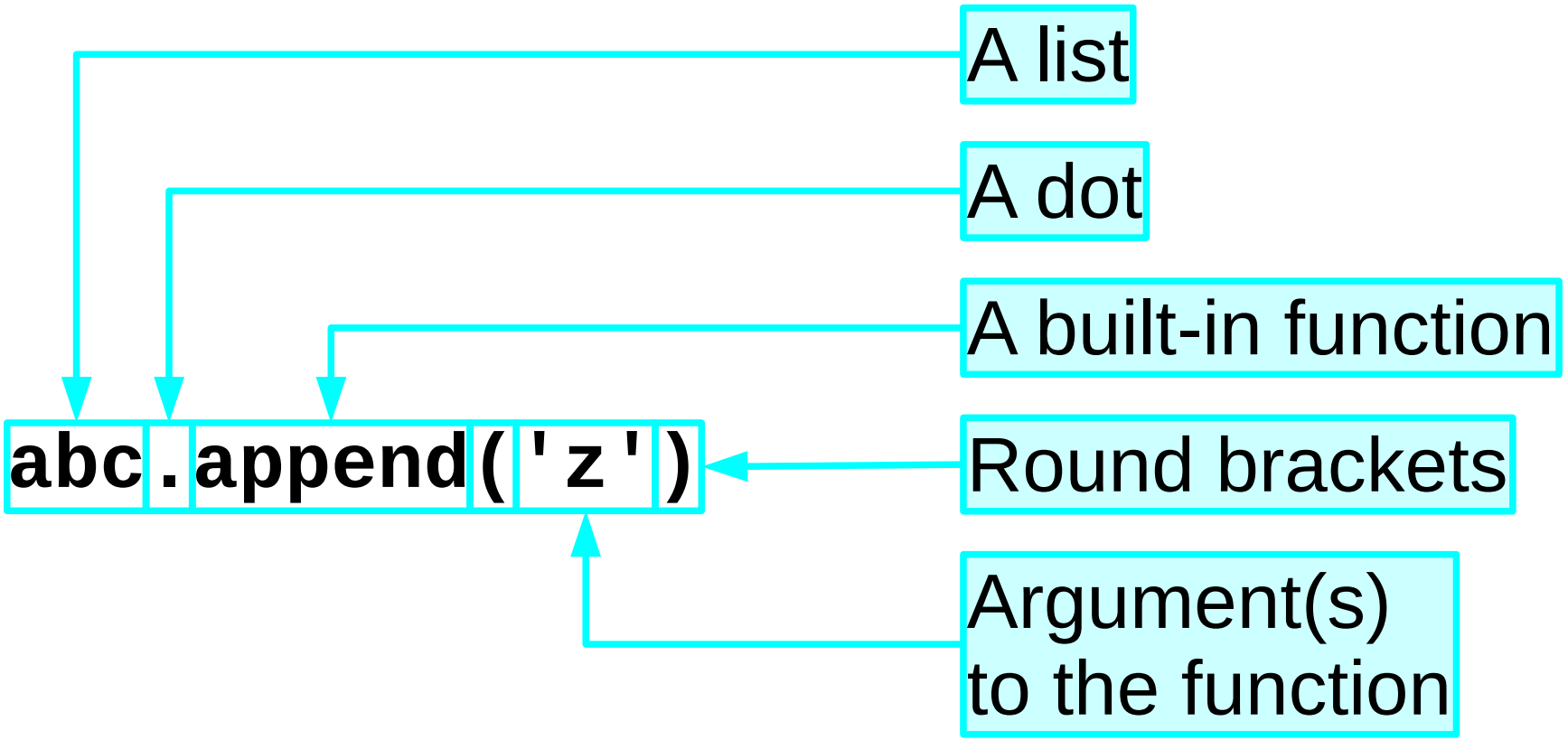
New length

Appending to a list

```
>>> abc = ['x','y']

>>> abc
['x','y']

>>> abc.append('z')

>>> abc
['x','y','z']
```

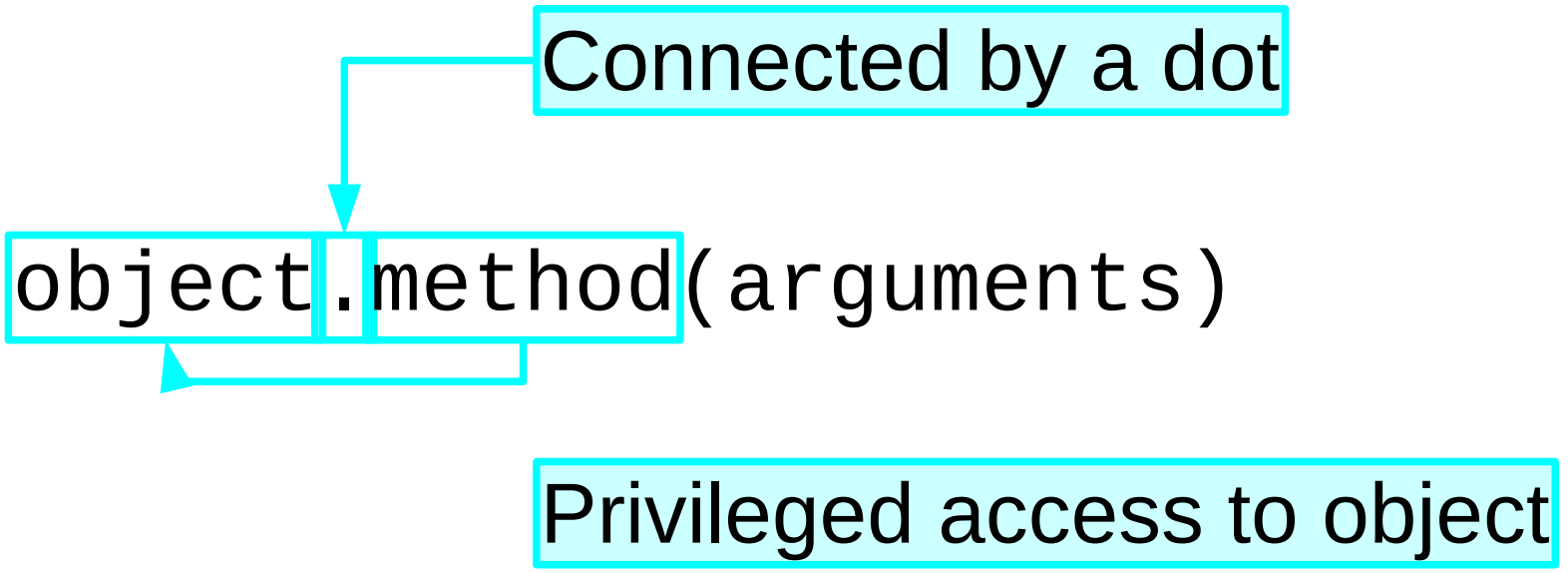
Add one element to the end of the list.

List “methods”



Built-in functions: “methods”

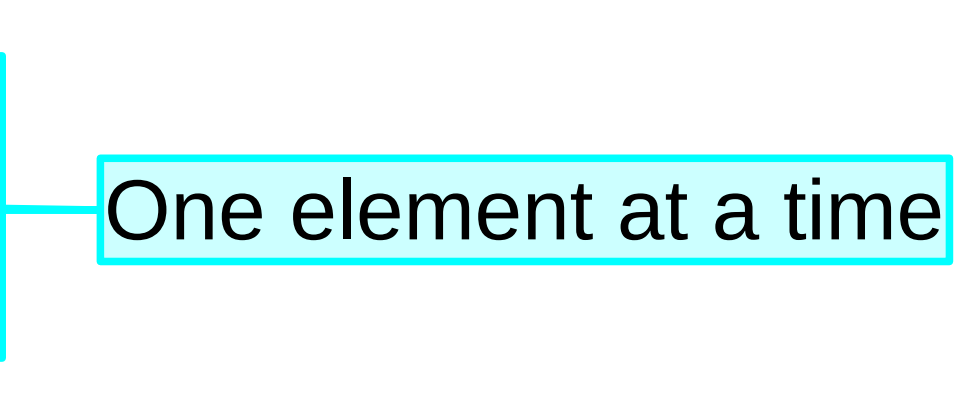
Methods



“Object-oriented programming”

The append() method

```
>>> abc = ['x','y','z']

>>> abc.append('A')
>>> abc.append('B')
>>> abc.append('C')

>>> abc
['x','y','z','A','B','C']
```

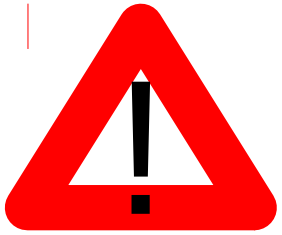


Beware!

```
>>> abc = ['x','y','z']

>>> abc.append(['A','B','C'])

>>> abc

['x', 'y', 'z', ['A','B','C']]
```

Appending a list

Get a list as the last item



“Mixed lists”

```
['x', 'y', 'z', ['A','B','C']]

['x', 2, 3.0]

['alpha', 5, 'beta', 4, 'gamma', 5]
```

The extend() method

```
>>> abc = ['x','y','z']
```

All in one go

```
>>> abc.extend(['A','B','C'])
```

rather unnecessary

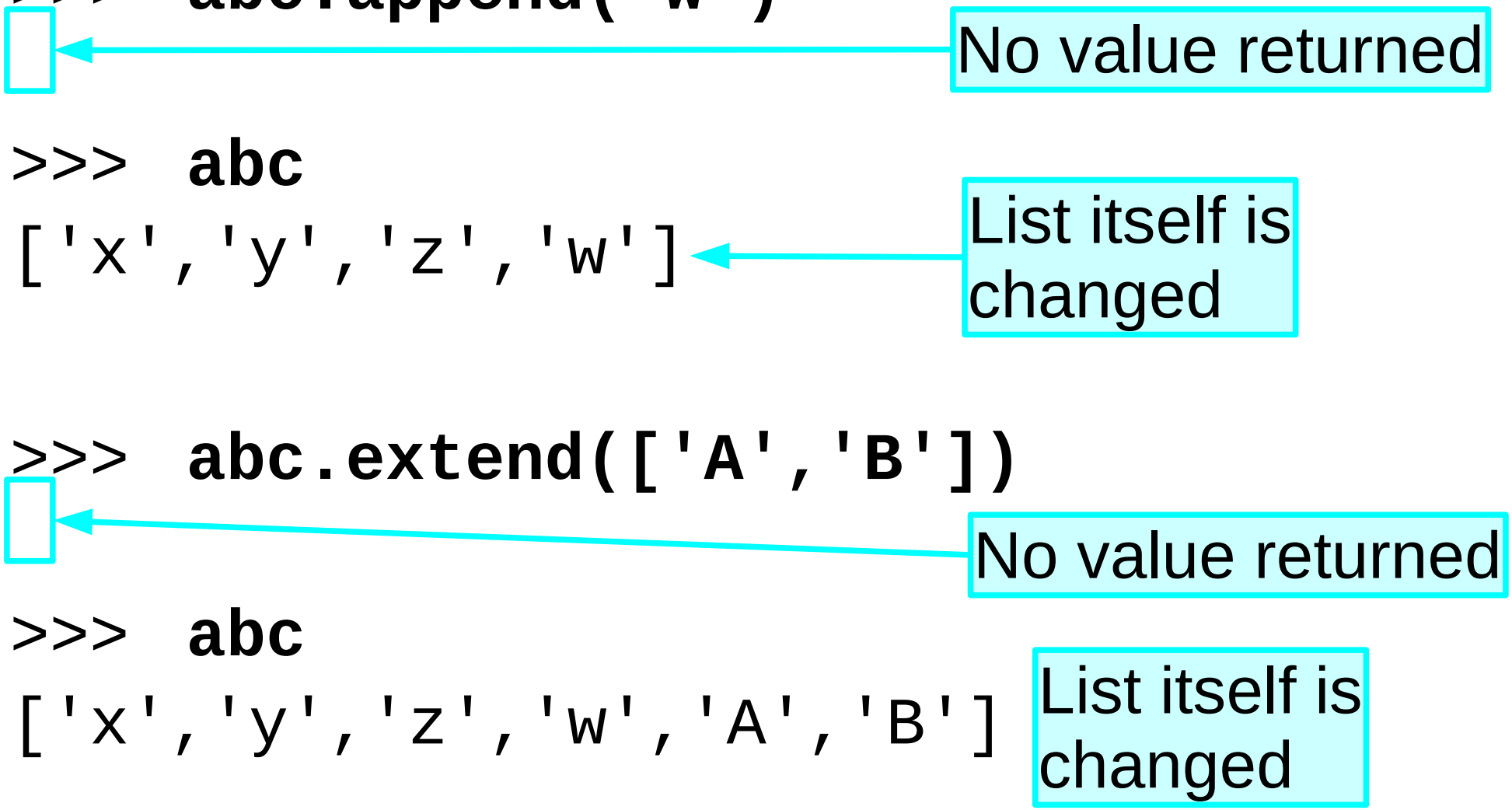
```
>>> abc
['x','y','z','A','B','C']
```

Avoiding extend()

```
>>> abc = ['x','y','z']

>>> abc = abc + ['A', 'B', 'C']

>>> abc
['x', 'y', 'z', 'A', 'B', 'C']
```

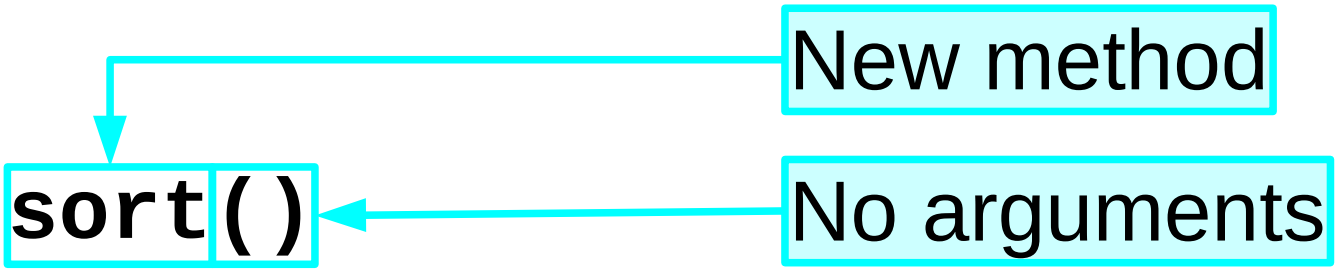
Changing the list “in place”

```
>>> abc.append('w')

>>> abc
['x','y','z','w']

>>> abc.extend(['A','B'])

>>> abc
['x','y','z','w','A','B']
```



Another list method: sort()

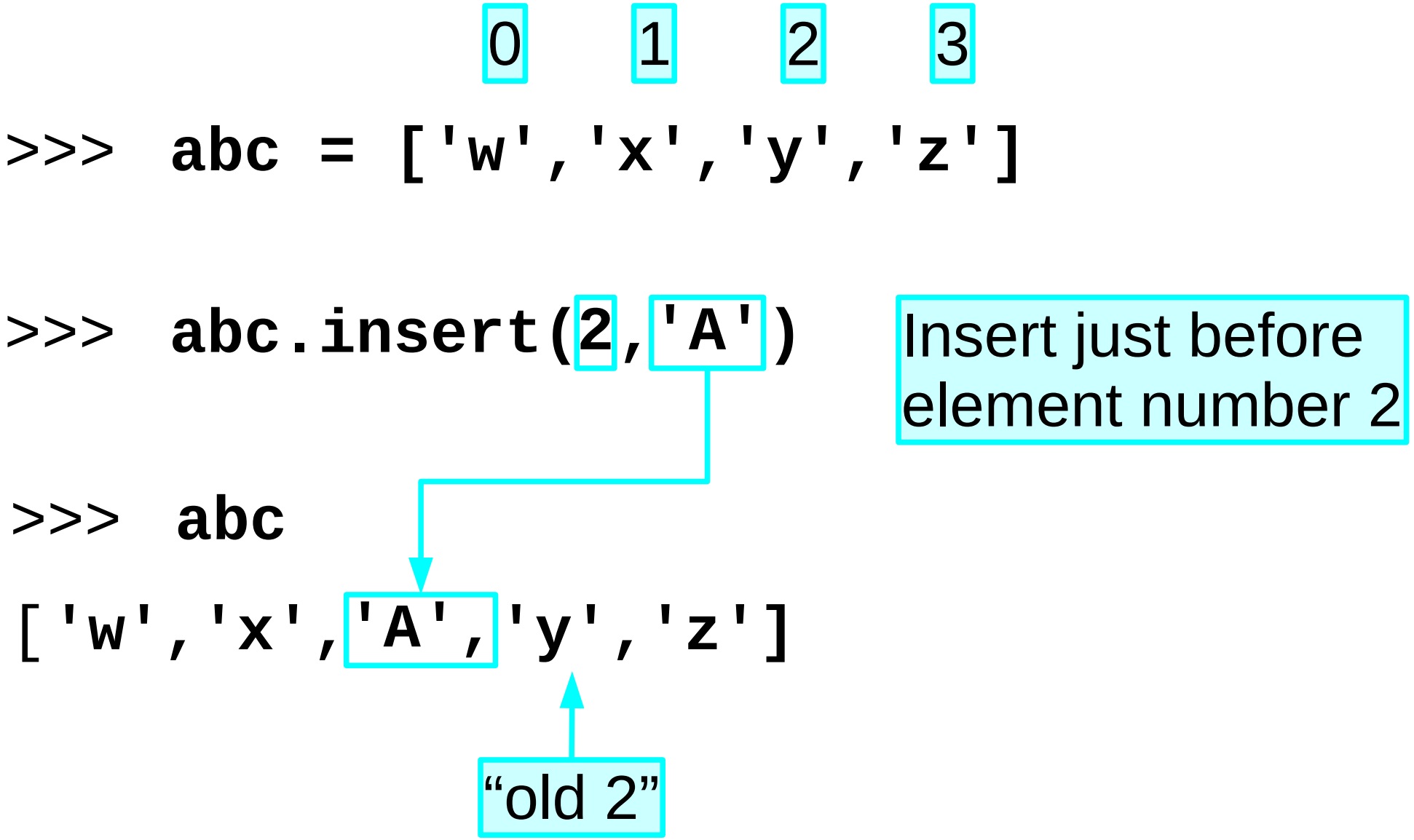
```
>>> abc = ['z','x','y']
>>> abc.sort()
>>> abc
['x','y','z']
```



Any type of sortable element

```
>>> abc = [3, 1, 2]
>>> abc.sort()
>>> abc
[1, 2, 3]

>>> abc = [3.142, 1.0, 2.718]
>>> abc.sort()
>>> abc
[1.0, 2.718, 3.142]
```

Another list method: insert()



Summary on lists

List methods:

Change the list itself

Don't return any result

```
list.append(item)
list.extend([item1, item2, item3])
list.sort()
list.insert(index, item)
```

Creating new lists

```
>>> numbers = [0,1,2,3,4,5,6,7,8,9]
>>> copy = []
>>> for number in numbers:
...     copy.append(number)
...
>>> copy
[0,1,2,3,4,5,6,7,8,9]
```

Simple copying

Creating new lists

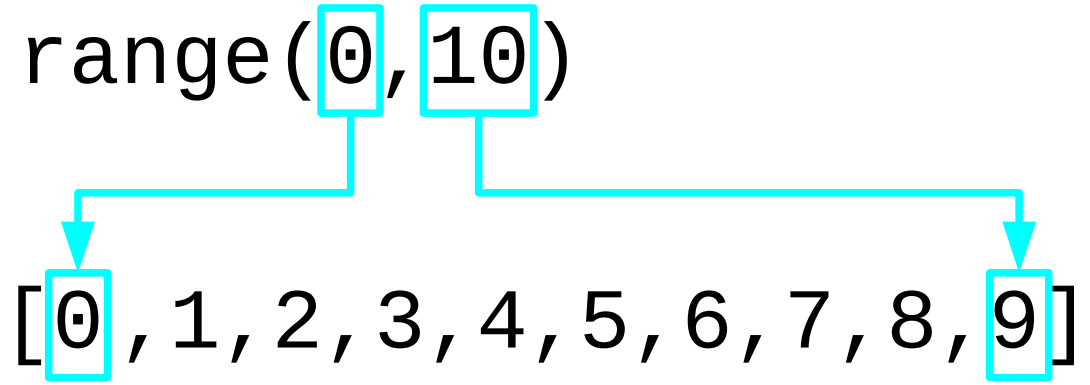
Boring!

```
>>> numbers = [0,1,2,3,4,5,6,7,8,9]
>>> squares = []
>>> for number in numbers:
...     squares.append(number**2)
...
>>> squares
[0,1,4,9,16,25,36,49,64,81]
```

Changing the value

Lists of numbers

```
>>> numbers = range(0,10)
>>> numbers
[0,1,2,3,4,5,6,7,8,9]
```

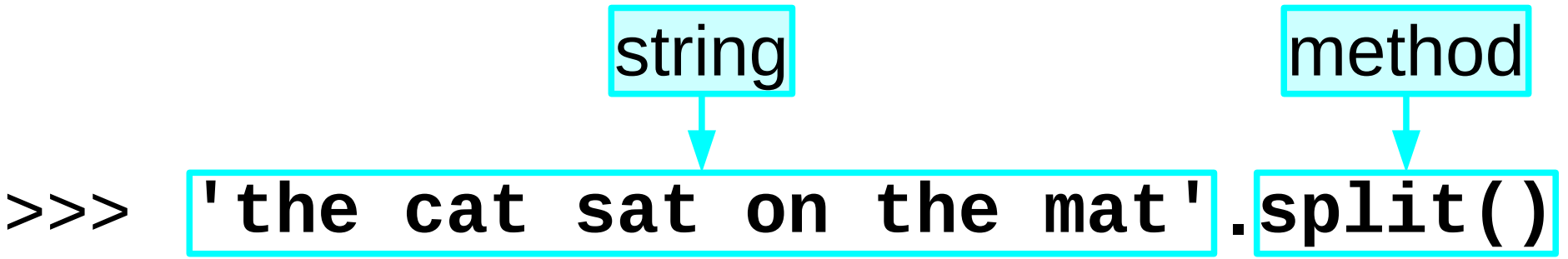


Creating new lists

Better!

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

Lists of words

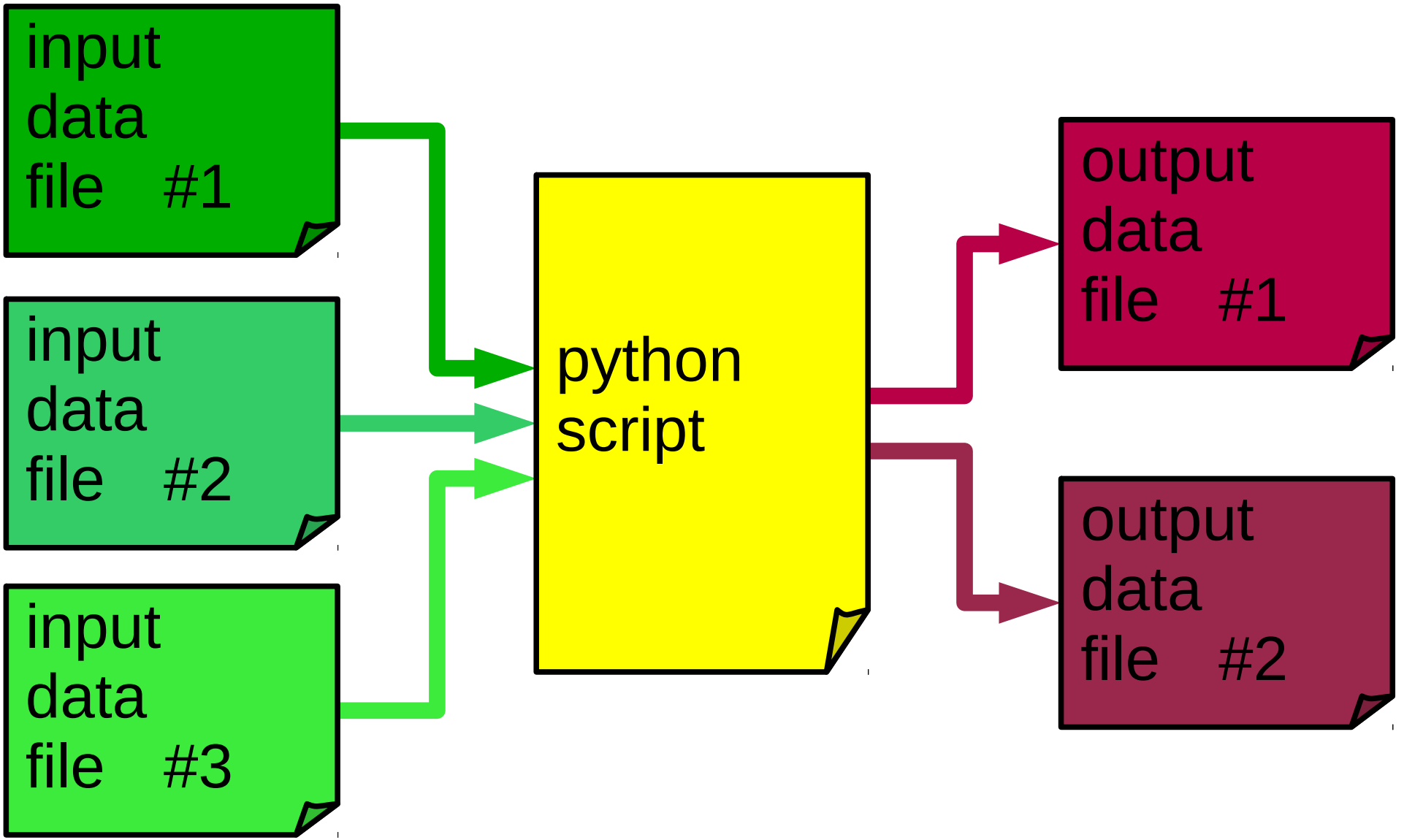



```
['the','cat','sat','on','the','mat']
```

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

No special handling for punctuation.

Files


input
data
file #1
input
data
file #2
input
data
file #3
python
script
output
data
file #1
output
data
file #2

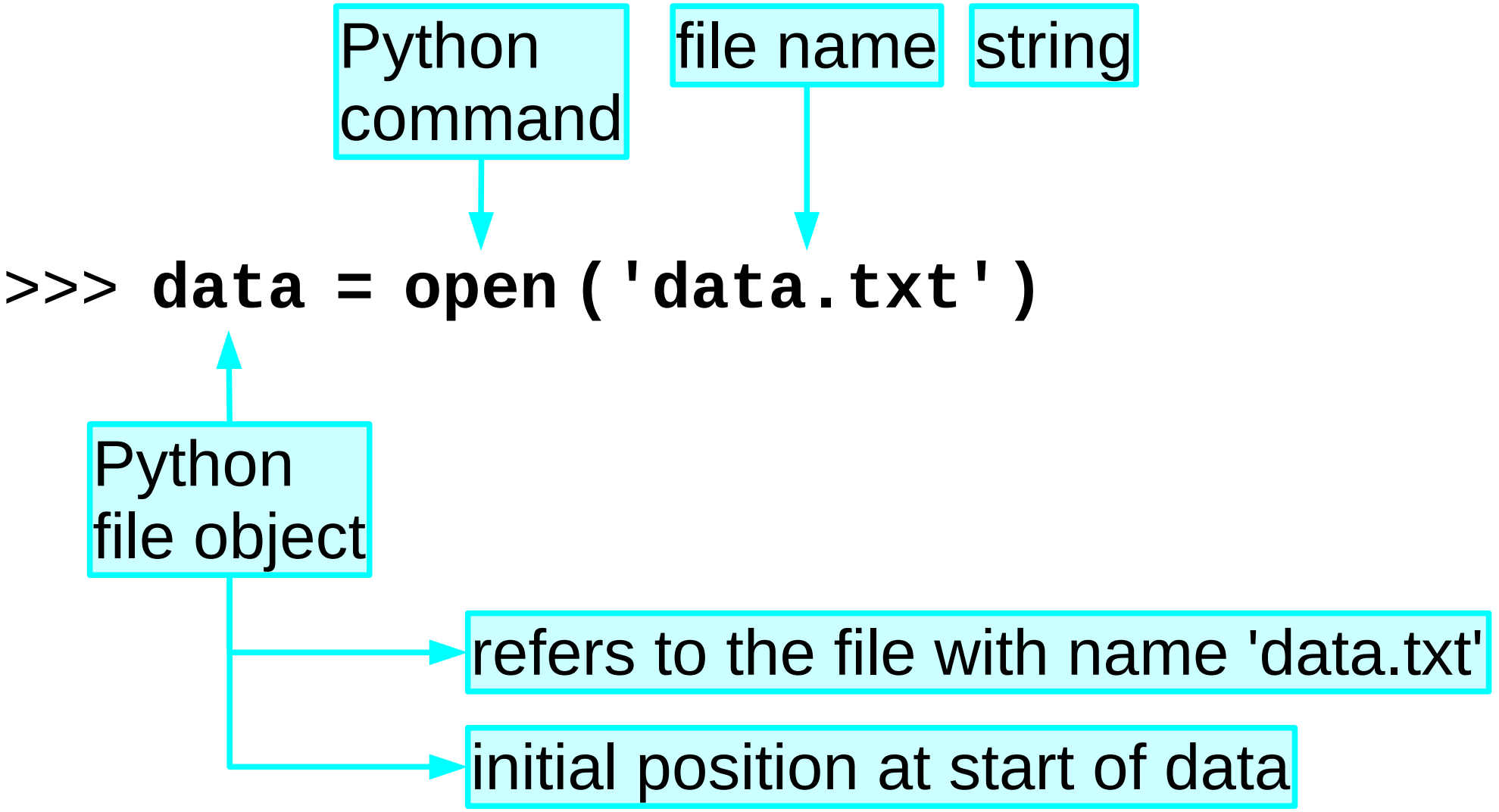
Reading a file

1. Opening a file

2. Reading from the file

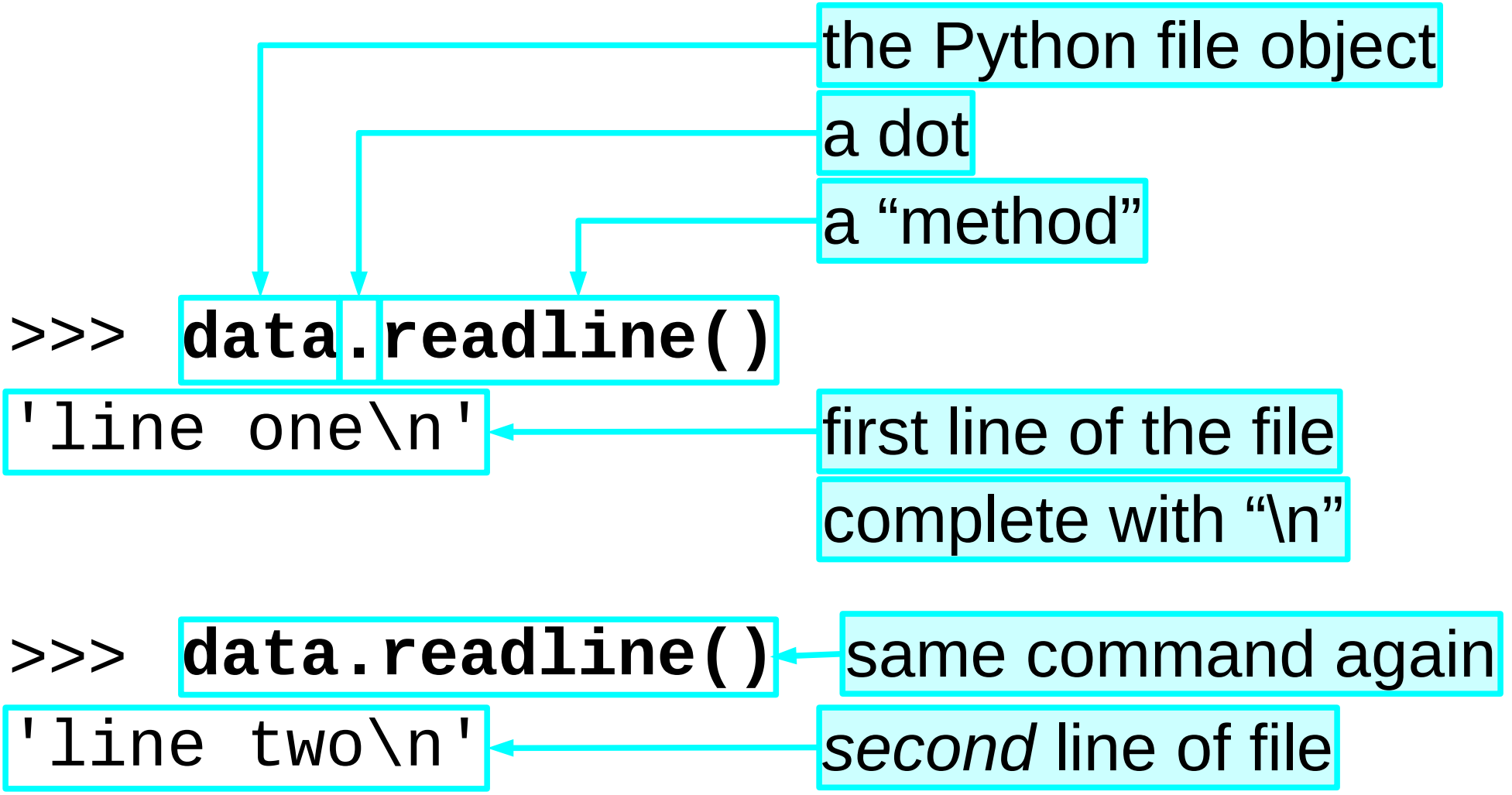
3. Closing the file

Opening a file

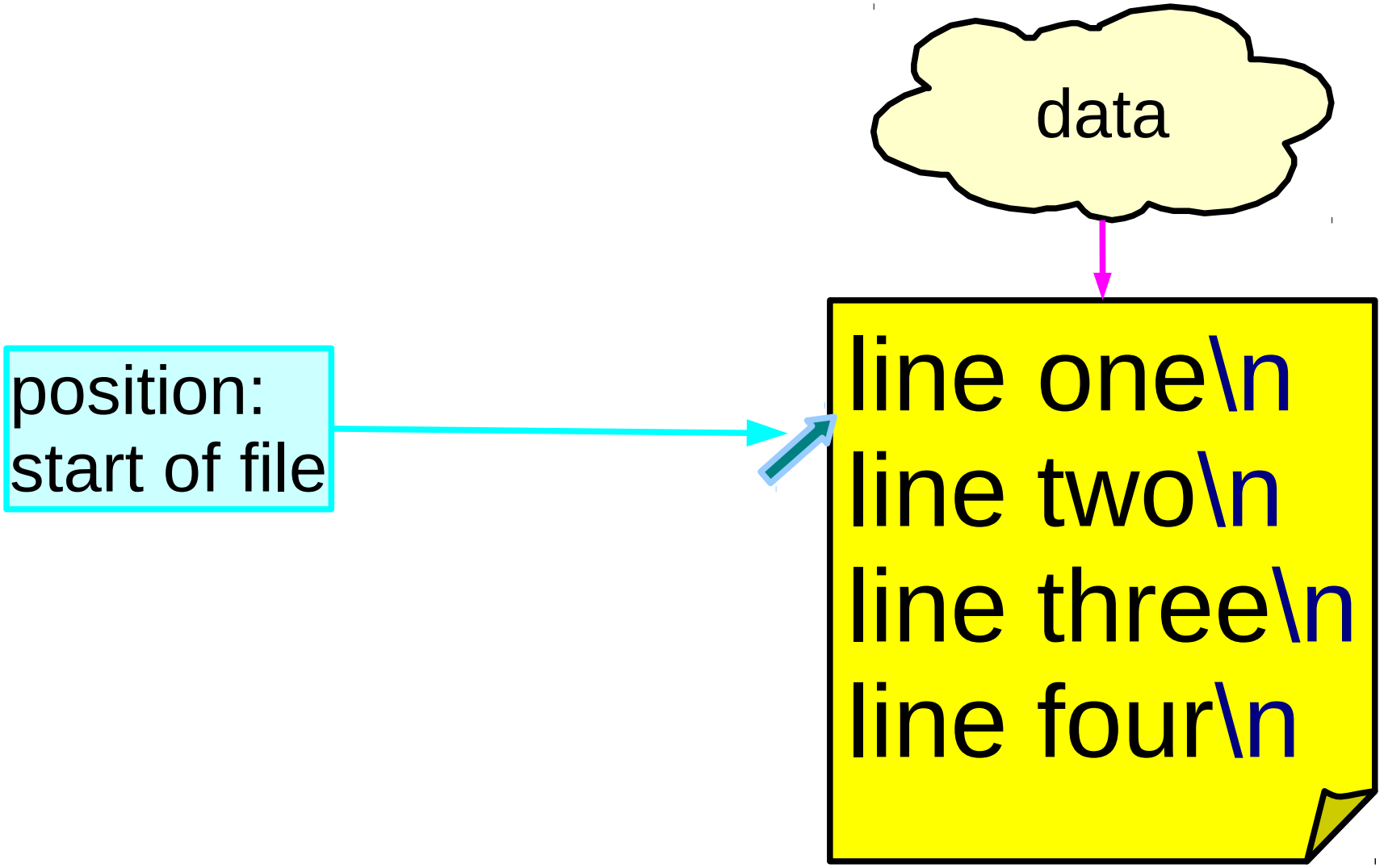


Reading from a file

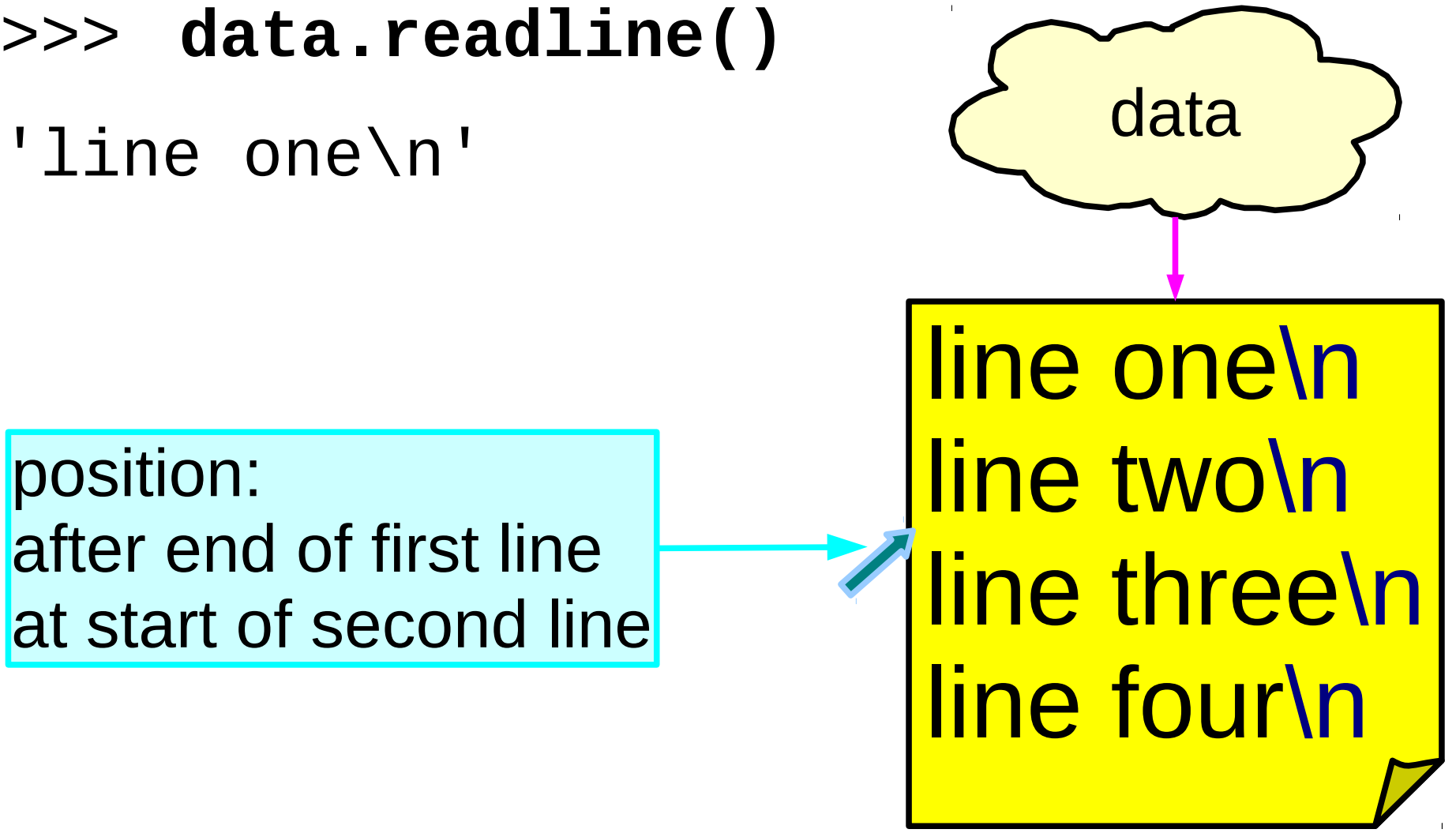
```
>>> data = open('data.txt')
```



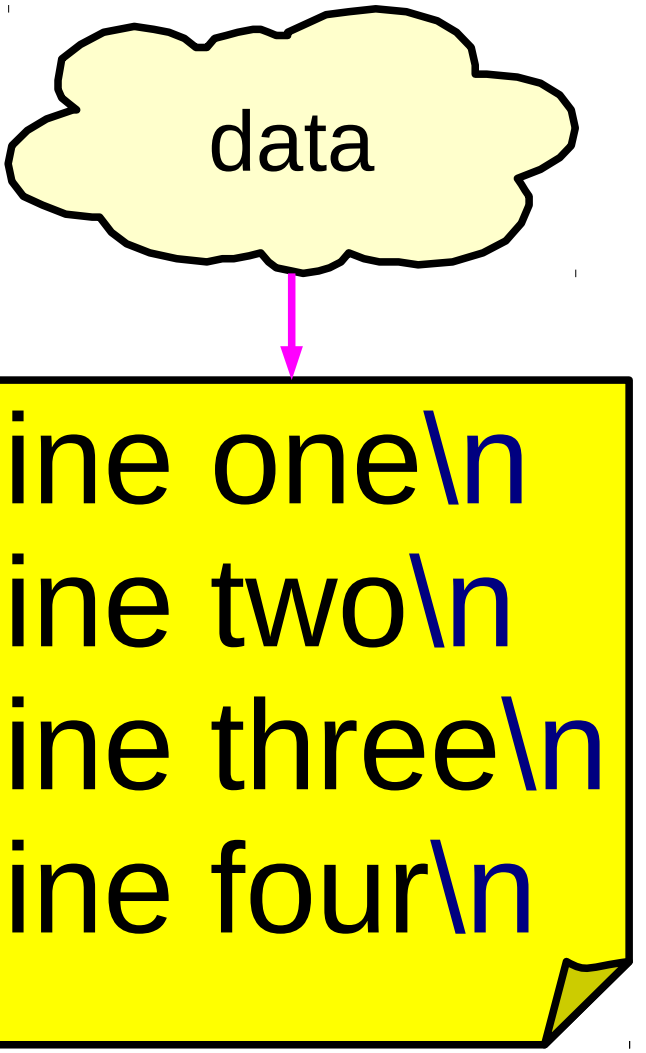
```
>>> data = open('data.txt')
```



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

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

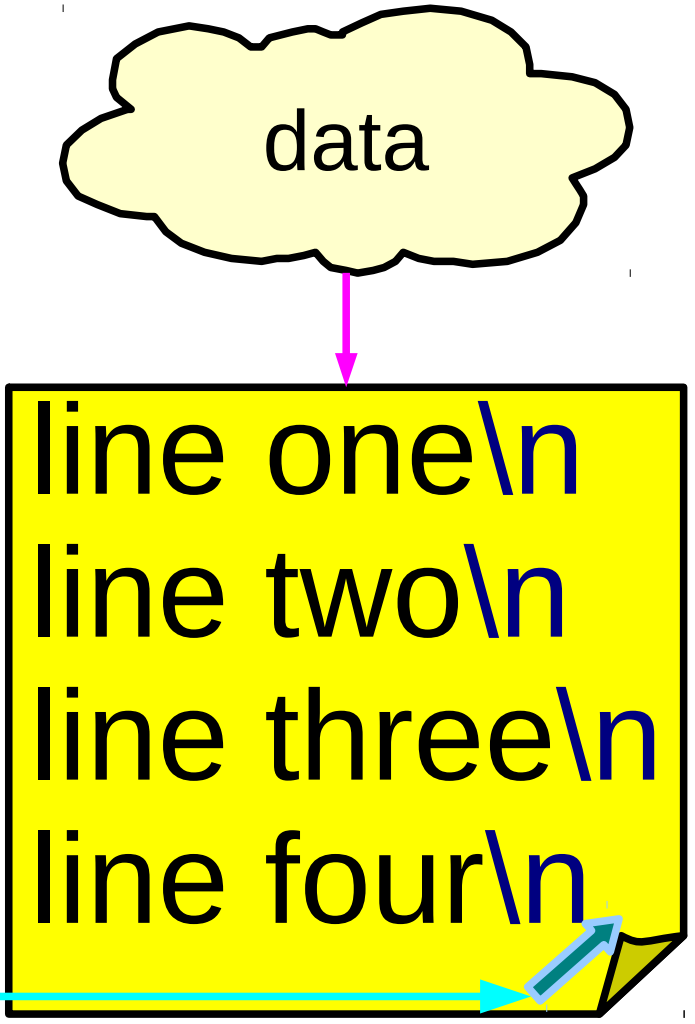


```
>>> data = open('data.txt')

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

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



after end of second line
at start of third line

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

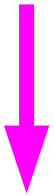


end of file

disconnect

Common trick

```
for line in data.readlines():
    stuff
```

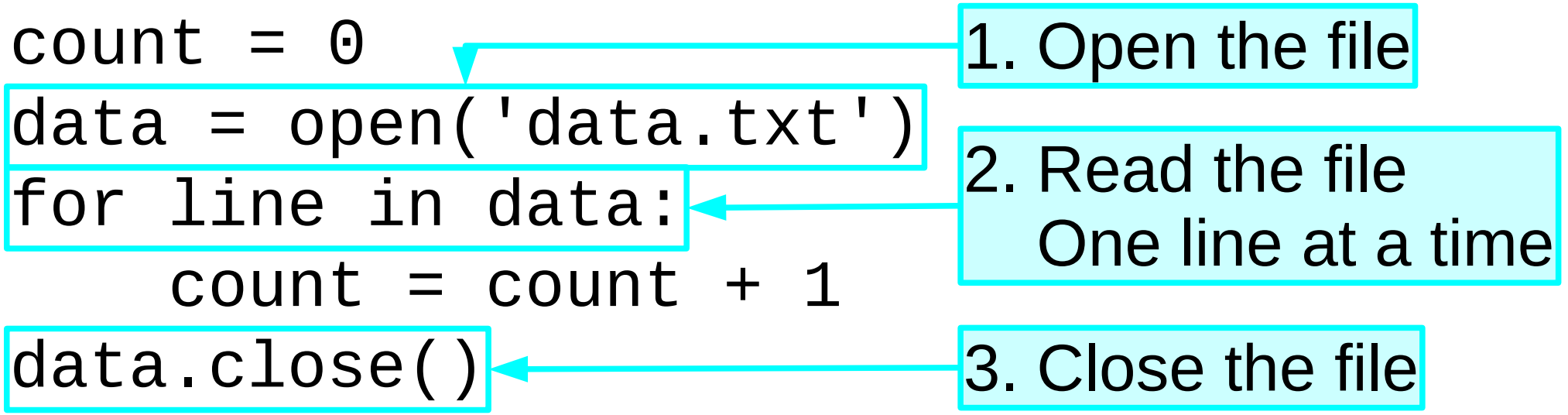


```
for line in data:
    stuff
```

Python “magic”: treat the file like a list and it will behave like a list

Simple example script

```
count = 0
data = open('data.txt')
for line in data:
    count = count + 1
data.close()
print(count)
```



Exercise

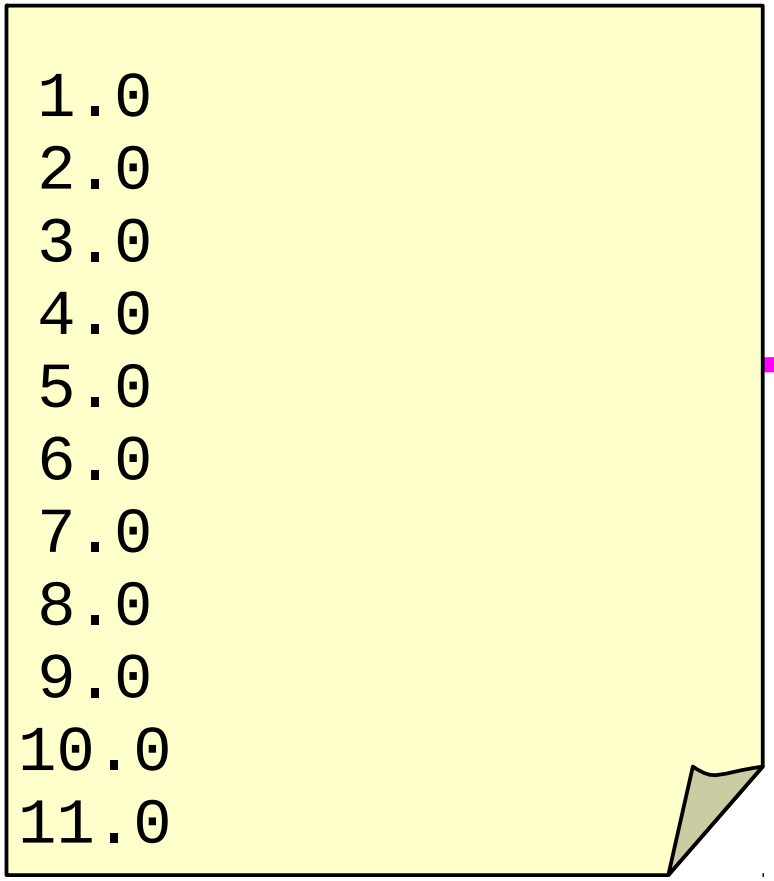
Write a **script** counting.py from scratch to do this:

Open the file file.txt.
Set three counters equal to zero:
    n_lines, n_words, n_chars
Read the file line by line.
For each line:
    increase n_lines by 1
    increase n_chars by the length of the line
    split the line into a list of words
    increase n_words by the length of the list
Close the file.
Print the three counters.

15 minutes

Converting the type of input

Problem:




```
['1.0\n','2.0\n',
 '3.0\n','4.0\n',
 '5.0\n','6.0\n',
 '7.0\n','8.0\n',
 '9.0\n','10.0\n',
 '11.0\n']
```

List of strings, not a list of numbers.

Type conversions

```
>>> float('1.0\n')
1.0
```

String → Float

```
>>> str(1.0)
'1.0'
```

Float → String

No newline

```
>>> float(1)
1.0
```

Int → Float

```
>>> int(-1.5)
-1
```

Float → Int

Rounding to zero

Type conversions to lists

```
>>> list('hello')
```

String → List

```
['h','e','l','l','o']
```

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

File → List

```
['line one\n', 'line two\n',
 'line three\n', 'line four\n']
```

Example script

```
sum = 0.0
data = open('numbers.dat')
for line in data:
    sum = sum + float(line)
data.close()
print sum
```

Writing to a file

```
output = open('output.txt' )
```

Default

Equivalent

```
output = open('output.txt','r')
```

Open for reading

```
output = open('output.txt','w')
```

Open for writing

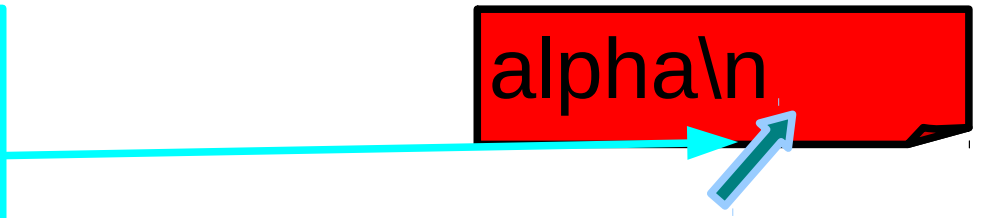
```
>>> output =  open('output.txt','w')
>>> output.write('alpha\n')
```

Method to write a lump of data

Lump of data

"Lump": need not be a line.

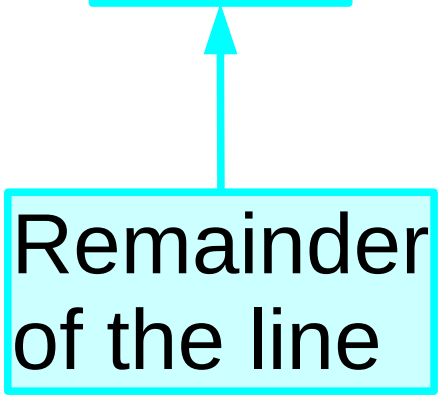
Current position changed



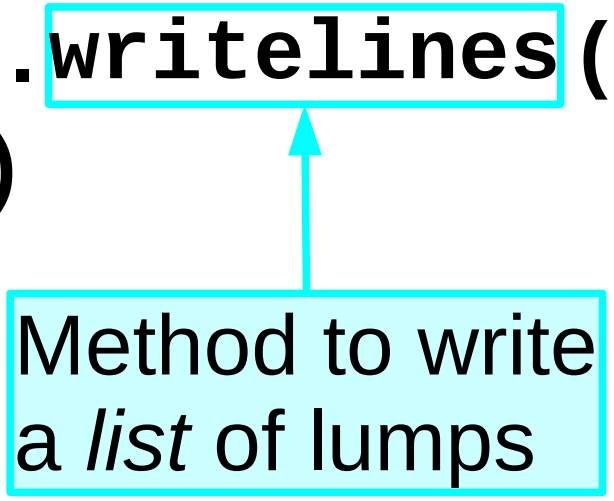
```
>>>  output =  open('output.txt','w')
>>>  output.write('alpha\n')
>>>  output.write('bet')
```

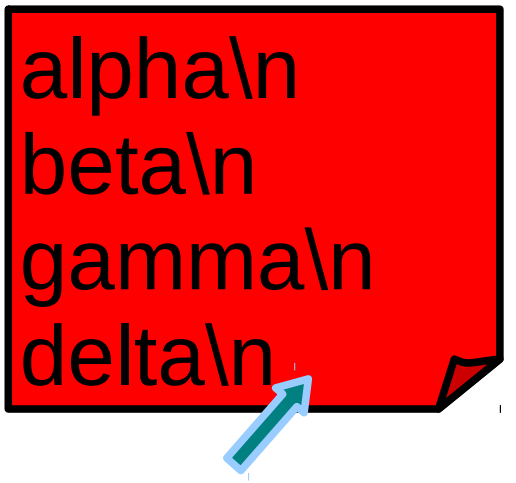
Lump of data to be written

```
>>> output =  open('output.txt','w')
>>> output.write('alpha\n')
>>> output.write('bet')
>>> output.write('a\n')
```



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

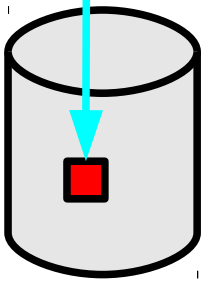




```
>>> output =  open('output.txt','w')
>>> output.write('alpha\n')
>>> output.write('a\n')
>>> output.writelines(['gamma\n',
'delta\n']
>>> output.close()
```

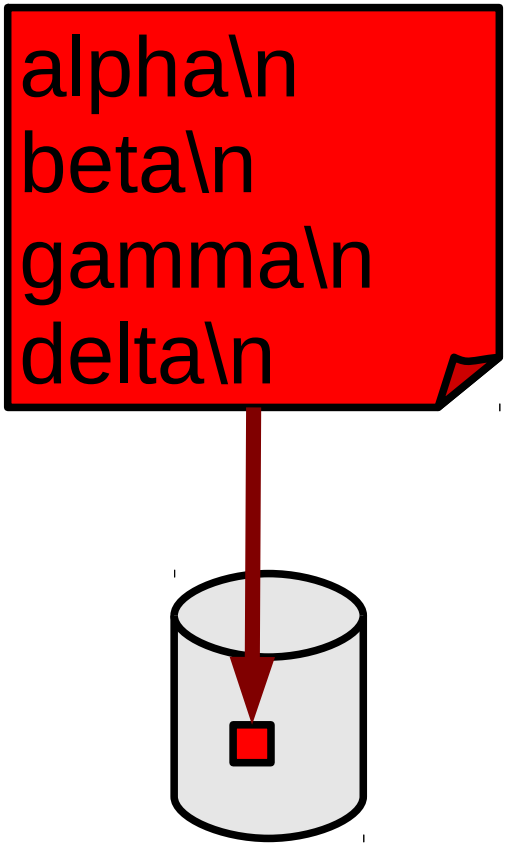
Python is done with this file.

Data may not be written to disc until close()!



Only on close() is it guaranteed that the data is on the disc!

```
>>> output.close()
```

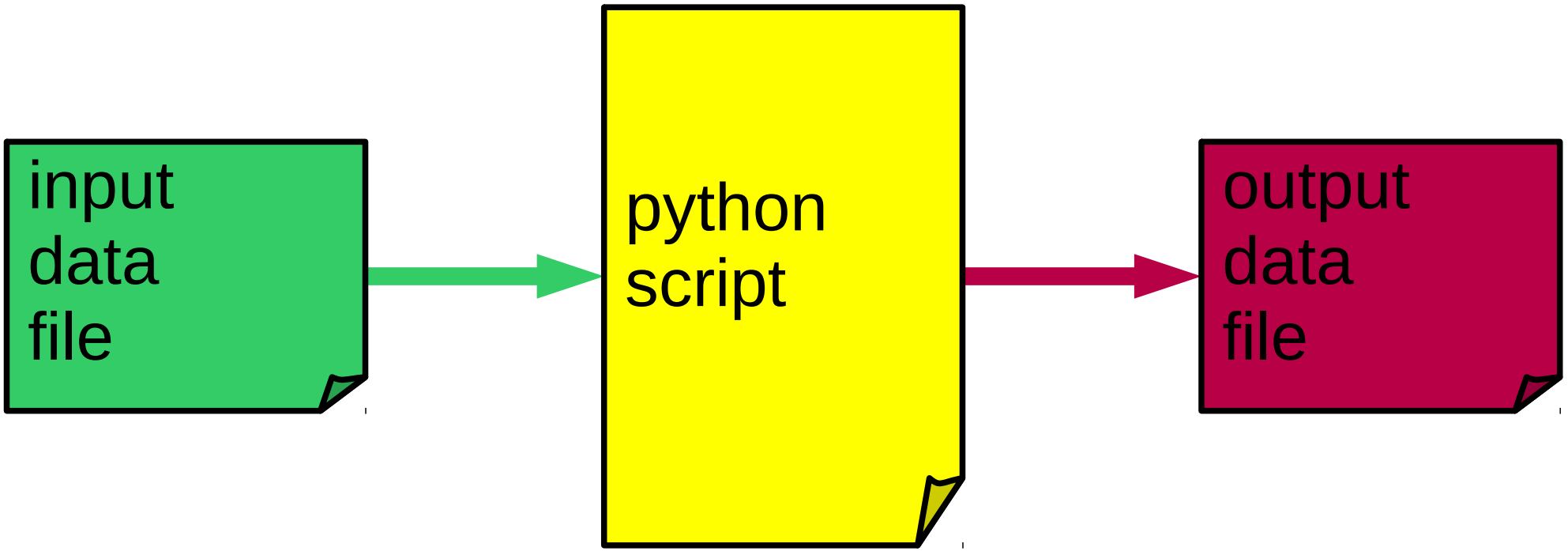


Example

```
output = open('output.txt', 'w')
output.write('Hello, world!\n')
output.close()
```

Example of a “filter”

Reads one file, writes another.



Example of a “filter”

```
input  = open('input.dat',  'r')
output = open('output.dat', 'w')
line_number = 0

for line in input:
    line_number = line_number + 1
    words = line.split()
    output.write('Line ')
    output.write(str(line_number))
    output.write(' has ')
    output.write(str(len(words)))
    output.write(' words.\n')

input.close()
output.close()
```

Setup

Ugly!

Shutdown

filter1.py

Exercise

Change counting.py to do this:

Read `file.txt` and write `file.out`.

For each line write to the output:

- line number
- number of words on the line
- number of characters in the line

separated by TABs.

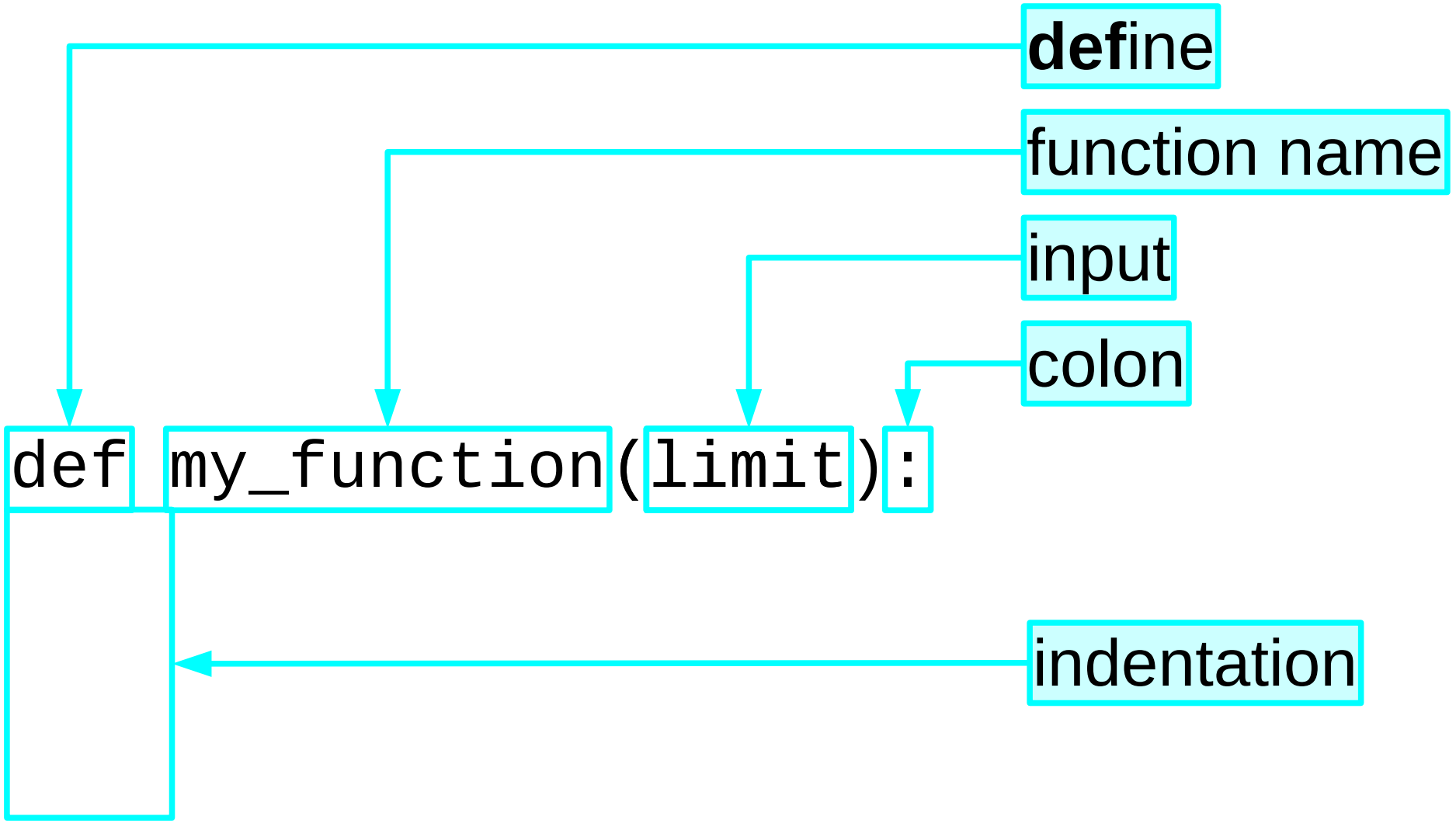
At the end output a summary line

- number of lines
- total number of words
- total number of characters

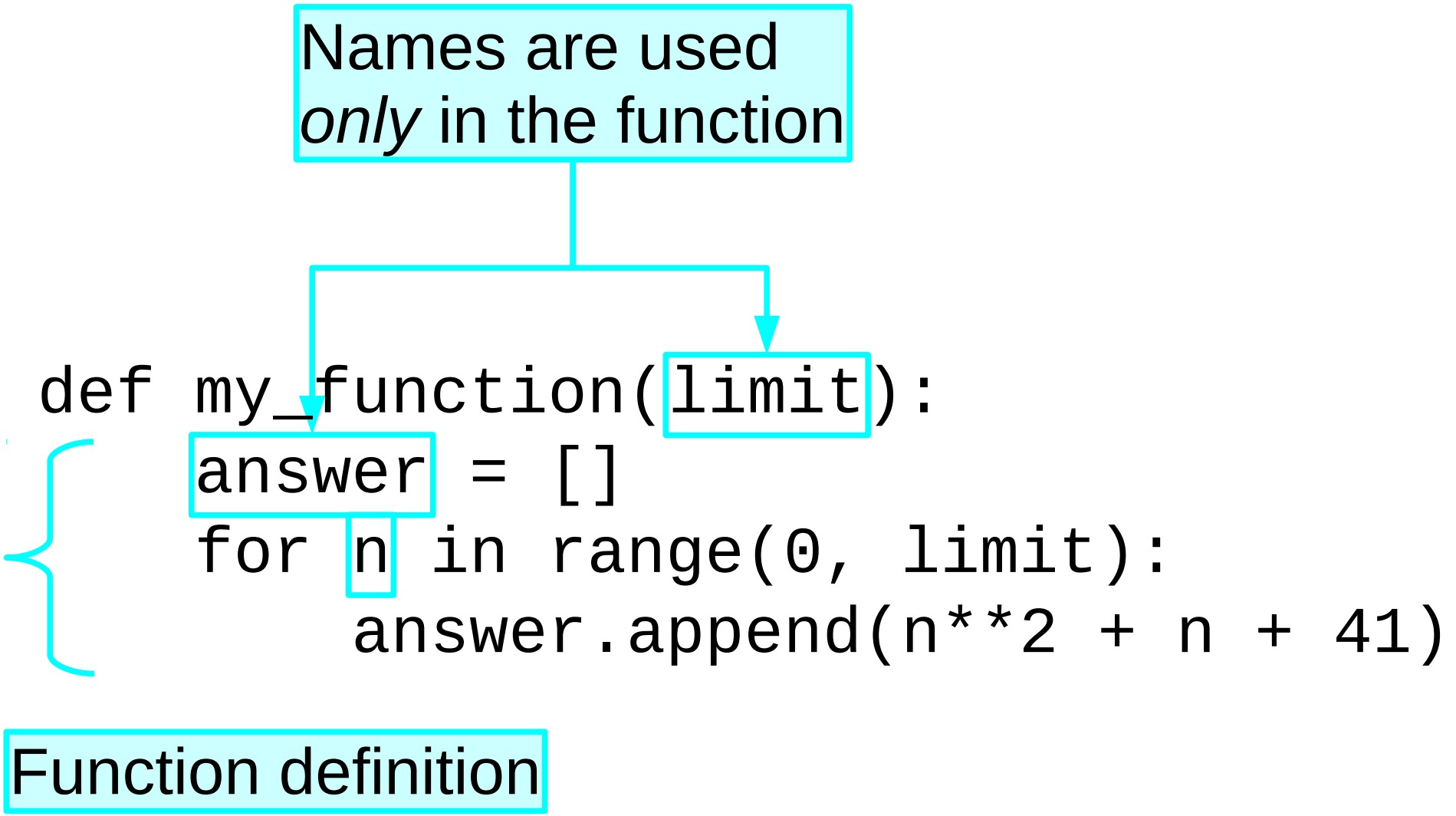
separated by TABs too.

15 minutes

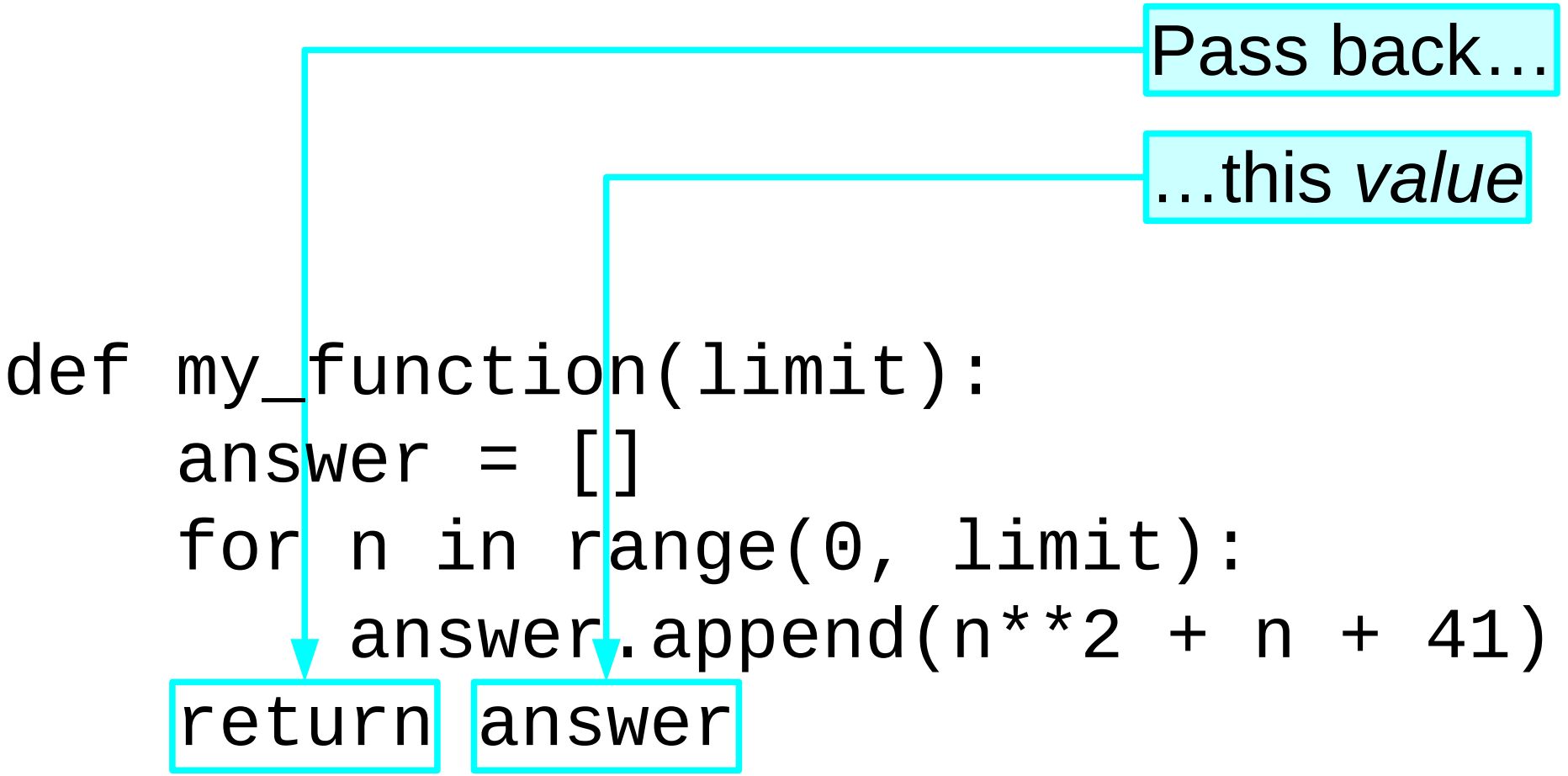
Defining our function



Defining our function



Defining our function



Using our function

```
def my_function(limit):
    answer = []
    for n in range(0, limit):
        answer.append(n**2 + n + 41)
    return answer

...

results = my_function(11)
```

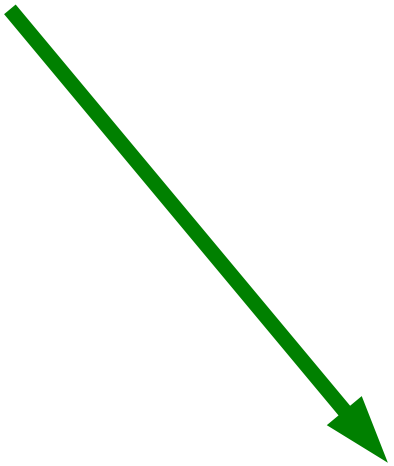
"answer"

"limit"

Why use functions?

Reuse

If you use a function in lots of places and have to change it, you only have to edit it in one place.



Clarity

Clearly separated components are easier to read.

Reliability

Isolation of variables leads to fewer accidental clashes of variable names.

Example 1

Write a function to take a list of floating point numbers and return the sum of the squares.

$$(a_i) \rightarrow \sum |a_i|^2$$

```
def norm2(values):

    sum = 0.0

    for value in values:
        sum = sum + value**2

    return sum
```

Example 1

```
print norm2([3.0, 4.0, 5.0])
```

```
50.0
```

```
$ python norm2.py
50.0                                    [3.0, 4.0, 5.0]
169.0                                   [12.0, 5.0]
```

Example 2

Write a function to pull the minimum value from a list.

$$(a_i) \rightarrow \min(a_i)$$

```
def minimum(a_list):

    a_min = a_list[0]
    for a in a_list:
        if a < a_min:
            a_min = a

    return a_min
```

Example 2

```
print minimum([2.0, 4.0, 1.0, 3.0])
```

```
1.0
```

```
$ python minimum.py
3.0                                [4.0, 3.0, 5.0]
5                                  [12, 5]
```

Example 3

Write a function to “dot product” two vectors.

$$(a_i, b_j) \rightarrow \sum a_k b_k$$

```
def dot(a_vec, b_vec):

    sum = 0.0
    for n in range(0,len(a_vec)):
        sum = sum + a_vec[n]*b_vec[n]

    return sum
```

Example 3

```
print dot([3.0, 4.0], [1.0, 2.0]))
```

```
11.0

$ python dot_product.py
11.0
115
```

Example 3 — version 2

```
def dot(a_vec, b_vec):

    if len(a_vec) != len(b_vec):
        print 'WARNING: lengths differ!'

    sum = 0.0
    for n in range(0,len(a_vec)):
        sum = sum + a_vec[n]*b_vec[n]

    return sum
```

Example 4

Write a function to filter out the positive numbers from a list.

e.g. [1, -2, 0, 5, -5, 3, 3, 6] ⟶ [1, 5, 3, 3, 6]

```
def positive(a_list):

    answer = []

    for a in a_list:
        if a > 0:
            answer.append(a)

    return answer
```

How to return more than one value?

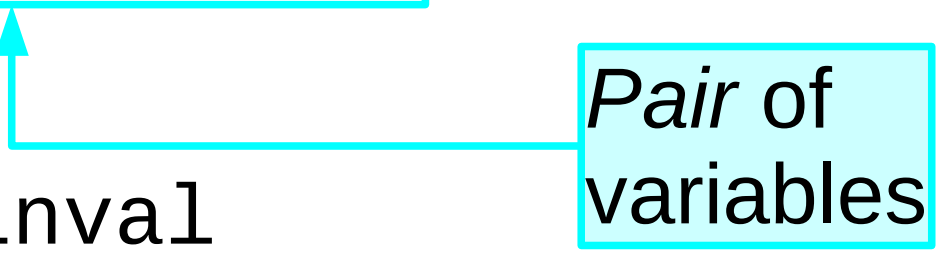
Write a function to pull the minimum *and* maximum values from a list.

```
def min_max(a_list):
    a_min = a_list[0]
    a_max = a_list[0]
    for a in a_list:
        if a < a_min:
            a_min = a
        if a > a_max:
            a_max = a
    return (a_min, a_max)
```

Pair of values

Receiving two values

```
...
values = [1, 2, 3, 4, 5, 6, 7, 8, 9]

(minval, maxval) = min_max(values)

print minval
print maxval
```



Pairs, triplets, …

singles
doubles
triples
quadruples
quintuples
…

“tuples”

Tuples ≠ Lists

Lists	**Tuples**
Concept of “next entry”	All items at once
Same types	Different types
Mutable	Immutable

Tuple examples

Pair of measurements of a tree

(*height*,*width*)	(7.2, 0.5)
(*width*,*height*)	(0.5, 7.2)

Details about a person

(*name*, *age*, *height*)	('Bob', 45, 1.91)
(*age*, *height, name*)	(45, 1.91, 'Bob')

Exercise

Copy the `min_max()` function.

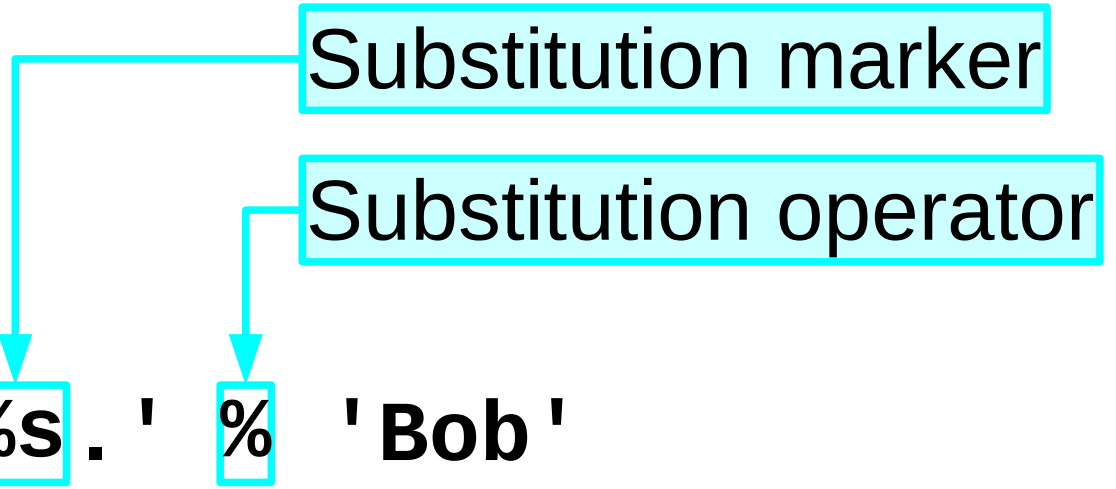
Extend it to return a triplet:
`(minimum, mean, maximum)`

10 minutes

Tuples and string substitution

“Hello, my name is Bob and I'm 46 years old.”

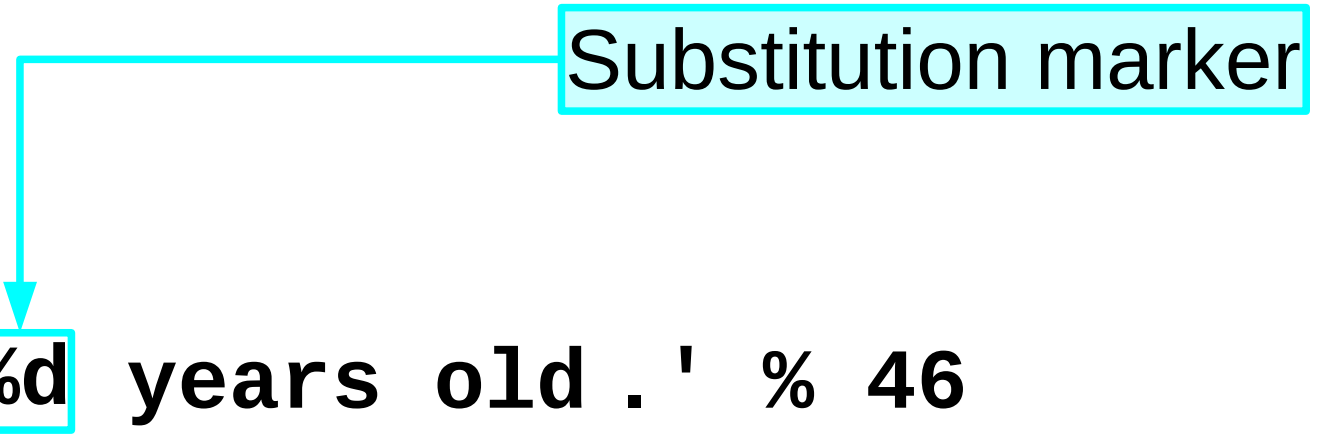
Simple string substitution




```
>>> 'My name is %s.' % 'Bob'
'My name is Bob.'
```

`%s` Substitute a string.

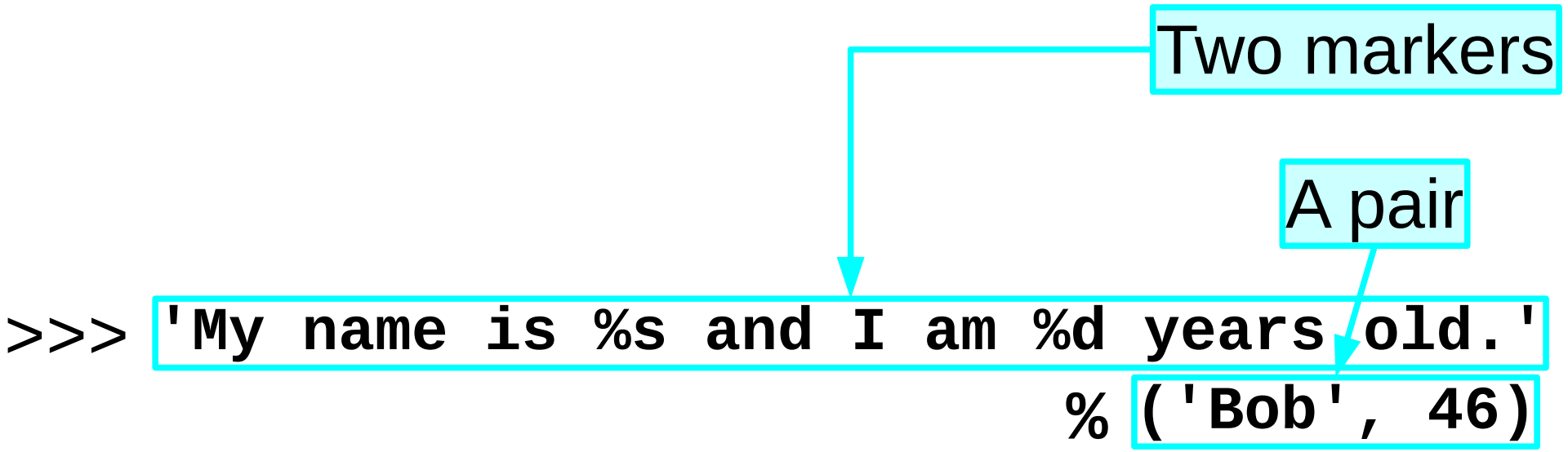
Simple integer substitution




```
>>> 'I am %d years old.' % 46
'I am 46 years old.'
```

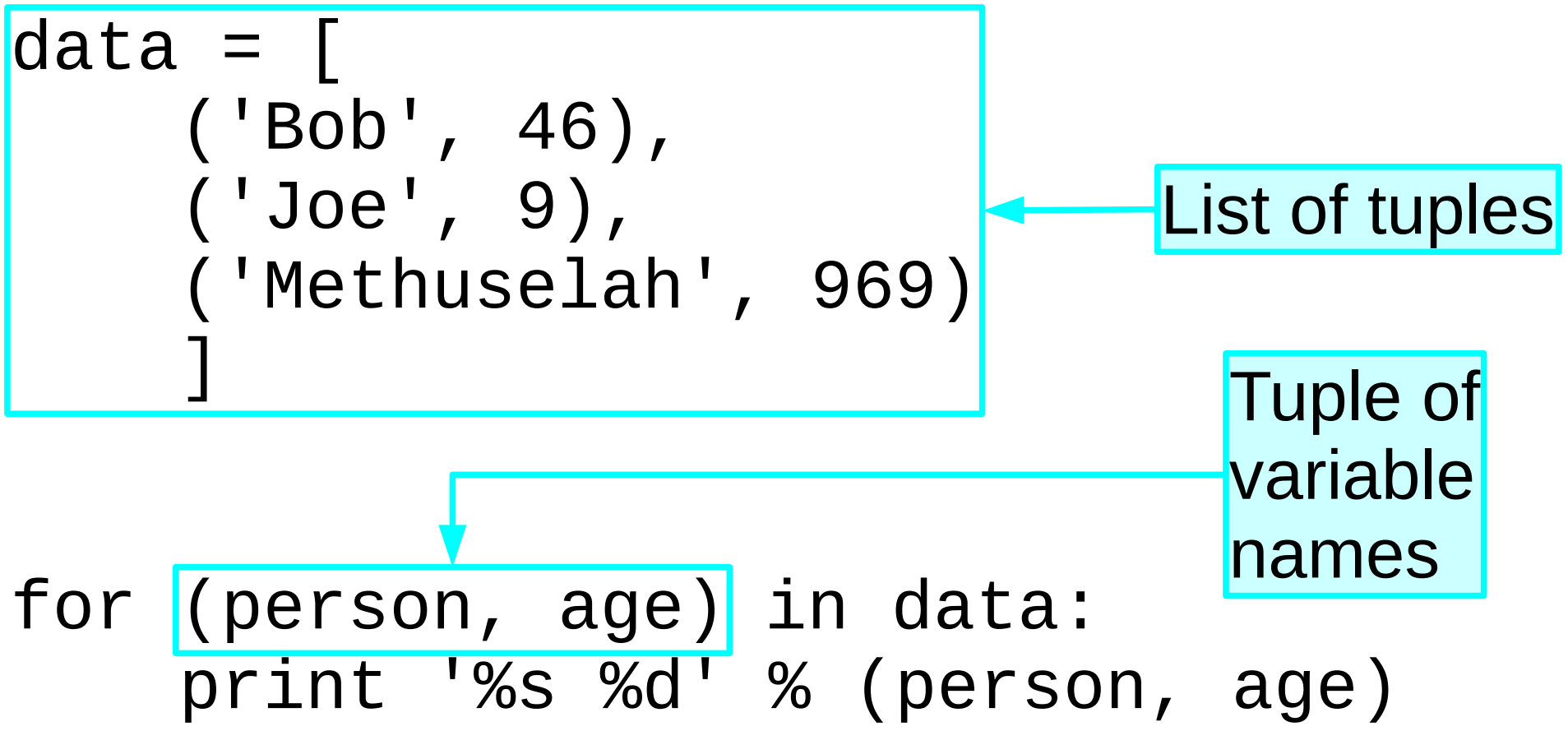
%d Substitute an integer.

Tuple substitution




```
'My name is Bob and I am 46 years old.'
```

Lists of tuples



Problem: ugly output

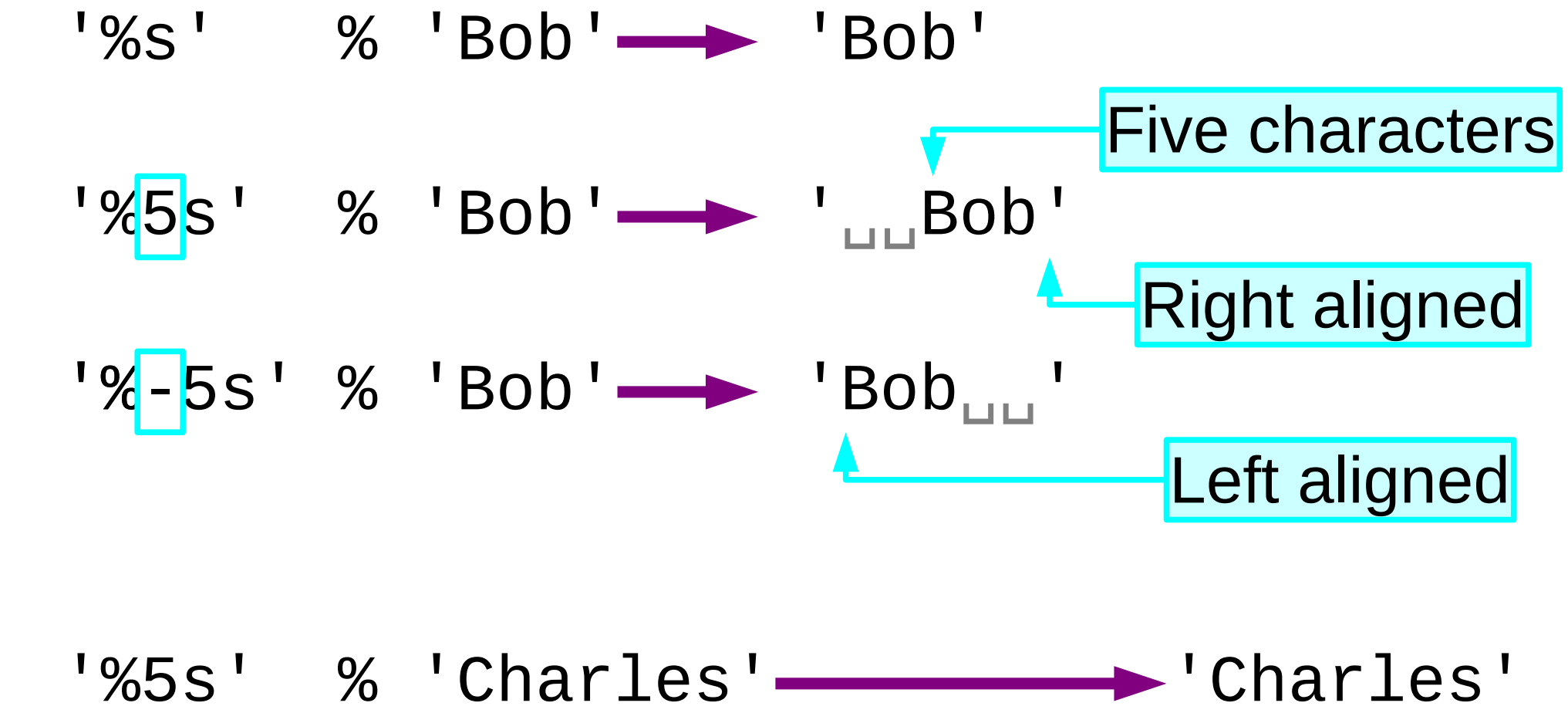
```
Bob 46
Joe 9
Methuselah 969
```

Columns should align

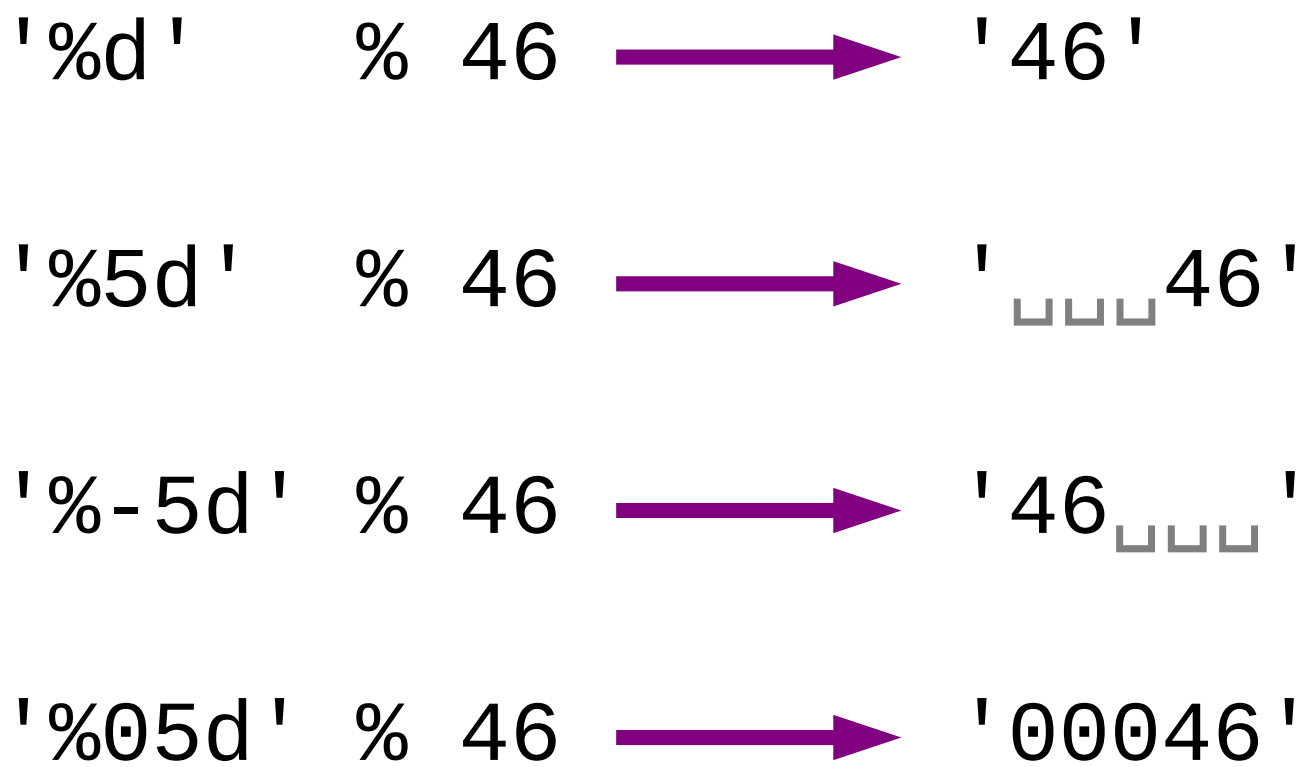
Columns of numbers should be right aligned

```
Bob          46
Joe           9
Methuselah  969
```

Solution: formatting

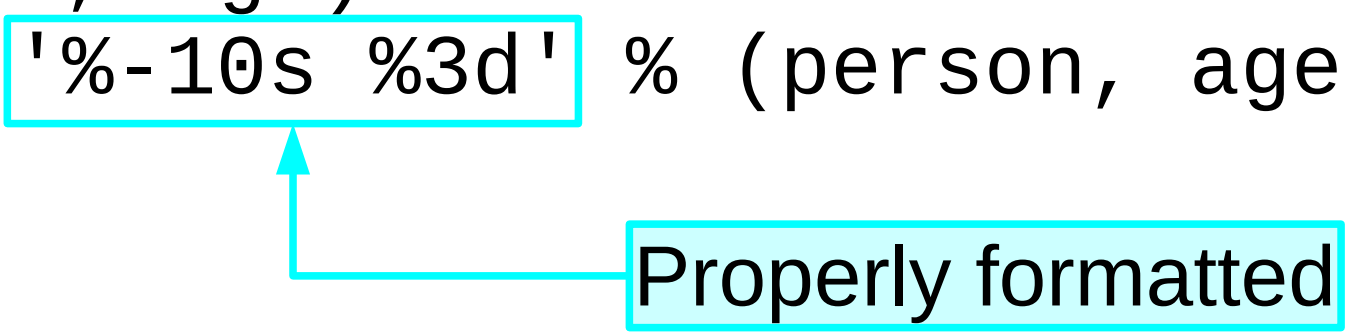


Solution: formatting



Columnar output

```
data = [
     ('Bob', 46),
     ('Joe', 9),
     ('Methuselah', 969)
     ]


for (person, age) in data:
     print '%-10s %3d' % (person, age)
```



Floats

```
'%f'    % 3.141592653589 → '3.141593'

'%.4f' % 3.141592653589 → '3.1416'

'%.4f' % 3.1 ——————————→ '3.1000'
```

Exercise

Complete the script `format1.py` to generate this output:

```
Alfred    46    1.90
Bess      24    1.75
Craig      9    1.50
Diana    100    1.66
↑        ↑      ↑
1        9      15
```

 5 minutes

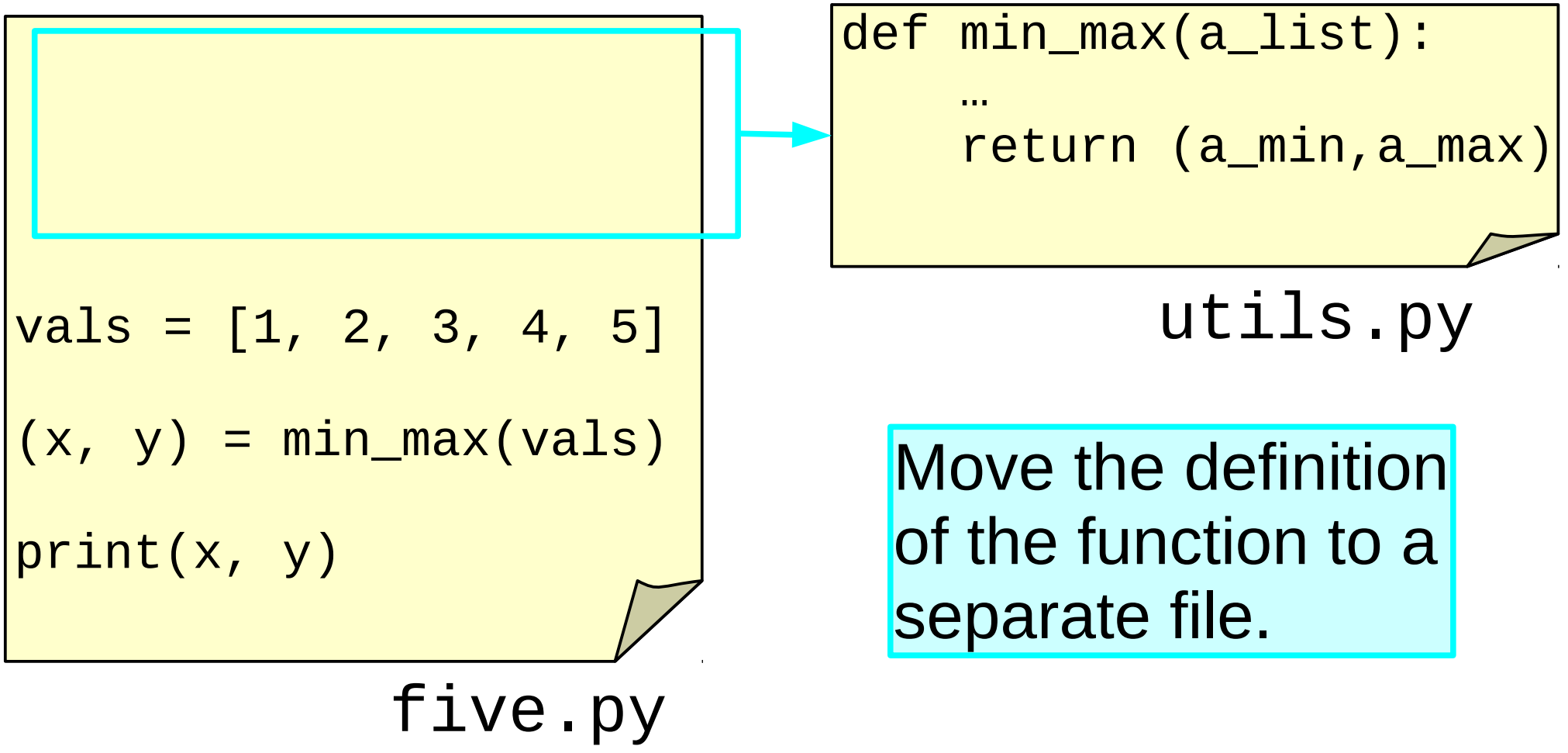
Reusing our functions

Want to use the same function in many scripts

```
def min_max(a_list):
    …
    return (a_min,a_max)


vals = [1, 2, 3, 4, 5]

(x, y) = min_max(vals)

print(x, y)
```

five.py

How to reuse — 1



How to reuse — 2

```
import utils



vals = [1, 2, 3, 4, 5]

(x, y) = min_max(vals)

print(x, y)
```

five.py

```
def min_max(a_list):
    …
    return (a_min,a_max)
```

utils.py

Identify the file with the functions in it.

How to reuse — 3

```
import utils



vals = [1, 2, 3, 4, 5]

(x, y) = utils.min_max(vals)

print(x, y)
```

five.py

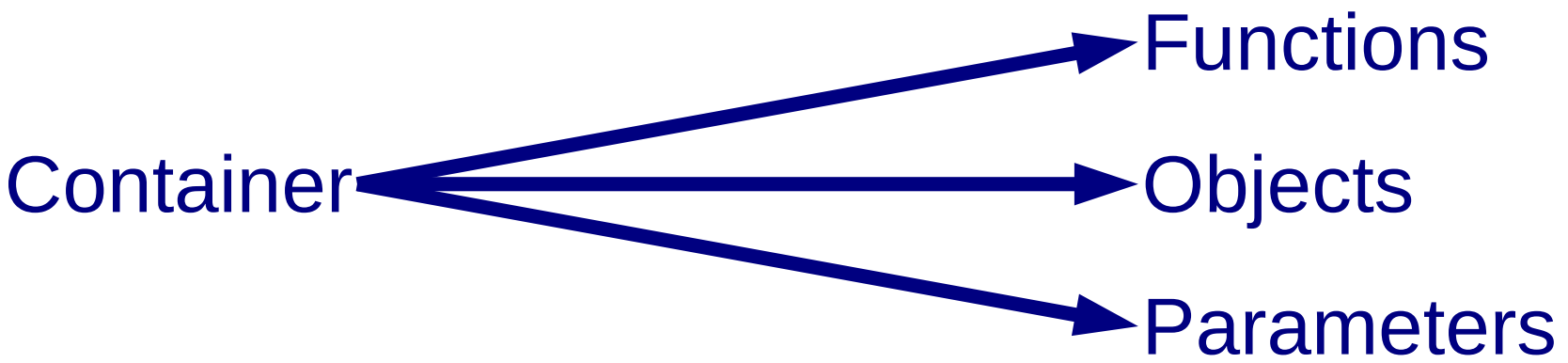
```
def min_max(a_list):
    …
    return (a_min,a_max)
```

utils.py

Indicate that the function comes from that import.

A library of our functions

"Module"



System modules

os	operating system access
subprocess	support for child processes
sys	general system functions
math	standard mathematical functions
numpy	numerical arrays and more
scipy	maths, science, engineering
csv	read/write comma separated values
re	regular expressions

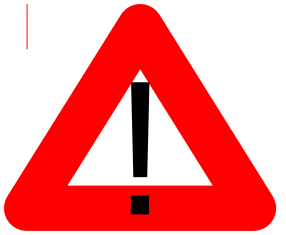
Using a system module

```
>>> import math

>>> math.sqrt(2.0)
1.4142135623730951

>>>
```

Keep track of the module with the function.

Don't do this

```
>>> from math import sqrt
>>> sqrt(2.0)
1.4142135623730951

>>> from math import *
>>> sqrt(2.0)
1.4142135623730951
```



Do do this

```
>>> import math

>>> help(math)

Help on module math:
NAME
   math
DESCRIPTION
   This module is always available. It
   provides access to the mathematical
   functions defined by the C standard.
```

Exercise

1. Edit your `utils.py` file.

2. Write a function `print_list()` that prints all the elements of a list, one per line.

3. Edit the `elements2.py` script to use this new function.

 5 minutes

Interacting with the **sys**tem

```
>>>  import sys
```

Standard input and output

```
>>> import sys
```

`sys.stdin` ← Treat like an open(..., 'r') file

`sys.stdout` ← Treat like an open(..., 'w') file

Line-by-line copying — 1

```
import sys
for line in sys.stdin:

    sys.stdout.write(line)
```

Import module

No need to `open()` `sys.stdin` or `sys.stdout`.
The module has done it for you at import.

Line-by-line copying — 2

```
import sys
for line in sys.stdin:

    sys.stdout.write(line)
```

Standard input

Treat a file like a list → Acts like a list of lines

Line-by-line copying — 3

```
import sys

for line in sys.stdin:

    sys.stdout.write(line)
```

Standard output

An open file

The file's `write()` method

Line-by-line copying — 4

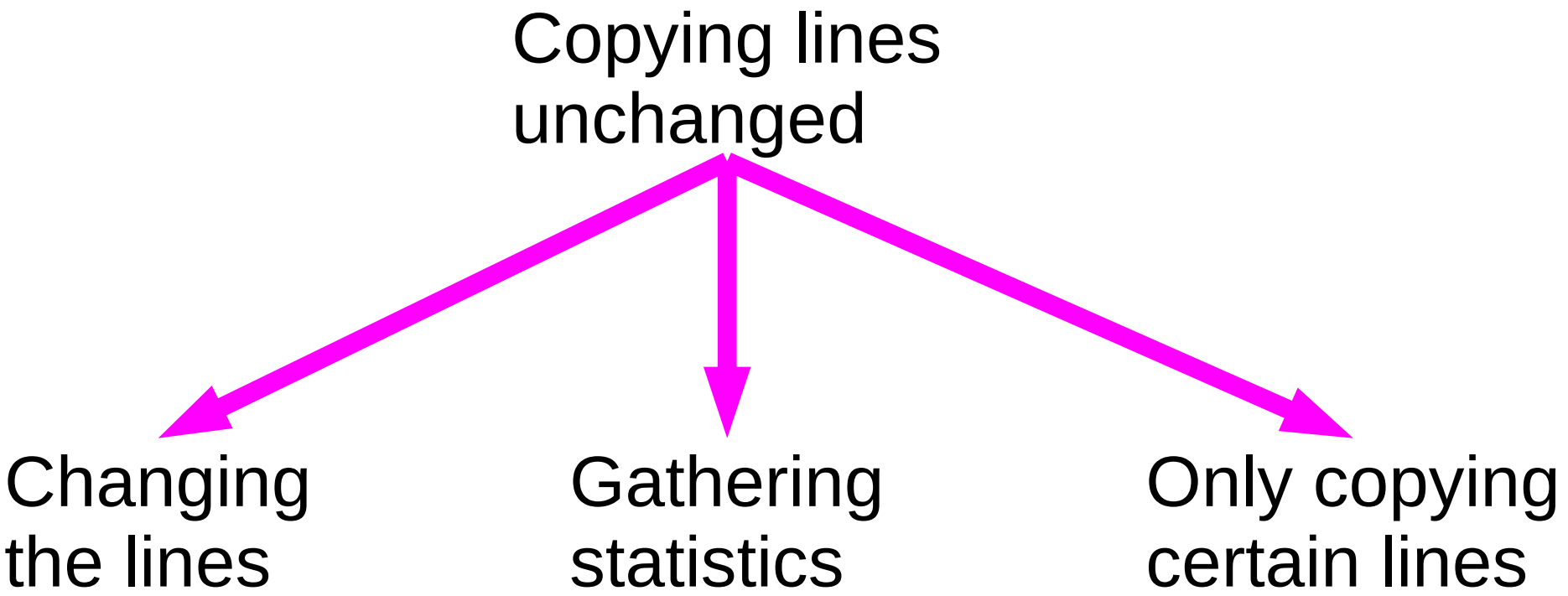
```
import sys

for line in sys.stdin:

    sys.stdout.write(line)
```

Lines in…
lines out

```
$ python copy.py < in.txt > out.txt
```

Copy

Line-by-line actions



Line-by-line rewriting

```
import sys
for input in sys.stdin:
    output = function(input)
    sys.stdout.write(output)
```

Define or import a function here

```
$ python process.py < in.txt > out.txt
```

Process

Line-by-line filtering

```
import sys
for input in sys.stdin:
    if test(input):
        sys.stdout.write(input)
```

Define or import a test function here

```
$ python filter.py < in.txt > out.txt
```

Filter

Exercise

Write a script that reads from standard input.

If should generate two lines of output:

```
Number of lines:         MMM
Number of blank lines: NNN
```

Hint: `len(line.split()) == 0` for blank lines.

 5 minutes

The command line

We are putting parameters in our scripts.

```
...
number = 1.25
...
```

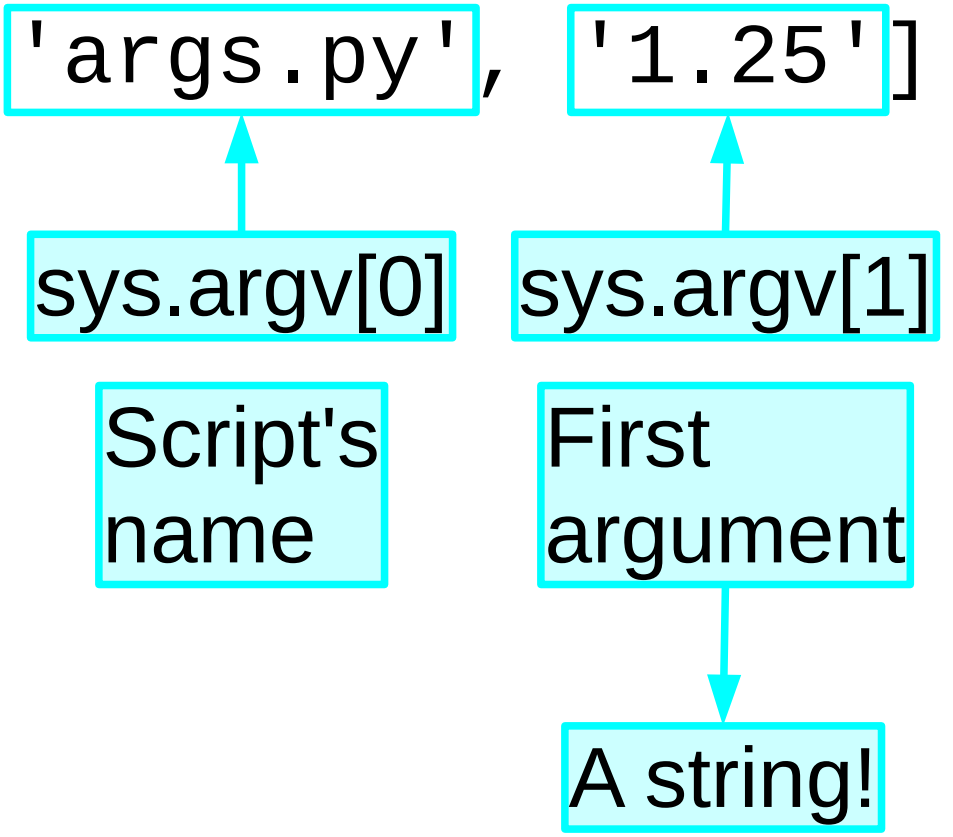
We want to put them on the command line.

```
$ python script.py 1.25
```

Reading the command line

```
import sys

print(sys.argv)
```

```
$ python args.py 1.25
['args.py', '1.25']
```



Command line strings

```
import sys

number = sys.argv[1]
number = number + 1.0

print(number)
```

```
Traceback (most recent call last):
  File "thing.py", line 3, in <module>
    number = number + 1.0
TypeError:
cannot concatenate 'str' and 'float' objects
```

Using the command line

```
import sys

number = float(sys.argv[1])
number = number + 1.0

print(number)
```

Enough arguments?

Valid as floats?

Worked example

Write a script to print points

(x, y) $y=x^r$ $x \in [0,1]$, uniformly spaced

Two command line arguments:

r	(float)	power
N	(integer)	number of points

General approach

1a. Write a function that parses the command line for a float and an integer.

1b. Write a script that tests that function.

2a. Write a function that takes (r, N) as (float, integer) and does the work.

2b. Write a script that tests that function.

3. Combine the two functions.

1a. Write a function that parses the command line for a float and an integer.

```
import sys

def parse_args():
   pow = float(sys.argv[1])
   num = int(sys.argv[2])

   return (pow, num)
```

curve.py

1b. Write a script that tests that function.

```
import sys

def parse_args():
    ...
(r, N) = parse_args()
print 'Power:  %f' % r
print 'Points: %d' % N
```

curve.py

1b. Write a script that tests that function.

```
$ python curve.py 0.5 5

Power:  0.500000
Points: 5
```

2a. Write a function that takes (r, N) as (float, integer) and does the work.

```
def power_curve(pow, num_points):

   for index in range(0, num_points):
      x = float(index)/float(num_points-1)
      y = x**pow
      print '%f %f' % (x, y)
```

curve.py

2b. Write a script that tests that function.

```
def power_curve(pow, num_points):
    ...

power_curve(0.5, 5)
```

curve.py

2b. Write a script that tests that function.

```
$ python curve.py
0.000000 0.000000
0.250000 0.500000
0.500000 0.707107
0.750000 0.866025
1.000000 1.000000
```

3. Combine the two functions.

```
import sys

def parse_args():
   pow = float(sys.argv[1])
   num = int(sys.argv[2])
   return (pow, num)

def power_curve(pow, num_points):
   for index in range(0, num_points):
      x = float(index)/float(num_points-1)
      y = x**pow
      print '%f %f' % (x, y)

(power, number) = parse_args()
power_curve(power, number)
```

curve.py

Exercise

Write a script that takes a command line of numbers and prints their minimum and maximum.

Hint: You have already written a min_max function. Reuse it.

5 minutes

Back to our own module

```
>>> import utils
>>> help(utils)
```

```
Help on module utils:
NAME
    utils
FILE
    /home/rjd4/utils.py
FUNCTIONS
    min_max(numbers)
...
```

We want to do better than this.

Function help

```
>>> import utils
>>> help(utils.min_max)
```

```
Help on function min_max in
module utils:

min_max(numbers)
```

Annotating a function

```
def min_max(numbers):
    minimum = numbers[0]
    maximum = numbers[0]
    for number in numbers:
        if number < minimum:
           minimum = number
  if number > maximum:
        maximum = number
    return (minimum, maximum)
```

Our current file

A “documentation string”

```
def min_max(numbers):
    """This functions takes a list
    of numbers and returns a pair
    of their minimum and maximum.
    """
    minimum = numbers[0]
    maximum = numbers[0]
    for number in numbers:
        if number < minimum:
            minimum = number
        if number > maximum:
            maximum = number
    return (minimum, maximum)
```

A string before the body of the function.

Annotated function

```
>>> import utils
>>> help(utils.min_max)

Help on function min_max in
module utils:

min_max(numbers)
    This functions takes a list
    of numbers and returns a pair
    of their minimum and maximum.
```

Annotating a module

```
"""A personal utility module
full of all the pythonic goodness
I have ever written.
"""
def min_max(numbers):
    """This functions takes a list
    of numbers and returns a pair
    of their minimum and maximum.
    """

    minimum = numbers[0]
    maximum = numbers[0]
    for number in numbers:
...
```

A string before any active part of the module.

Annotated module

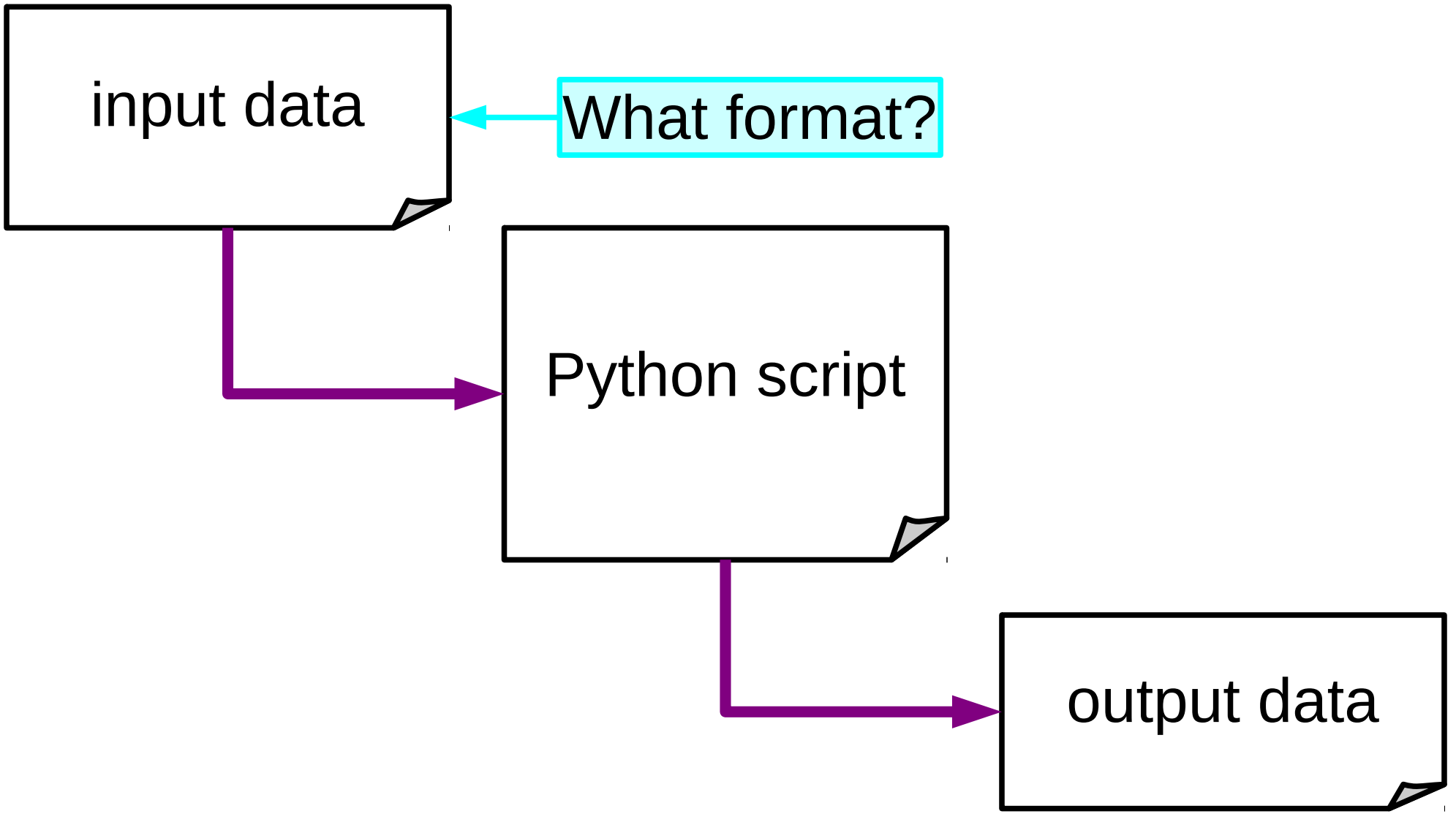
```
>>> import utils
>>> help(utils)

Help on module utils:
NAME
    utils
FILE
    /home/rjd4/utils.py
DESCRIPTION
    A personal utility module
    full of all the pythonic goodness
    I have ever written.
```

Exercise

Annotate your `utils.py` and the functions in it.

3 minutes

Simple data processing



Comma Separated Values

input data

```
A101,Joe,45,1.90,100
G042,Fred,34,1.80,92
H003,Bess,56,1.75,80
...
```

```
1.0,2.0,3.0,4.0
2.0,4.0,8.0,16.0
3.0,8.0,24.0,64.0
...
```

Quick and dirty .csv — 1

CSV: “comma separated values”

More likely to have come from sys.stdin

```
>>> line = '1.0, 2.0, 3.0, 4.0\n'
```

```
>>> line.split(',')
```

Split on commas rather than spaces.

```
['1.0', ' 2.0', ' 3.0', ' 4.0\n']
```

Note the leading and trailing white space.

Quick and dirty .csv — 2

```
>>> line = '1.0, 2.0, 3.0, 4.0\n'
>>> strings = line.split(',')
>>> numbers = []
>>> for string in strings:
...     numbers.append(float(string))
...
>>> numbers
[1.0, 2.0, 3.0, 4.0]
```

Quick and dirty .csv — 3

Why “quick and dirty”?

Can't cope with common cases:

- Quotes `'"1.0","2.0","3.0","4.0"'`
- Commas `'A,B\,C,D'`

Dedicated module: `csv`

Proper .csv

Dedicated module: `csv`

```
import csv
import sys

input = csv.reader(sys.stdin)
output = csv.writer(sys.stdout)

for [id, name, age, height, weight] in input:
    output.writerow([id, name, float(height)*100])
```

Processing data

Storing data in the program

id	name	age	height	weight
A101	Joe	45	1.90	100
G042	Fred	34	1.80	92
H003	Bess	56	1.75	80
...				

? id → (name, age, height, weight) ?

Simpler case

Storing data in the program

id	name
A101	Joe
G042	Fred
H003	Bess
...	

? id → name ?

Not the same as a list…

index	name
0	Joe
1	Fred
2	Bess
...	

names[1] = 'Fred'

['Joe', 'Fred', 'Bess', …]

…but similar: a “dictionary”

id	name
A101	Joe
G042	Fred
H003	Bess
...	

names['G042'] = 'Fred'

{'A101':'Joe', 'G042':'Fred', 'H003':'Bess', …}

Dictionaries

“key” → “value”

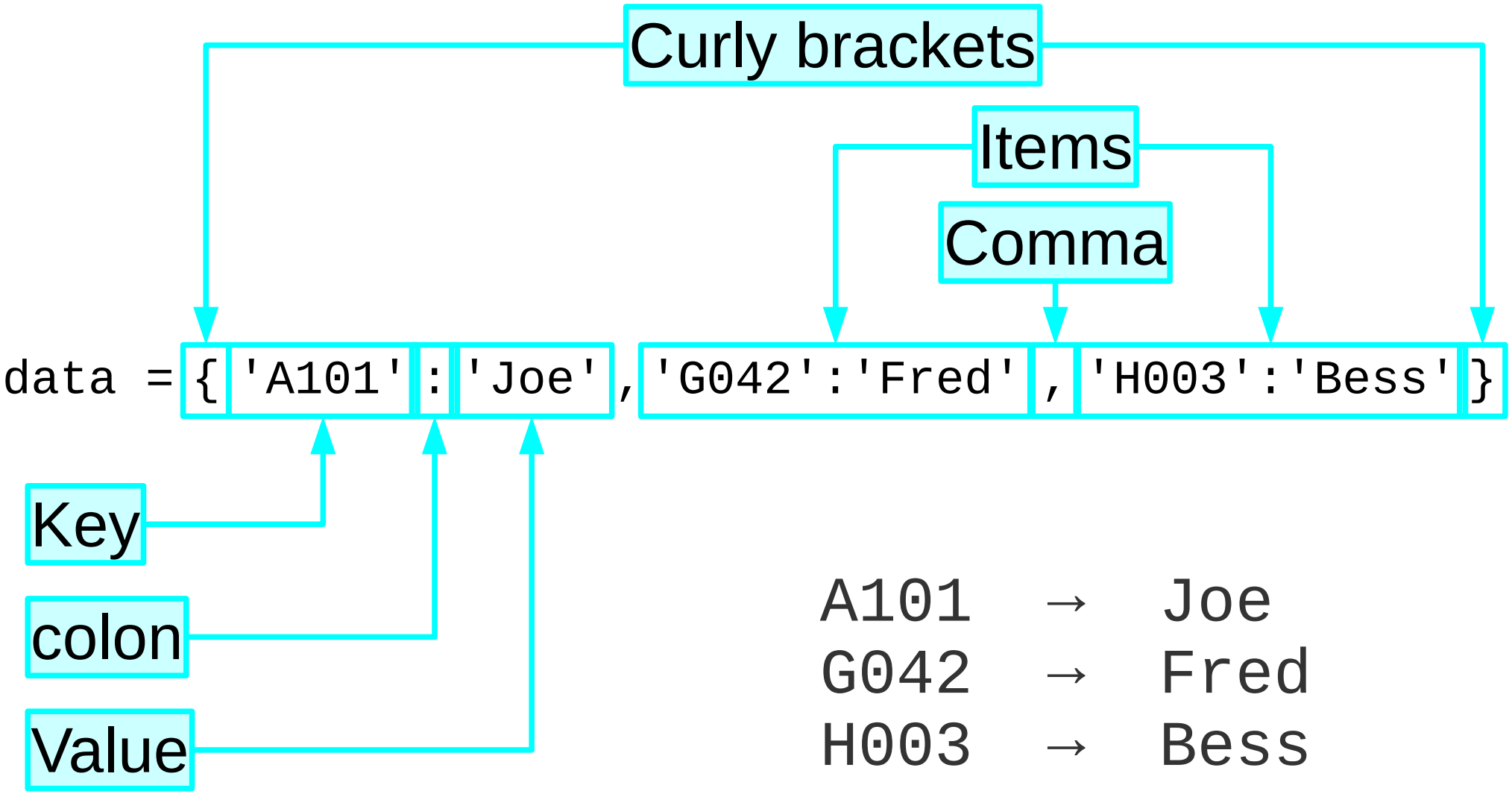
Generalized look up

Python object (immutable) → Python object (arbitrary)

'G042' →	'Fred'
1700045 →	29347565
'G042' →	('Fred', 34)
(34, 56) →	'treasure'
(5,6) →	[5, 6, 10, 12]

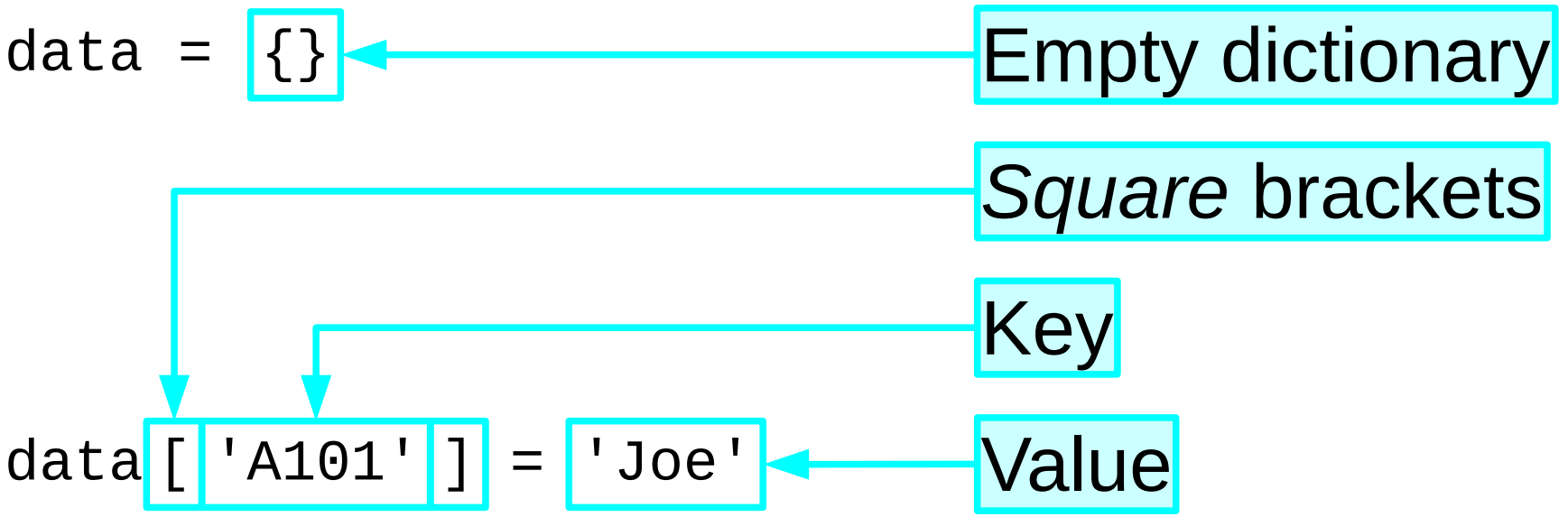
string →	string
int →	int
string →	tuple
tuple →	string
tuple →	list

Building a dictionary — 1




```
A101  →  Joe
G042  →  Fred
H003  →  Bess
```

Building a dictionary — 2




```
data['G042'] = 'Fred'

data['H003'] = 'Bess'
```

```
A101  →  Joe
G042  →  Fred
H003  →  Bess
```

Example — 1

```
>>> data = {'A101':'Joe', 'F042':'Fred'}

>>> data
{'F042': 'Fred', 'A101': 'Joe'}
```

Order is not preserved!

Example — 2

```
>>> data['A101']
'Joe'

>>> data['A101'] = 'James'

>>> data
{'F042': 'Fred', 'A101': 'James'}
```

Square brackets in Python

`[…]`	Defining literal lists
`numbers[N]`	Indexing into a list
`numbers[M:N]`	Slices
`values[key]`	**Looking up in a dictionary**

Example — 3

```
>>> data['X123'] = 'Bob'

>>> data['X123']
'Bob'

>>> data
{'F042': 'Fred', 'X123': 'Bob',
'A101': 'James'}
```

More dictionaries

```
data =
{'G042':('Fred',34), 'A101':('Joe',45)}
```

```
data['G042']
```

```
('Fred',34)
```

```
data['H003'] = ('Bess ', 56)
```

Exercise

Write a script that:

1. Creates an empty dictionary, “elements”.
2. Adds an entry 'H'→'Hydrogen'.
3. Adds an entry 'He'→'Helium'.
4. Adds an entry 'Li'→'Lithium'.
5. Prints out the value for key 'He'.
6. Tries to print out the value for key 'Be'.

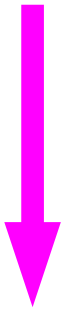
 10 minutes

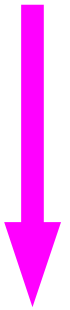
Worked example — 1

Reading a file to populate a dictionary

```
H    Hydrogen
He   Helium
Li   Lithium
Be   Beryllium
B    Boron
C    Carbon
N    Nitrogen
O    Oxygen
F    Fluorine
...
```

`elements.txt` — File





`symbol_to_name` — Dictionary

Worked example — 2

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

Open file

```
symbol_to_name = {}
```

Empty dictionary

Read data

```
for line in data:
    [symbol, name] = line.split()
    symbol_to_name[symbol] = name
```

Populate dictionary

```
data.close()
```

Close file

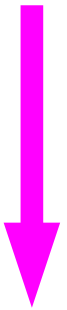
Now ready to use the dictionary

Worked example — 3

Reading a file to populate a dictionary

```
A101 Joe
F042 Fred
X123 Bob
K876 Alice
J000 Maureen
A012 Stephen
X120 Peter
K567 Anthony
F041 Sally
...
```

`names.txt`



`key_to_name`

Worked example — 4

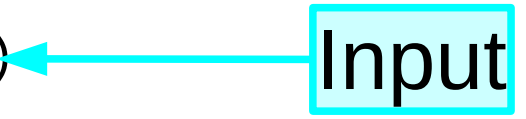
```
data = open('names.txt')
key_to_name = {}

for line in data:
    [key, person] = line.split()
    key_to_name[key] = person

data.close()
```

Make it a function!

```
symbol_to_name = {}

data = open('elements.txt')

for line in data:
  [symbol, name] = line.split()
   symbol_to_name[symbol] = name

data.close()
```

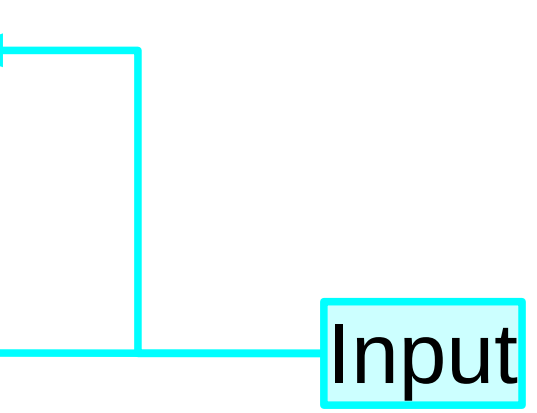
Make it a function!

```
symbol_to_name = {}

data = open('elements.txt')

for line in data:
  [symbol, name] = line.split()
   symbol_to_name[symbol] = name

data.close()
```



Make it a function!

```
def filename_to_dict(filename):

    symbol_to_name = {}

    data = open(filename)

    for line in data:
      [symbol, name] = line.split()
       symbol_to_name[symbol] = name

    data.close()
```



Make it a function!

```
def filename_to_dict(filename):

    symbol_to_name = {}

    data = open(filename )

    for line in data:
      [symbol, name] = line.split()
       symbol_to_name[symbol] = name

    data.close()
```

Make it a function!

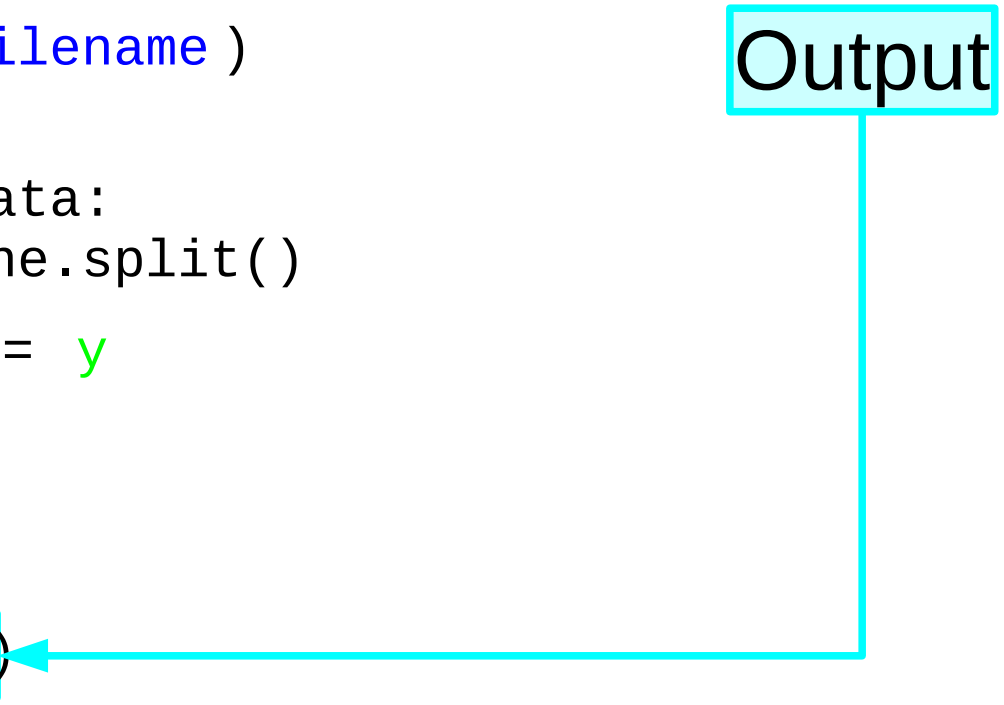
```
def filename_to_dict(filename):

    x_to_y  = {}

    data = open(filename )

    for line in data:
      [x, y] = line.split()
       x_to_y[x] = y

    data.close()
```

Output

Make it a function!

```
def filename_to_dict(filename):

    x_to_y  = {}

    data = open(filename )

    for line in data:
      [x, y] = line.split()
       x_to_y[x] = y

    data.close()

    return(x_to_y)
```

Output



Exercise

1. Write `filename_to_dict()` in your `utils` module.

2. Write a script that does this:

   a. Loads the file elements.txt as a dictionary (This maps 'Li' → 'lithium' for example.)

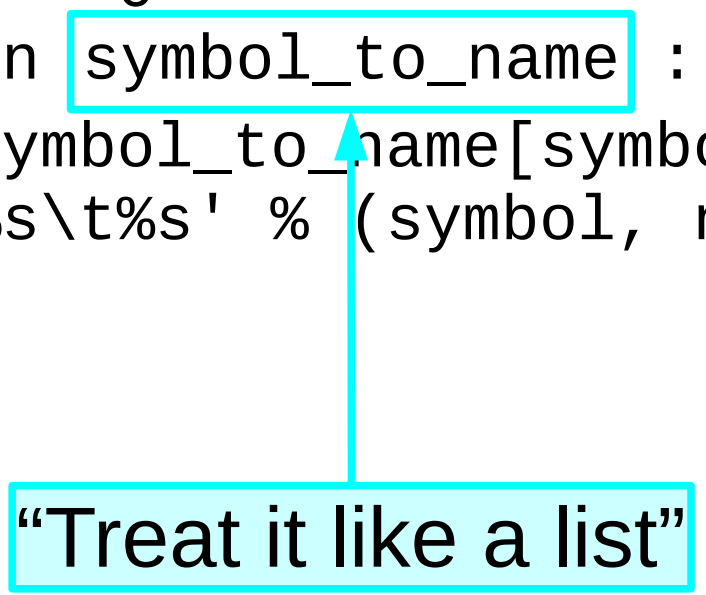
   b. Reads each line of inputs.txt (This is a list of chemical symbols.)

   c. For each line, prints out the element name

 10 minutes

Keys in a dictionary?

```
total_weight = 0
for symbol in symbol_to_name :
    name = symbol_to_name[symbol]
    print '%s\t%s' % (symbol, name)
```



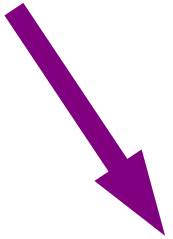
“Treat it like a list”

“Treat it like a list and it behaves like a (useful) list.”

File	→	List of lines
String	→	List of letters
Dictionary	→	**List of keys**

“Treat it like a list”

```
for item in list:
    blah blah
    …item…
    blah blah
```



```
for key in dictionary:
    blah blah
    …dictionary[key]…
    blah blah
```

Missing key?

```
>>> data = {'a':'alpha', 'b':'beta'}
>>> data['g']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'g'
```

Dictionary equivalent of "index out of range"

Convert to a list

```
keys = list(data)
print(keys)
```

```
['b', 'a']
```

`list(`*`dictionary`*`)` ⟶ `[`*`keys`*`]`

Exercise

Write a function `invert()`
in your `utils` module.

`symbol_to_name` 'Li' → 'Lithium'

```
name_to_symbol = invert(symbol_to_name)
```

`name_to_symbol` 'Lithium' → 'Li'

 10 minutes

One last example

Word counting

Given a text, what words appear and how often?

Word counting algorithm

Run through file line-by-line

Run through line word-by-word

Clean up word

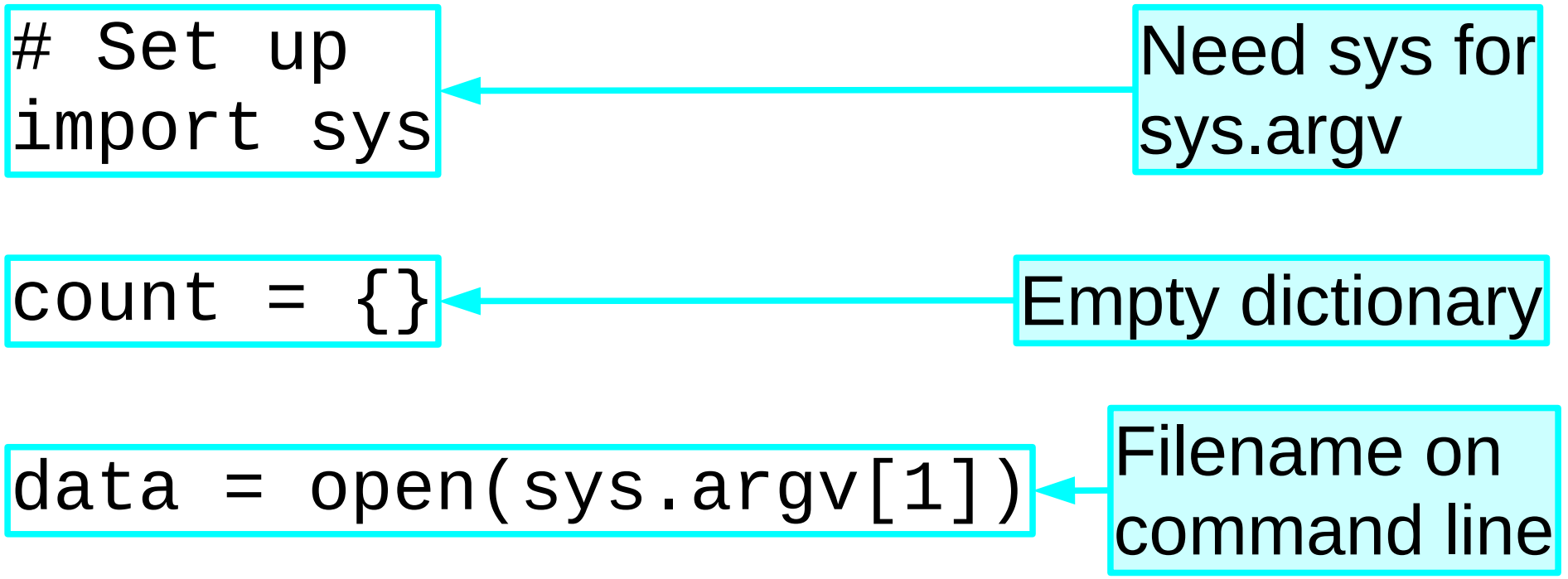
Is word in dictionary?

If not: add word as key with value 0

Increment the counter for that word

Output words alphabetically

Word counting in Python: 1



Word counting in Python: 2

```
for line in data:
    for word in line.split():
        clean_word = cleanup(word)
```

Lines

Words

We need to write this function.

Word counting in Python: 3

Insert at *start* of script

"Placeholder" function

```
def cleanup(word_in):
    word_out = word_in.lower()
    return word_out
```

Word counting in Python: 4

```
clean_word = cleanup(word)
```

Two levels indented

```
if not clean_word in count :
    count[clean_word] = 0
```

Create new entry in dictionary?

```
count[clean_word] = count[clean_word] + 1
```

Increment count for word

Word counting in Python: 5

```
        count[clean_word] = count[...
```

```
data.close()
```

Be tidy!

```
words = list(count)
```

All the words

```
words.sort()
```

Alphabetical order

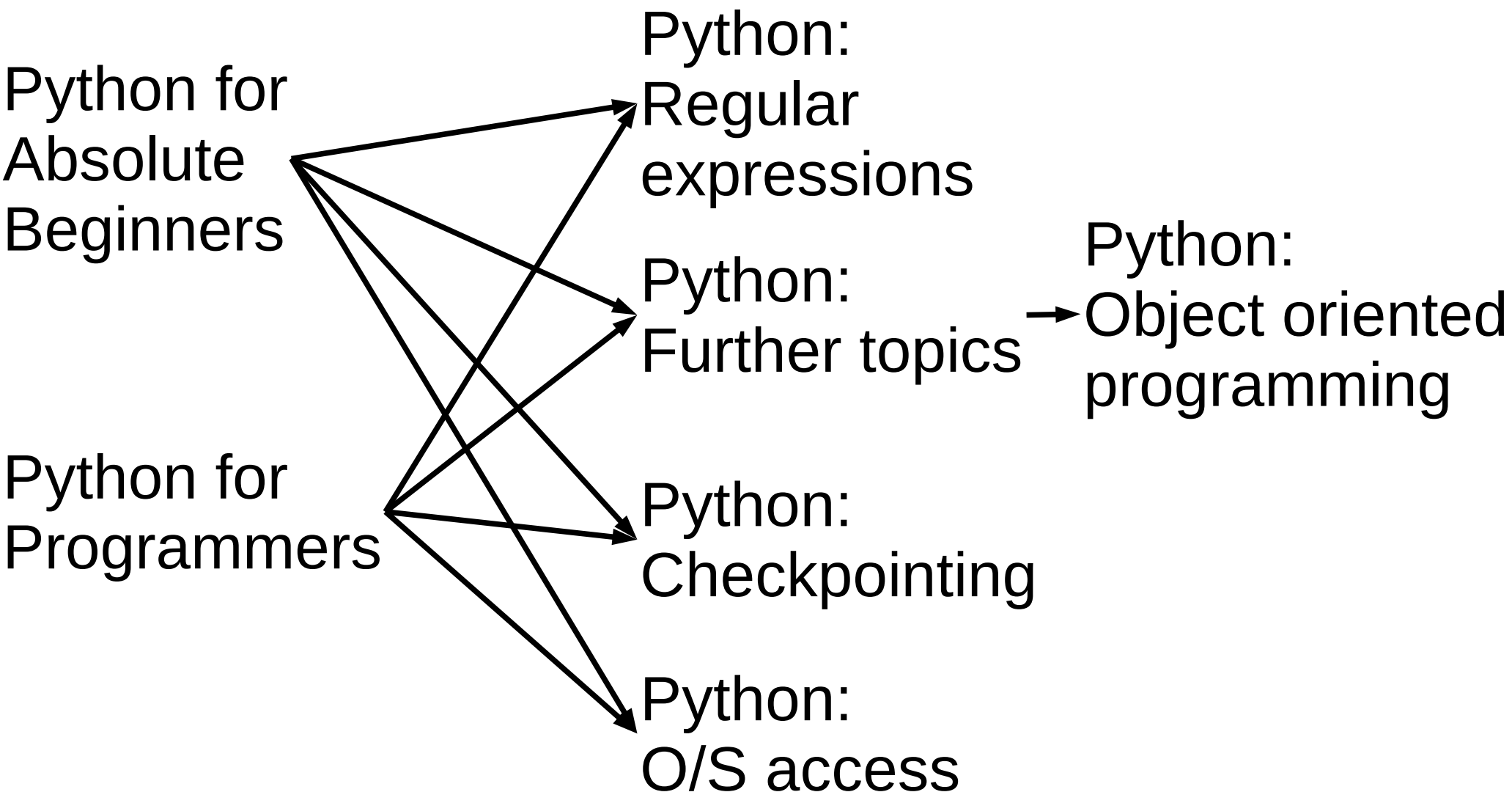
Word counting in Python: 6

```
words.sort()
```

Alphabetical order

```
for word in words:
    print('%s\t%d' % (word,count[word]))
```

More Python


Python for
Absolute
Beginners
Python for
Programmers
Python:
Regular
expressions
Python:
Further topics
Python:
Checkpointing
Python:
O/S access
Python:
Object oriented
programming