

An introduction to Python for absolute beginners

Bob Dowling
University Information Services
scientific-computing@ucs.cam.ac.uk

<http://www.ucs.cam.ac.uk/docs/course-notes/unix-courses/PythonAB>

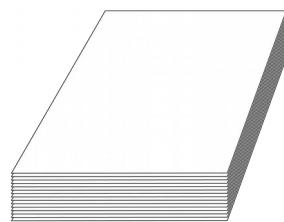
But first...



Fire escapes



Toilets



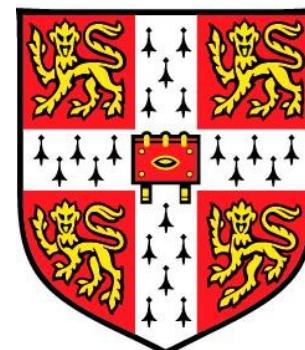
Notes



No drinks or snacks
in the room, please.



Signing in



Feedback



Course outline — 1

Who uses Python & what for
What sort of language it is

How to launch Python
Python scripts

Text
Names for values
Reading in user data
Numbers
Conversions
Comparisons
Truth & Falsehood

Course outline — 2



Assignment
Names

Our first “real” program

Loops
if... else...

Indentation

Comments

Course outline — 3



Lists

Indices

Lengths

Changing items

Extending lists

Methods

Creating lists

Testing lists

Removing from lists

for... loop

Iterables

Slices

Course outline — 4



Files

Reading & writing

Writing our own functions

Tuples

Modules

System modules

External modules

Dictionaries

Formatted text

Who uses Python?

On-line games



Web services

django

Applications



Science



Instrument control

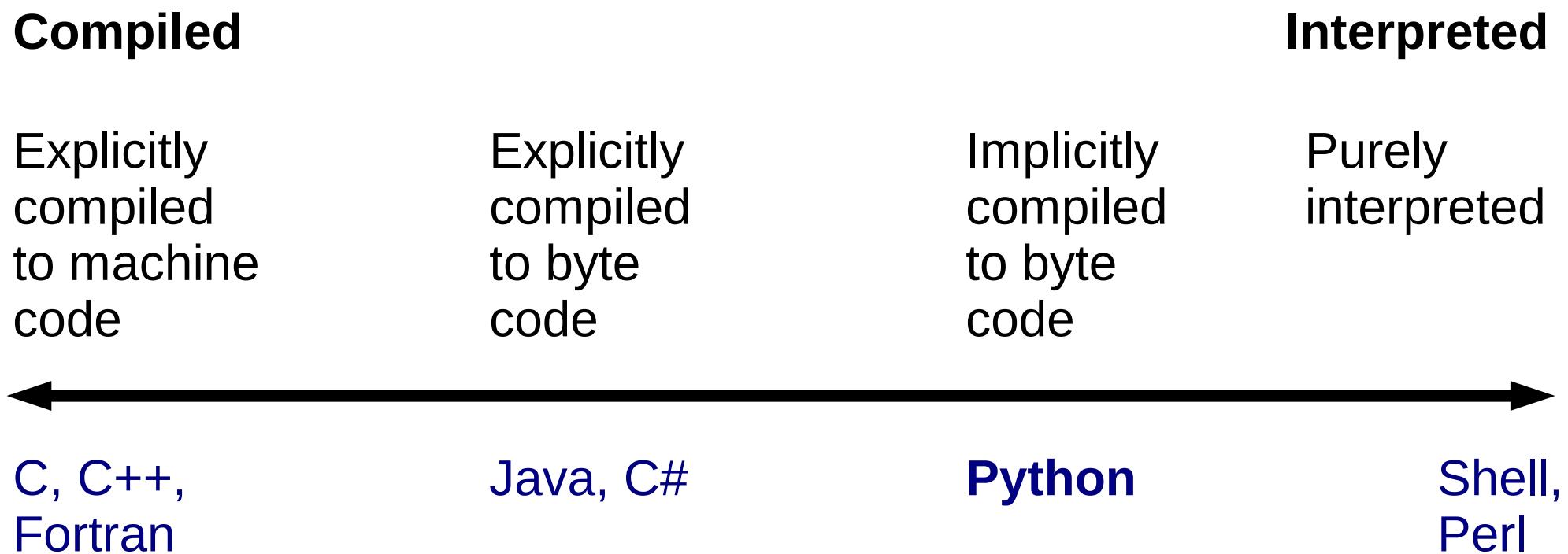


Embedded systems

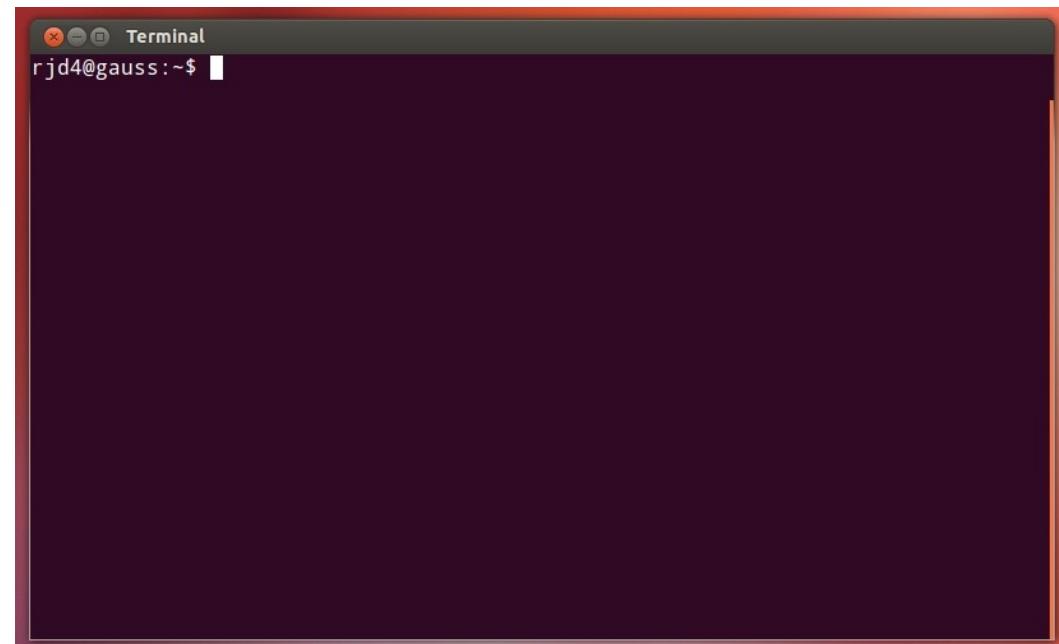


en.wikipedia.org/wiki/List_of_Python_software

What sort of language is Python?



Running Python — 1



Running Python — 2

```
$ python3
Python 3.2.3 (default, May 3 2012, 15:54:42)
[GCC 4.6.3] on linux2
>>>
```

Unix prompt

Unix command

Introductory blurb

Python version

Python prompt

Quitting Python

```
>>> exit()
```

```
>>> quit()
```

```
>>>  + 
```

Any one
of these

A first Python command

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

Hello, world!

```
>>>
```

The diagram illustrates the interaction between a Python prompt and a command. It shows a vertical sequence of text elements: a Python prompt, a command, the resulting output, and another Python prompt. Blue arrows point from each element to a corresponding label in a light blue box. The first arrow points from the top '>>>' to the label 'Python prompt'. The second arrow points from the command 'print('Hello, world!')' to the label 'Python command'. The third arrow points from the output 'Hello, world!' to the label 'Output'. The fourth arrow points from the bottom '>>>' back up to the label 'Python prompt'.

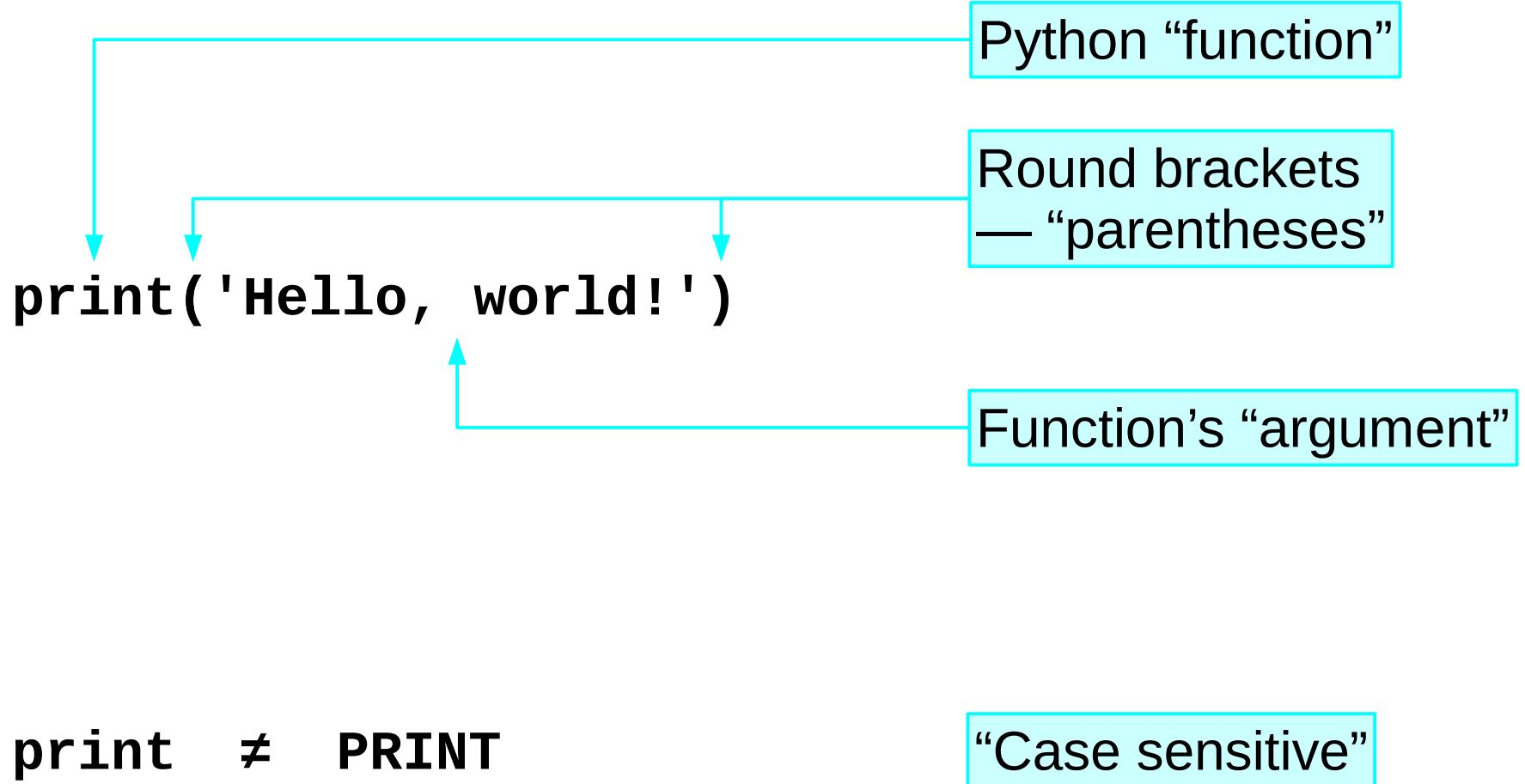
Python prompt

Python command

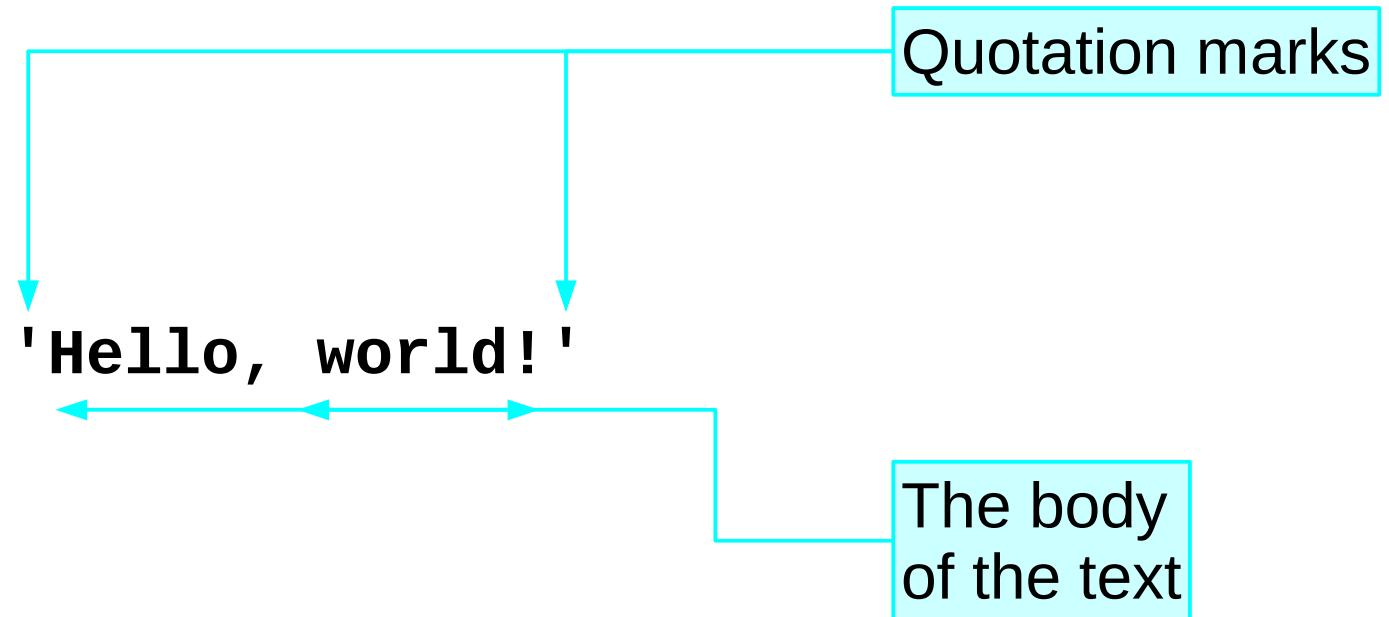
Output

Python prompt

Python commands



Python text



The quotes are not part of the text itself.

Quotes?

print → Command

'print' → Text

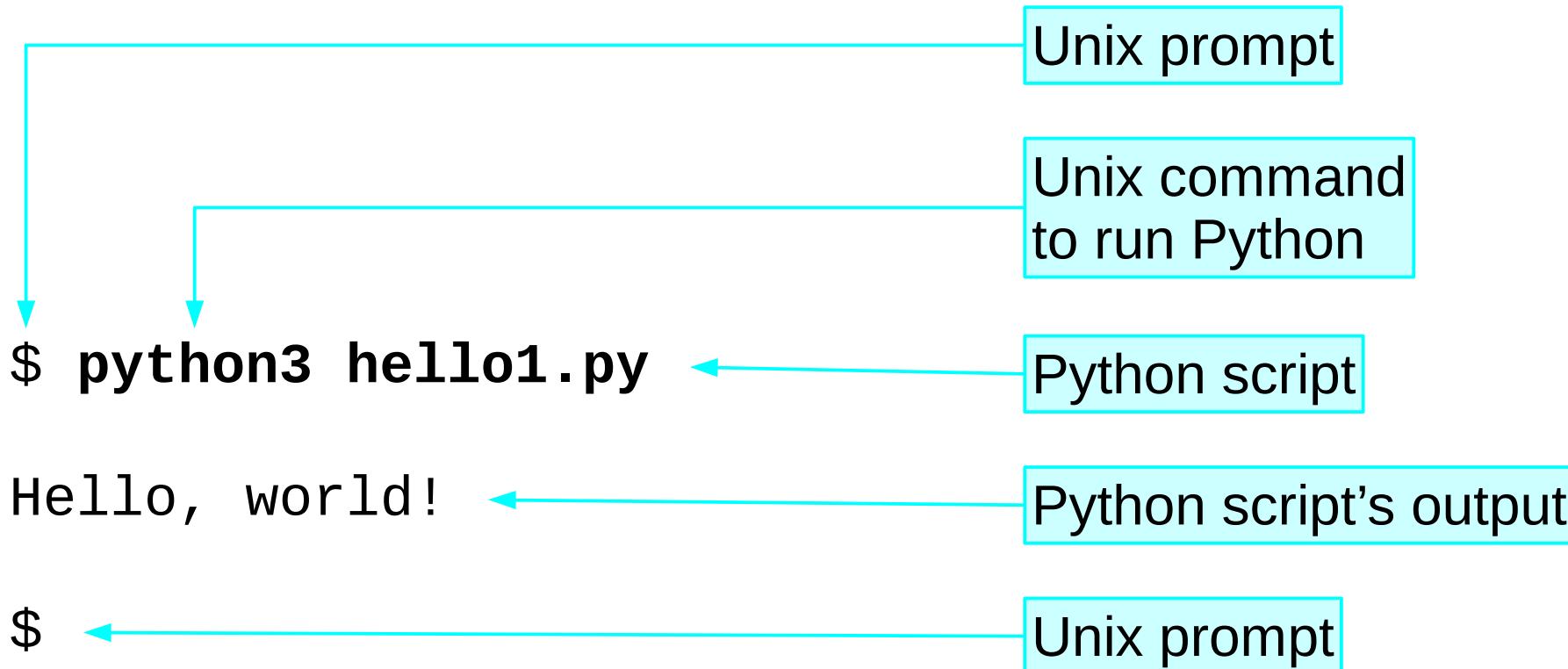
Python scripts

File in home directory

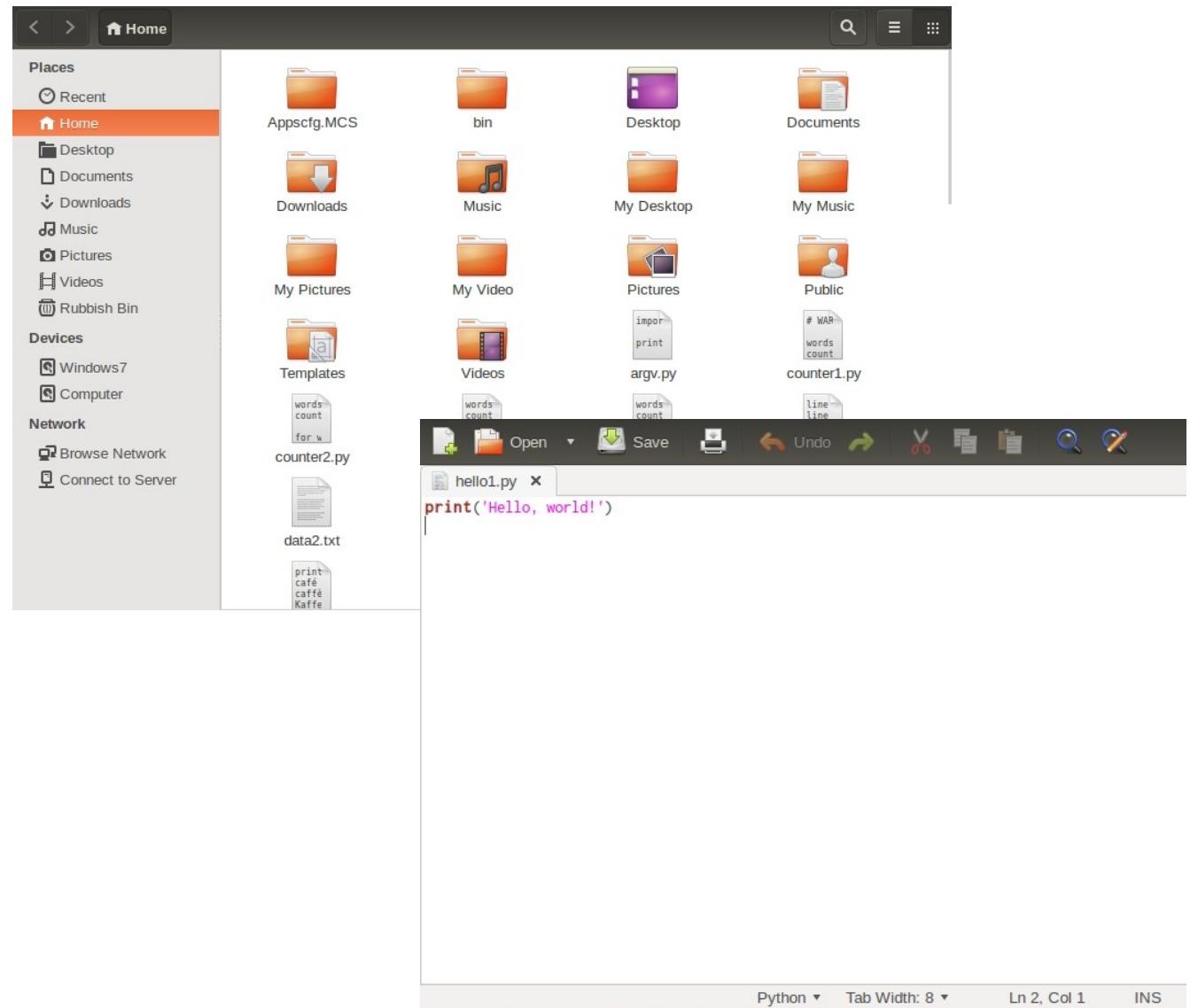
```
print('Hello, world!')
```

Run from *Unix* prompt

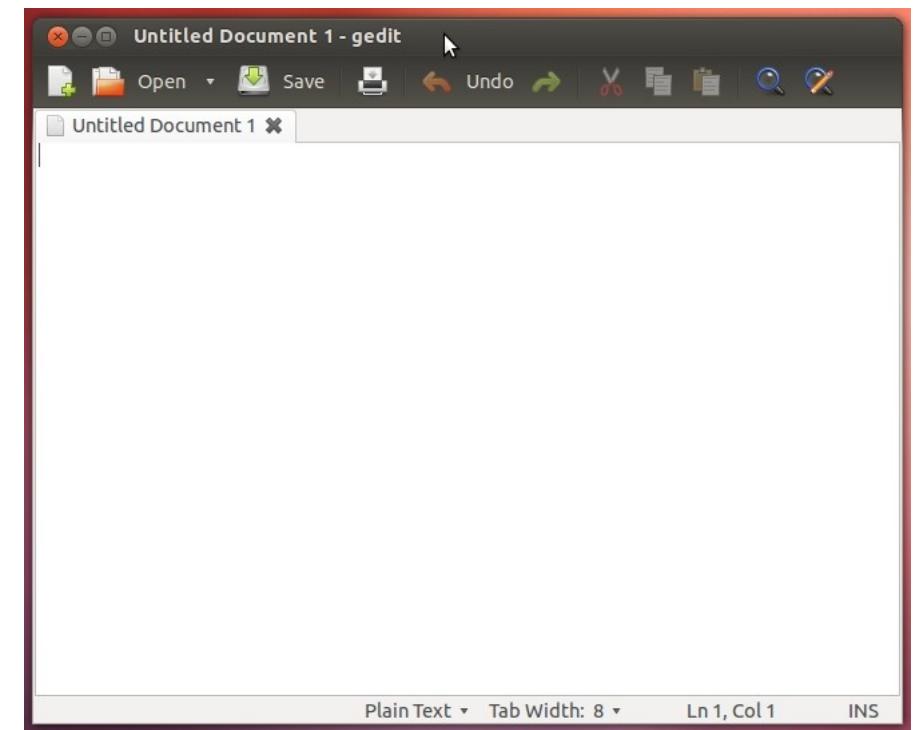
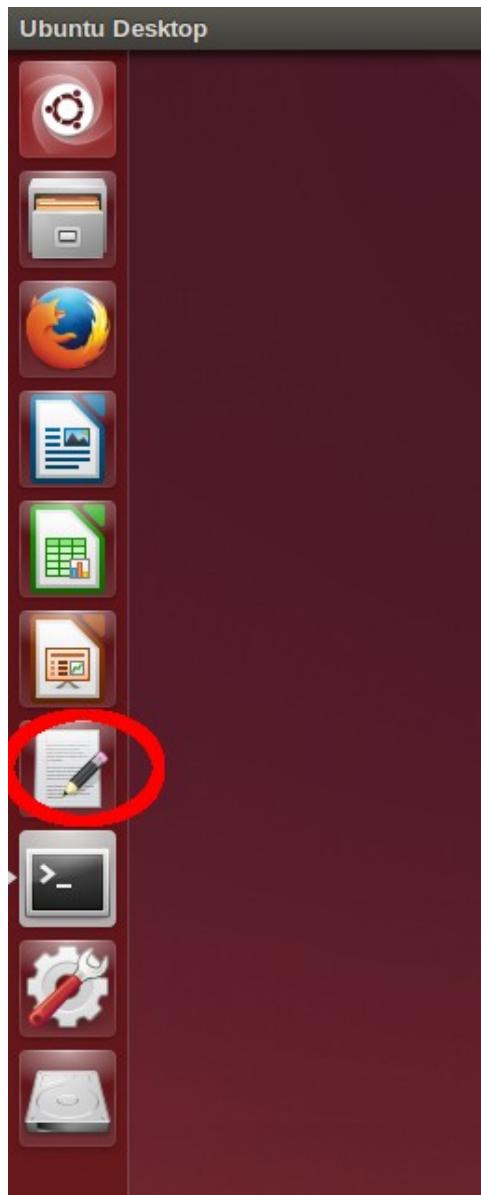
hello1.py



Editing Python scripts — 1



Editing Python scripts — 2



Progress

Interactive Python

Python scripts

`print()` command

Simple Python text

Exercise 1

1. Print “Goodbye, cruel world!” from interactive Python.
 2. Edit `exercise1.py` to print the same text.
 3. Run the modified `exercise1.py` script.
- Please ask if you have questions.



A little more text

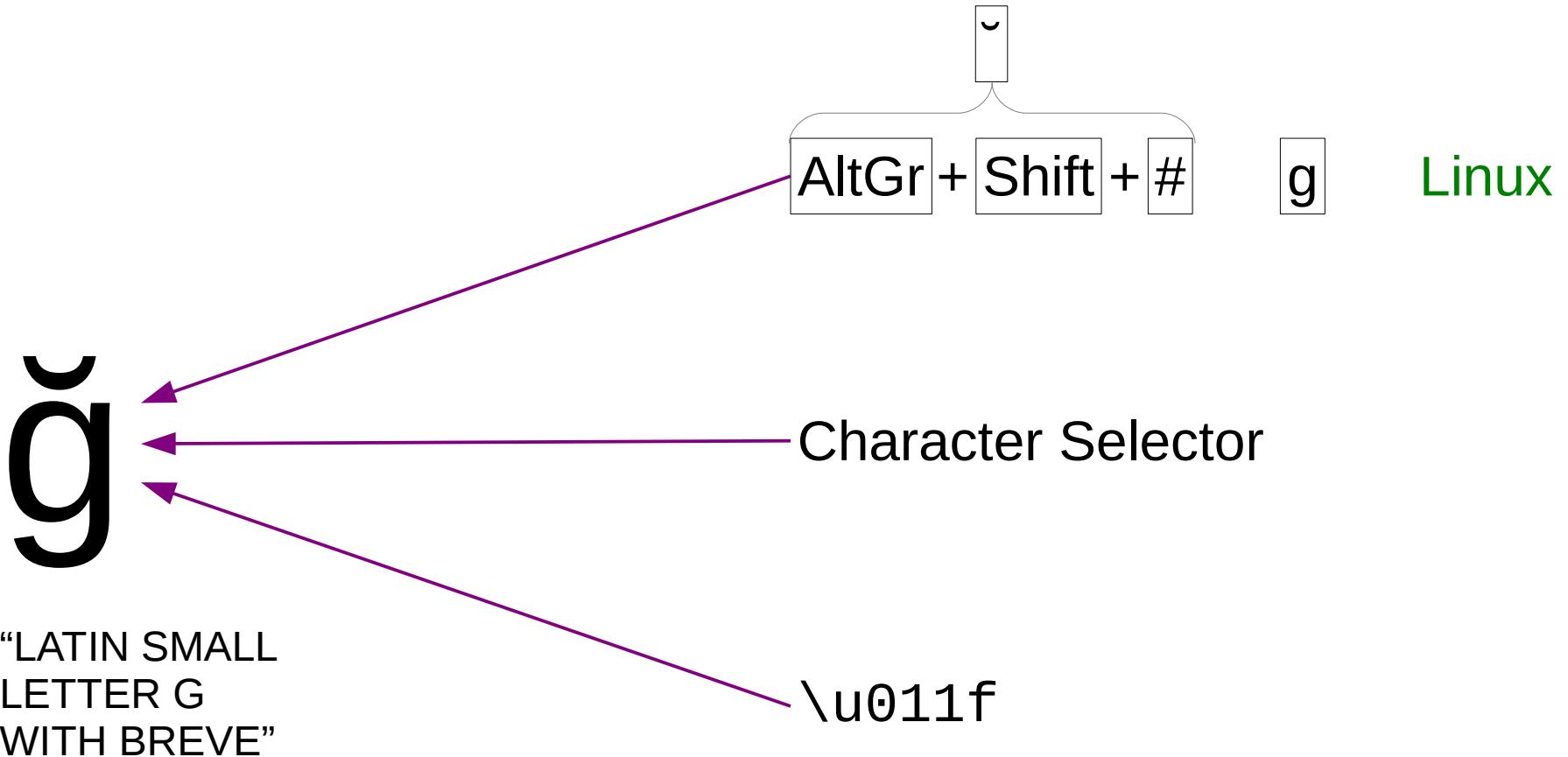
Full “Unicode” support

www.unicode.org/charts/

```
print('həllo, ω☺πø!!')
```

hello2.py

Getting characters



Text: a “string” of characters

```
>>> type('Hello, world!')
```

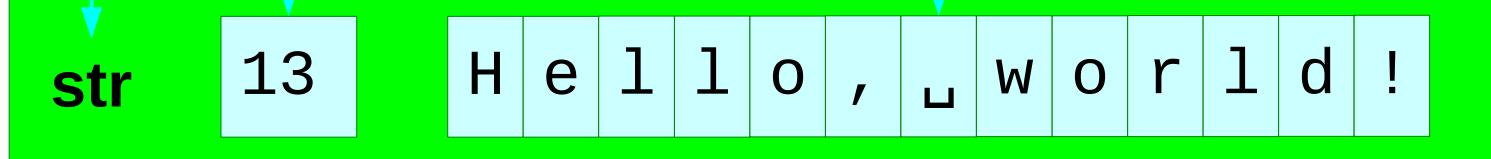
```
<class 'str'>
```

A string of characters

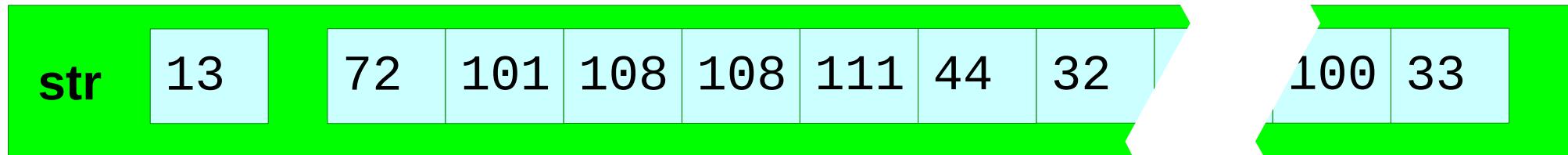
Class: string

Length: 13

Letters



Text: “behind the scenes”



```
>>> '\u011f'
```

011f₁₆

'g'

```
>>> ord('g')
```

287₁₀

287

```
>>> chr(287)
```

'g'

g

Adding strings together: +

“Concatenation”

```
print('Hello, ' + 'world!')
```

hello3.py

```
>>> 'Hello, ' + 'world!'
```

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

```
>>>
```

Pure concatenation

```
>>> 'Hello, ' + 'world! '
```

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

```
>>> 'Hello, ' + 'world! '
```

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

```
>>> 'Hello, ' + 'world! '
```

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

Only simple concatenation

No spaces added automatically.

Single & double quotes

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

Single quotes

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

Single quotes

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

Double quotes

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

Single quotes

Python strings: input & output

'Hello, world!'

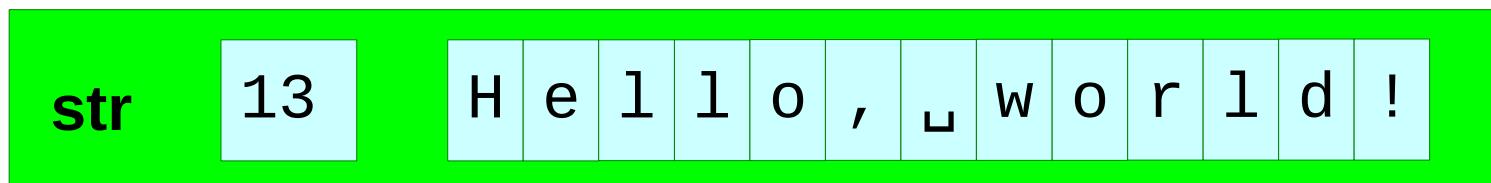
Single or double quotes on input.

"Hello, world!"

Create same string object.

'Hello, world! '

Single quotes on output.



Uses of single & double quotes

```
>>> print('He said "hello" to her.')
```

He said "hello" to her.

```
>>> print("He said 'hello' to her.")
```

He said 'hello' to her.

Why we need different quotes

```
>>> print('He said 'hello' to her.')
```

```
File "<stdin>", line 1
  print('He said 'hello' to her.')
                  ^
SyntaxError: invalid syntax
```



Adding arbitrary quotes

```
>>> print('He said \'hello\' to her.')
```

He said 'hello' to her.

\ ' → '

Just an ordinary character.

\ " → "

“Escaping”

str 23 He « said « ' hello ' « to « her .

Putting line breaks in text

Hello,
world!

What we want

```
>>> print('Hello,  world')
```

Try this

```
>>> print('Hello,  File "<stdin>", line 1  
      print('Hello,  
           ^
```

SyntaxError: EOL while
scanning string literal



“EOL”: End Of Line

Inserting “special” characters

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

```
Hello,  
world!
```

Treated as
a new line.

\n

Converted into a
single character.

```
str [13 H e l l o , ↵ w o r l d !]
```

```
>>> len('Hello,\nworld!')
```

```
13
```

`len()` function: gives
the length of the object

The backslash

Special → Ordinary

\ ' → '

\ " → "

Ordinary → Special

\n → ↴

\t → ↗

\n: unwieldy for long text

'SQUIRE TRELAWNEY, Dr. Livesey, and the\nrest of these gentlemen having asked me\nto write down the whole particulars\nabout Treasure Island, from the\nbeginning to the end, keeping nothing\nback but the bearings of the island,\nand that only because there is still\ntreasure not yet lifted, I take up my\npen in the year of grace 17\u00b7 and go\nback to the time when my father kept\nthe Admiral Benbow inn and the brown\nold seaman with the sabre cut first\ntook up his lodging under our roof.'



Single
line

Special input method for long text

'''SQUIRE TRELAWEY, Dr. Livesey, and the rest of these gentlemen having asked me to write down the whole particulars about Treasure Island, from the beginning to the end, keeping nothing back but the bearings of the island, and that only because there is still treasure not yet lifted, I take up my pen in the year of grace 17__ and go back to the time when my father kept the Admiral Benbow inn and the brown old seaman with the sabre cut first took up his lodging under our roof.'''

*Triple
quotes*

*Multiple
lines*

Python's “secondary” prompt

```
>>> '''Hello,  
... world'''
```

Python asking for more
of the same command.

It's still just text!

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

```
'Hello\nworld'
```



Python uses \n to represent line breaks in strings.

```
>>> '''Hello,  
... world!'''
```

```
'Hello\nworld'
```

Exactly the same!

Your choice of input quotes:

Four inputs:

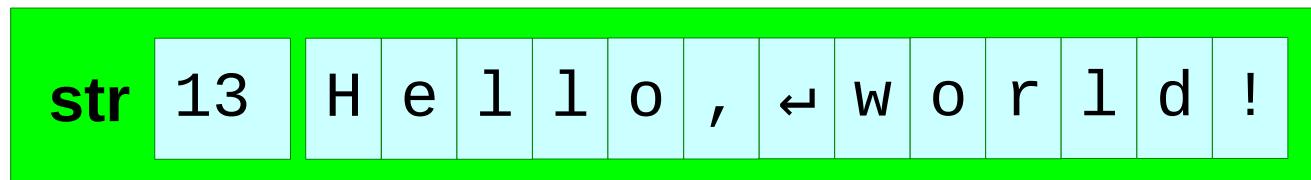
'Hello, \nworld! '

"Hello, \nworld!"

'''Hello,
world!'''

"""Hello,
world!"""

Same result:



Progress

International text

`print()`

Concatenation of strings

Special characters

Long strings

Exercise 2

1. Replace **XXXX** in `exercise2.py` so it prints the following text (with the line breaks) and then run the script.

coffee

café

caffè

Kaffee

é

\u00e8

AltGr + ; e

è

\u00e9

AltGr + # e



3 minutes

Attaching names to values

“variables”

```
>>> message='Hello, world!'
```

```
>>> message
```

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

```
>>> type(message)
```

```
<class 'str'>
```

```
message = 'Hello, world!'  
print(message)
```

hello3.py

message

str 13 H e l l o , w o r l d !

Attaching names to values

```
>>> type(print)
```

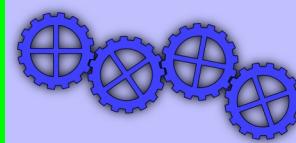
```
<class 'builtin_function_or_method'>
```

```
message = 'Hello, world!'
print(message)
```

hello4.py

print

function



message

str

13

H e l l o , u w o r l d !

Reading some text into a script

```
message = input('Yes? ')  
print(message)
```

```
$ python3 input1.py
```

```
input('Yes? ')
```

```
Yes? Boo!
```

```
message = ...
```

```
Boo!
```

```
print(message)
```

Can't read numbers directly!

```
$ python3 input2.py
```

```
N? 10
```

```
number = input('N? ')  
print(number + 1)
```



input2.py

```
Traceback (most recent call last):  
  File "input2.py", line 2, in <module>  
    print(number + 1)  
TypeError:  
  Can't convert 'int' object  
  to str implicitly
```

string

integer

input(): strings only

```
$ python3 input2.py
```

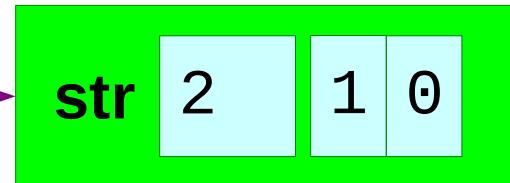
```
N? 10
```

```
number = input('N? ')  
print(number + 1)
```



input2.py

```
input('N? ')
```



≠



Some more types

```
>>> type('Hello, world!')
```

```
<class 'str'> ← string of characters
```

```
>>> type(42)
```

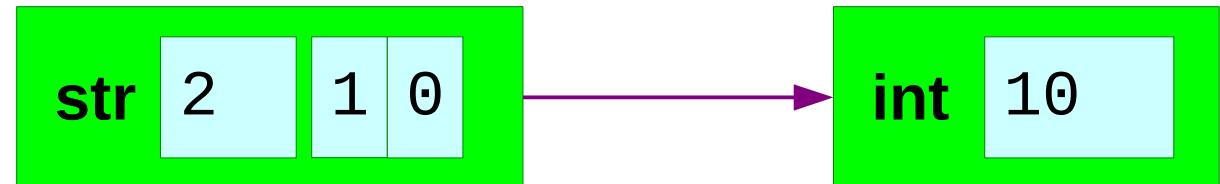
```
<class 'int'> ← integer
```

```
>>> type(3.14159)
```

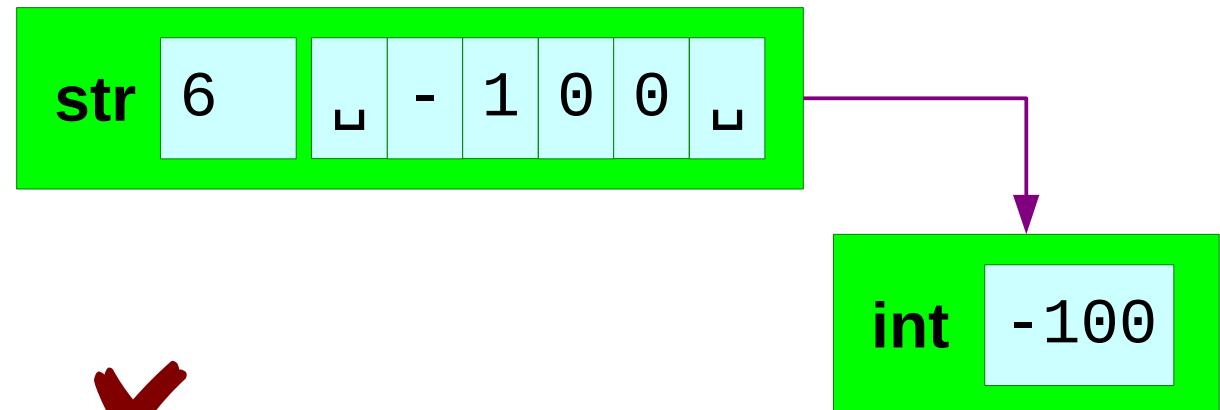
```
<class 'float'> ← floating point number
```

Converting text to integers

```
>>> int('10')  
10
```



```
>>> int(' -100 ')  
-100
```



```
>>> int('100-10')
```



```
ValueError:  
invalid literal for int() with base 10: '100-10'
```

Converting text to floats

```
>>> float('10.0')
```

'10.0' is a string

```
10.0
```

10.0 is a floating
point number

```
>>> float('10.')  
10.0
```

Converting between ints and floats

```
>>> float(10)
```

```
10.0
```

```
>>> int(10.9)
```

```
10
```

Truncates
fractional part

```
>>> int(-10.9)
```

```
-10
```

Converting into text

```
>>> str(10)
```

integer → string

```
'10'
```

```
>>> str(10.000)
```

float → string

```
'10.0'
```

Converting between types

<code>int()</code>	anything → integer
<code>float()</code>	anything → float
<code>str()</code>	anything → string

Functions named after the type they convert *into*.

Reading numbers into a script

```
$ python3 input3.py
```

N? 10

11

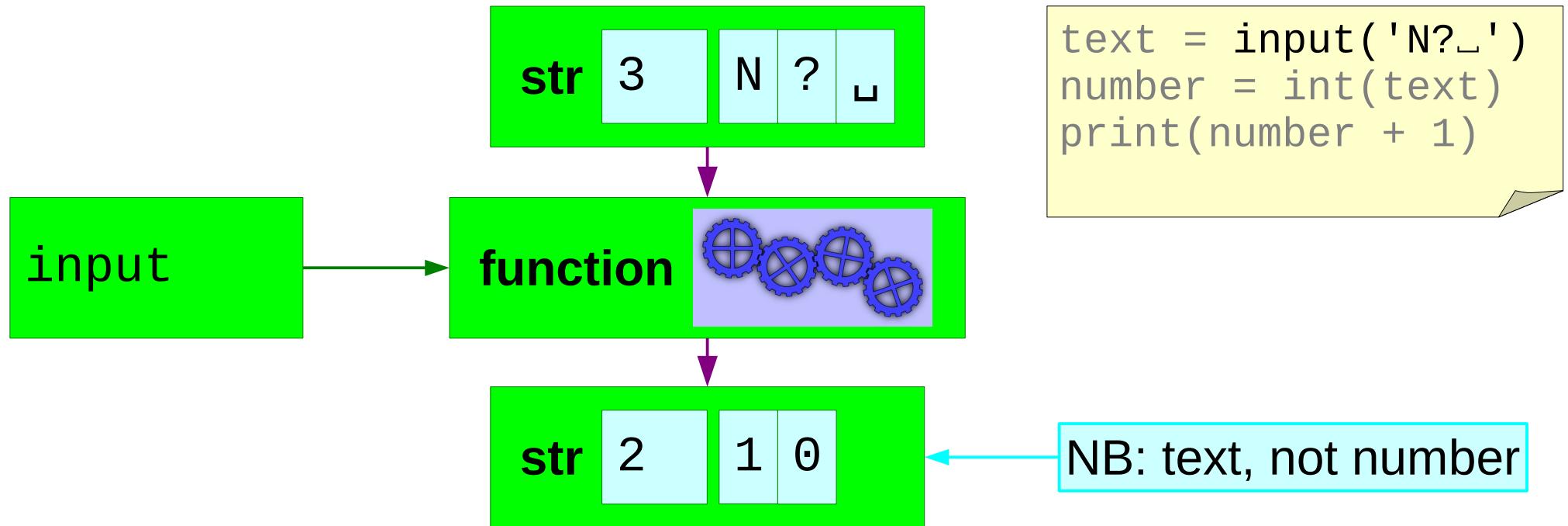
```
text = input('N? ')
number = int(text)
print(number + 1)
```

Stepping through our script — 1

```
str 3 N ? ↵
```

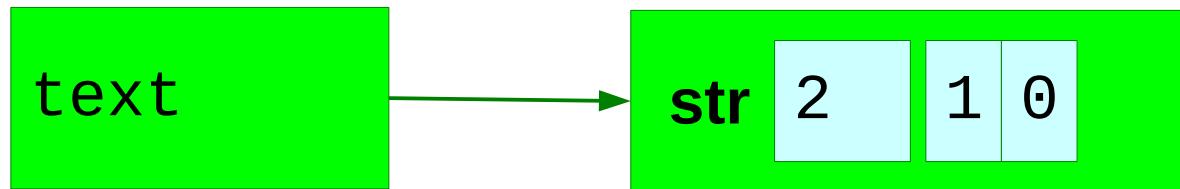
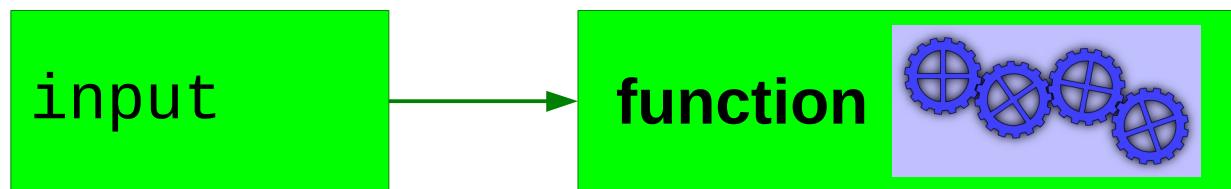
```
text = input('N?')  
number = int(text)  
print(number + 1)
```

Stepping through our script — 2



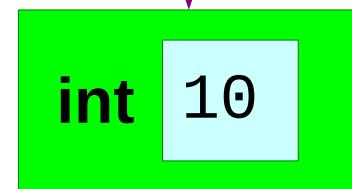
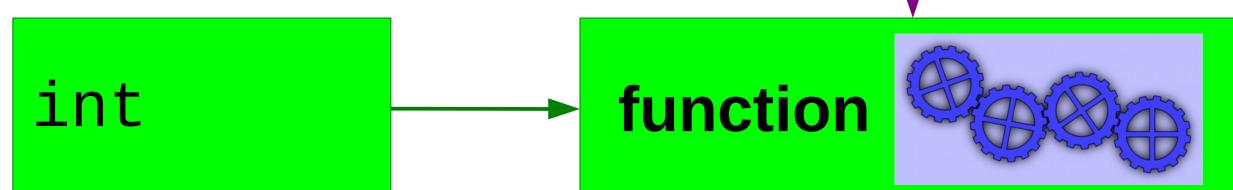
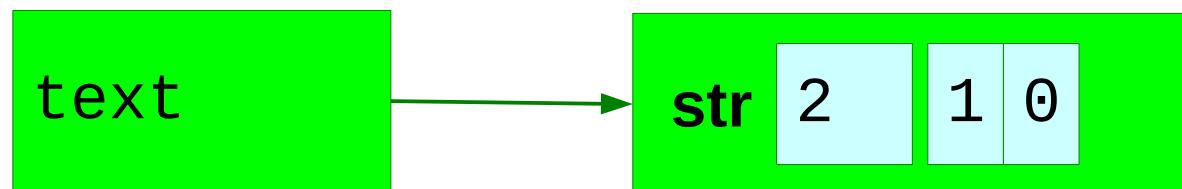
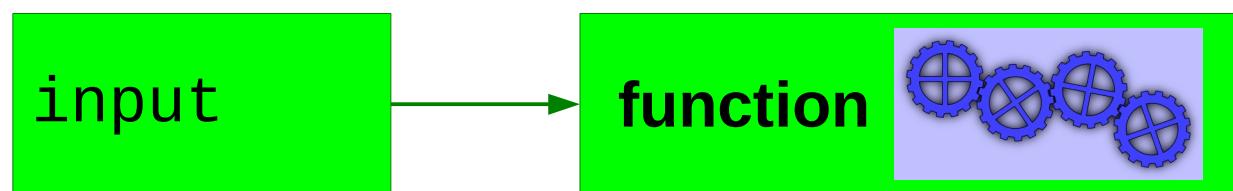
Stepping through our script — 3

```
text = input('N?_')
number = int(text)
print(number + 1)
```



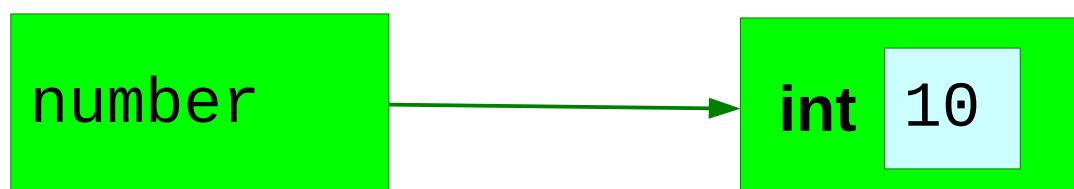
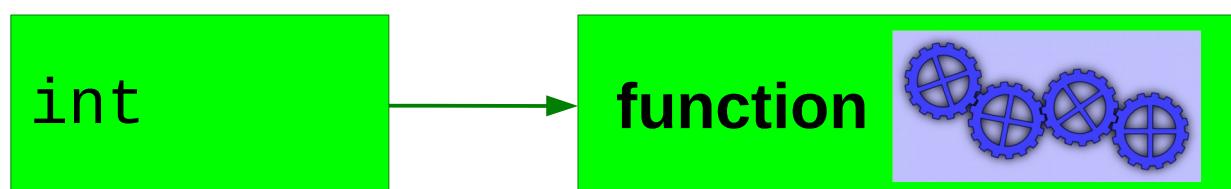
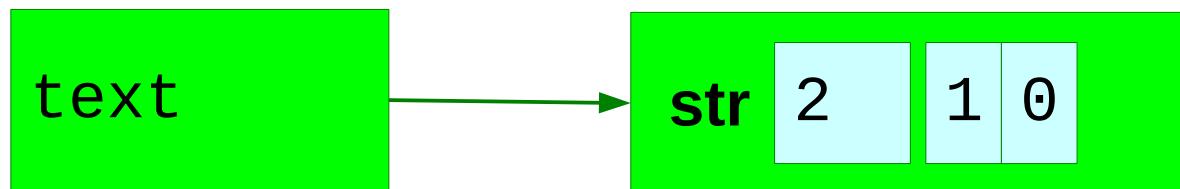
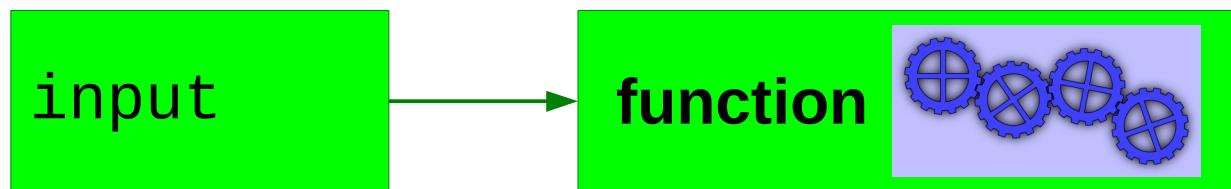
Stepping through our script — 4

```
text = input('N?')
number = int(text)
print(number + 1)
```

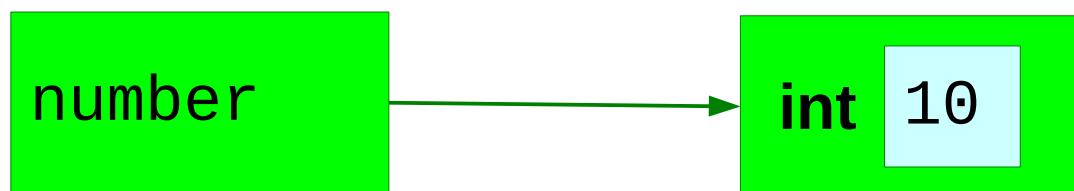


Stepping through our script — 5

```
text = input('N?')
number = int(text)
print(number + 1)
```

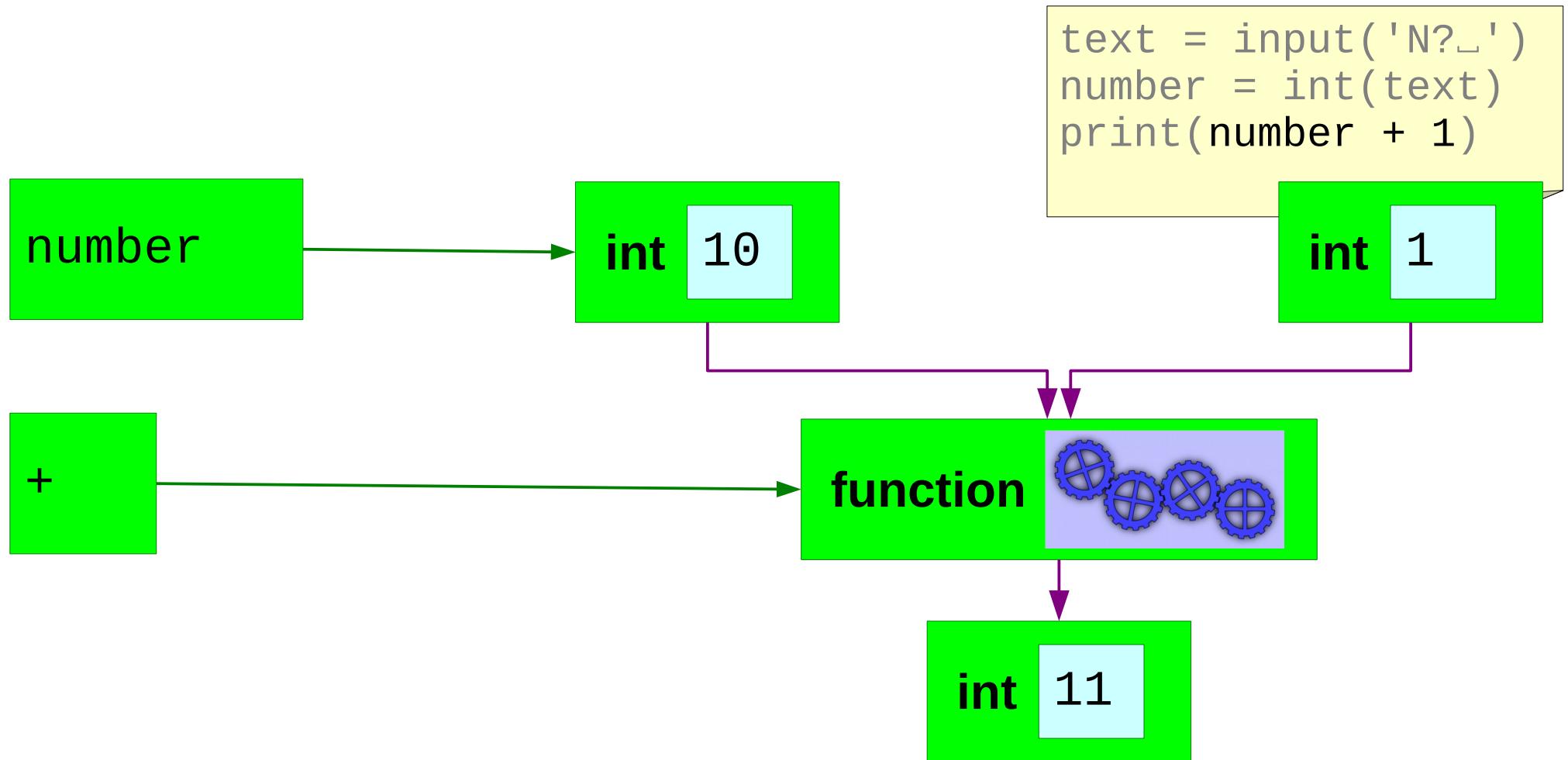


Stepping through our script — 6



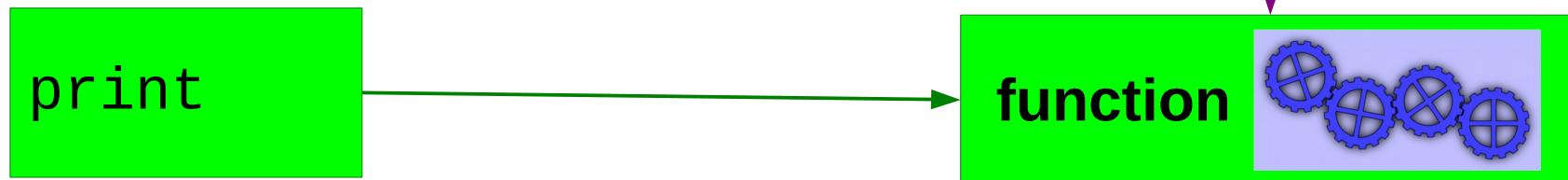
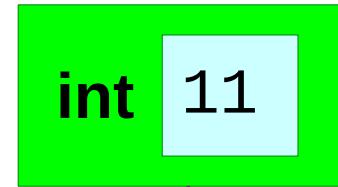
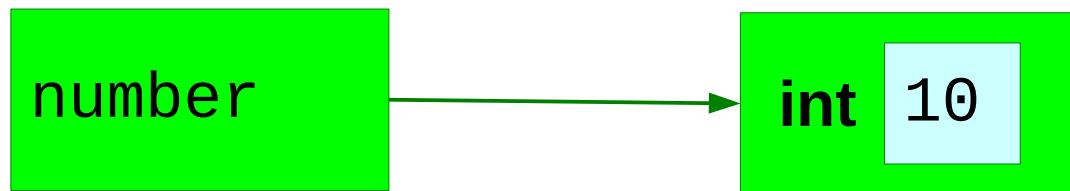
```
text = input('N?_')
number = int(text)
print(number + 1)
```

Stepping through our script — 7



Stepping through our script — 6

```
text = input('N?')
number = int(text)
print(number + 1)
```



Progress

Names → Values

name = value

Types

strings

integers

floating point numbers

Reading in text

`input(prompt)`

Type conversions

`str()` `int()` `float()`

Exercise 3

Replace the two **XXXX** in `exercise3.py` to do the following:

1. Prompt the user with the text “**How much?** ”.
2. Convert the user’s answer to a floating point number.
3. Print 2.5 plus that number.



Integers

$$\mathbb{Z} \{ \dots -2, -1, 0, \\ 1, 2, 3, 4 \dots \}$$

Integer addition & subtraction

```
>>> 20+5
```

```
25
```

```
>>> 20 - 5
```

```
15
```

Spaces around the operator don't matter.

“No surprises”

Integer multiplication

There is no “ \times ” on the keyboard.

Use “ $*$ ” instead

```
>>> 20 * 5
```

```
100
```

Linux:

AltGr + Shift + ,

Still no surprises

Integer division

There is no “÷” on the keyboard.

Use “/” instead

```
>>> 20 / 5
```

```
4.0
```

Linux:

AltGr + Shift + .

This is a floating point number!

Surprise!

Integer division gives floats !



Fractions → Floats sometimes

Consistency → Floats **always**

```
>>> 20 / 40
```

```
0.5
```

```
>>> 20 / 30
```

```
0.6666666666666666
```

Integer powers

There is no “ 4^2 ” on the keyboard.

Use “`**`” instead

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

Spaces around the operator don't matter.

```
>>> 4 * * 2  
SyntaxError: invalid syntax
```

Spaces in the operator do!

Integer remainders

e.g. Is a number even or odd?

Use “%”

```
>>> 4 % 2
```

```
0
```

```
>>> 5 % 2
```

```
1
```

```
>>> -5 % 2
```

```
1
```

Remainder is always non-negative

How big can a Python integer be?

```
>>> 2**2
```

```
4
```

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

```
16
```

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

```
256
```

```
>>> 256**2
```

```
65536
```

```
>>> 65536**2
```

```
4294967296
```

How big can a Python integer be?

```
>>> 4294967296**2
```

```
18446744073709551616
```

```
>>> 18446744073709551616**2
```

```
340282366920938463463374607431768211456
```

```
>>> 340282366920938463463374607431768211456**2
```

```
1157920892373161954235709850086879078532699846  
65640564039457584007913129639936
```

```
>>> 115792089237316195423570985008687907853269  
984665640564039457584007913129639936**2
```

```
1340780792994259709957402499820584612747936582  
0592393377723561443721764030073546976801874298  
1669034276900318581864860508537538828119465699  
46433649006084096
```

How big can a Python integer be?

10443888814131525066917527107166243825799642490473837803842334832839
53907971557456848826811934997558340890106714439262837987573438185793
60726323608785136527794595697654370999834036159013438371831442807001
18559462263763188393977127456723346843445866174968079087058037040712
84048740118609114467977783598029006686938976881787785946905630190260
94059957945343282216020202020002142050025015072020202771 421554169383555
98852914863182379 413490084170616
75093668333850551 213796825837188
09183365675122131 259567449219461
70238065059132456 982023131690176
78006675195485079921636419370285375124784014907159135459982790513399
6115517942711068311340905842728842797915548497829543225215170652226
9061394905987693002122963395687782878948440616007412 Except for 05
7164237715481632138063104590291613692670834285644073 machine 81
4657634732238502672530598997959960907994692017746248 memory 65
9250178329070473119433165550807568221846571746373296 74
5700244092661691087414838507841192980452298185733897 , 04010512000003
00130241346718972667321649151113160292078173803343609024380470834040
3154190336

There is no limit!

Except for machine memory	05 81 65 74
---------------------------------	----------------------

Big integers

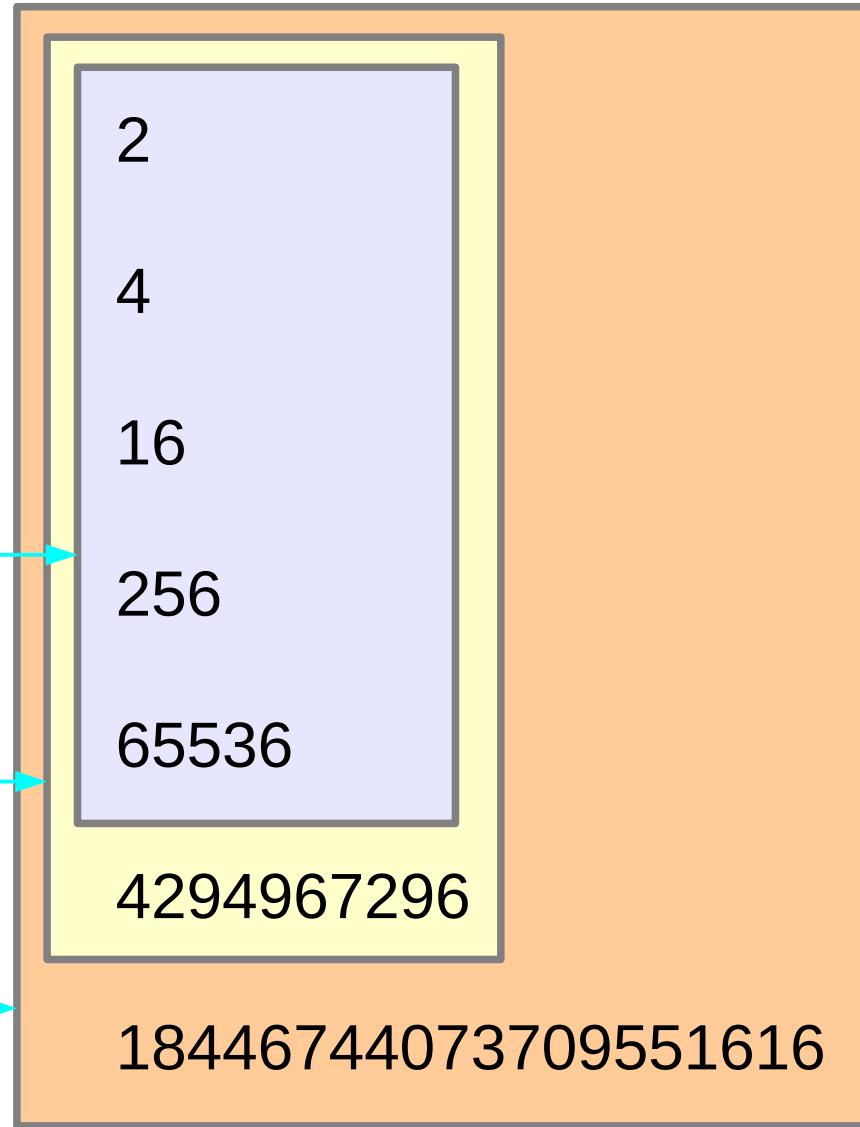
C / C++
Fortran

int
INTEGER*4

long
INTEGER*8

long long
INTEGER*16

Out of the reach
of C or Fortran!



3402823669209384634...
63374607431768211456

Floating point numbers

~~R~~

1.0

0.33333333

3.14159265

2.71828182

Basic operations

```
>>> 20.0 + 5.0
```

```
25.0
```

```
>>> 20.0 * 5.0
```

```
100.0
```

```
>>> 20.0 ** 5.0
```

```
3200000.0
```

```
>>> 20.0 - 5.0
```

```
15.0
```

```
>>> 20.0 / 5.0
```

```
4.0
```

Equivalent to integer arithmetic

Floating point imprecision

```
>>> 1.0 / 3.0
```

```
0.3333333333333333
```

```
>>> 10.0 / 3.0
```

```
3.333333333333335
```

If you are relying on
this last decimal place,
you are doing it wrong!

≈ 17 significant figures

Hidden imprecision



```
>>> 0.1
```

```
0.1
```

```
>>> 0.1 + 0.1
```

```
0.2
```

```
>>> 0.1 + 0.1 + 0.1
```

```
0.30000000000000004
```

Really: if you are relying on
this last decimal place,
you are doing it wrong!



How big can a Python float be? — 1

```
>>> 65536.0**2
```

```
4294967296.0
```

So far, so good.

```
>>> 4294967296.0**2
```

```
1.8446744073709552e+19
```

Switch to
“scientific notation”

1.8446744073709552 e+19

1.8446744073709552 ×10¹⁹

Floats are not exact

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

Floating point

```
>>> 4294967296**2  
18446744073709551616
```

Integer

$1.8446744073709552 \times 10^{19}$ → 18446744073709552000

- 18446744073709551616

How big can a Python float be? — 2

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

```
3.402823669209385e+38
```

```
>>> 3.402823669209385e+38**2
```

```
1.157920892373162e+77
```

```
>>> 1.157920892373162e+77**2
```

```
1.3407807929942597e+154
```

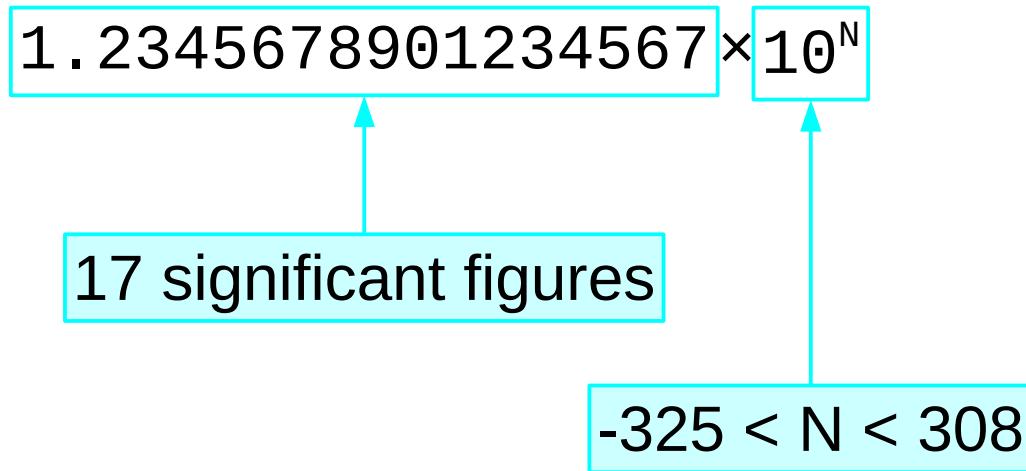
```
>>> 1.3407807929942597e+154**2
```

```
OverflowError: (34,  
'Numerical result out of range')
```

So far, so good.

Too big!

Floating point limits

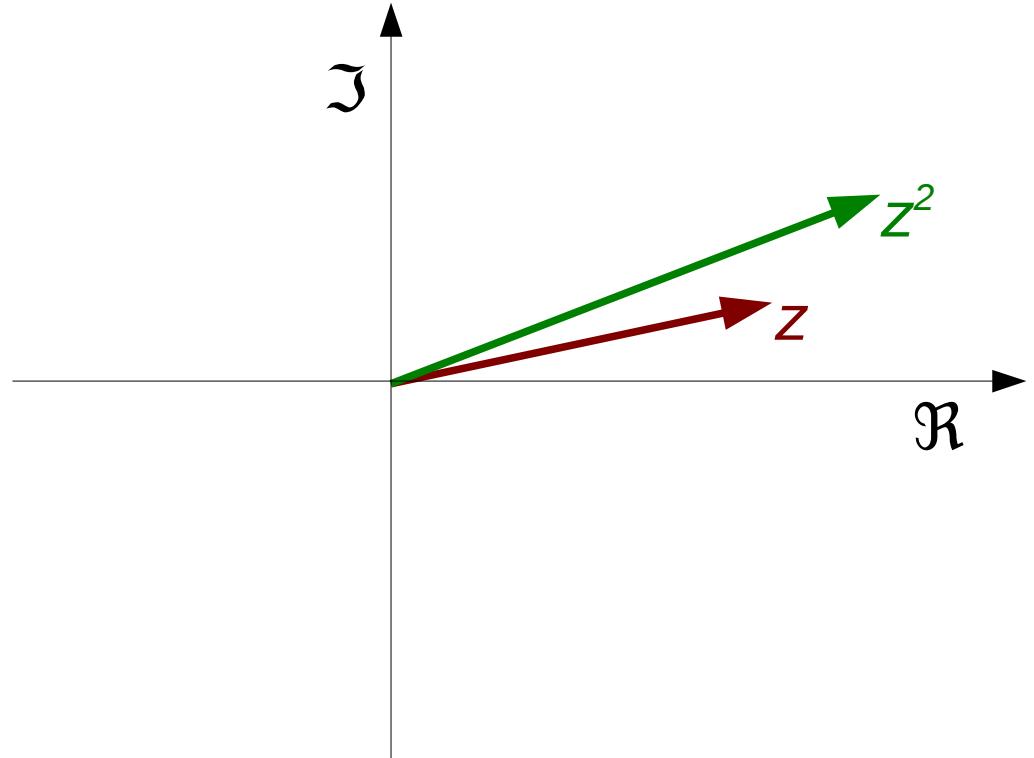


Positive values:

$$4.94065645841 \times 10^{-324} < N < 8.98846567431 \times 10^{307}$$

Complex numbers

C



```
>>> (1.25+0.5j)**2  
(1.3125+1.25j)
```

Progress

Arithmetic

+ - * / ** %

Integers

No limits!

Floating point numbers

Limited size

Limited precision

Complex numbers

Exercise 4

Replace the **XXXX** in `exercise4.py` to evaluate and print out the following calculations:

1. $223 \div 71$
2. $(1 + 1/10)^{10}$
3. $(1 + 1/100)^{100}$
4. $(1 + 1/1000)^{1000}$



Comparisons

$5 < 10$



$5 > 10$



Comparisons

```
>>> 5 < 10
```

Asking the question

True ✓

```
>>> 5 > 10
```

Asking the question

False ✗

True & False

```
>>> type(True)
```

```
<class 'bool'>
```

“Booleans”

5 + 10

15

int

int

int

5 < 10

True

bool

True & False

bool ✓

bool ✗

True

False

Only two values

Six comparisons

Maths

=

≠

<

>

≤

≥

Python

==

!=

<

>

<=

>=

Double equals sign

Equality comparison & assignment

==

`name = value`

Attach a name to a value.

==

`value1 == value2`

Compare two values

Textual comparisons

```
>>> 'cat' < 'dog'
```

Alphabetic ordering

True

```
>>> 'Cat' < 'cat'
```

Uppercase before lowercase

True

```
>>> 'Dog' < 'cat'
```

All uppercase before lowercase

True

Ordering text is *complicated*

Python inequalities use Unicode character numbers.

This is over-simplistic for “real” use.

“Collation” is a whole field of computing in itself

Alphabetical order?

German: z < ö

Swedish: ö < z

“Syntactic sugar”

$0 < \text{number} < 10$  $0 < \text{number}$ and
 $\text{number} < 10$

```
>>> number = 5
```

```
>>> 0 < number < 10
```

True

Converting to booleans

float()

Converts to floating point numbers

`<class 'float'>`

int()

Converts to integers

`<class 'int'>`

str()

Converts to strings

`<class 'str'>`

bool()

Converts to booleans

`<class 'bool'>`

Useful conversions

'' → False

Empty string

'Fred' → True

Non-empty string

0 → False

Zero

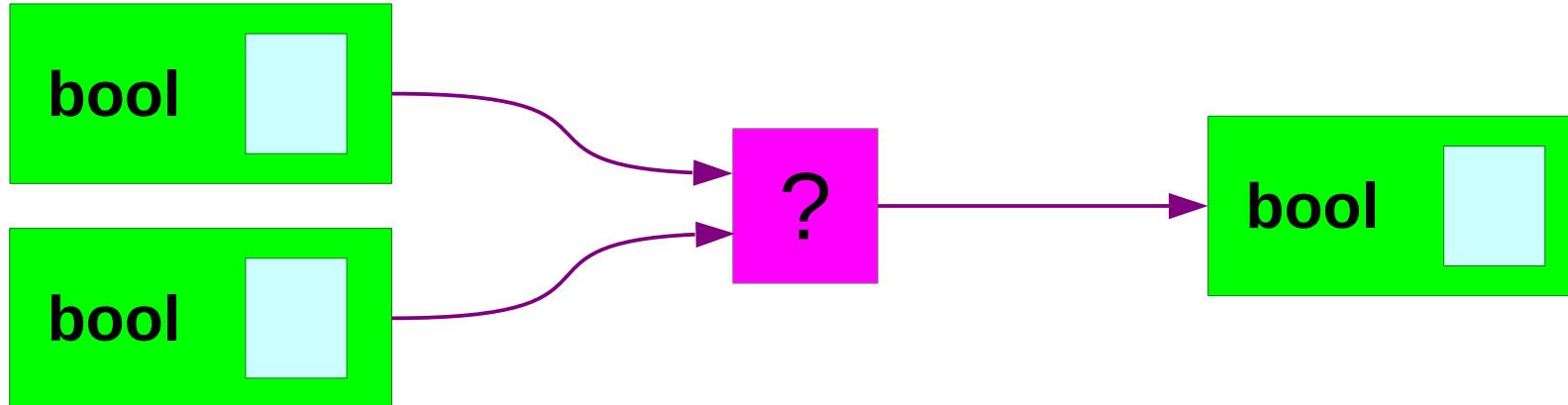
1 → True

Non-zero

12 → True

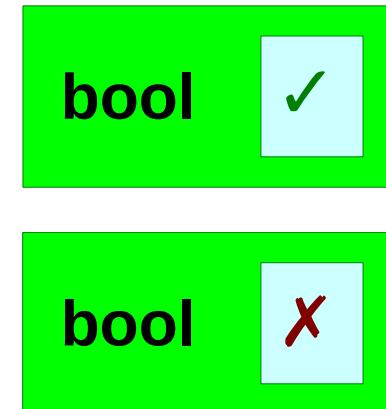
-1 → True

Boolean operations

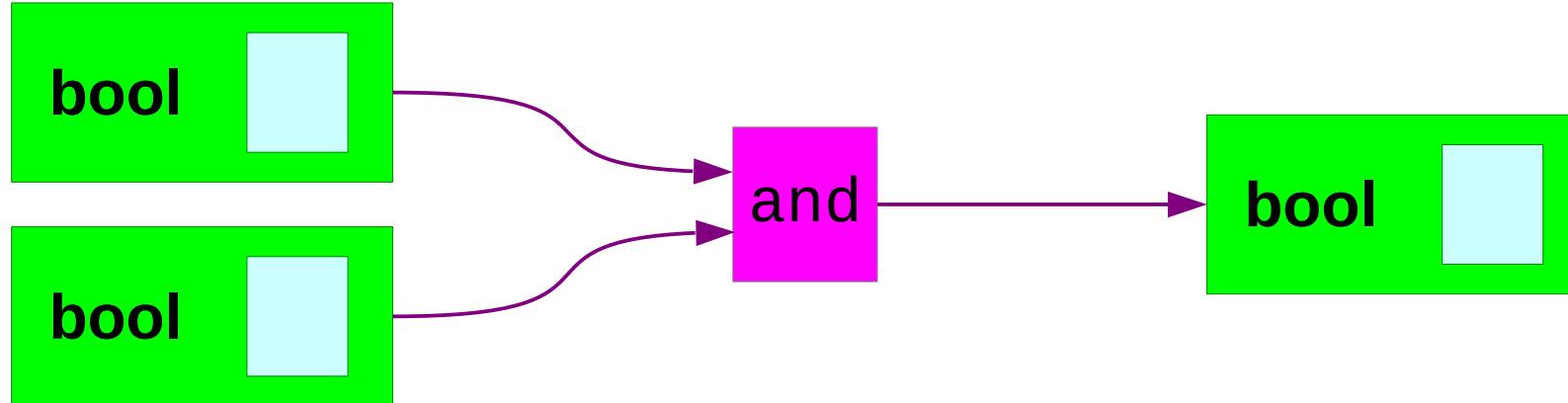


Numbers have $+$, $-$, $*$...

What do booleans have?



Boolean operations — “and”



True and True



True

Both have
to be True

True and False



False

False and True



False

False and False



False

Boolean operations — “and”

```
>>> 4 < 5 and 6 < 7
```

```
True
```

4 < 5 → True

6 < 7 → True

} and → True

```
>>> 4 < 5 and 6 > 7
```

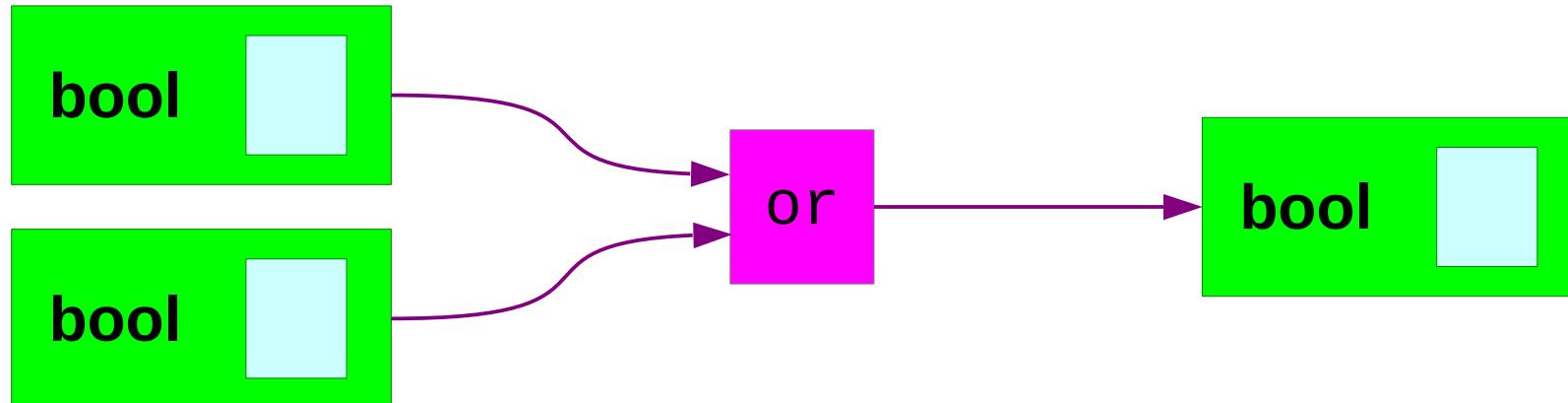
```
False
```

4 < 5 → True

6 > 7 → False

} and → False

Boolean operations — “or”



True or True → True

True or False → True

False or True → True

False or False → False

At least one has to be True

Boolean operations — “or”

```
>>> 4 < 5 or 6 < 7
```

```
True
```

$4 < 5 \rightarrow \text{True}$

$6 < 7 \rightarrow \text{True}$

} or $\rightarrow \text{True}$

```
>>> 4 < 5 or 6 > 7
```

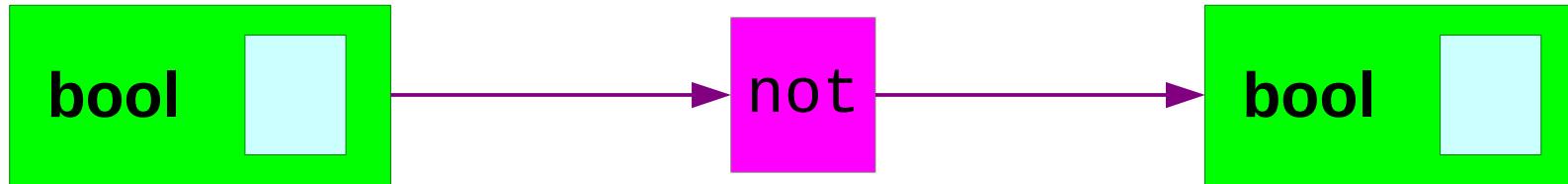
```
True
```

$4 < 5 \rightarrow \text{True}$

$6 > 7 \rightarrow \text{False}$

} or $\rightarrow \text{True}$

Boolean operations — “not”



not True → False

not False → True

Boolean operations — “not”

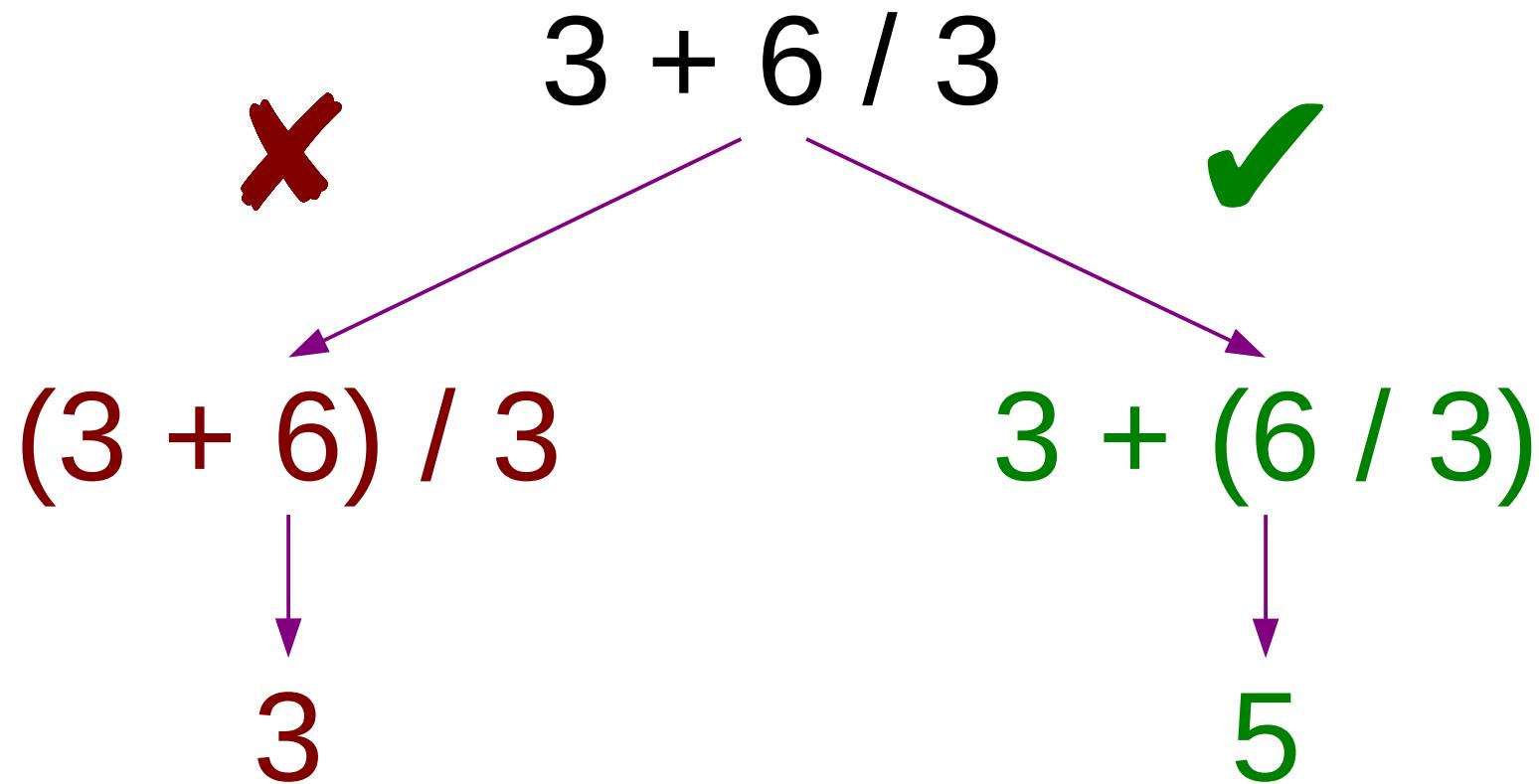
```
>>> not 6 < 7           6 < 7 → True — not → False
```

False

```
>>> not 6 > 7           6 > 7 → False — not → True
```

True

Ambiguity in expressions



Division before addition

Division first

$$3 + 6 / 3$$



$$3 + 2$$



$$5$$

Addition second

“Order of precedence”

First

$x^{**}y$ $-x$ $+x$ $x\%y$ x/y x^*y $x-y$ $x+y$

$x==y$ $x!=y$ $x>=y$ $x>y$ $x<=y$ $x<y$

not x x **and** y x **or** y

Last

Progress

Comparisons

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

Numerical comparison

`5 < 7`

Alphabetical ordering

`'dig' < 'dug'`

Booleans

`True False`

Boolean operators

`and or not`

Conversions

`'' → False
0 → False
0.0 → False`

`other → True`

Exercise 5

Predict whether these expressions will evaluate to True or False.
Then try them.

1. 'sparrow' > 'eagle'

2. 'dog' > 'Cat' or 45 % 3 == 0

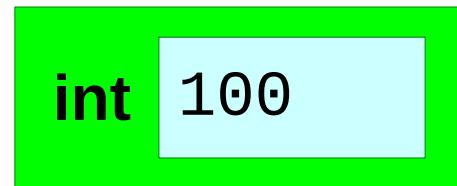
3. 60 - 45 / 5 + 10 == 1



Names and values: “assignment”

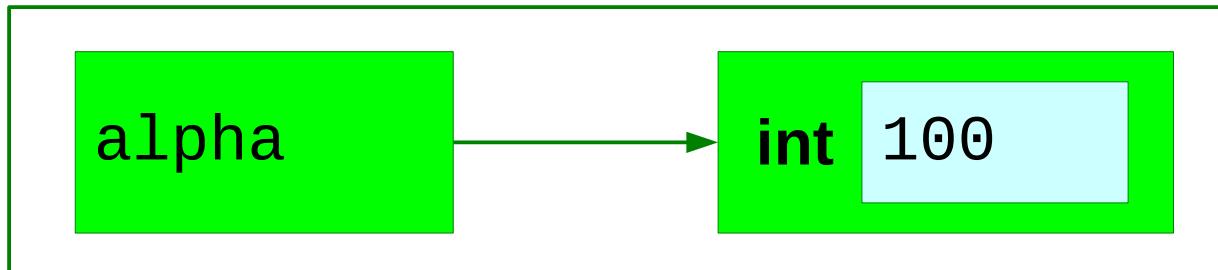
```
>>> alpha = 100
```

1. `alpha = 100`



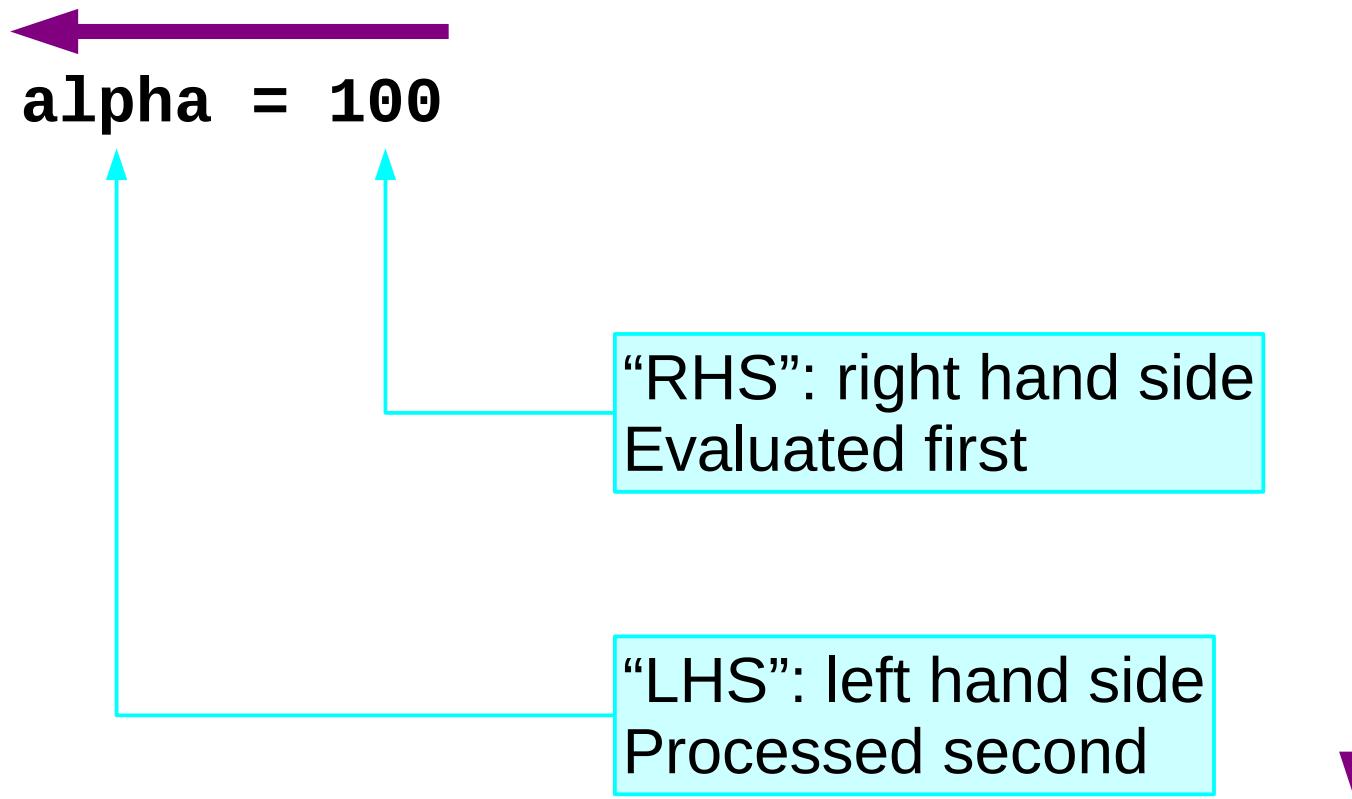
Python creates an “integer 100” in memory.

2. `alpha = 100`



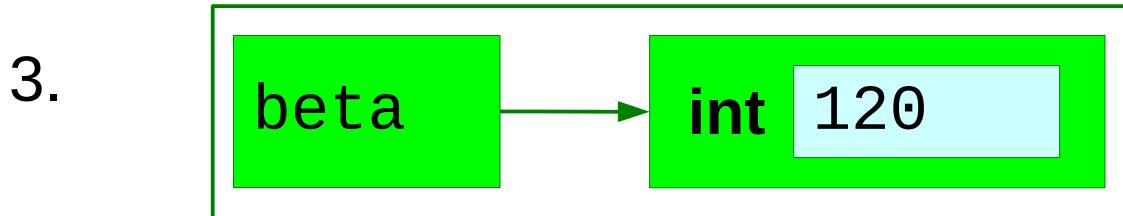
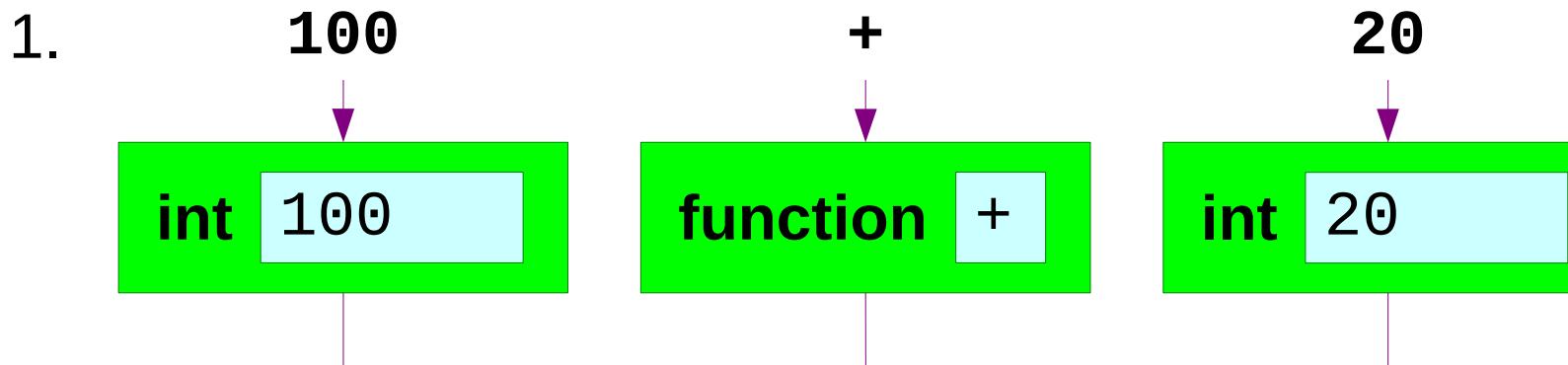
Python attaches the name “alpha” to the value.

Assignment: right to left



Simple evaluations

```
>>> beta = 100 + 20
```



Changing value — 1

```
>>> gamma = 1
```

```
>>> gamma = 2
```

Changing value — 2

RHS

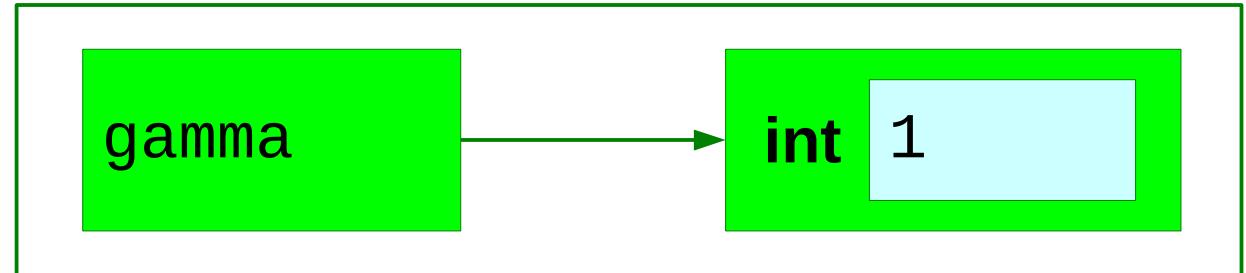
```
>>> gamma = 1
```

int 1

Changing value — 3

LHS

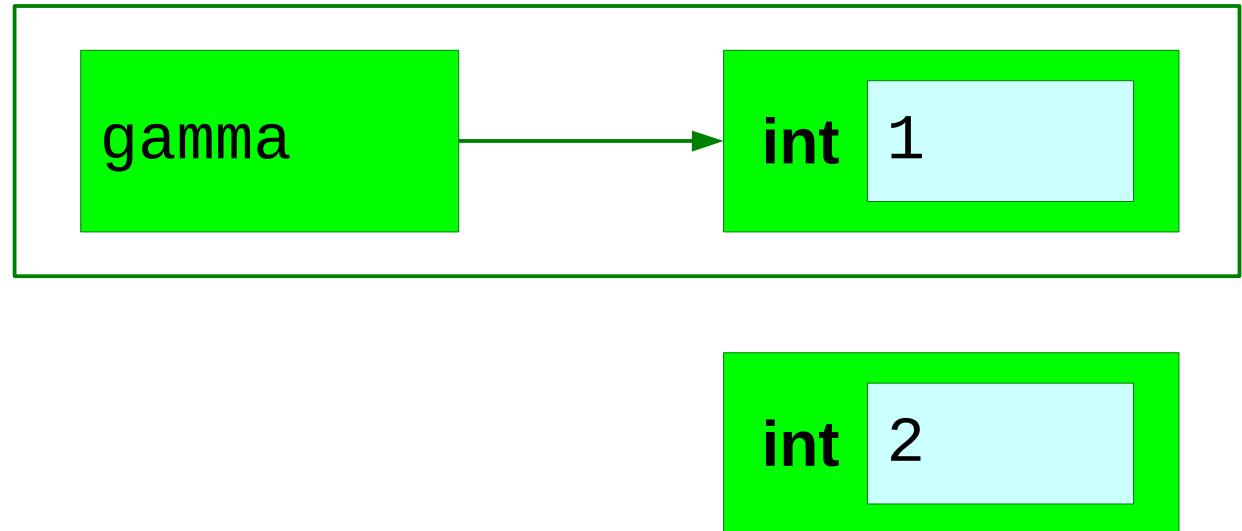
```
>>> gamma = 1
```



Changing value — 4

RHS

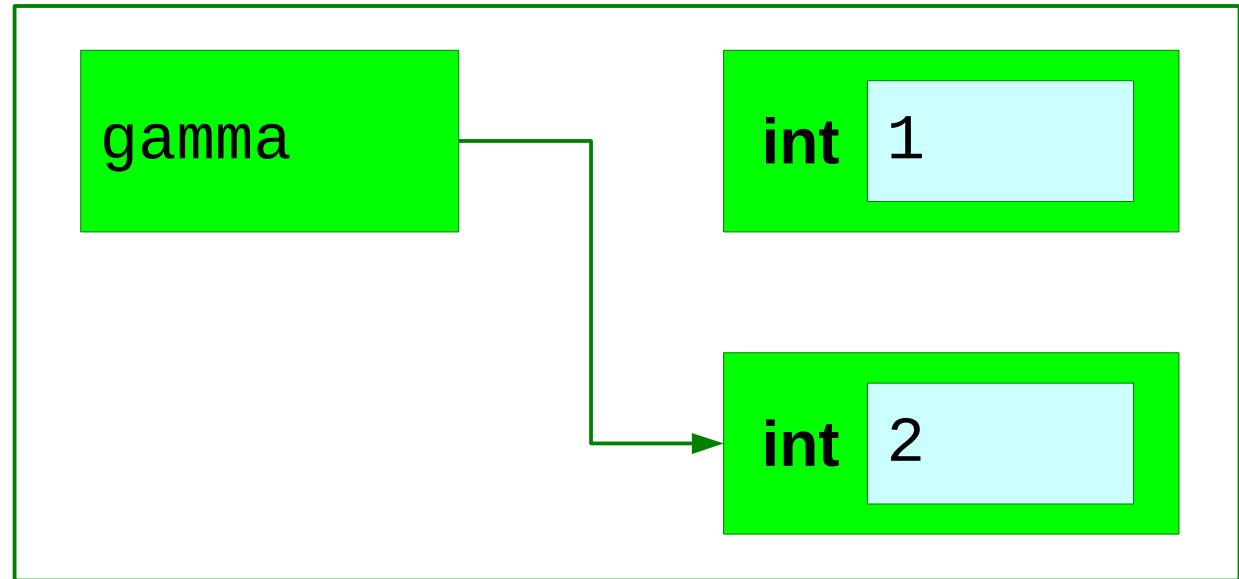
```
>>> gamma = 1  
>>> gamma = 2
```



Changing value — 5

LHS

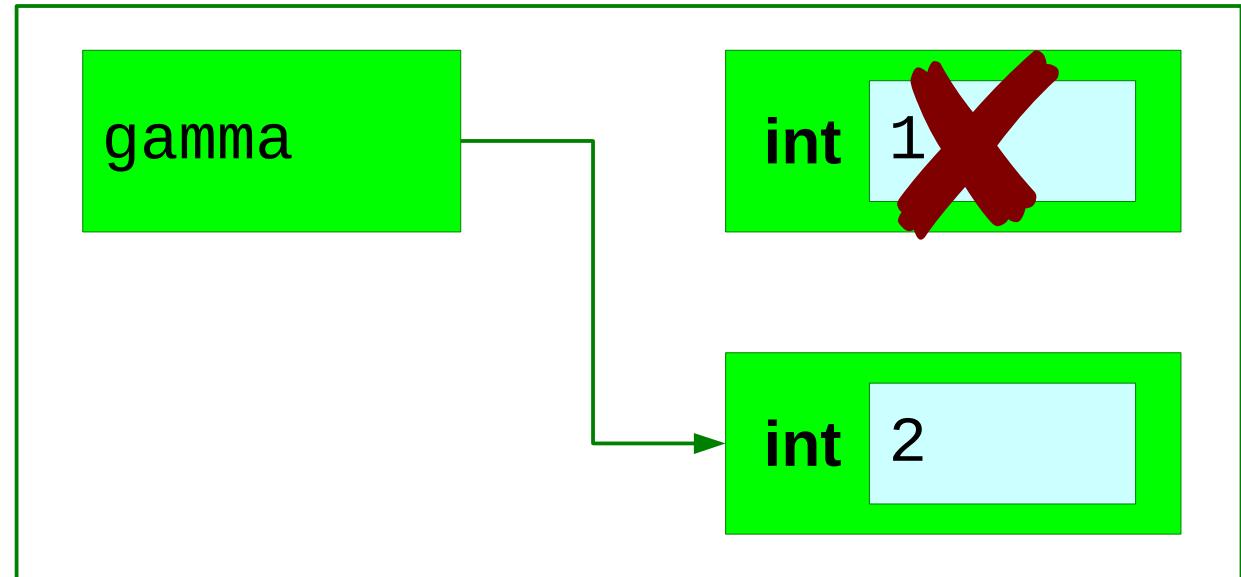
```
>>> gamma = 1  
>>> gamma = 2
```



Changing value — 6

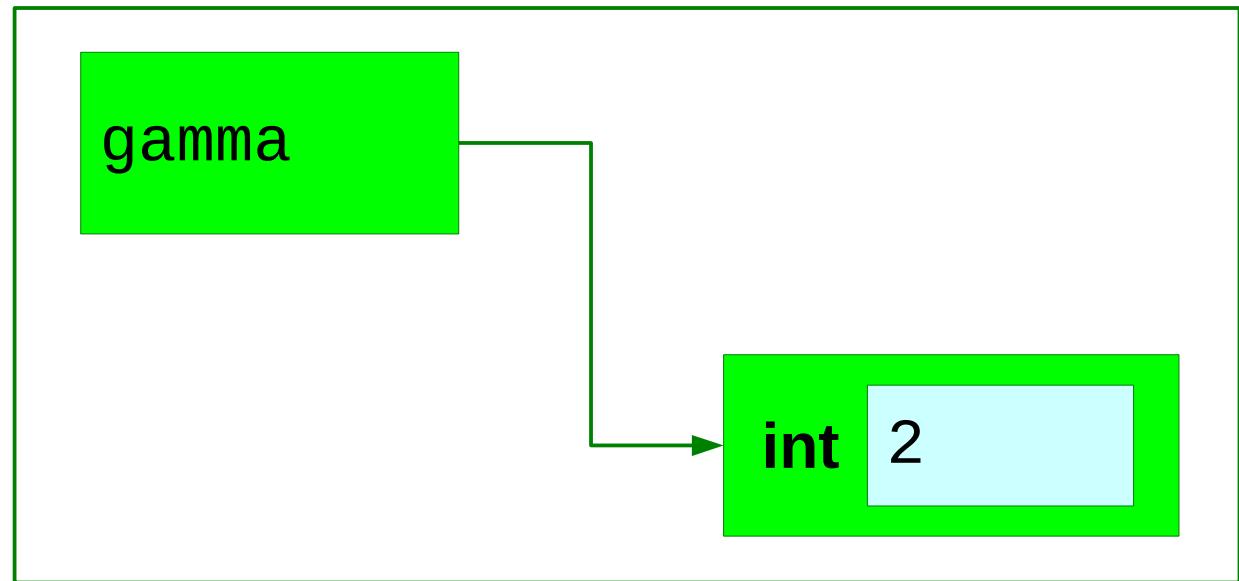
```
>>> gamma = 1  
>>> gamma = 2
```

garbage collection



Changing value — 7

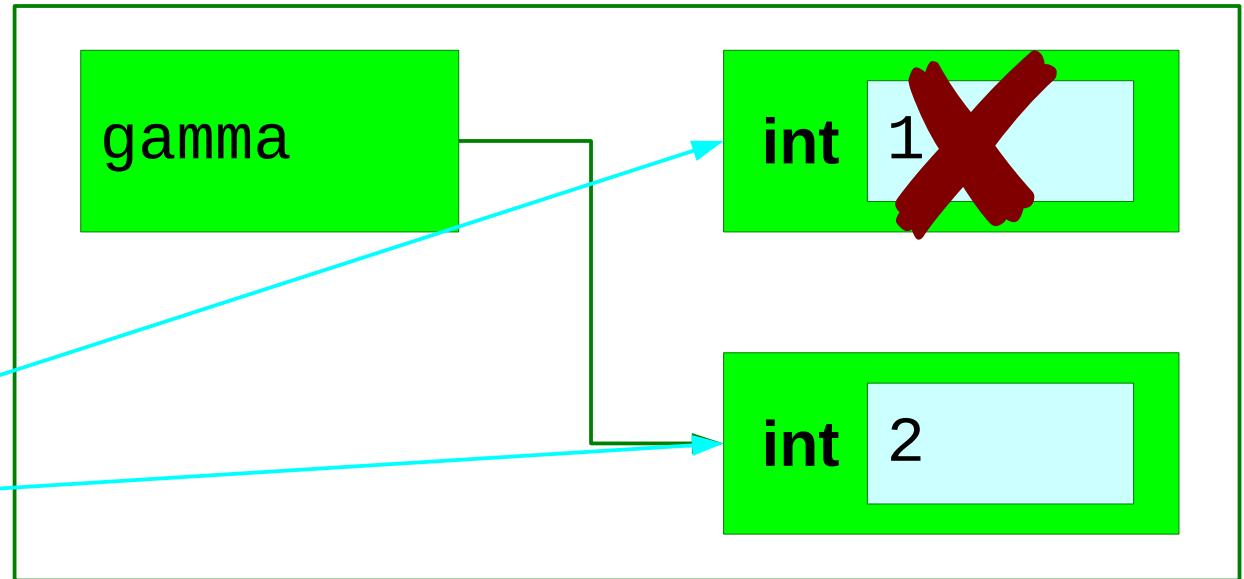
```
>>> gamma = 1  
>>> gamma = 2
```



Changing value — a subtlety

```
>>> gamma = 1  
>>> gamma = 2
```

Two separate integer objects.

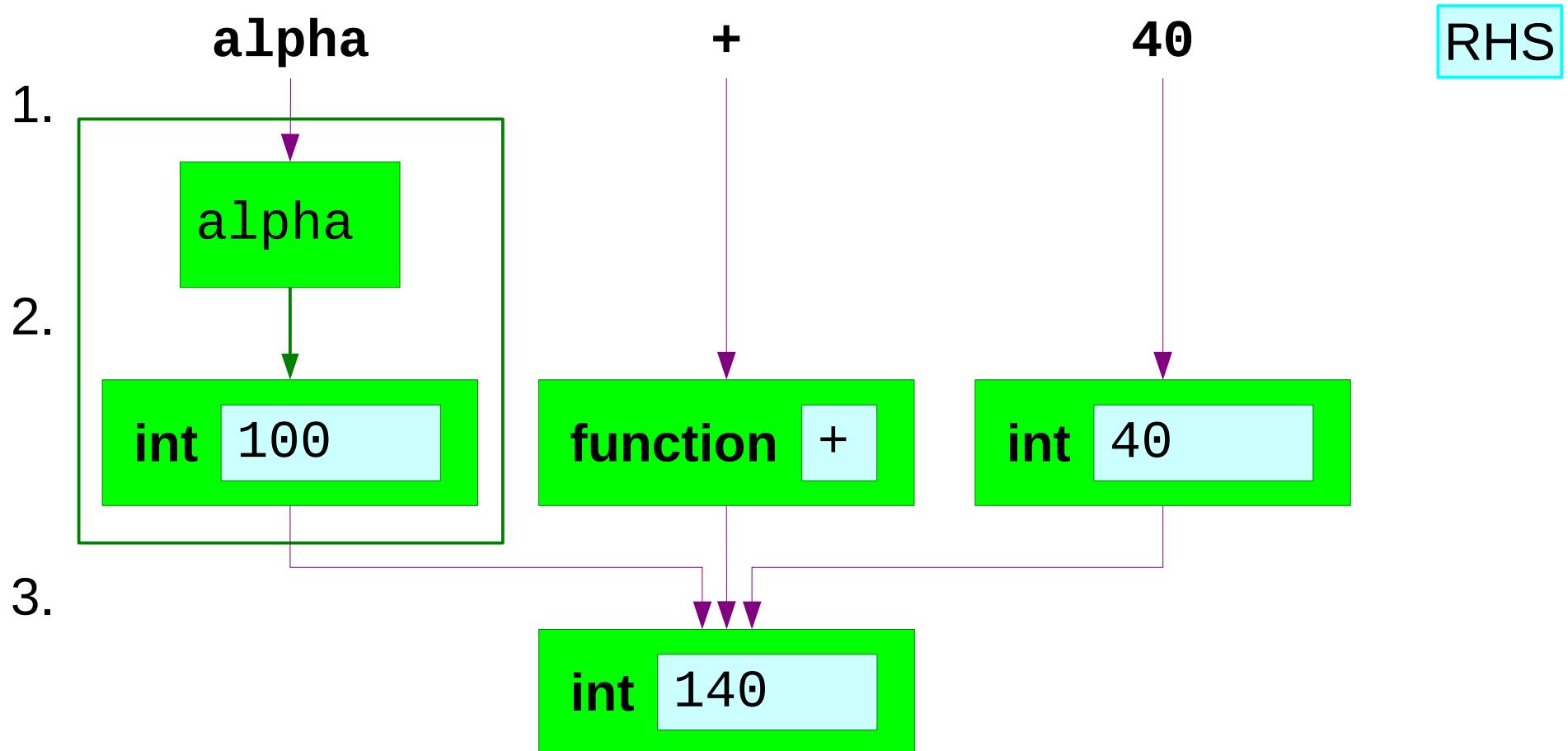


Python doesn't change an integer's value;
Python *replaces* the integer.

Python integers are “**immutable**”

Names on the RHS — 1

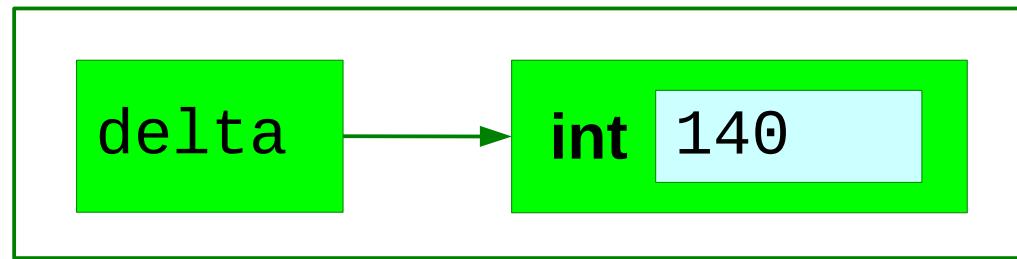
```
>>> delta = alpha + 40
```



Names on the RHS — 2

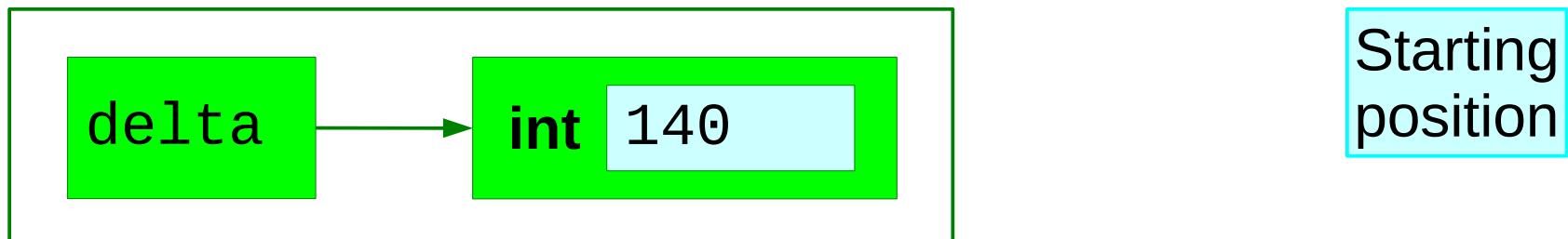
```
>>> delta = alpha + 40
```

4.



LHS

Same name on both sides — 0



```
>>> print(delta)
```

140

Same name on both sides — 1

```
>>> delta = delta + 10
```

RHS

1. **delta**

+

10

2.

delta

3.

int 140

function +

int 10

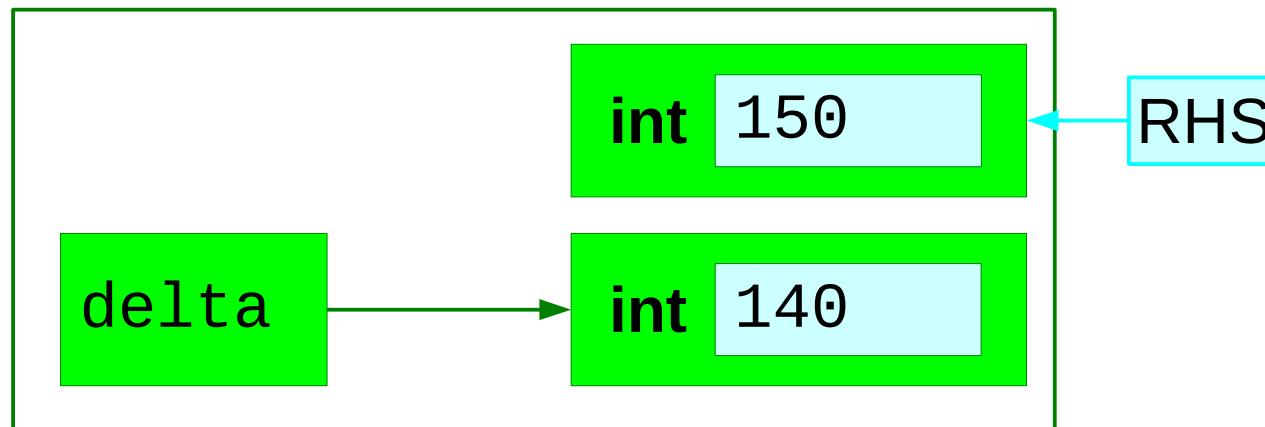
4.

int 150

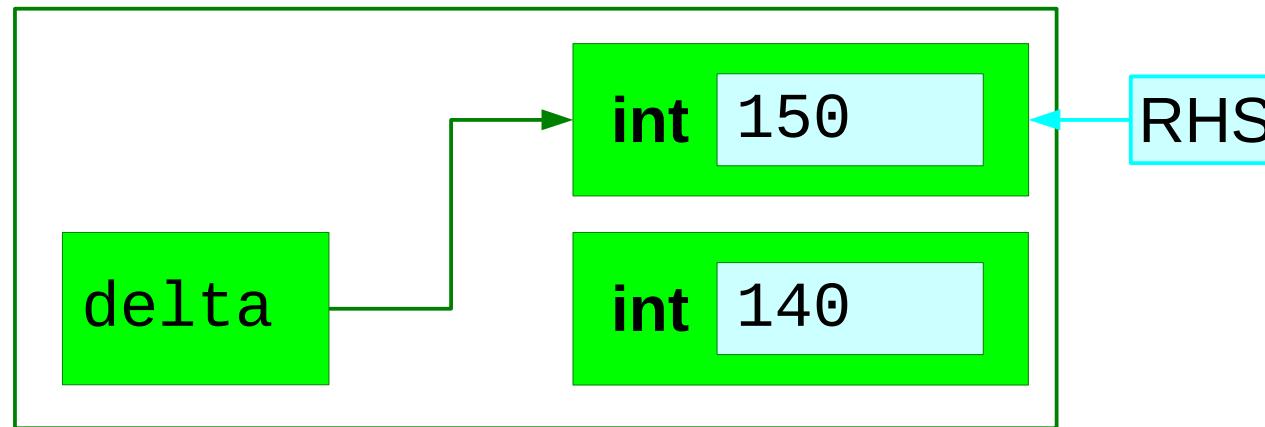
Same name on both sides — 2

```
>>> delta = delta + 10
```

5.



6.

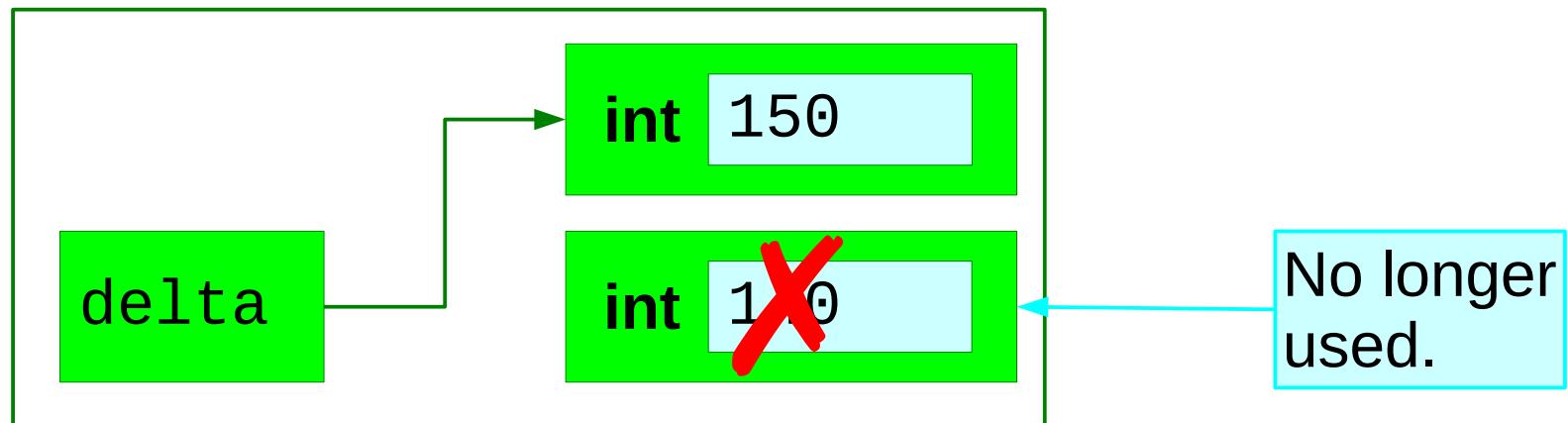


Same name on both sides — 3

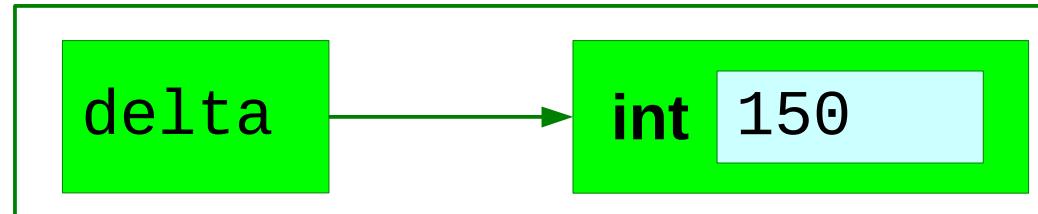
```
>>> delta = delta + 10
```

LHS

7.



8.



“Syntactic sugar”

thing `+=` 10

thing `-=` 10

thing `*=` 10

thing `/=` 10

thing `**=` 10

thing `%=` 10

is equivalent to

thing = thing `+` 10

thing = thing `-` 10

thing = thing `*` 10

thing = thing `/` 10

thing = thing `**` 10

thing = thing `%` 10

Deleting a name — 1

```
>>> print(thing)
```

```
Traceback (most recent call last):
```

```
  File "<stdin>", line 1, in <module>
```

```
NameError: name 'thing' is not defined
```

```
>>> thing = 1
```

```
>>> print(thing)
```

1

Unknown
variable

Deleting a name — 2

```
>>> print(thing)
```

```
1
```

Known
variable

```
>>> del thing
```

```
>>> print(thing)
```

Traceback (most recent call last):

 File "<stdin>", line 1, in <module>

NameError: name 'thing' is not defined

Unknown
variable

Progress

Assignment

`thing = thing + 10`

Deletion

`del thing`

Strictly right to left

`thing = thing + 10`

2nd

1st

`+ =` etc. “syntactic sugar”

`thing += 10`

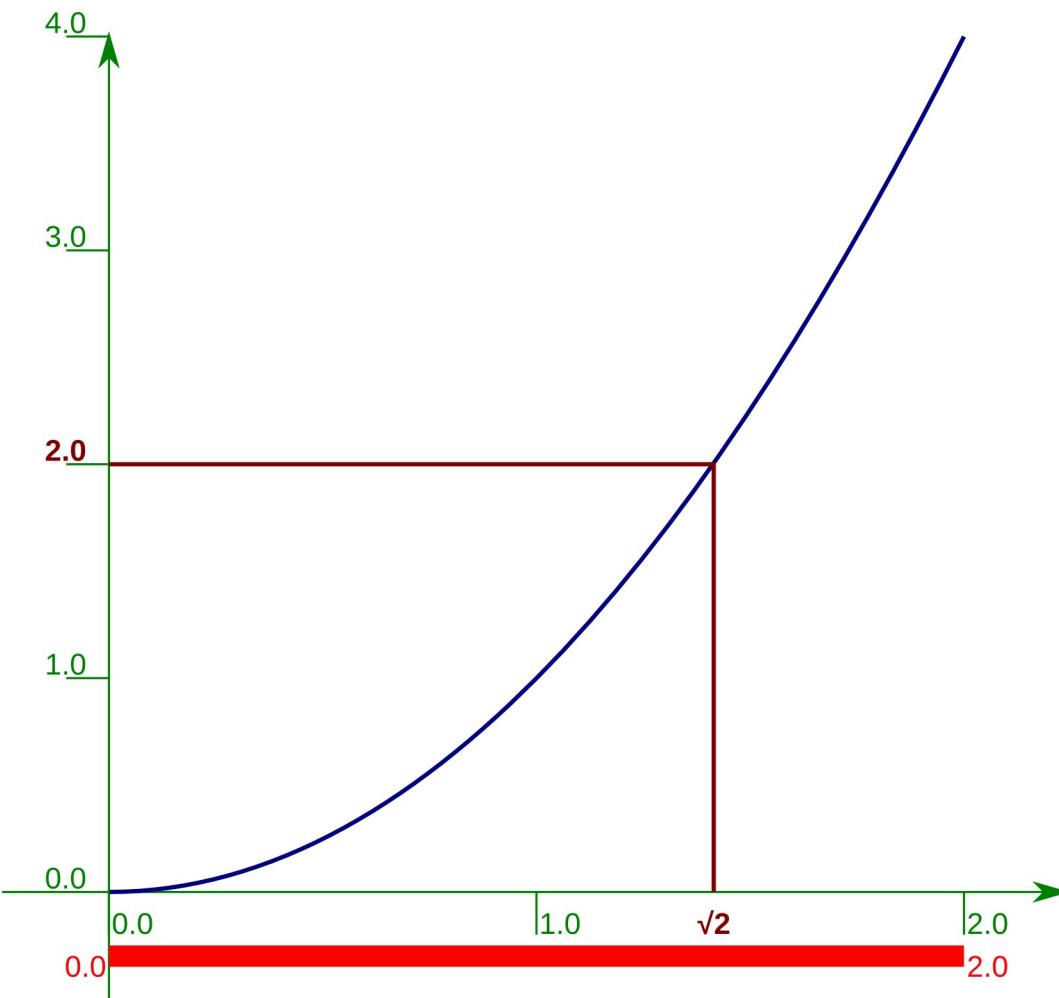
Our first “real” program

```
$ python3 sqrt.py  
Number? 2.0  
1.414213562373095
```

We have to write `sqrt.py`

First, the maths.
Then, the Python.

Square root of 2.0 by “bisection”



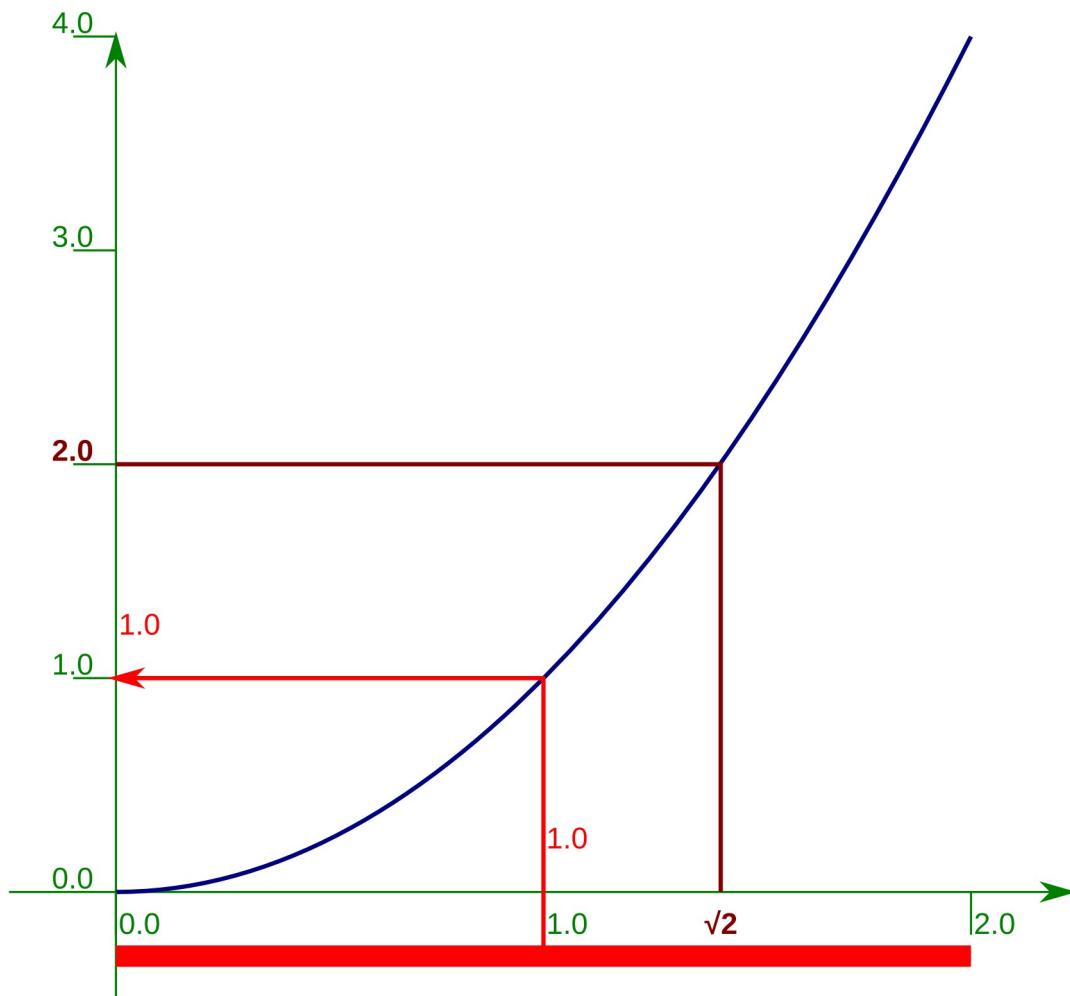
0.0 too small for $\sqrt{2}$

2.0 too large for $\sqrt{2}$

“Interval of uncertainty”

$0.0 < \sqrt{2} < 2.0$

Square root of 2.0 by bisection — 1



$$(0.0 + 2.0) / 2.0$$

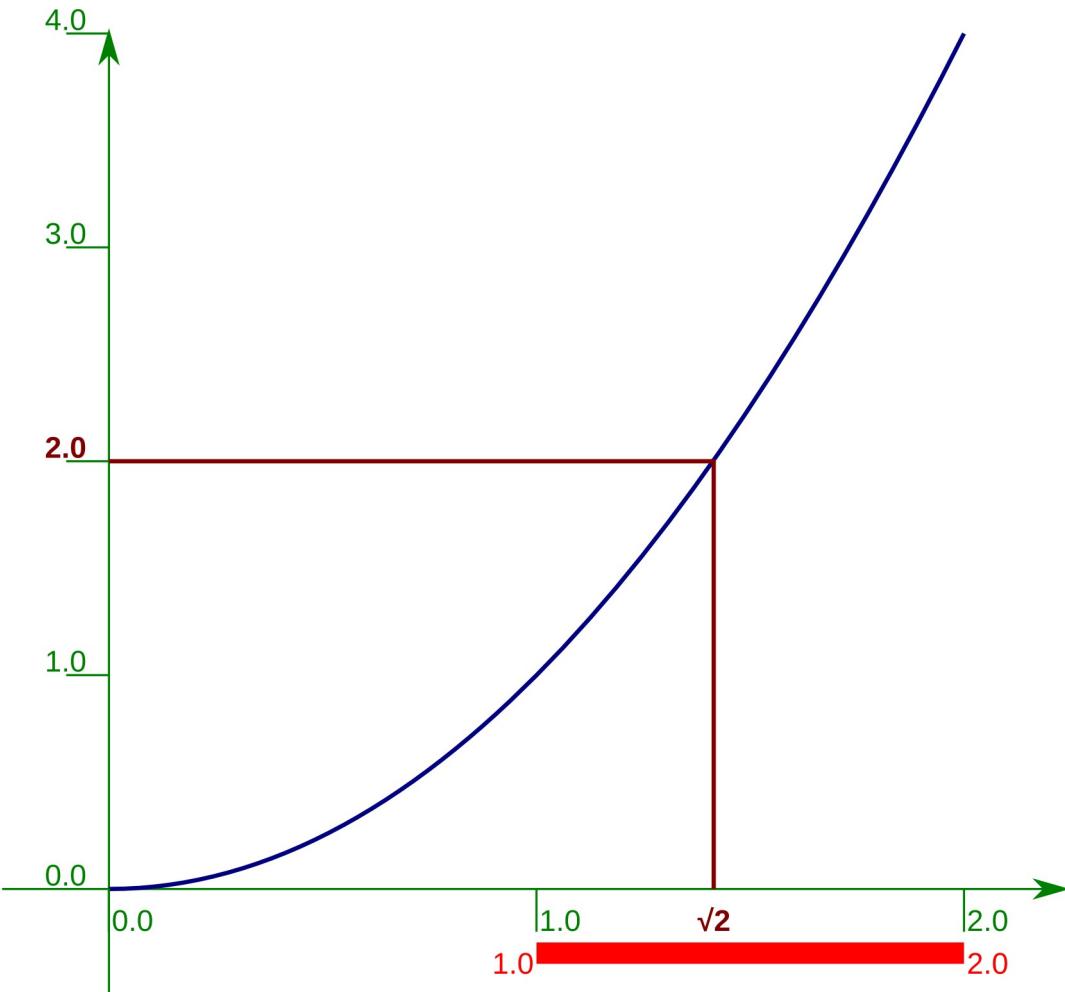
1.0

$$1.0^{**2}$$

1.0

Mid-point: 1.0

Square root of 2.0 by bisection — 2



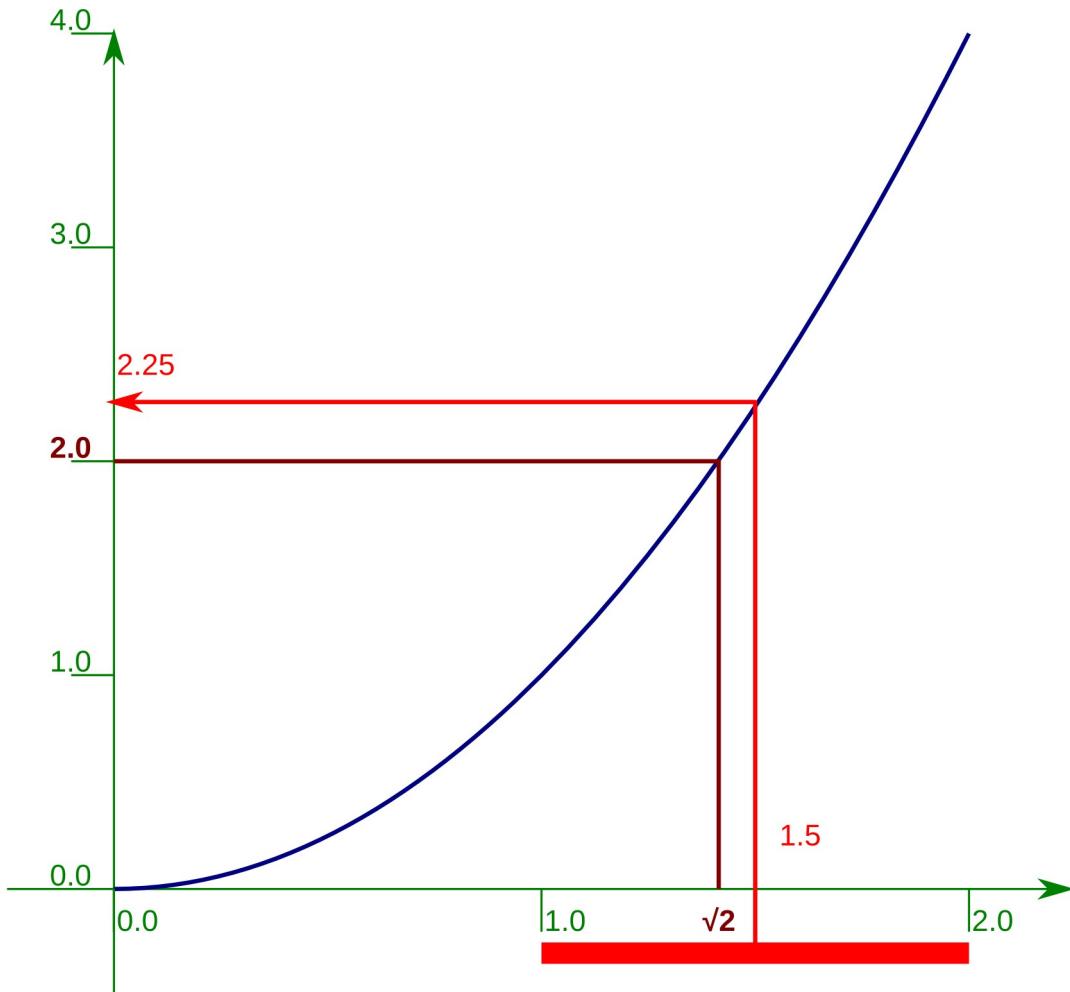
$$\text{midpoint}^2 < \sqrt{2}^2$$

$$1.0^{**2} < 2.0$$

so change **lower** bound

$$1.0 < \sqrt{2} < 2.0$$

Square root of 2.0 by bisection — 3



$$(1.0 + 2.0) / 2.0$$

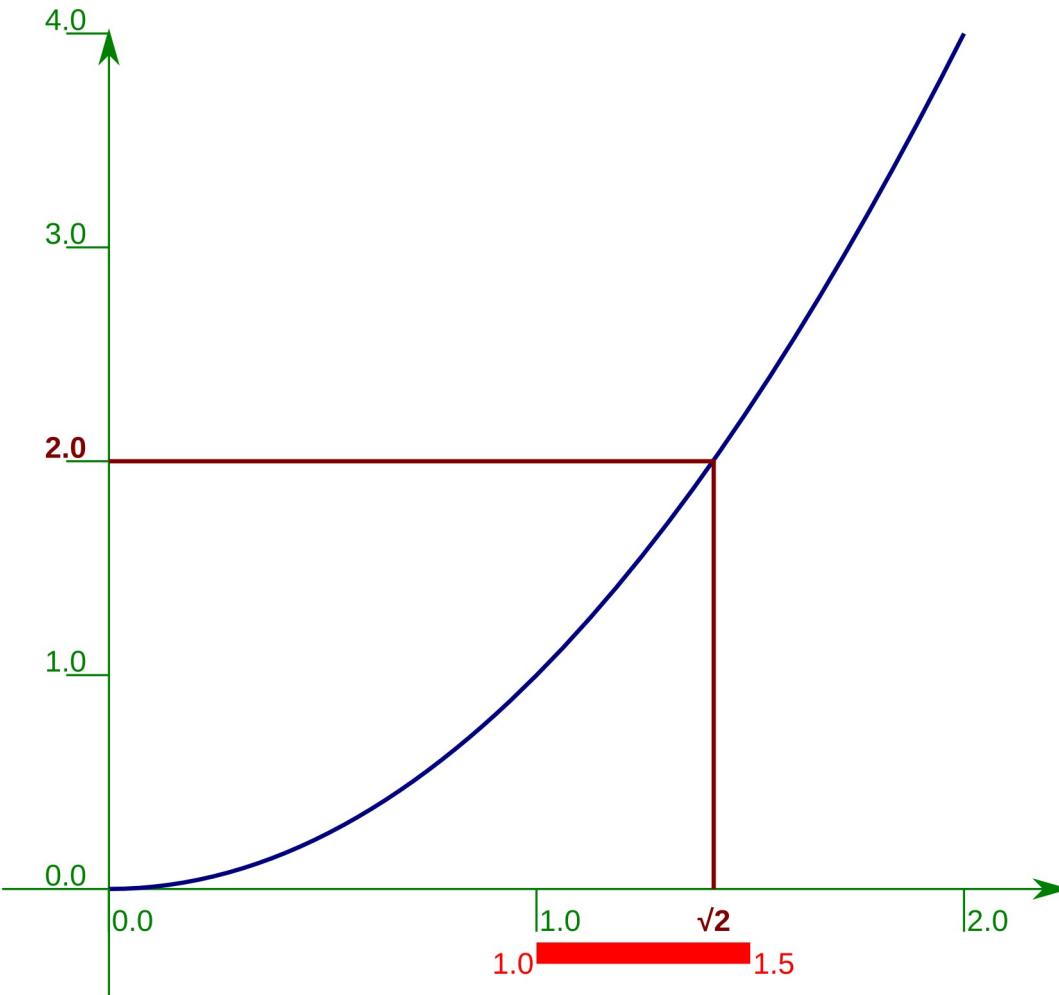
1.5

$$1.5^{**}2$$

2.25

Mid-point: 1.5

Square root of 2.0 by bisection — 4

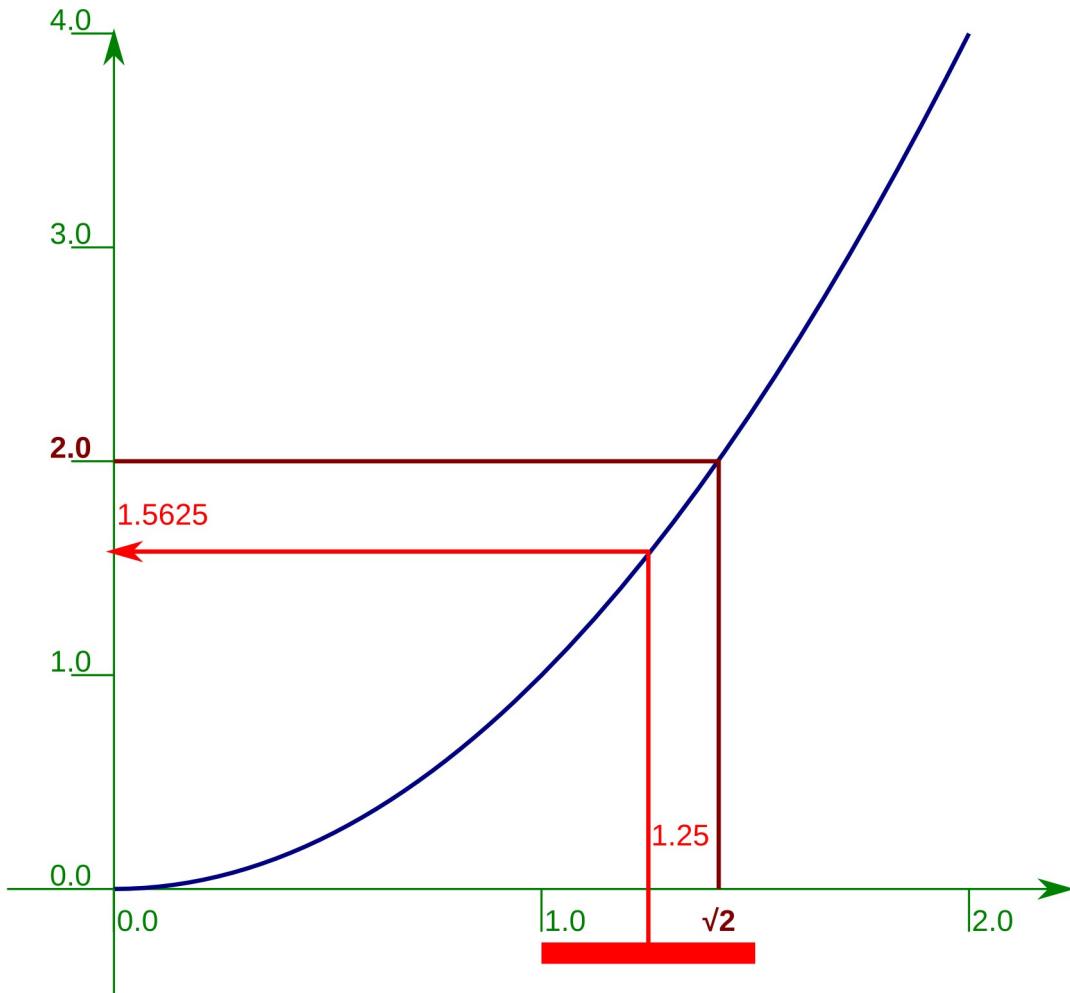


$$1.5^{**2} > 2.0$$

so change **upper** bound

$$1.0 < \sqrt{2} < 1.5$$

Square root of 2.0 by bisection — 5



$$(1.0 + 1.5) / 2.0$$

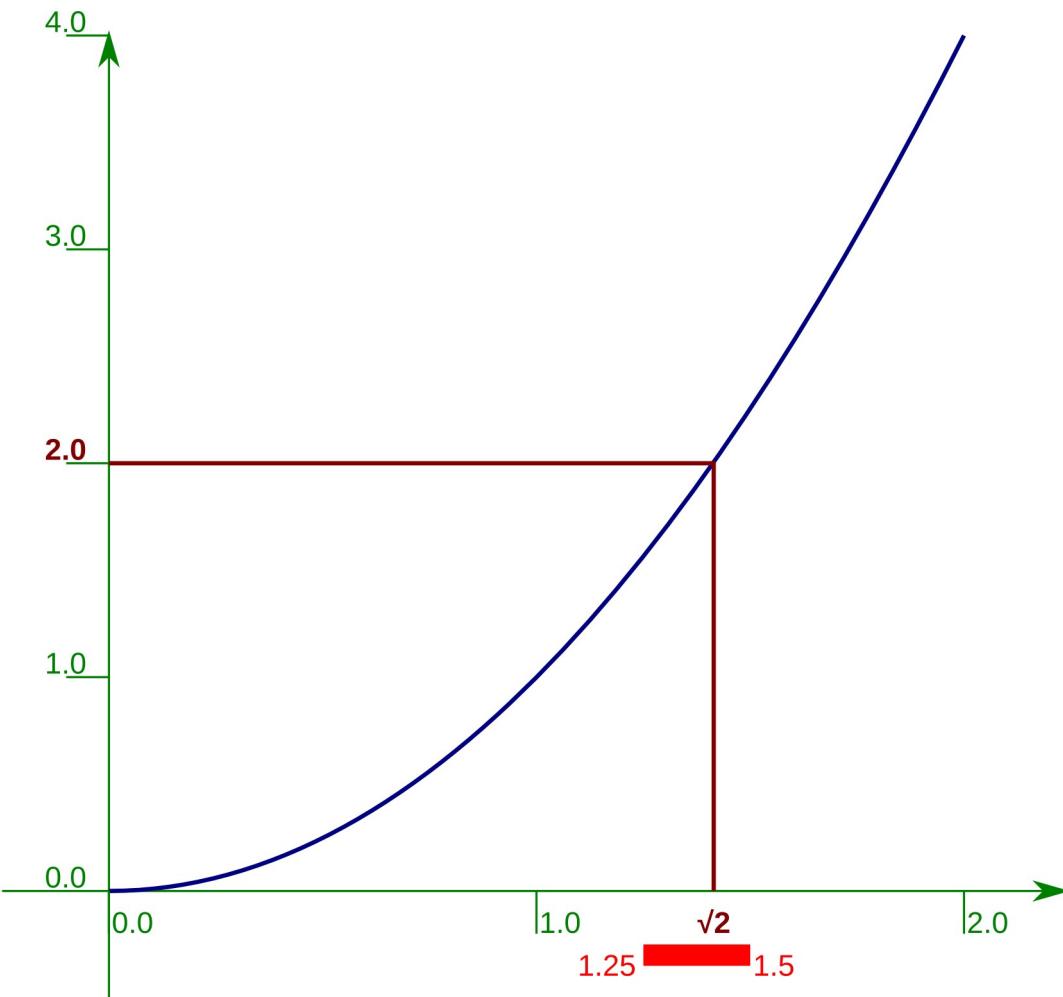
1.25

$$1.25^{**} 2$$

1.5625

Mid-point: 1.25

Square root of 2.0 by bisection — 6

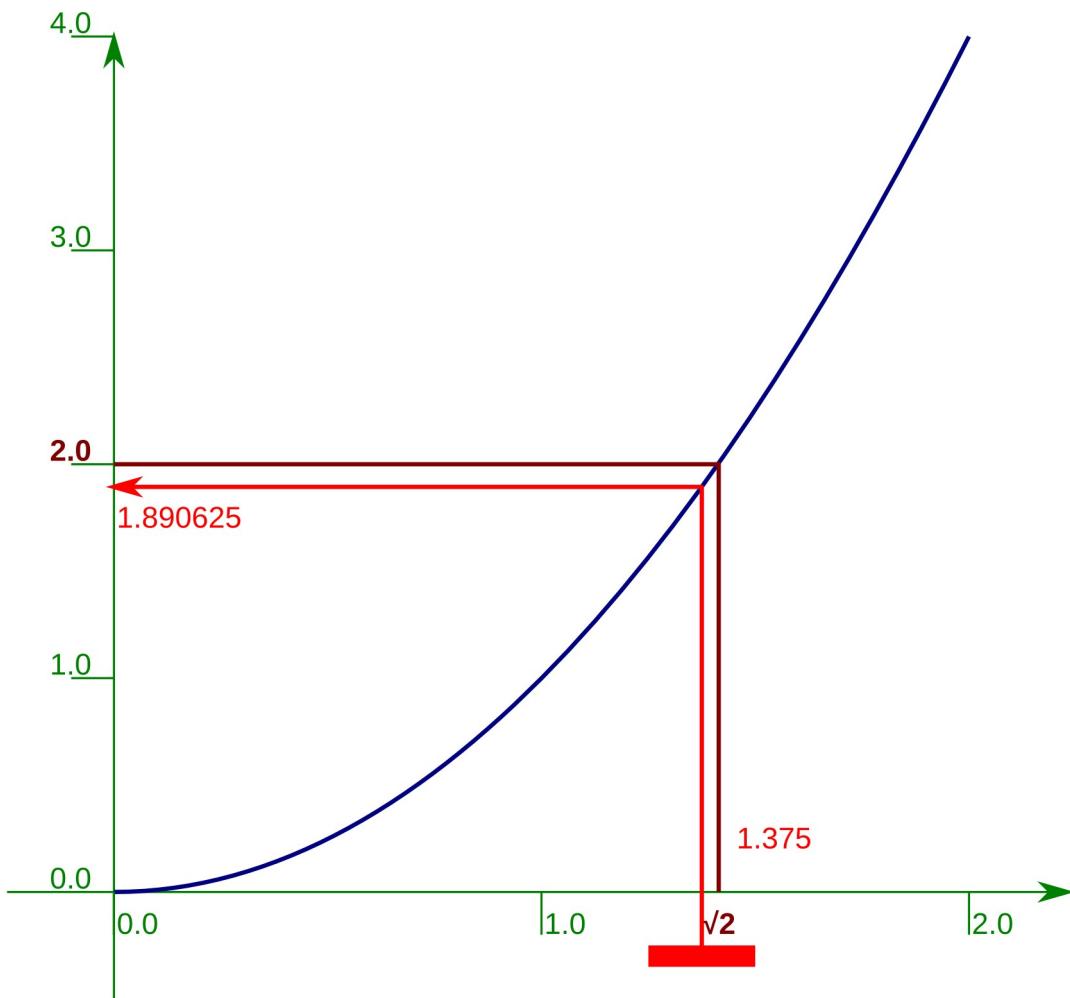


$$1.25^{**2} < 2.0$$

so change **lower** bound

$$1.25 < \sqrt{2} < 1.5$$

Square root of 2.0 by bisection — 7



$$(1.25 + 1.5) / 2.0$$

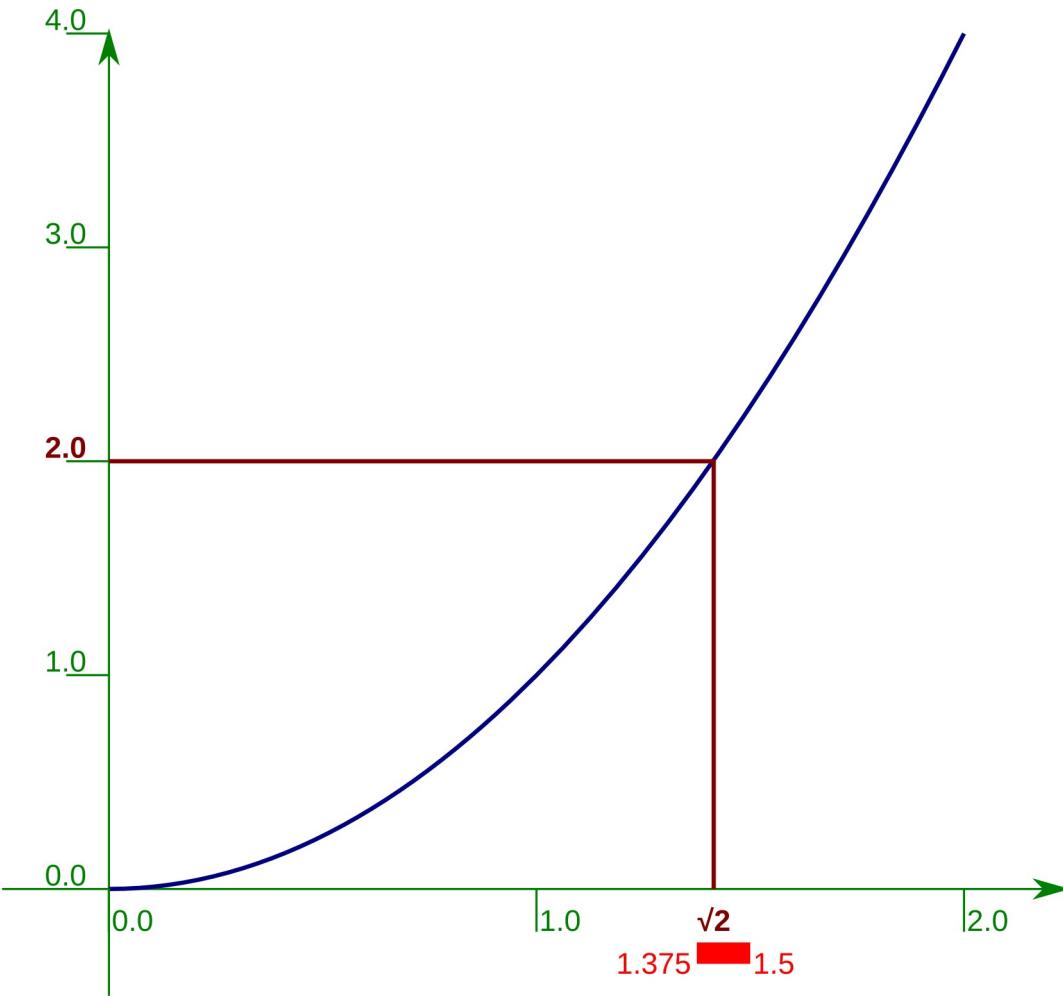
$$1.375$$

$$1.375^{**2}$$

$$1.890625$$

Mid-point: 1.375

Square root of 2.0 by bisection — 8

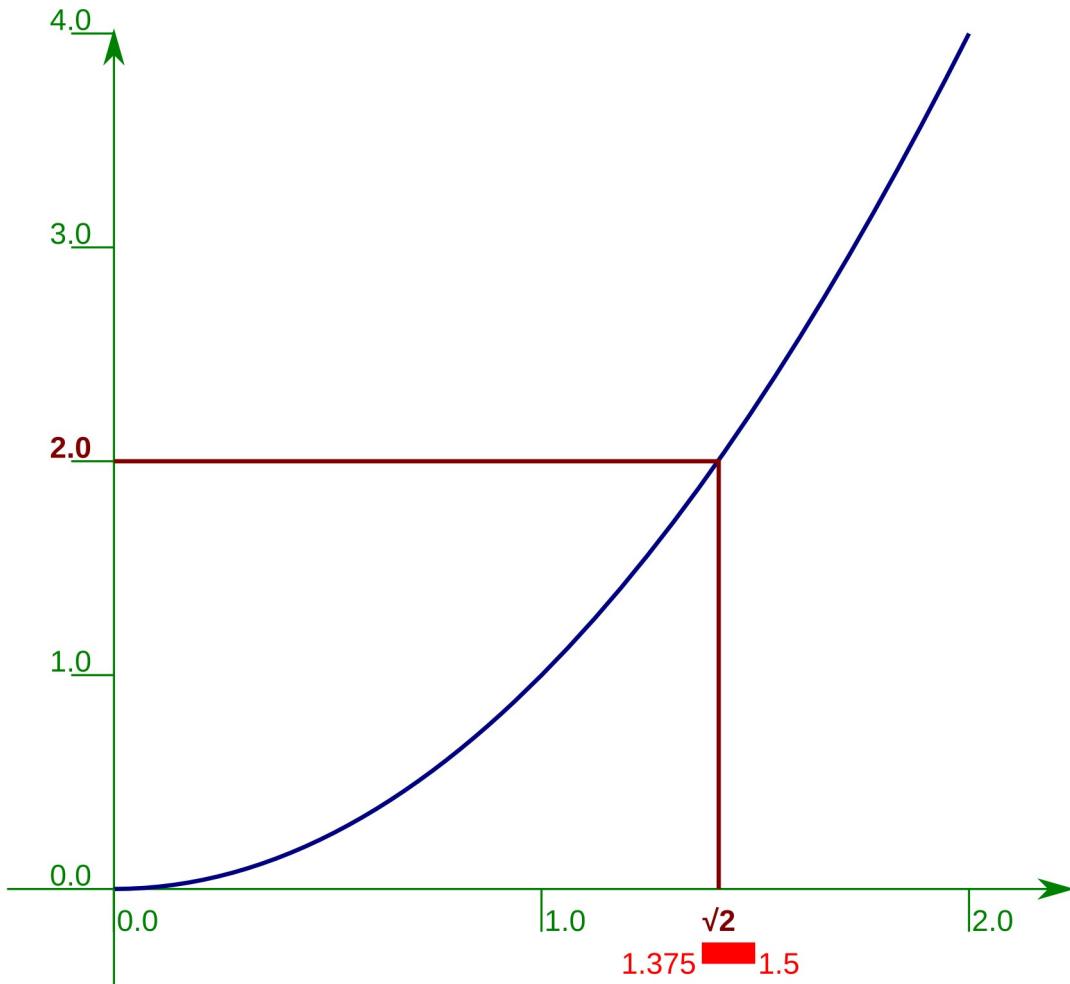


$$1.375^{**2} < 2.0$$

so change **lower** bound

$$1.375 < \sqrt{2} < 1.5$$

Exercise 6



$$1.375 < \sqrt{2} < 1.5$$

One more iteration.

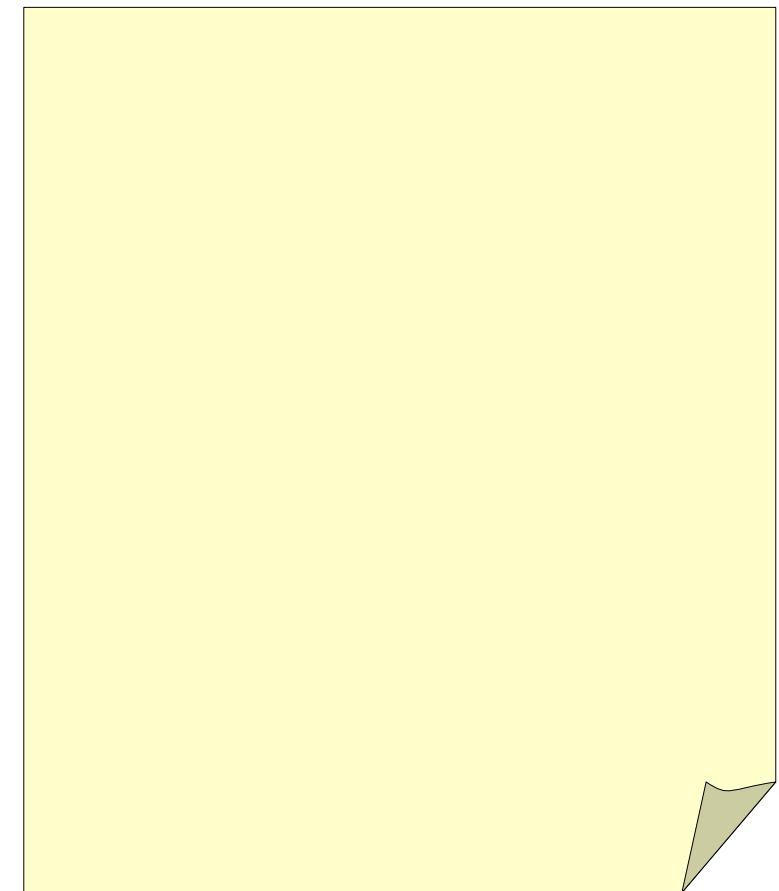
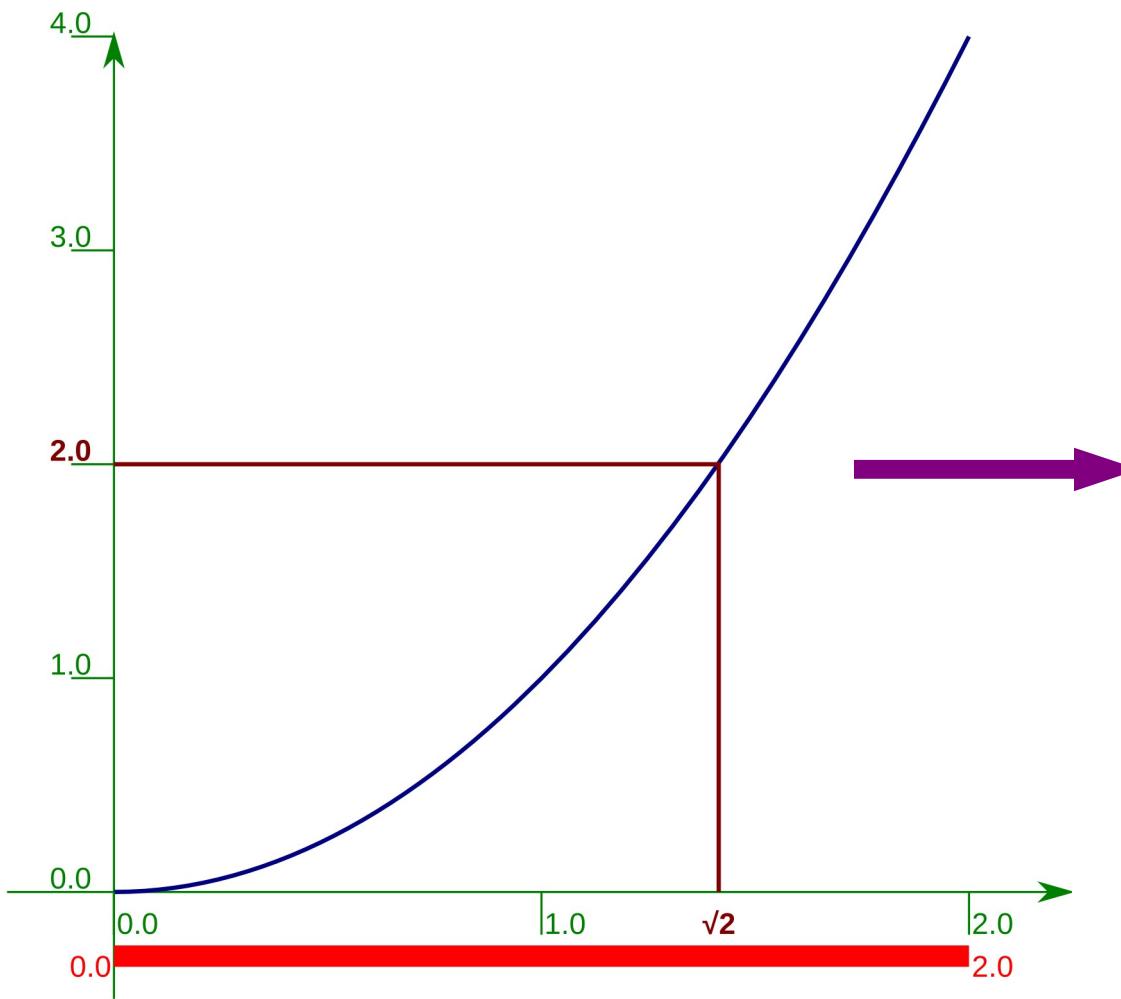
Find the mid-point.
Square it.
Compare the square to 2.0.

**Do you change the
lower or upper bound?**

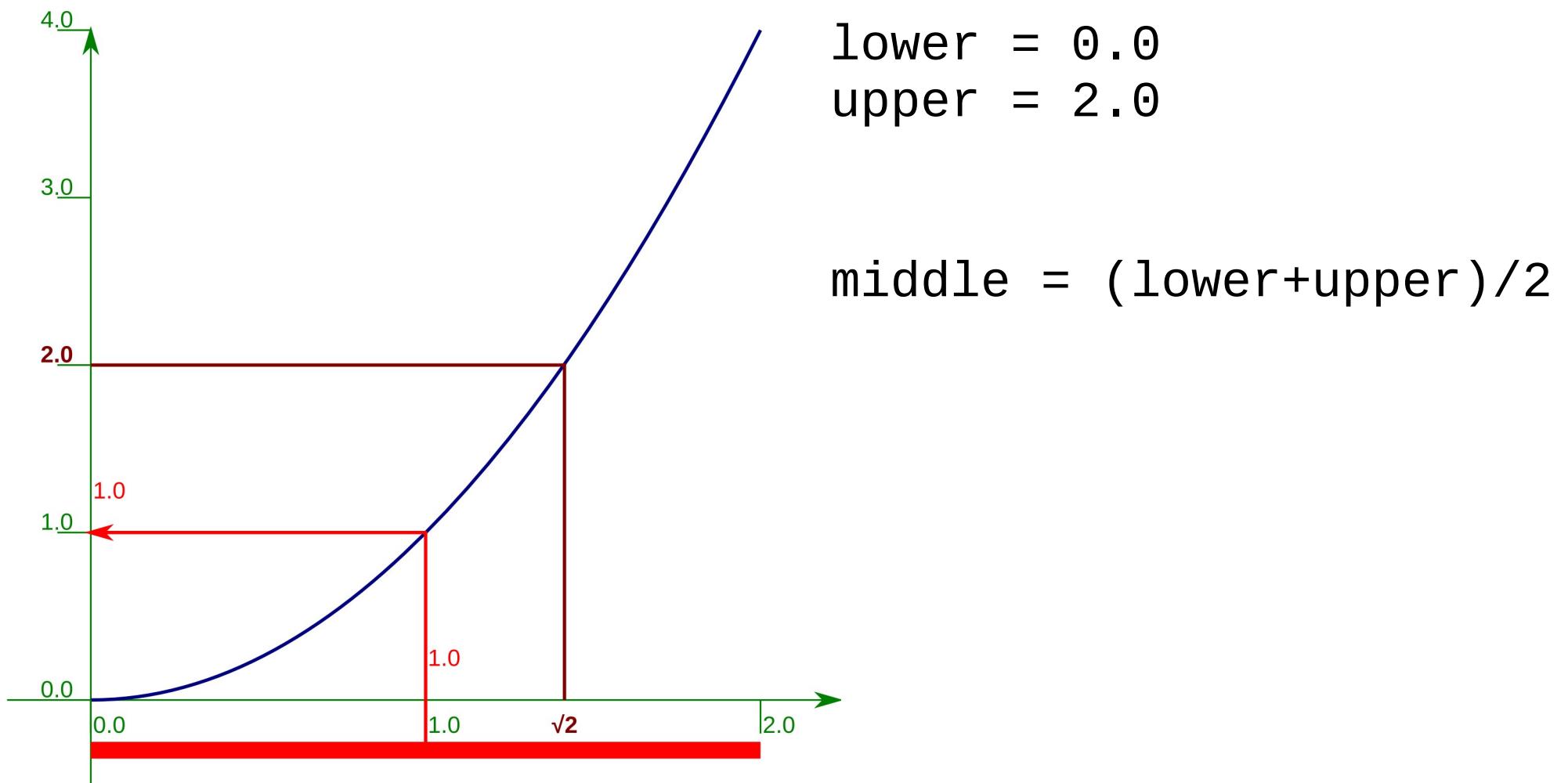


2 minutes

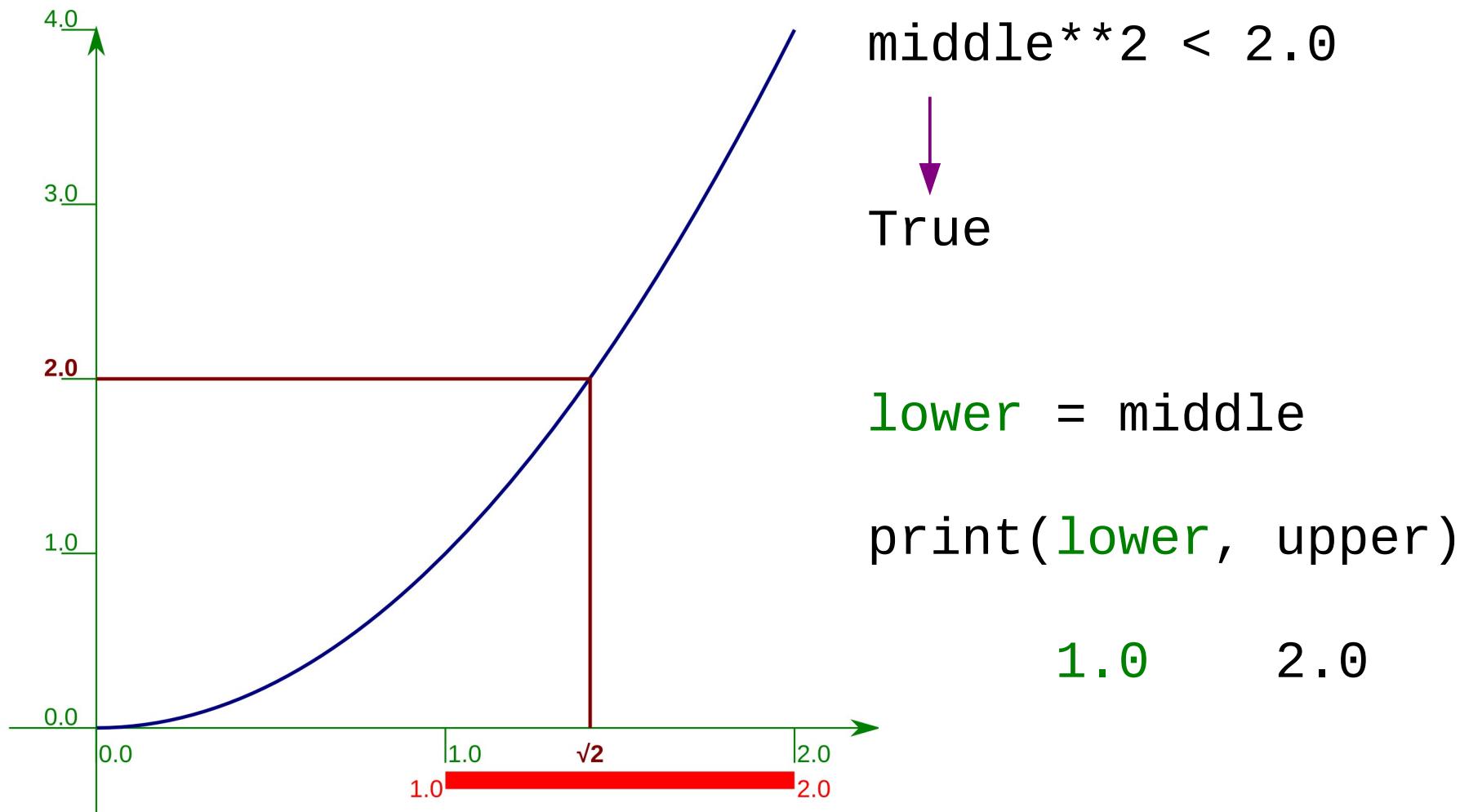
Understanding before Programming



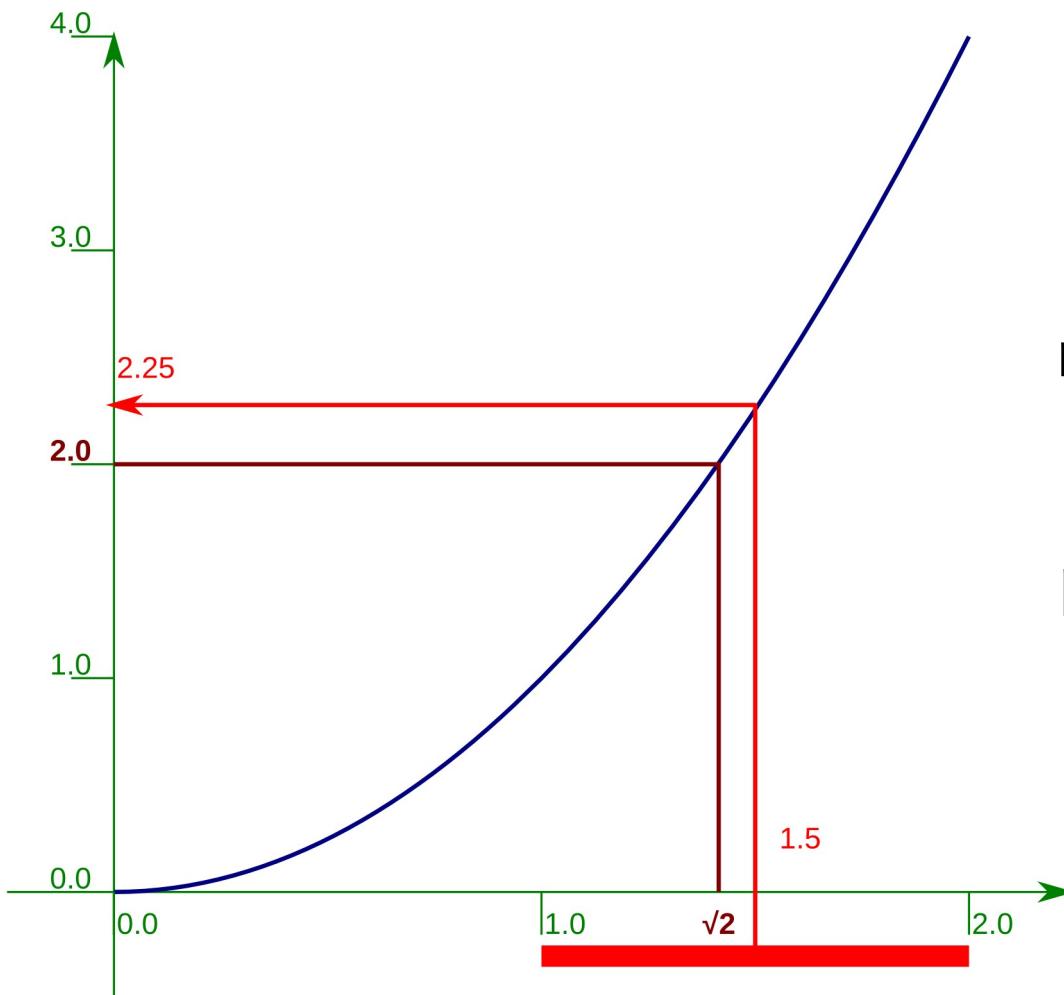
And now using Python!



And now using Python — 2



And now using Python — 3



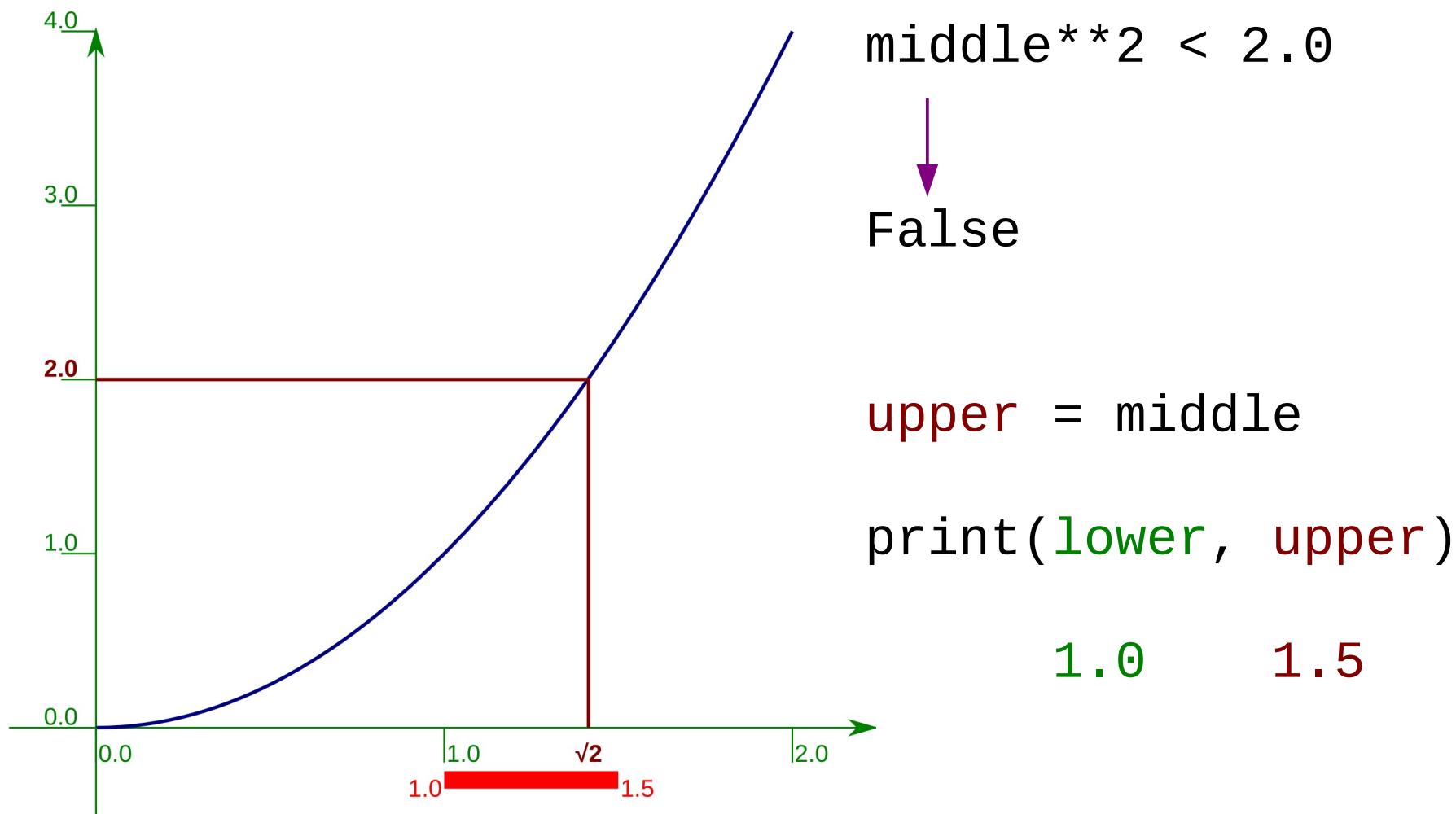
```
middle = (lower+upper)/2
```

```
print(middle, middle**2)
```

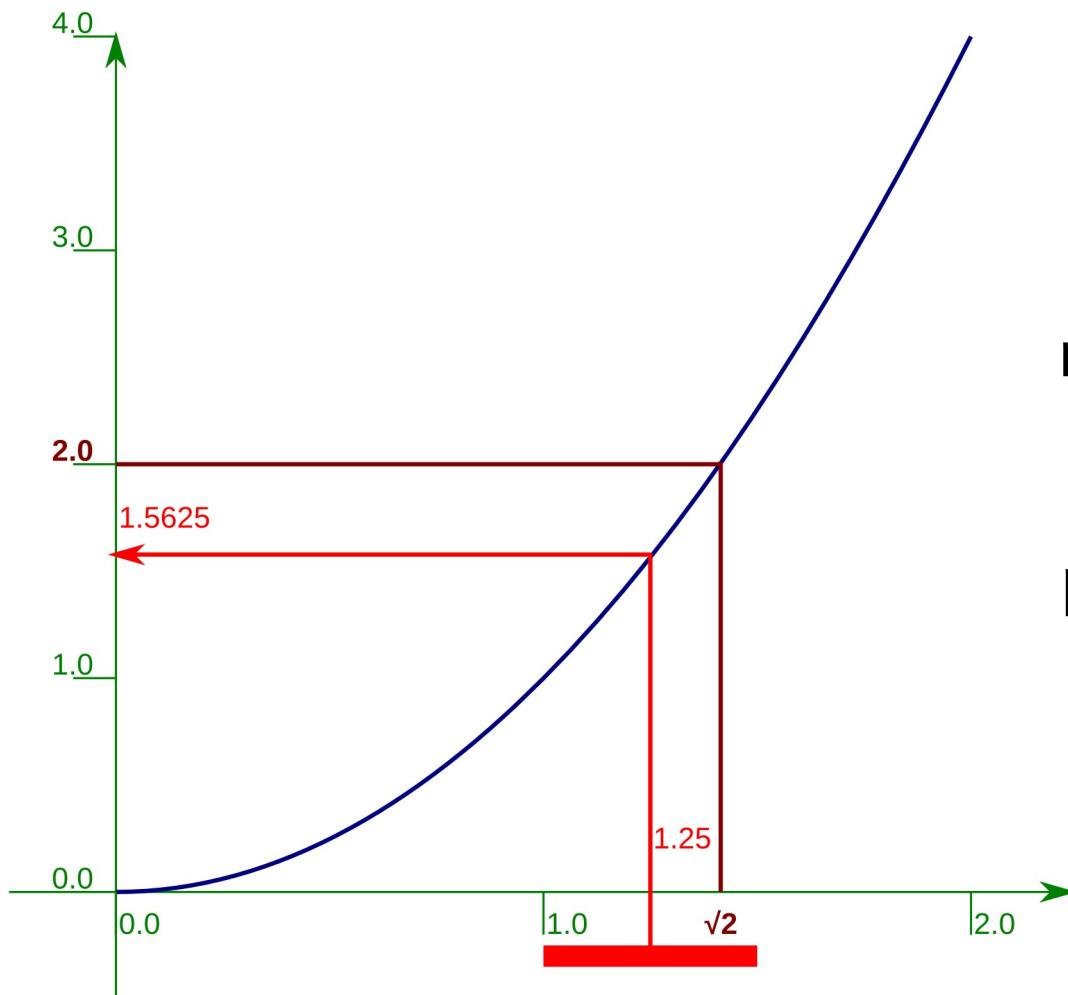
1.5

2.25

And now using Python — 4



And now using Python — 5



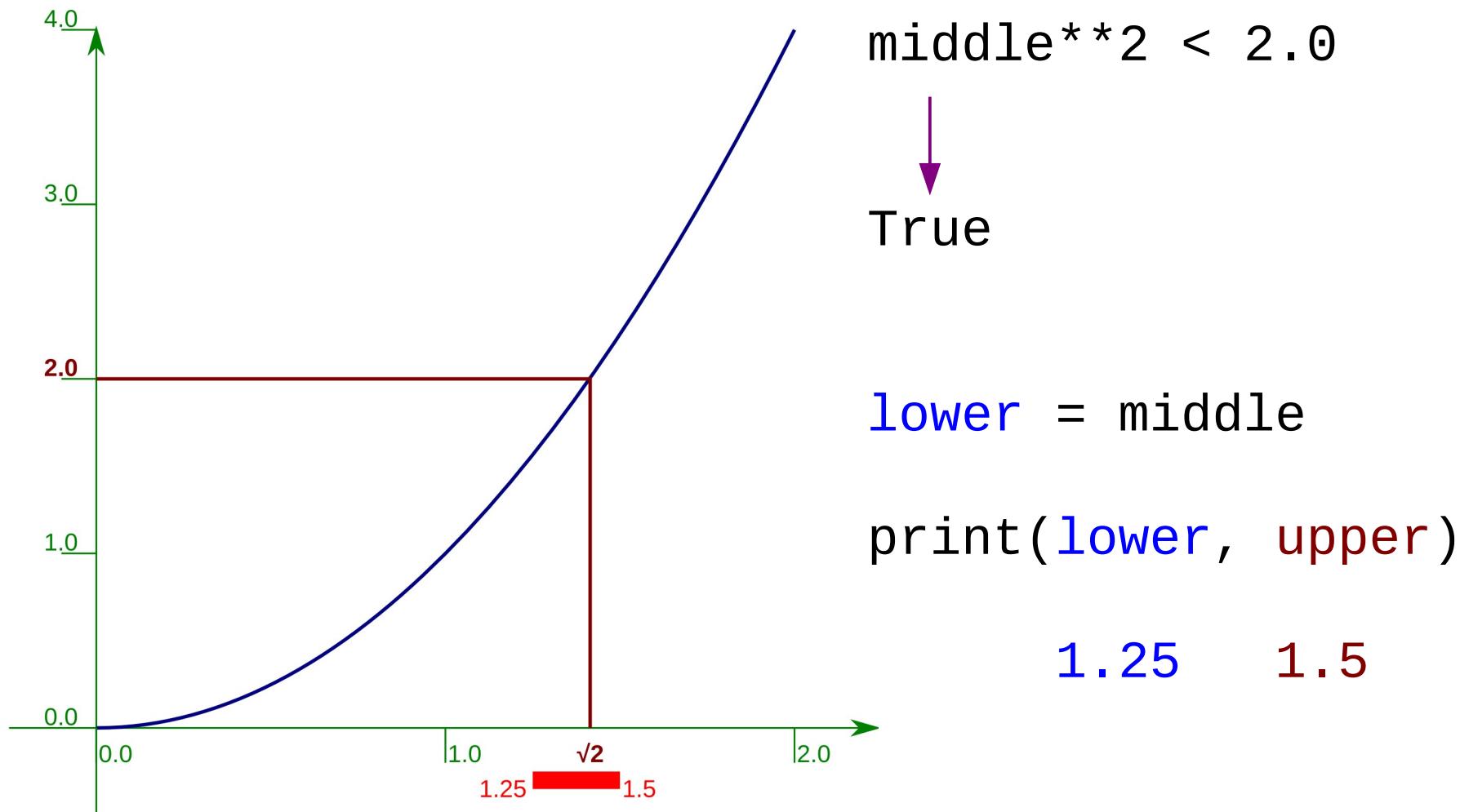
`middle = (lower+upper)/2`

`print(middle, middle**2)`

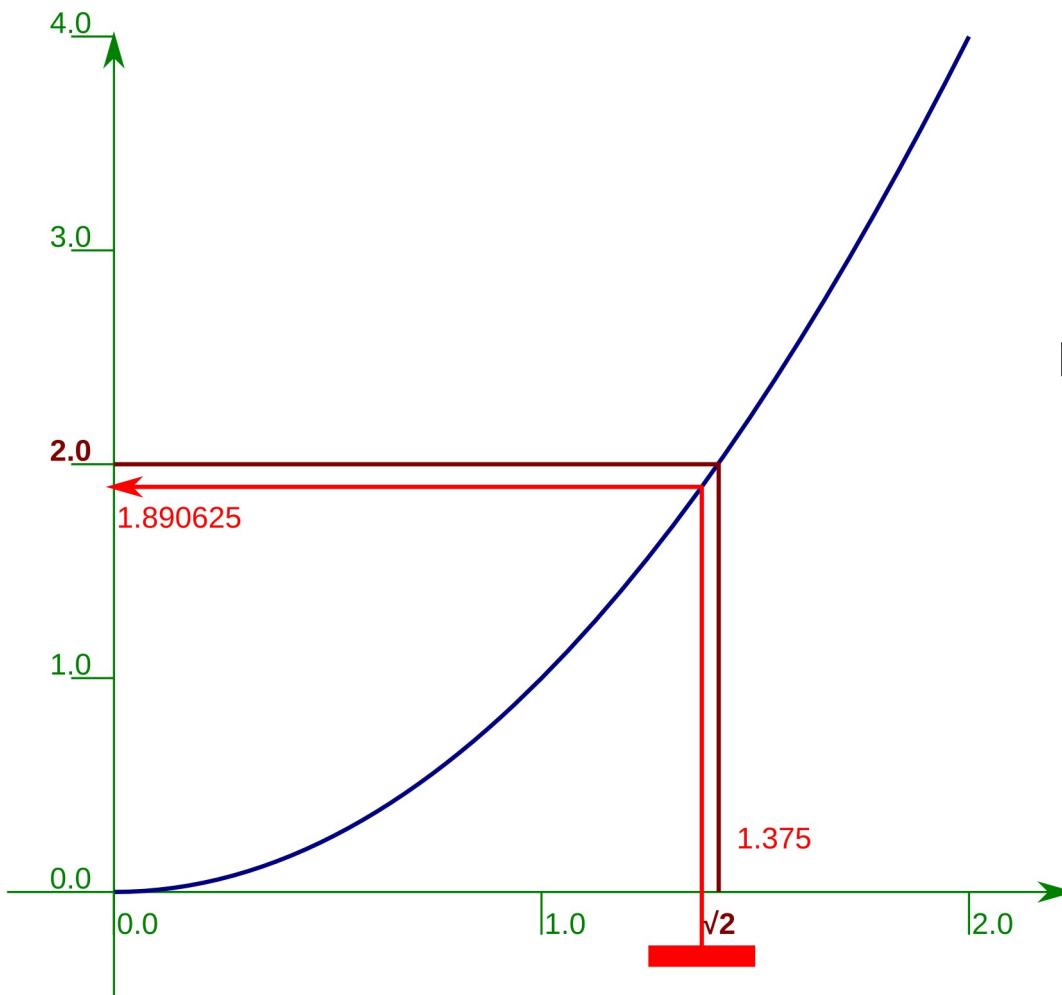
1.25

1.5625

And now using Python — 6



And now using Python — 7



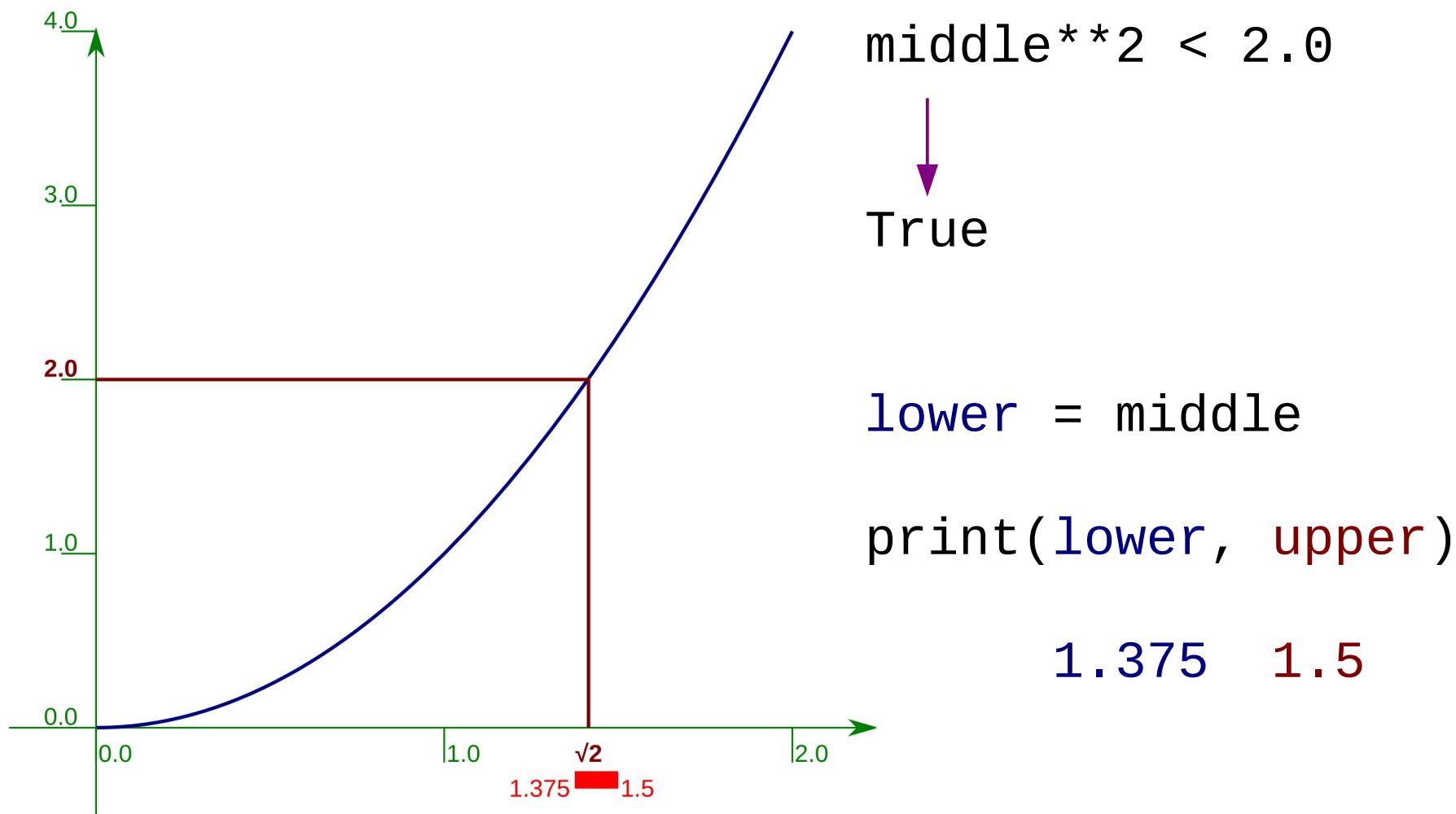
`middle = (lower+upper)/2`

`print(middle, middle**2)`

1.375

1.890625

And now using Python — 8



Looking at the Python code

```
lower = 0.0  
upper = 2.0
```

```
middle = (lower+upper)/2  
print(middle, middle**2)
```

```
middle**2 < 2.0
```

```
lower = middle ? upper = middle
```

```
print(lower, upper)
```

Looking at the Python structures

```
lower = 0.0  
upper = 2.0
```

Set up

```
middle = (lower+upper)/2  
print(middle, middle**2)
```

Loop

```
middle**2 < 2.0
```

Choice

```
lower = middle
```



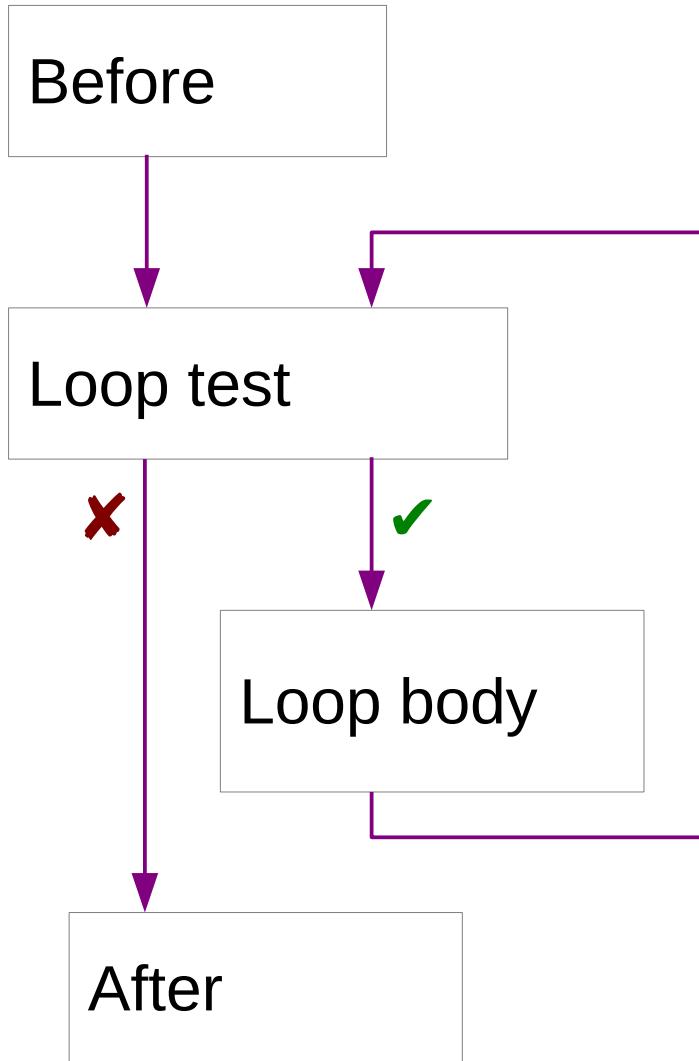
?

```
upper = middle
```

✗

```
print(lower, upper)
```

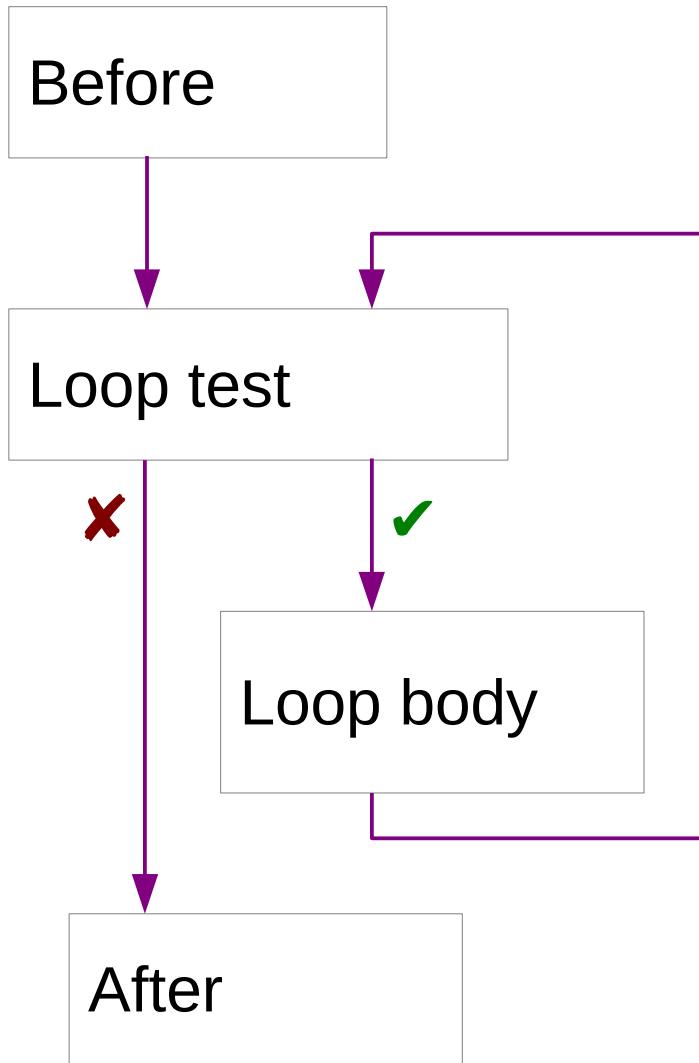
Looping



Should the
loop run
(again)?

What is
run each
loop?

Loop example: Count from 1 to 10



number = 1

number <= 10

X

print(number)
number += 1

print('Done!')

Loop example: Count from 1 to 10

```
number = 1
```

```
while number <= 10 :
```

```
    print(number)  
    number += 1
```

```
print('Done!')
```

```
number = 1
```

```
number <= 10
```

✗

```
    print(number)  
    number += 1
```

✓

```
print('Done!')
```

Loop test: Count from 1 to 10

```
number = 1
```

```
while number <= 10 :
```

The diagram illustrates the structure of a Python while loop. It starts with the assignment statement 'number = 1'. A blue bracket groups the entire line starting with 'while' up to the colon. Three cyan arrows point from this bracketed area to three cyan-bordered boxes: '“while” keyword', 'loop test', and 'colon'.

```
    print(number)  
    number += 1
```

```
print('Done!')
```

Loop body: Count from 1 to 10

```
number = 1
```

```
while number <= 10 :
```

The diagram illustrates the structure of a Python loop. A blue box labeled "loop body" contains the code "print(number)" and "number += 1". Two blue arrows point from the text "loop body" to these two lines. Below the box, a blue arrow points upwards to the opening brace of the loop, indicating the scope of the indentation. The code "print('Done!')" is shown below the loop body.

```
print(number)
number += 1
```

loop body

indentation

```
print('Done!')
```

Loop example: Count from 1 to 10

```
number = 1

while number <= 10 :

    print(number)
    number += 1

print('Done!')
```

while1.py

```
$ python3 while1.py
```

```
1
2
3
4
5
6
7
8
9
10
Done!
$
```

Python's use of indentation

```
number = 1  
  
while number <= 10 :  
    print(number)  
    number += 1  
  
print('Done!')
```

Four spaces' indentation indicate a “block” of code.

The block forms the repeated lines.

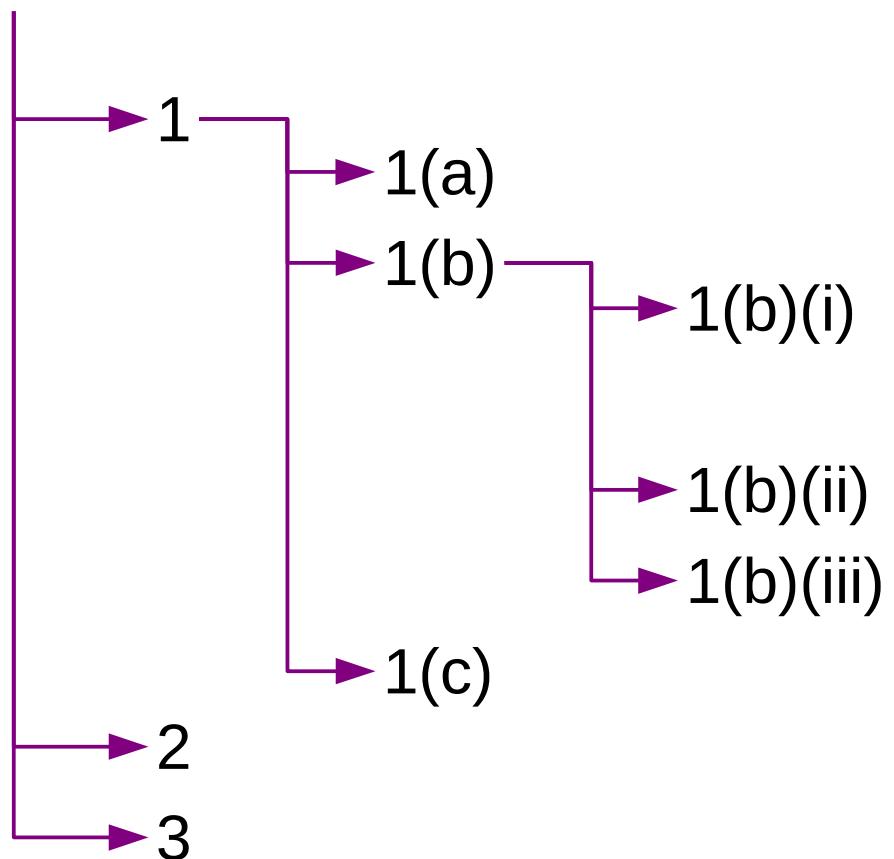
The first unindented line marks the end of the block.

c.f. “legalese”

CHAPTER

BOARDS AND S

1. There shall be in the University
 - (a) such Boards and Syndicates as may by ar maintained;
 - (b) the following Boards and Syndicates, the co
 - (i) the Board of Graduate Studies, which sh of students as Graduate Students and th respect of graduate study or contributio assigned to it by Ordinance;
 - (ii) the Board of Examinations, which shall e of University examinations and other co
 - (iii) the Local Examinations Syndicate, w examinations in schools and other instit
 - (c) any other Boards or Syndicates the compo the University.
2. Any Board or Syndicate constituted by Statute shall have the right of reporting to the University.
3. No person shall be appointed or reappointed a or Managers even though it be not expressly calle occasional Syndicate, who at the commencement of service, as the case may be, would have attained the



Other languages

Shell

```
while ...
do
  ...
done
```

```
do ... done
```

Syntax

```
  ...
```

Clarity

C

```
while ...
{
  ...
}
```

```
{ ... }
```

Syntax

```
  ...
```

Clarity

Progress

`while . . . :`

test to keep looping

code blocks

 indentation

before

`while test :`
 `action1`
 `action2`
 `action3`

afterwards

Exercise 7

For each script:

□□□□ Predict what it will do.

`while2.py`

□□□□ Run the script.

`while3.py`

□□□□ Were you right?

`while4.py`

To kill a running script:

`Ctrl` + `C`

`while5.py`

`while6.py`



5 minutes

Back to our square root example

uncertainty

0.0 ■ 2.0



1.0 ■ 2.0



1.0 ■ 1.5



1.25 ■ 1.5



1.375 ■ 1.5

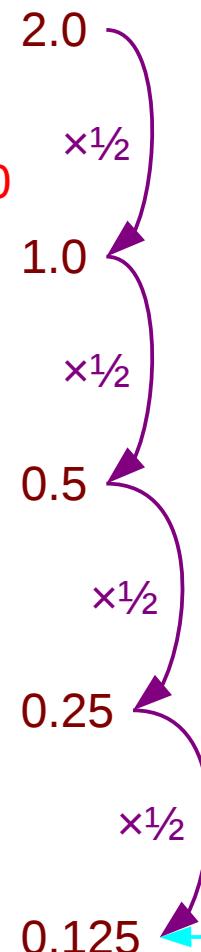


tolerance

■ ← →

1.0×10^{-15}

What we want



What we get

Keep looping while ... ?

uncertainty > tolerance

```
while uncertainty > tolerance :
```

```
    Do stuff.
```

```
    
```

```
    
```

```
    
```

Square root: the loop

```
tolerance = 1.0e-15
```

Set up

```
lower = 0.0
```

Loop

```
upper = 2.0
```

```
uncertainty = upper - lower
```

```
while uncertainty > tolerance :
```

```
    middle = (lower + upper)/2
```

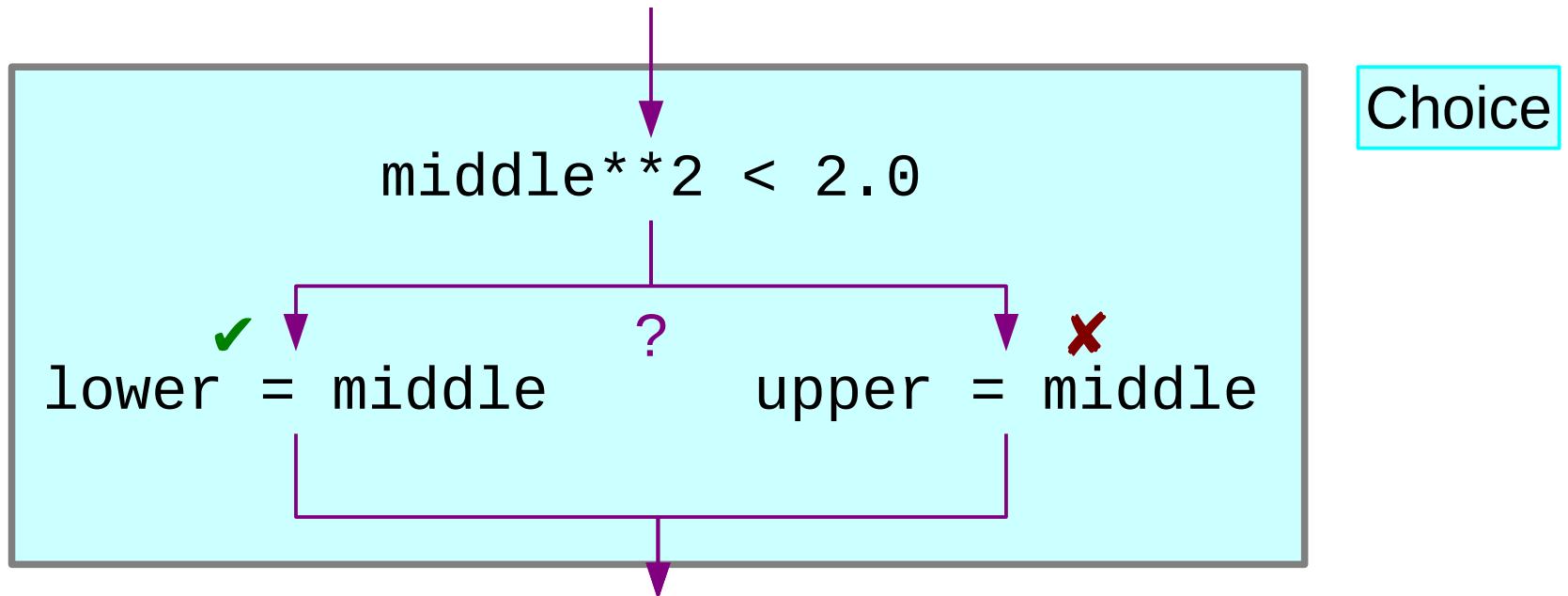
Choice

?

```
    print(lower, upper)
```

```
    uncertainty = upper - lower
```

Choosing



`middle ** 2 < 2.0` → True or False

True → `lower = middle`

False → `upper = middle`

Simple example

```
text = input('Number? ')
number = int(text)

if number % 2 == 0:
    print('Even number')
else:
    print('Odd number')

print('That was fun!')
```

ifthenelse1.py

```
$ python3 ifthenelse1.py
Number? 8
Even number
That was fun
```

```
$ python3 ifthenelse1.py
Number? 7
Odd number
That was fun
```

if...then... else... — 1

The diagram illustrates the structure of an if-else statement. It starts with the 'if' keyword, followed by a test condition 'number % 2 == 0', which is preceded by the 'Test' label. A colon follows the condition, indicated by the 'Colon' label. The code block is enclosed in a light blue box. Arrows point from the labels to their corresponding parts in the code.

```
if number % 2 == 0 :  
    print('Even number')  
  
else :  
    upper = middle  
  
print('That was fun!')
```

if...then... else... — 2

```
if number % 2 == 0 :  
    print('Even number')  
else :  
    upper = middle  
    print('That was fun!')
```

Run if test is **True**

Indentation

if...then... else... — 3

```
if number % 2 == 0 :  
    print('Even number')  
  
else :  
    upper = middle  
    print('That was fun!')
```

else: keyword

Run if test is **False**

Indentation

if...then... else... — 4

```
if number % 2 == 0 :  
    print('Even number')  
  
else :  
    upper = middle  
  
    print('That was fun!')
```

Run afterwards
regardless of test

Our square root example

```
middle = (lower + upper)/2
```

Before

```
if middle**2 < 2.0 :
```

```
    lower = middle
```

```
else :
```

```
    upper = middle
```



if... block

```
print(lower, upper)
```

After

Progress

if ... :

before

else:

if *test* :
 *action*₁
 *action*₂
else:
 *action*₃

choice of two
code blocks

afterwards

 indentation

Exercise 8

For each script:

uuuu Predict what it will do.

uuuu Run the script.

uuuu Were you right?

`ifthenelse2.py`

`ifthenelse3.py`

`ifthenelse4.py`



5 minutes

Back to our example

```
tolerance = 1.0e-15  
lower = 0.0  
upper = 2.0  
uncertainty = upper - lower
```

```
while uncertainty > tolerance :
```

```
    middle = (lower + upper)/2
```

```
        if middle**2 < 2.0 :
```

```
            lower = middle
```

```
        else :
```

```
            upper = middle
```

```
        print(lower, upper)
```

```
        uncertainty = upper - lower
```

if starts
indented

Doubly
indented

Levels of indentation

```
tolerance = 1.0e-15  
lower = 0.0  
upper = 2.0  
uncertainty = upper - lower
```

```
while uncertainty > tolerance :
```

```
    middle = (lower + upper)/2
```

4 spaces

```
    if middle**2 < 2.0 :
```

```
        lower = middle
```

8 spaces

```
    else :
```

```
        upper = middle
```

```
    print(lower, upper)
```

```
    uncertainty = upper - lower
```

Trying it out

```
tolerance = 1.0e-15
lower = 0.0
upper = 2.0
uncertainty = upper - lower

while uncertainty > tolerance :
    middle = (lower + upper)/2

    if middle**2 < 2.0:
        lower = middle
    else:
        upper = middle

    print(lower, upper)
    uncertainty = upper - lower
```

sqrt1.py

```
$ python3 sqrt1.py
```

1.0 2.0

1.0 1.5

1.25 1.5

1.375 1.5

1.375 1.4375

1.40625 1.4375

1.40625 1.421875

...

1.414213... 1.414213...



Script for the square root of 2.0

```
tolerance = 1.0e-15
lower = 0.0
upper = 2.0 ← √2.0
uncertainty = upper - lower

while uncertainty > tolerance :
    middle = (lower + upper)/2
    if middle**2 < 2.0 : ← √2.0
        lower = middle
    else :
        upper = middle
    print(lower, upper)
    uncertainty = upper - lower
```

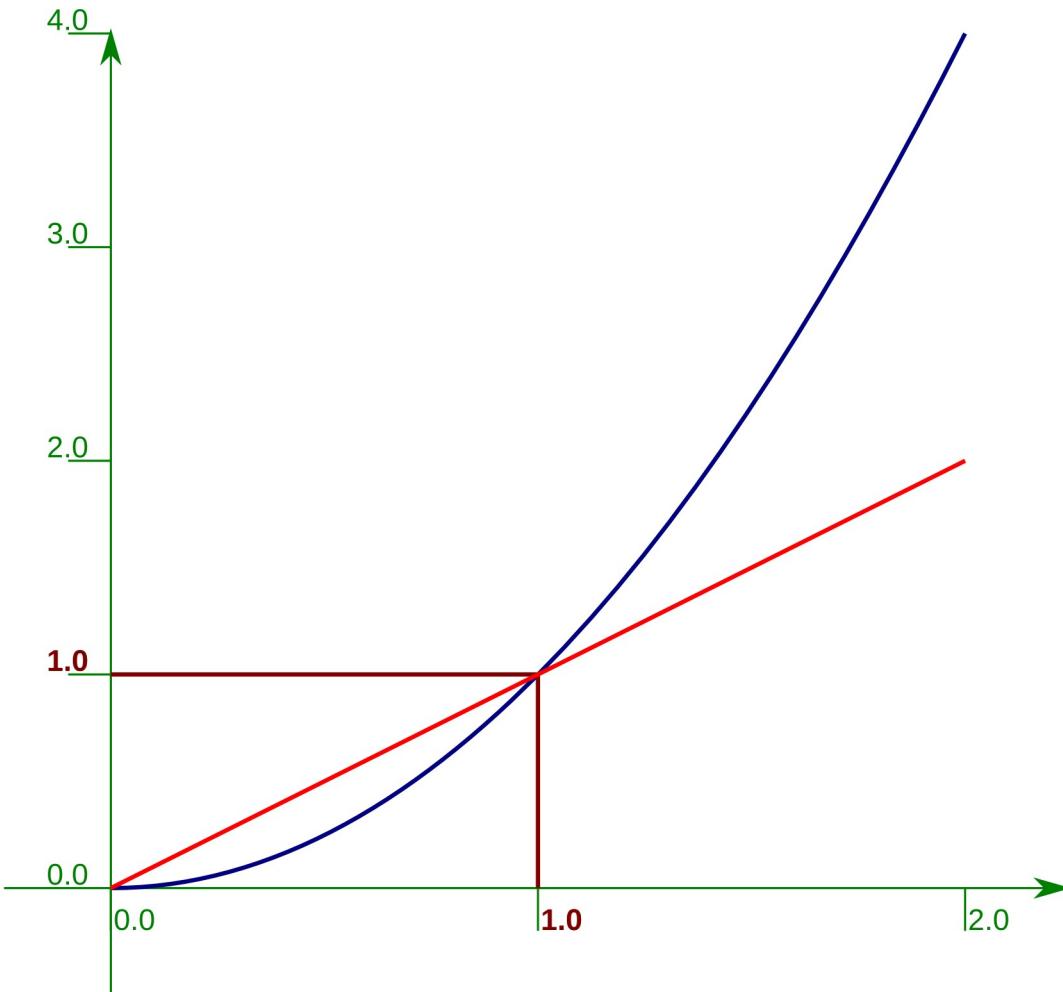
Input target

```
text = input('Number? ')  
number = float(text)
```

...

```
if middle**2 < number :
```

Initial bounds?



lower = ?
upper = ?

$x > 1.0 \rightarrow 1.0 < \sqrt{x} < x$

$1.0 > x \rightarrow 1.0 > \sqrt{x} > x$

if...then...else...

Initial bounds

```
if number < 1.0 :  
    lower = number  
    upper = 1.0  
else :  
    lower = 1.0  
    upper = number
```

Generic square root script?

```
text = input('Number? ')
number = float(text)

if number < 1.0:
    lower = number
    upper = 1.0
else:
    lower = 1.0
    upper = number

tolerance = 1.0e-15
uncertainty = upper - lower

while uncertainty > tolerance:
    middle = (lower+upper)/2.0
    if middle**2 < number:
        lower = middle
    else:
        upper = middle

    uncertainty = upper - lower

print(lower, upper)
```

User input

Initialization

Processing

Output

Negative numbers?

Need to catch negative numbers

```
if number < 0.0:  
    print('Number must be positive!')  
    exit()
```

Quit immediately

"else" is optional

“Chained” tests

```
text = input('Number? ')  
number = float(text)
```

User input

```
if number < 0.0:  
    print('Number must be positive!')  
    exit()
```

Input validation

```
if number < 1.0:  
    lower = number  
    upper = 1.0  
else:  
    lower = 1.0  
    upper = number
```

Initialization

...

“Chained” tests — syntactic sugar

```
text = input('Number? ')
number = float(text)

if number < 0.0:
    print('Number must be positive!')
    exit()

elif number < 1.0: ←
    lower = number
    upper = 1.0
else:
    lower = 1.0
    upper = number
```

elif: “else if”

...

Without elif...

```
text = input('Number? ')
number = float(text)

if number < 0.0:
    print('Number is negative.')
else:
    if number < 1.0:
        print('Number is between zero and one.')
    else:
        if number < 2.0:
            print('Number is between one and two.')
        else:
            if number < 3.0:
                print('Number is between two and three.')
            else:
                print('Number is three or more.)
```

Stacked clauses get unwieldy

With elif...

```
text = input('Number? ')
number = float(text)

if number < 0.0:
    print('Number is negative.')
elif number < 1.0:
    print('Number is between zero and one.')
elif number < 2.0:
    print('Number is between one and two.')
elif number < 3.0:
    print('Number is between two and three.')
else:
    print('Number is three or more.')
```

Progress

Nested structures

```
while ... :  
    if ... :
```

Chained tests

```
if ... :  
    ...  
elif ... :
```

Testing inputs to scripts

```
    ...  
elif ... :  
    ...  
else:  
    ...
```

```
exit()
```

Exercise 9

**Only attempt each part after
you have the previous part working!**

exercise9.py

1. Edit the square root script to catch negative numbers.
2. Edit the square root script to ask for the tolerance.
3. Edit the square root script to catch negative tolerances.



10 minutes

Comments

We have written our first real Python script

What did it do?

Why did it do it?

Need to annotate the script

Python comment character

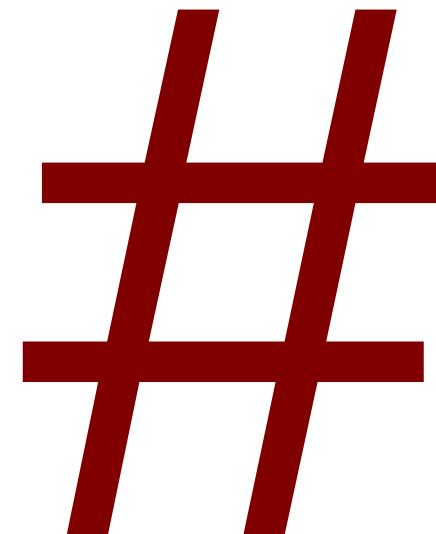
The “hash” character

a.k.a. “pound”, “number”, “sharp”

Lines starting with “#” are ignored

Partial lines starting “#” are ignored

Used for annotating scripts



Python commenting example

```
# Script to calculate square roots by bisection
# (c) Bob Dowling 2012. Licensed under GPL v3.0
text = input('Number?')
number = float(text) # Need a real number

# Test number for validity,
# set initial bounds if OK.
if number < 0.0:
    print('Number must be non-negative!')
    exit()
elif number < 1.0:
    lower = number
    upper = 1.0
else:
    lower = 1.0
    upper = number
```

On a *real* Unix system...

```
#!/usr/bin/python3
```

```
# Script to calculate square roots by bisection
# (c) Bob Dowling 2012. Licensed under GPL v3.0
text = input('Number? ')
number = float(text) # Need a real number
```

Magic line for executable files

\$ **fubar.py**

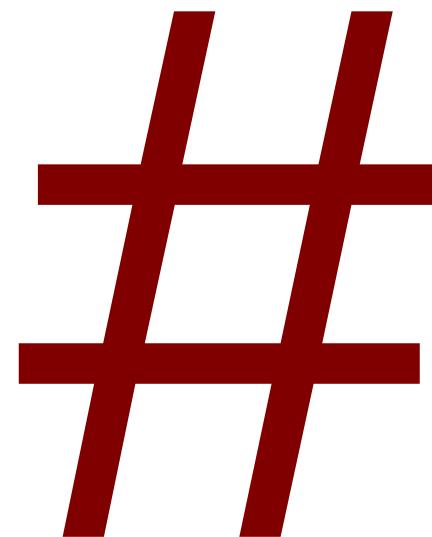
instead of

\$ **python3 fubar.py**

Progress

Comments

“#” character



Exercise 10

Comment your square root script from exercise 9.



2 minutes

Recap: Python types so far

Whole numbers

-127

Floating point numbers

3.141592653589793

Complex numbers

(1.0 + 2.0j)

Text

'The cat sat on the mat.'

Booleans

True

False

Lists

```
[ 'hydrogen', 'helium', 'lithium', 'beryllium',
'boron', ..., 'thorium', 'protactinium', 'uranium' ]
```

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

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

What is a list?

hydrogen, helium, lithium, beryllium, ..., protactinium, uranium

A sequence of values

The names of the elements

Values stored in order

Atomic number order

Individual value identified
by position in the sequence

“helium” is the name of the
second element

What is a list?

2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59

A sequence of values

The prime numbers
less than sixty

Values stored in order

Numerical order

Individual value identified
by position in the sequence

7 is the fourth prime

Creating a list in Python

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

A literal list

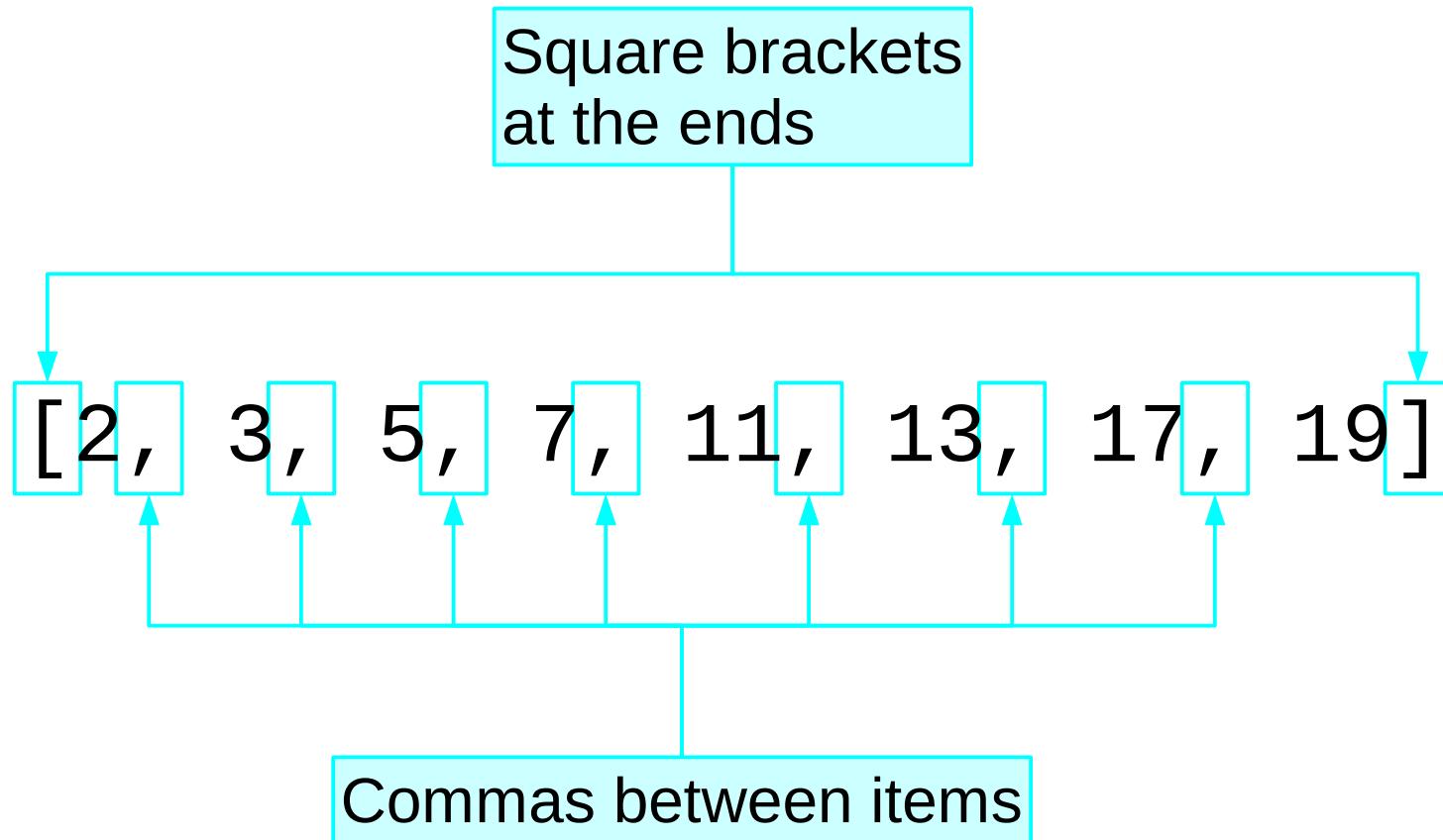
```
>>> primes  
[2, 3, 5, 7, 11, 13, 17, 19]
```

The whole list

```
>>> type(primes)  
<class 'list'>
```

A Python type

How Python presents lists

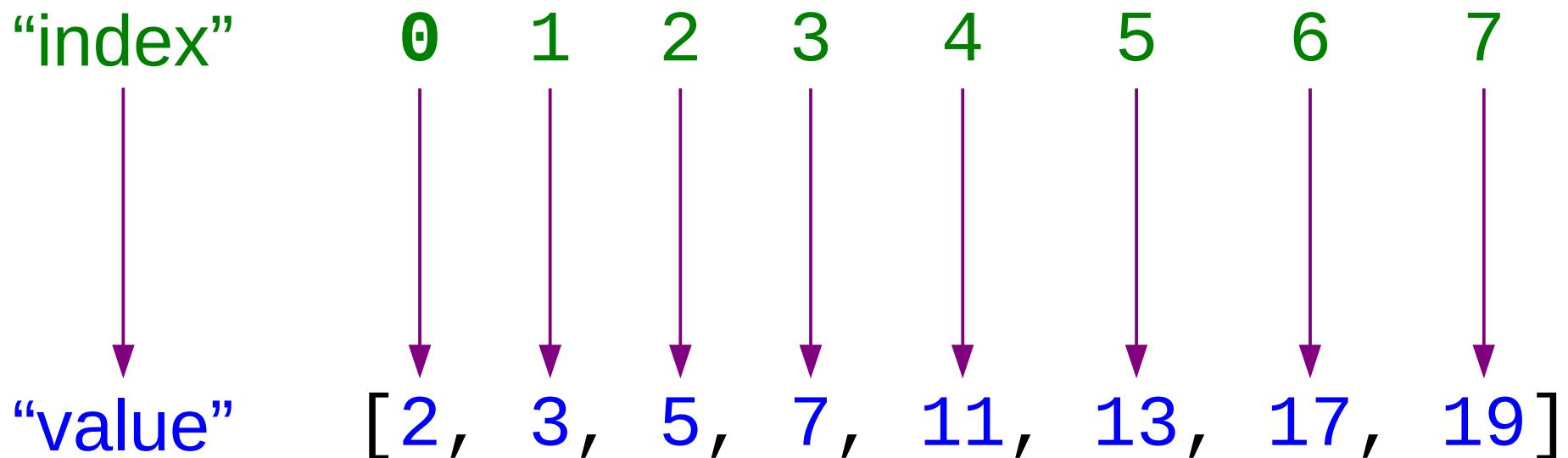


Square brackets

```
primes = [2, 3, 5, 7, 11]
```

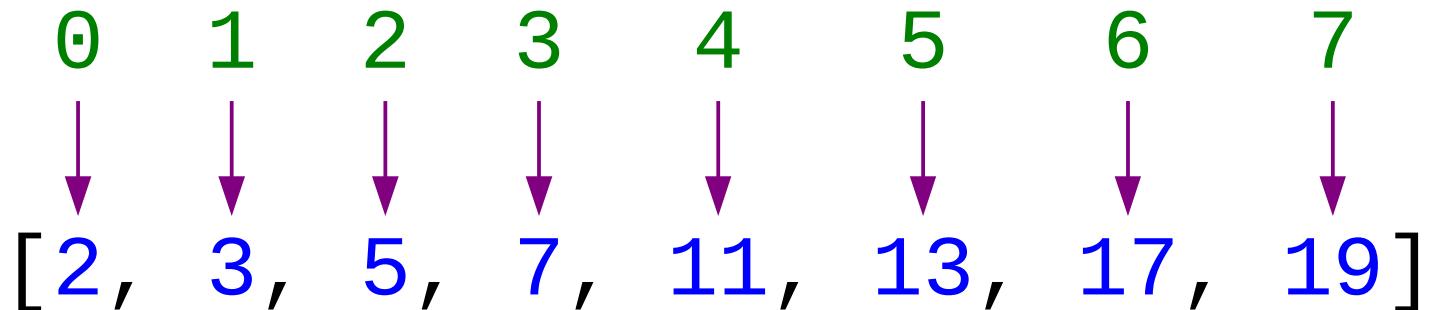
Literal list

Python counts from zero



Looking things up in a list

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



index

```
>>> primes[0]
```

2

square brackets

```
>>> primes[6]
```

17

204

Square brackets

`primes = [2, 3, 5, 7, 11]`

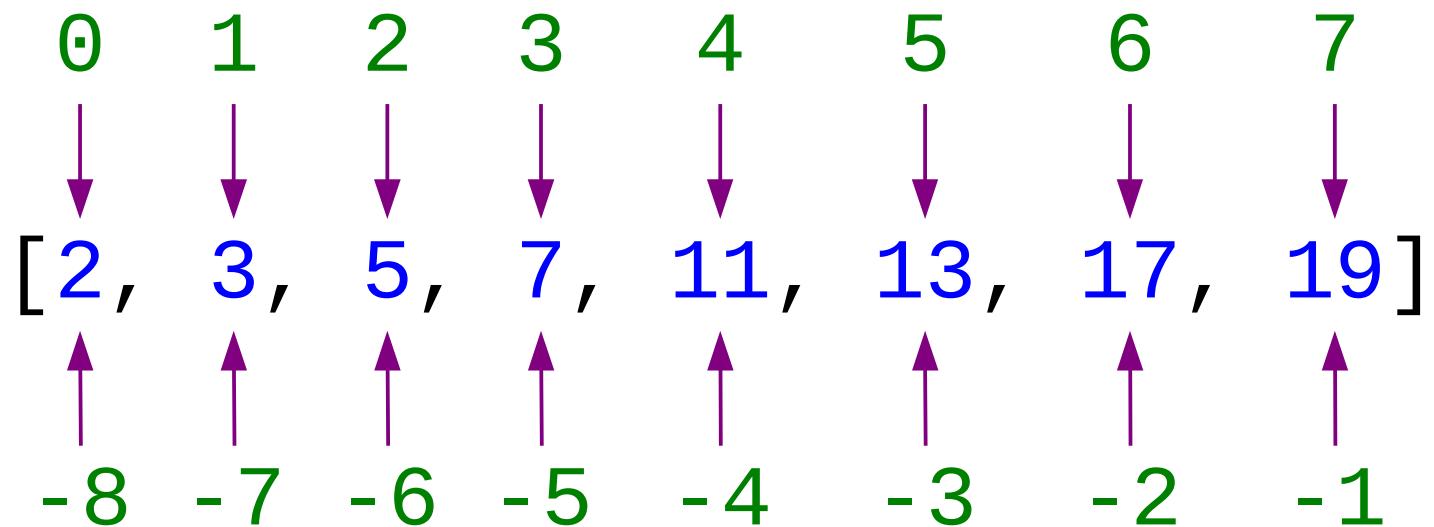
Literal list

`primes[3]`

Index into list

Counting from the end

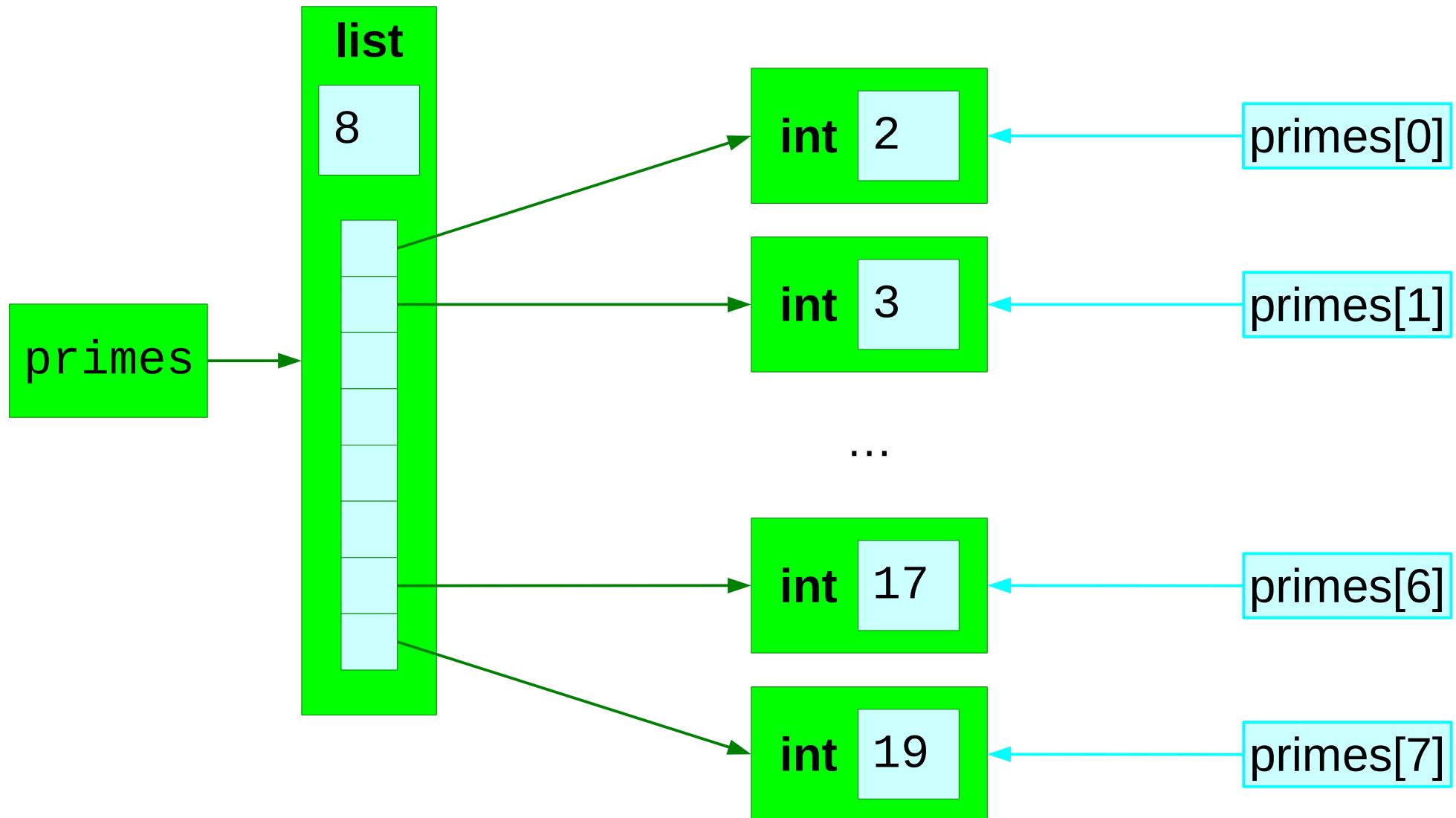
```
>>> primes = [ 2, 3, 5, 7, 11, 13, 17, 19]
```



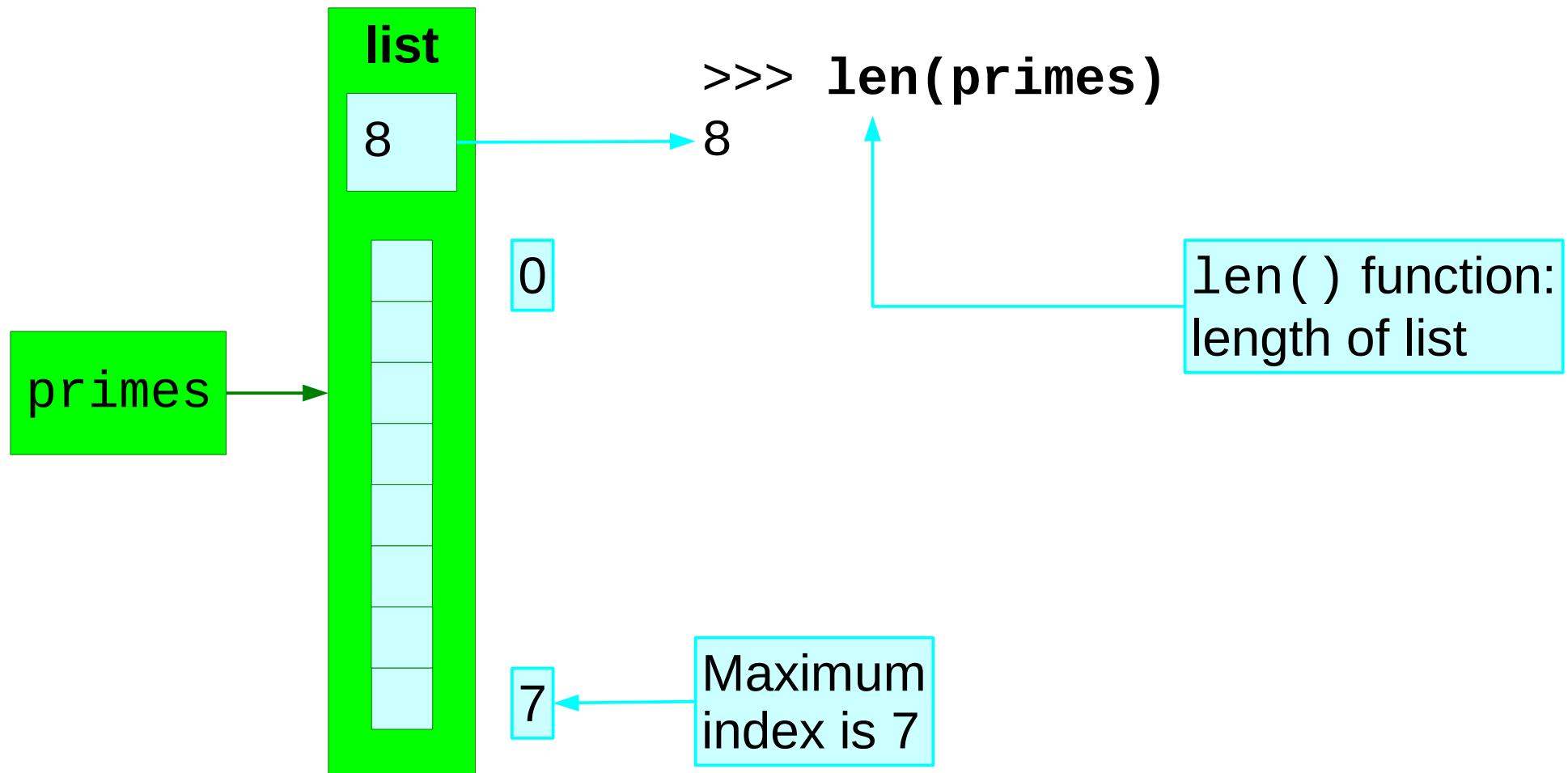
getting at the last item

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

Inside view of a list



Length of a list



Changing a value in a list

```
>>> data = ['alpha', 'beta', 'gamma']
```

The list

```
>>> data[2]  
'gamma'
```

Initial value

```
>>> data[2] = 'G'
```

Change value

```
>>> data[2]  
'G'
```

Check change

```
>>> data  
['alpha', 'beta', 'G']
```

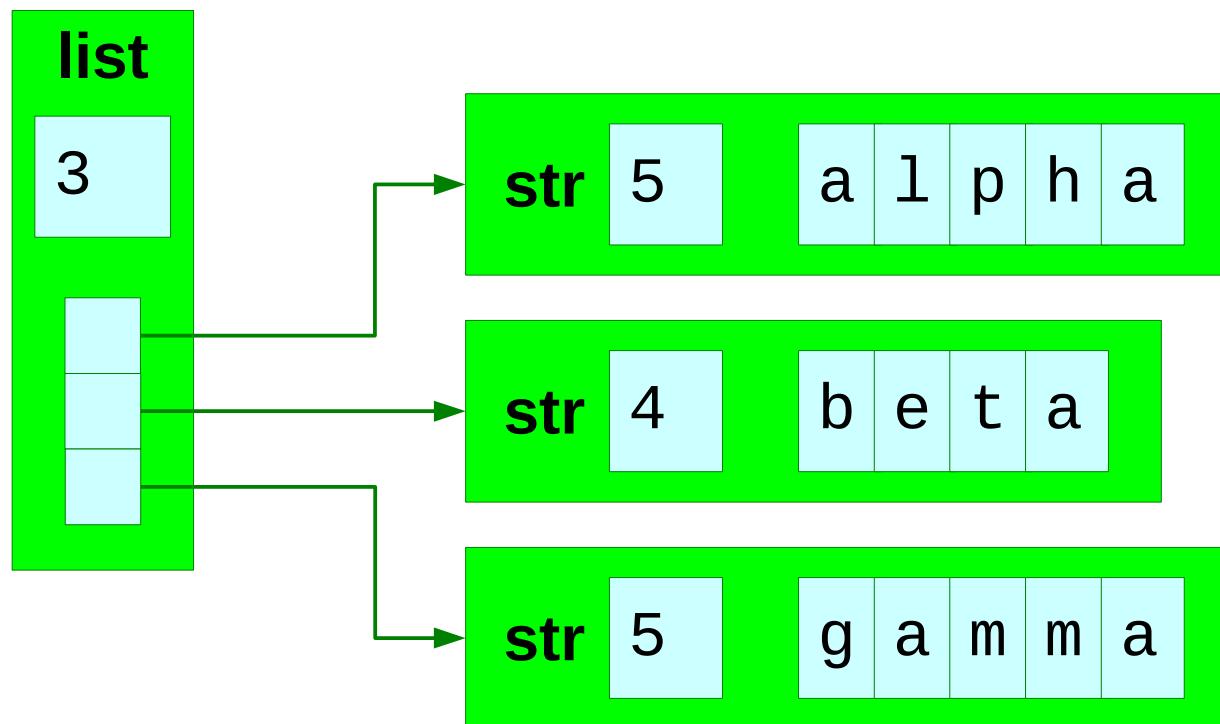
Changed list

Changing a value in a list — 1



Right to left

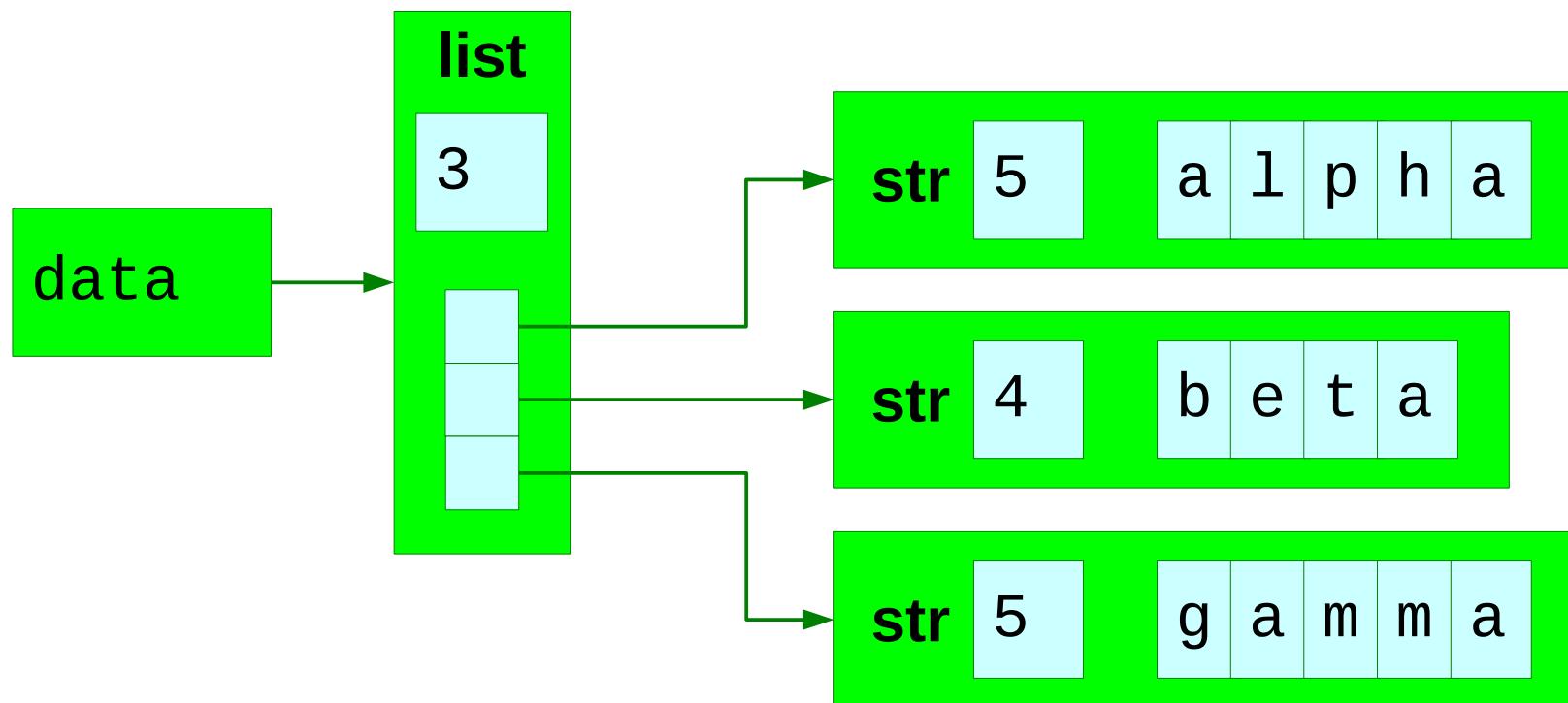
```
>>> data = ['alpha', 'beta', 'gamma']
```



Changing a value in a list — 2

```
<-->  
=> >>> data = ['alpha', 'beta', 'gamma']
```

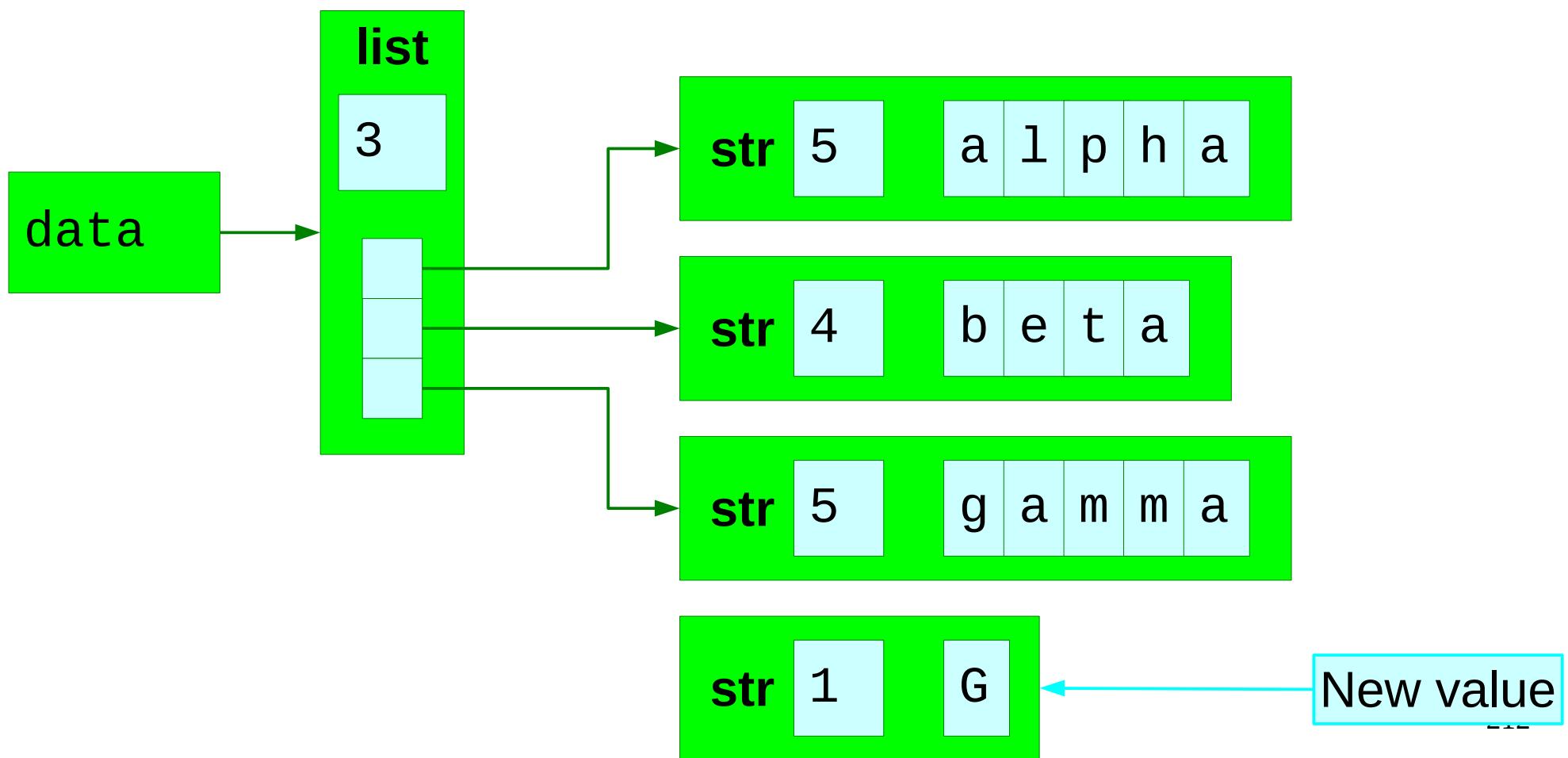
Right to left



Changing a value in a list — 3

```
<-->  
">>>> data[2] = 'G'
```

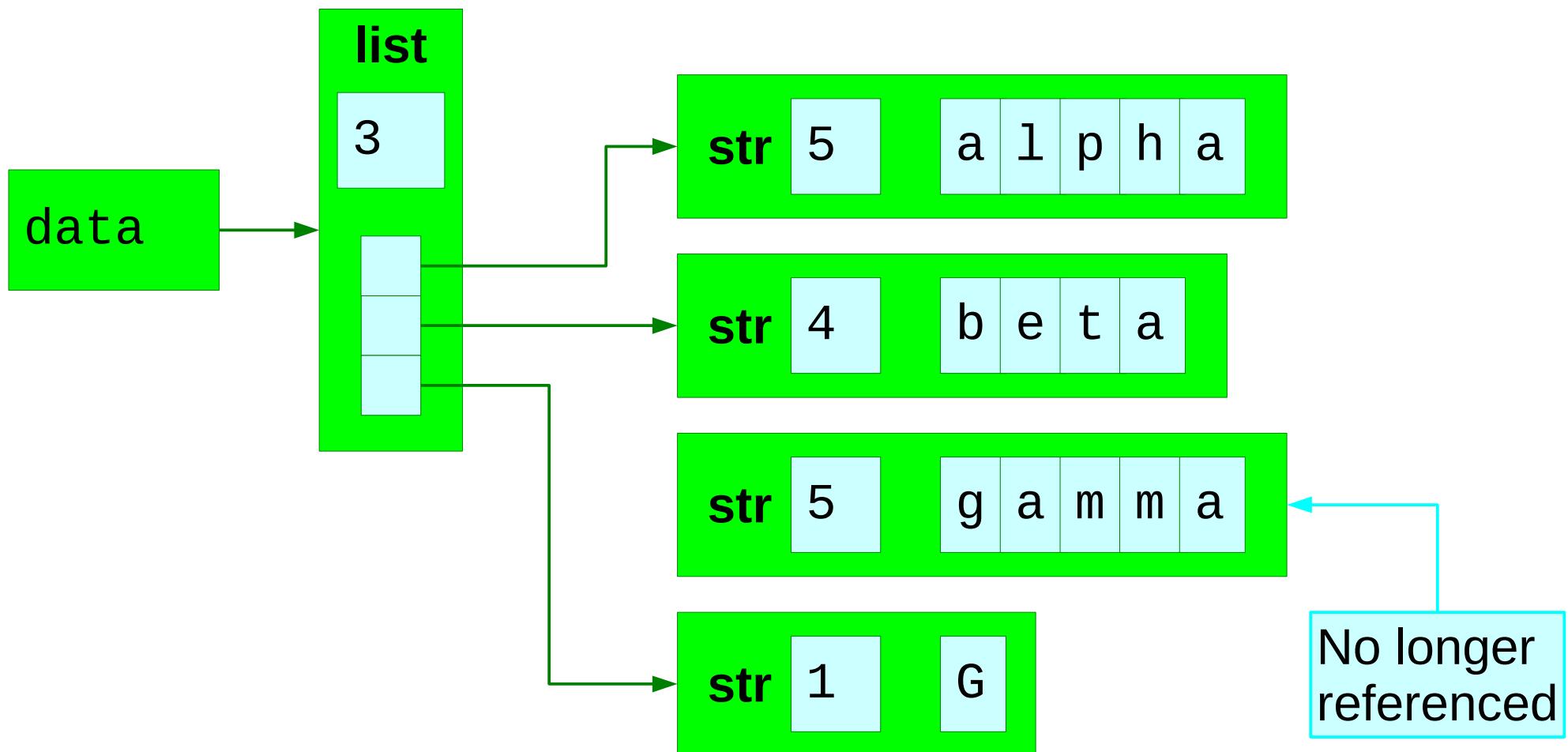
Right to left



Changing a value in a list — 4

```
>>> data[2] = 'G'
```

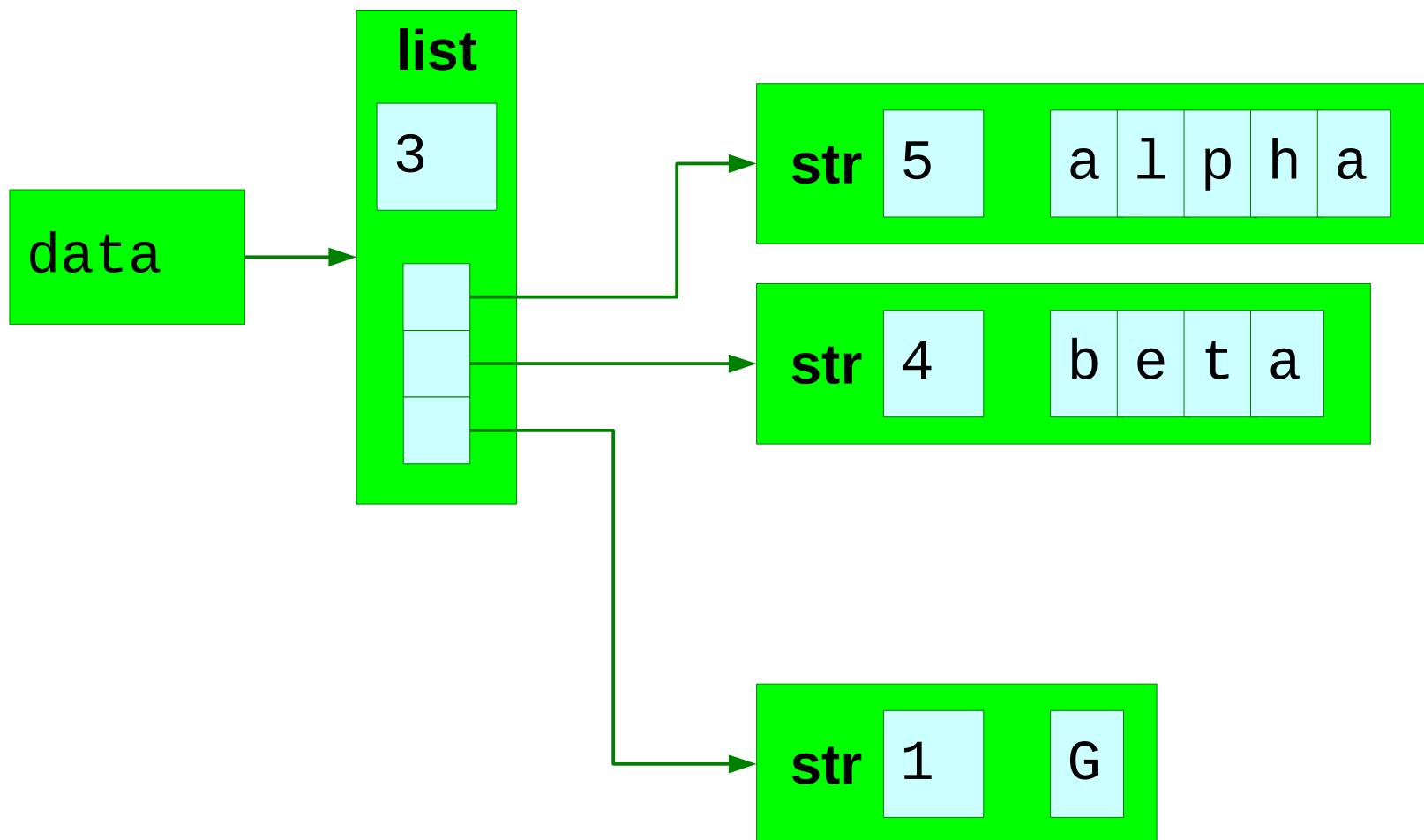
Right to left



Changing a value in a list — 5

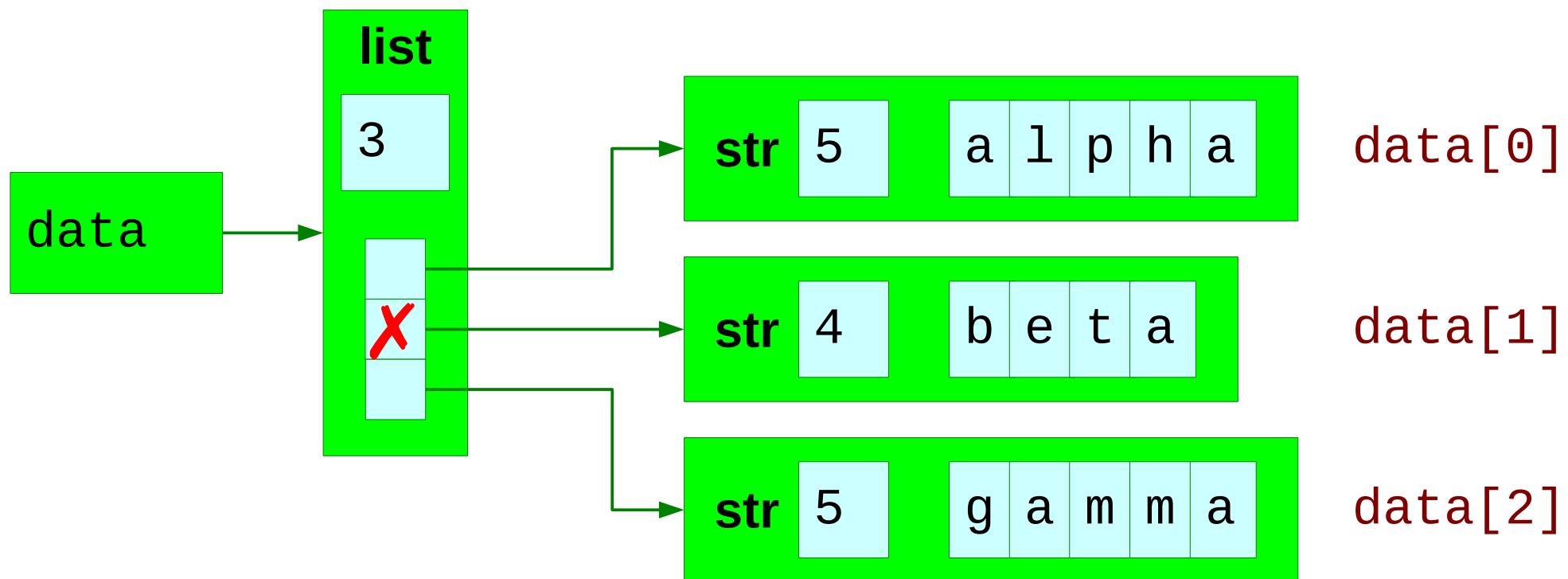
```
<-->  
">>>> data[2] = 'G'
```

Right to left



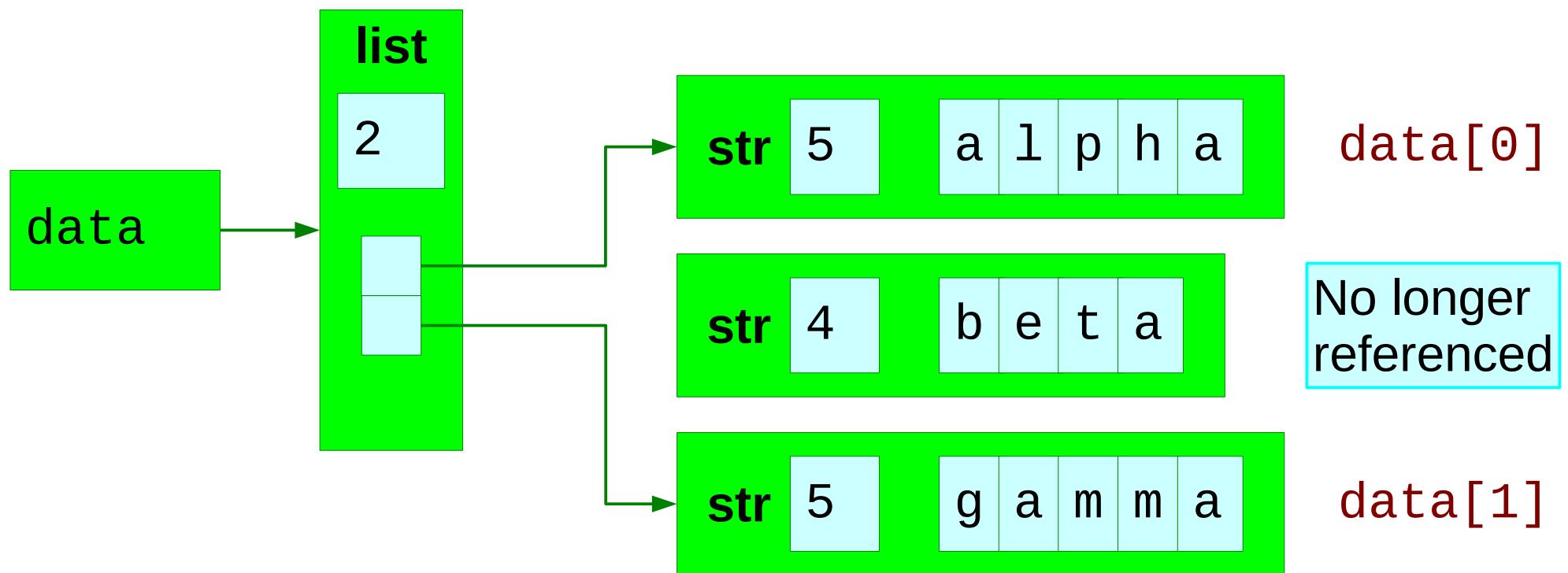
Removing an entry from a list — 1

```
>>> del data[1]
```



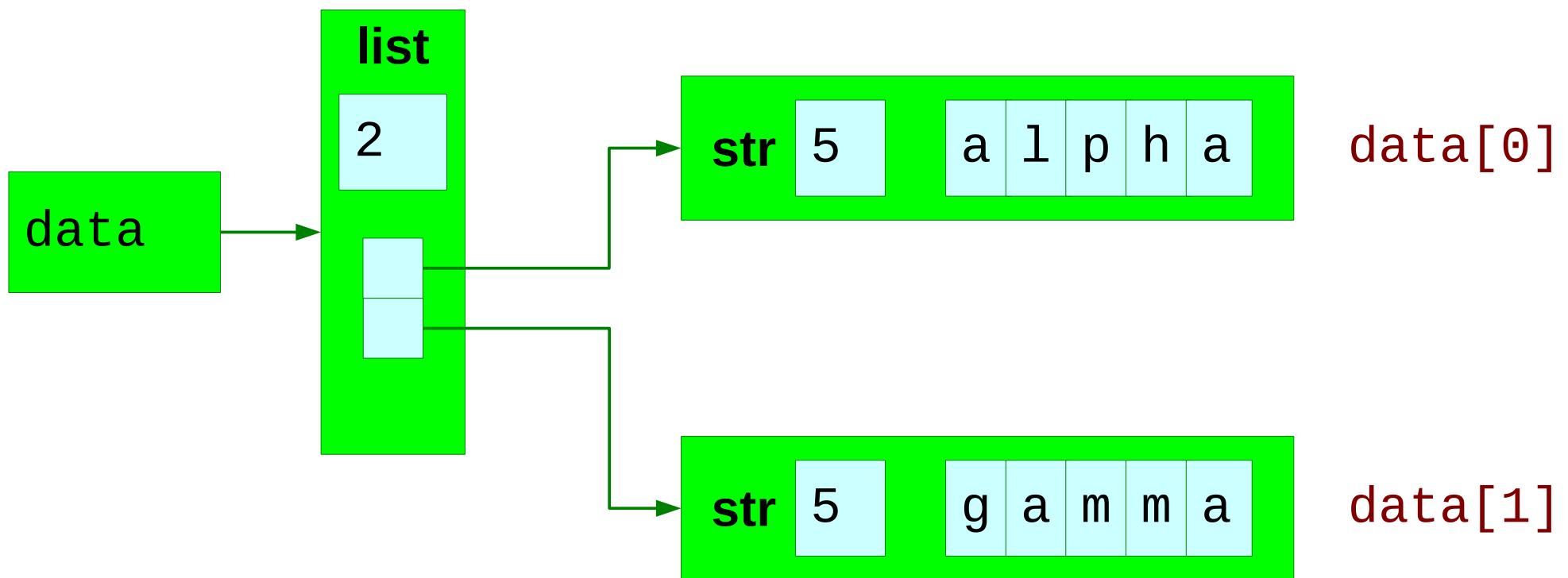
Removing an entry from a list — 2

```
>>> del data[1]
```

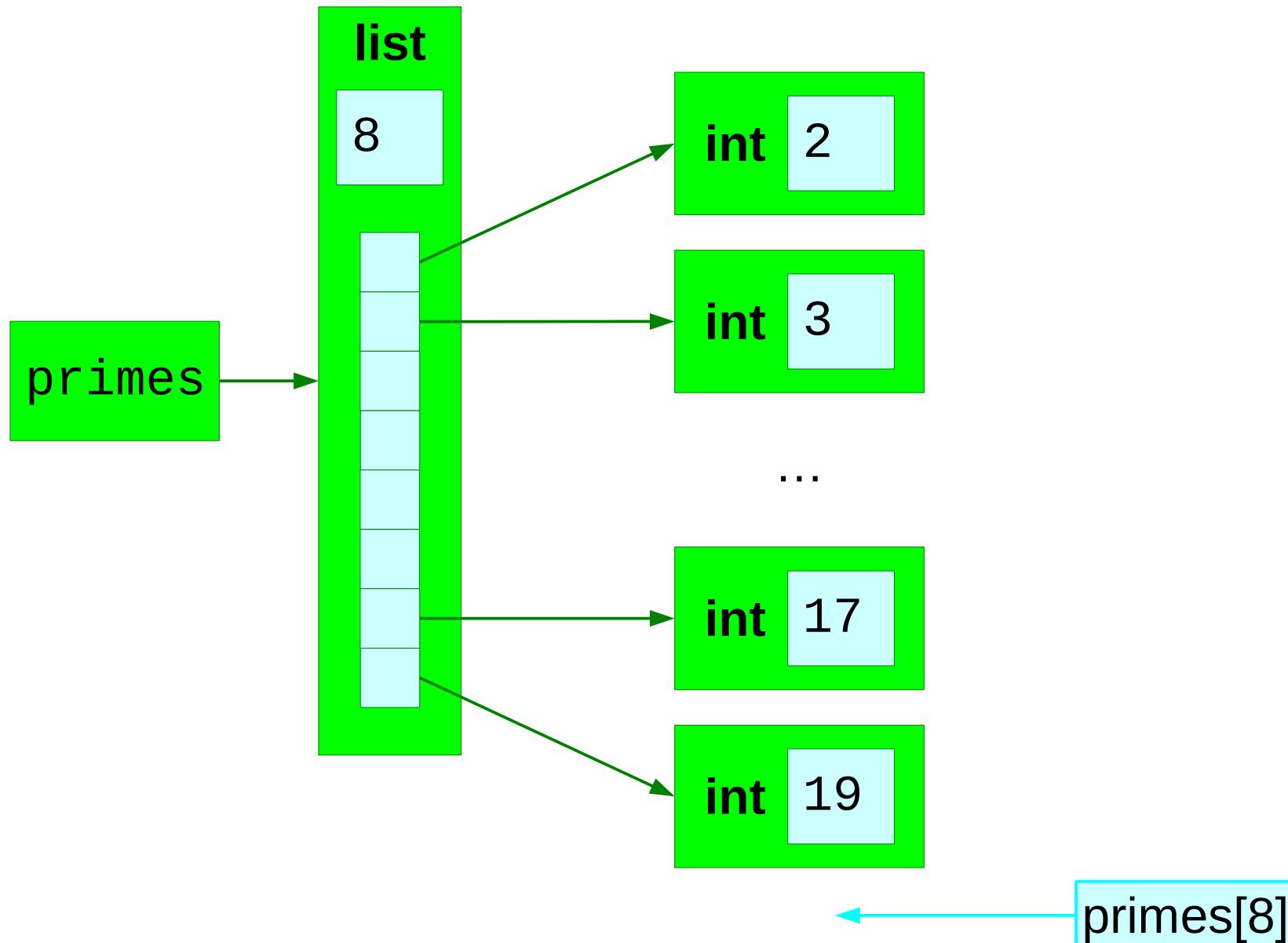


Removing an entry from a list — 3

```
>>> del data[1]
```



Running off the end



Running off the end

```
>>> len(primes)
```

```
8
```

```
>>> primes[7]
```

```
19
```

```
>>> primes[8]
```

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

Type of error

Description of error

Running off the end

```
>>> primes[8] = 23
```

```
Traceback (most recent call last):
```

```
  File "<stdin>", line 1, in <module>
```

```
IndexError: list assignment index out of range
```

Same type
of error

Similar description of error
but with “assignment”

Progress

Lists

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

index

primes[4]

Count from zero

primes[0]

Deletion

del primes[6]

Length

len(primes)

Over-running

primes[8]

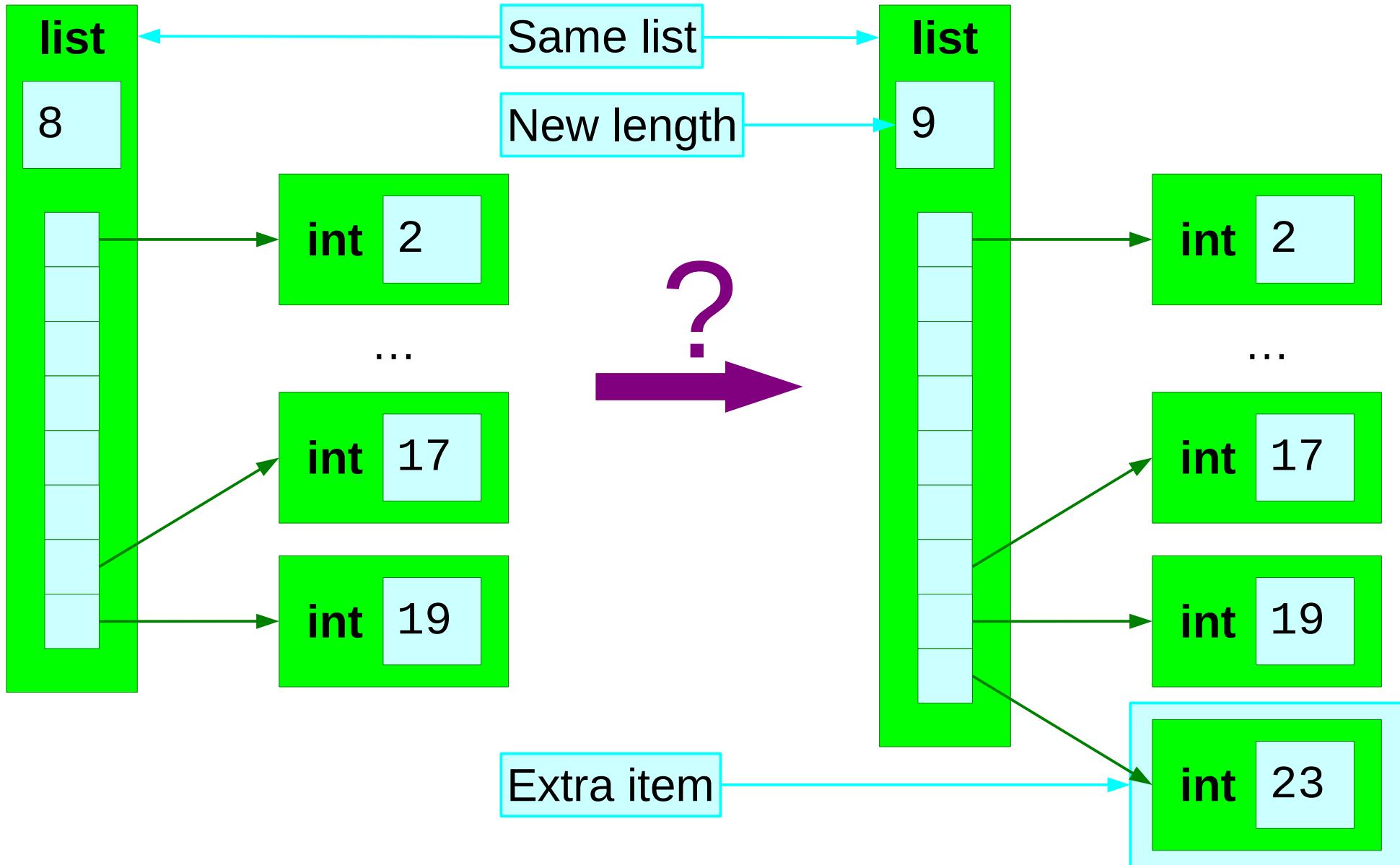
Exercise 11

Track what the value of `numbers` is at each stage of this sequence of instructions.

- 1 `>>> numbers = [5, 7, 11, 13, 17, 19, 29, 31]`
- 2 `>>> numbers[1] = 3`
- 3 `>>> del numbers[3]`
- 4 `>>> numbers[3] = 37`
- 5 `>>> numbers[4] = numbers[5]`



How can we add to a list?



Appending to a list

```
>>> primes
```

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

A function built into a list

```
>>> primes.append(23)
```

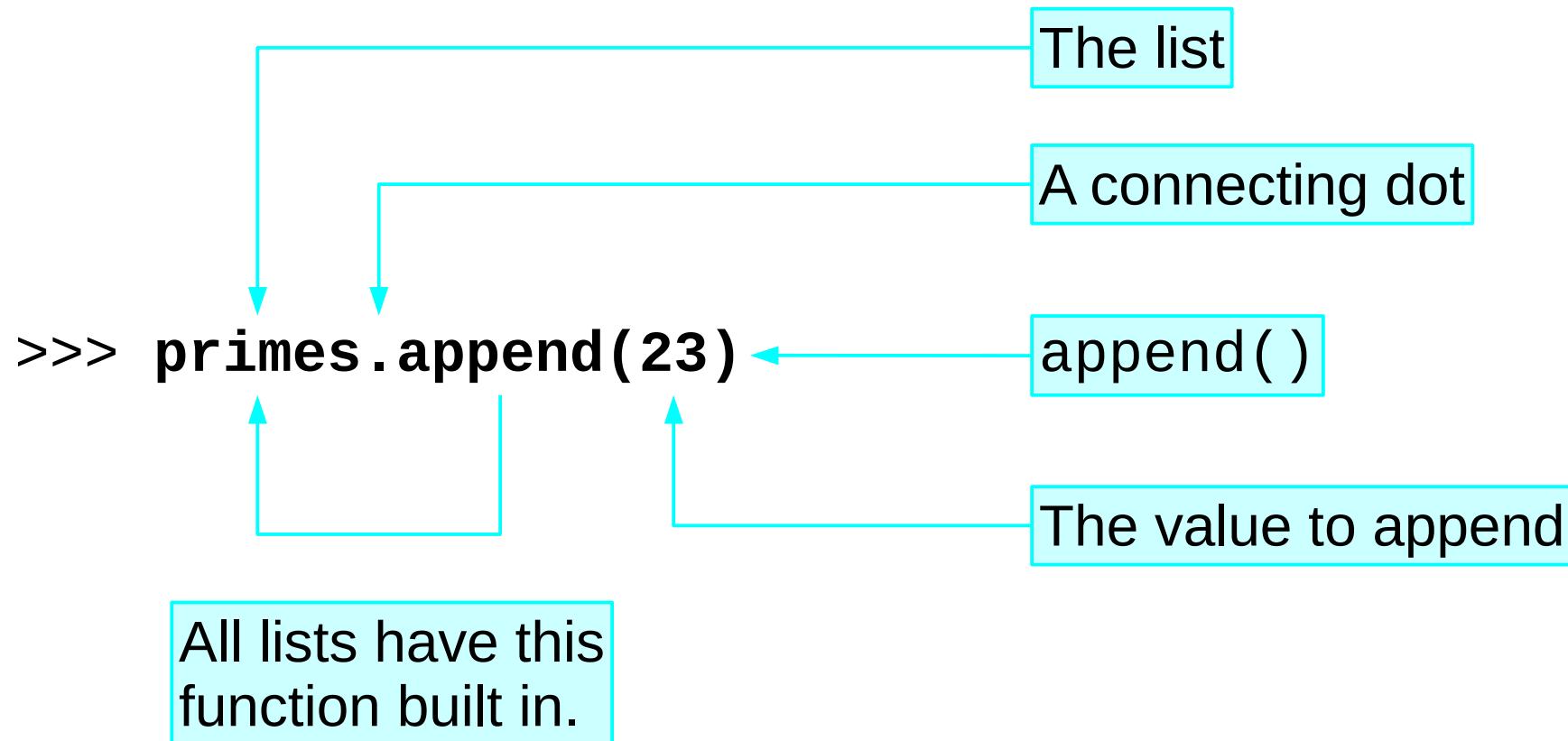
The list is now updated

```
>>> primes
```

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

23]

primes.append() ?



“Methods”

Behaves just
like a function

object . *function* (*arguments*)

a function that has
special access to
the object's data.

Using the append() method

```
>>> print(primes)
```

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

```
>>> primes.append(23)
```

```
>>> primes.append(29)
```

```
>>> primes.append(31)
```

```
>>> primes.append(37)
```

```
>>> print(primes)
```

The function doesn't return any value.

It modifies the list itself.

```
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37]
```

Other methods on lists: reverse()

```
>>> numbers = [4, 7, 5, 1]
```

```
>>> numbers.reverse()
```

The function doesn't return any value.

```
>>> print(numbers)
```

It modifies the list itself.

```
[1, 5, 7, 4]
```

Other methods on lists: sort()

```
>>> numbers = [4, 7, 5, 1]
```

```
>>> numbers.sort()
```

The function does not return the sorted list.

```
>>> print(numbers)
```

```
[1, 4, 5, 7]
```

It sorts the list itself.

Numerical order.

Other methods on lists: sort()

```
>>> greek = ['alpha', 'beta', 'gamma', 'delta']
```

```
>>> greek.sort()
```

```
>>> print(greek)
```

```
['alpha', 'beta', 'delta', 'gamma']
```

Alphabetical order
of the words.

Other methods on lists: insert()

```
0           1           2  
>>> greek = ['alpha', 'gamma', 'delta']
```

```
>>> greek.insert(1, 'beta')
```

Where to insert

What to insert

```
>>> greek
```

```
['alpha', 'beta', 'gamma', 'delta']
```

0

1

Displaced

Other methods on lists: remove()

```
>>> numbers = [7, 4, 8, 7, 2, 5, 4]
```

```
>>> numbers.remove(8) Value to remove
```

```
>>> print(numbers)
```

```
[7, 4, 7, 2, 5, 4]
```

```
c.f. del numbers[2] Index to remove
```

Other methods on lists: remove()

```
>>> print(numbers)
```

```
[7, 4, 7, 2, 5, 4]
```

There are two instances of 4.

```
>>> numbers.remove(4)
```

```
>>> print(numbers)
```

```
[7, 7, 2, 5, 4]
```

Only the first instance is removed

What methods are there?

```
>>> help(numbers)
```

Help on list object:

```
class list(object)
```

```
...
```

```
| append(...)
```

```
|     L.append(object) -- append object to end
```

```
...
```

Pagination:



next page



back one page



quit

Help on a single method

```
>>> help(numbers.append)
```

Help on built-in function append:

append(...)

L.append(object) -- append object to end

Sorting a list *redux*

```
>>> greek = ['alpha', 'beta', 'gamma', 'delta']
```

```
>>> greek.sort()
```

Recall: `greek.sort()`
sorts the list “in place”.

```
>>> print(greek)
```

```
['alpha', 'beta', 'delta', 'gamma']
```

Sorting a list *redux*: “sorted()”

```
>>> greek = ['alpha', 'beta', 'gamma', 'delta']
```

```
>>> print(sorted(greek))
```

```
['alpha', 'beta', 'delta', 'gamma']
```

sorted() function
returns a sorted list...

```
>>> print(greek)
```

```
['alpha', 'beta', 'gamma', 'delta']
```

...and leaves the
list alone

Adding to a list *redux*: “+”

```
>>> primes
```

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

Concatenation
operator

List to add

```
>>> primes + [23, 29, 31]
```

```
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31]
```

Concatenation

Create a new list

```
>>> newlist = primes + [23, 29, 31]
```

Update the list

```
>>> primes = primes + [23, 29, 31]
```

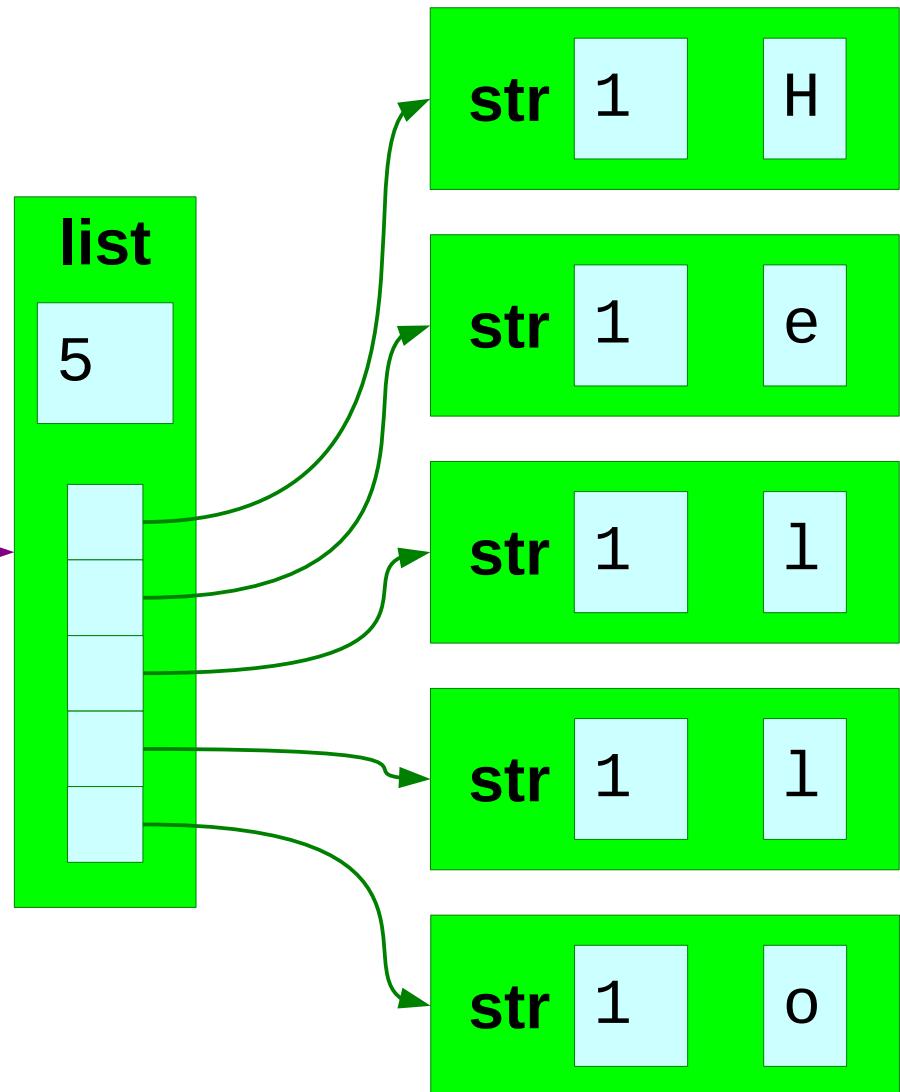
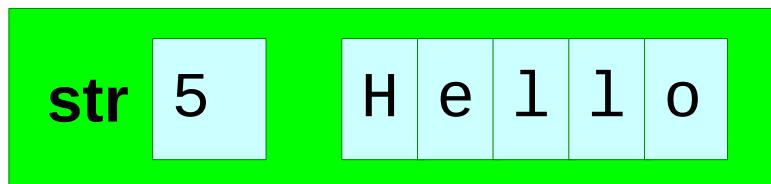
Augmented assignment

```
>>> primes += [23, 29, 31]
```

Creating lists from text — 1

```
>>> list('Hello')
```

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



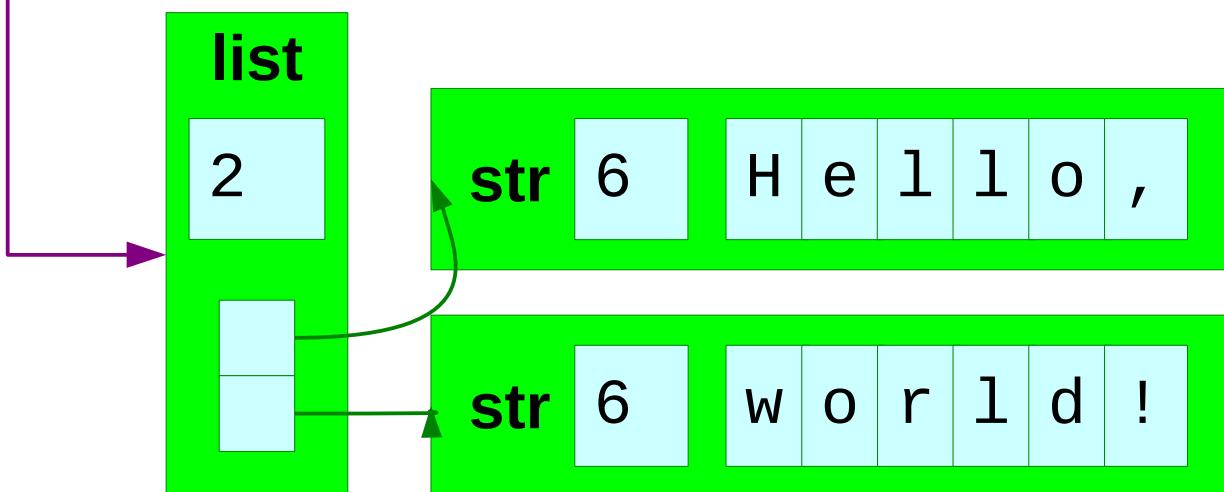
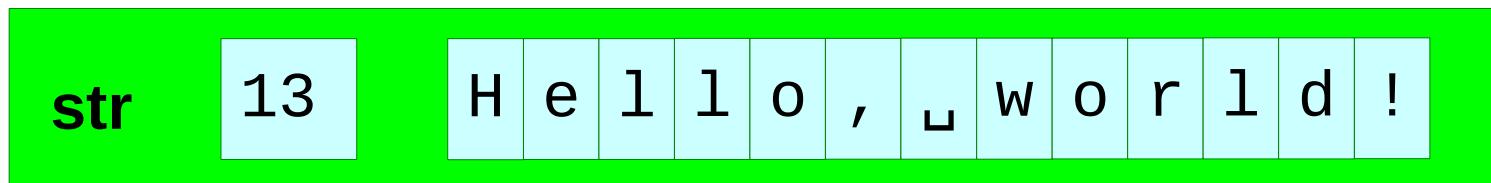
Creating lists from text — 2

```
>>> 'Hello, world!'.split()
```

```
[ 'Hello, ', 'world! ']
```

Built in method

Splits on spaces



Progress

“Methods”

```
append(item)  
reverse()  
sort()  
insert(index, item)  
remove(item)
```

object.method(arguments)

Help

```
help(object)  
help(object.method)
```

Sorting

```
list.sort()  
sorted(list)
```

Concatenation

```
+[  
+=
```

```
[1,2,3] + [4,5,6]  
primes += [29, 31]
```

Exercise 12



5 minutes

Is an item in a list? — 1

```
>>> odds = [3, 5, 7, 9]
```

Does not include 2

```
>>> odds.remove(2)
```

Try to remove 2

```
Traceback (most recent call last):
```

Hard error

```
  File "<stdin>", line 1, in <module>
```

```
    ValueError: list.remove(x): x not in list
```

X

x must be in the list before it can be removed

Is an item in a list? — 2

```
>>> odds = [3, 5, 7, 9]
```

```
>>> 2 in odds
```

False

```
>>> 3 in odds
```

True

```
>>> 2 not in odds
```

True

Precedence

First

$x^{**}y$ $-x$ $+x$ $x\%y$ x/y x^*y $x-y$ $x+y$

$x==y$ $x!=y$ $x>=y$ $x>y$ $x<=y$ $x<y$

$x \text{ not in } y$ $x \text{ in } y$

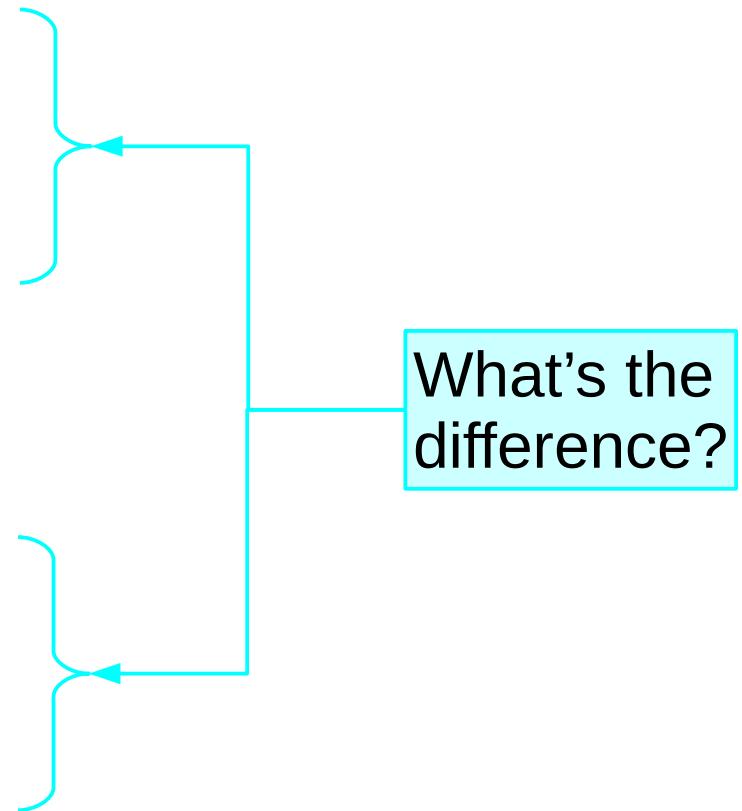
$\text{not } x$ $x \text{ and } y$ $x \text{ or } y$

Last

The list now contains every operator we meet in this course.

Safe removal

```
if number in numbers :  
    numbers.remove(number)
```



```
while number in numbers :  
    numbers.remove(number)
```

Working through a list — 1

e.g. Printing each element on a line

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



```
The  
cat  
sat  
on  
the  
mat.
```

Working through a list — 2

e.g. Adding the elements of a list

[45, 76, -23, 90, 15]



203

What is the sum of an empty list?

[] → ?

Working through a list — 3

e.g. Squaring every number in a list

[4, 7, -2, 9, 1]



[16, 49, 4, 81, 1]

The “for loop” — 1

```
name of list
list
words = ['The', 'cat', 'sat', 'on', 'the', 'mat.']
for word in words :
    print(word)
```

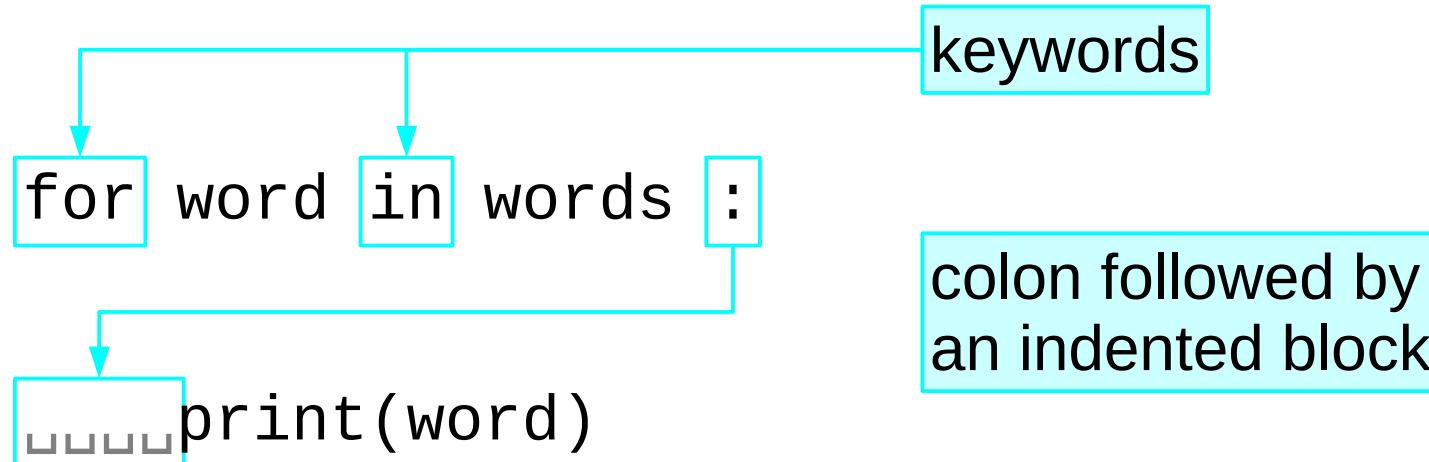
A new Python looping construct

print: What we want to do with the list items.

```
graph TD; A["name of list"] --> B["words"]; A --> C["list"]; D["A new Python looping construct"] --> E["{ word }"]; F["print: What we want to do with the list items."] --> G["print(word)"]
```

The “for loop” — 2

```
words = ['The', 'cat', 'sat', 'on', 'the', 'mat. ']
```



The “for loop” — 3

```
words = ['The', 'cat', 'sat', 'on', 'the', 'mat. ']
```

```
for word in words :
```

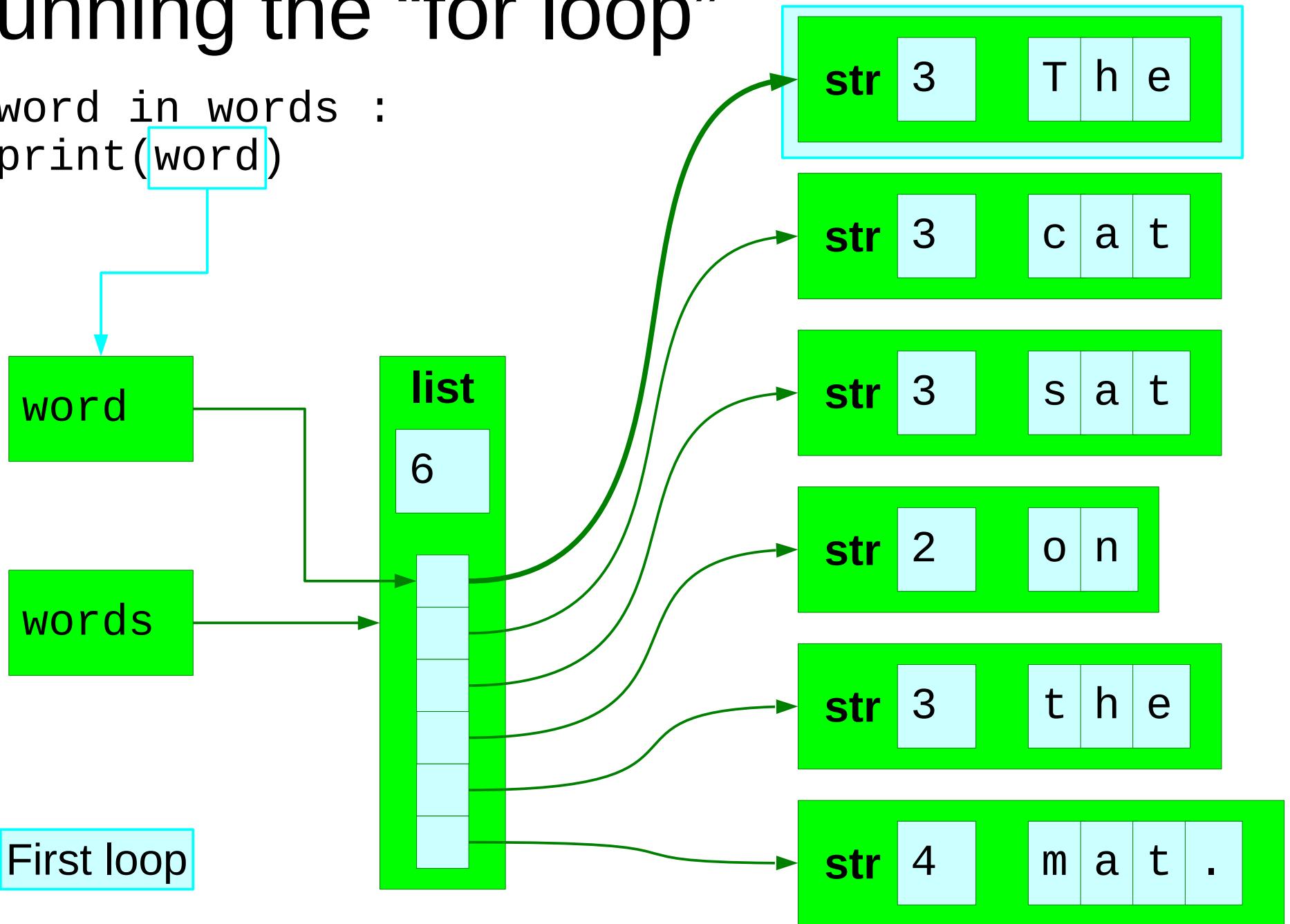
```
    print(word)
```

Defining the **loop variable**

Using the **loop variable**

Running the “for loop”

```
for word in words :  
    print(word)
```



Running the “for loop”

```
for word in words :  
    print(word)
```

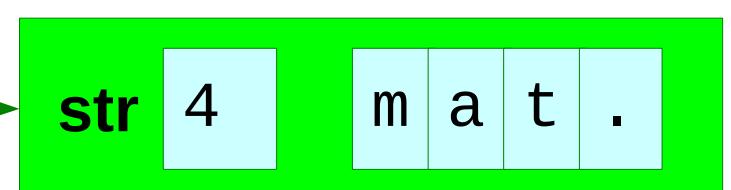
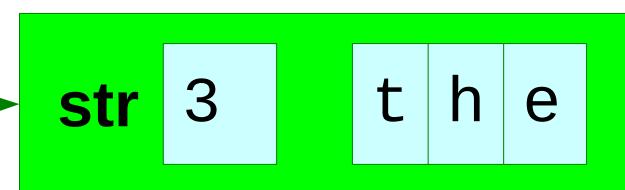
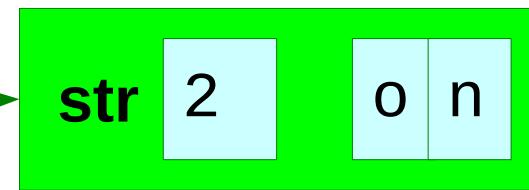
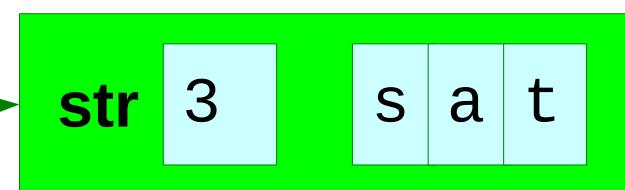
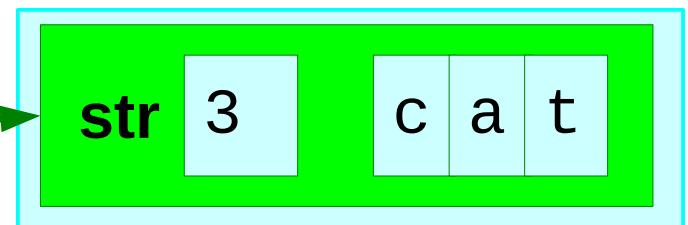
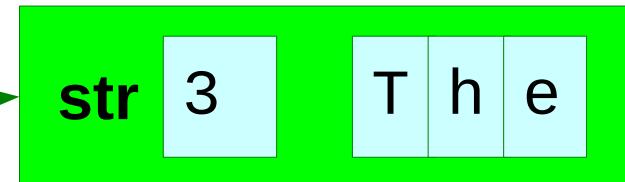
word

words

list

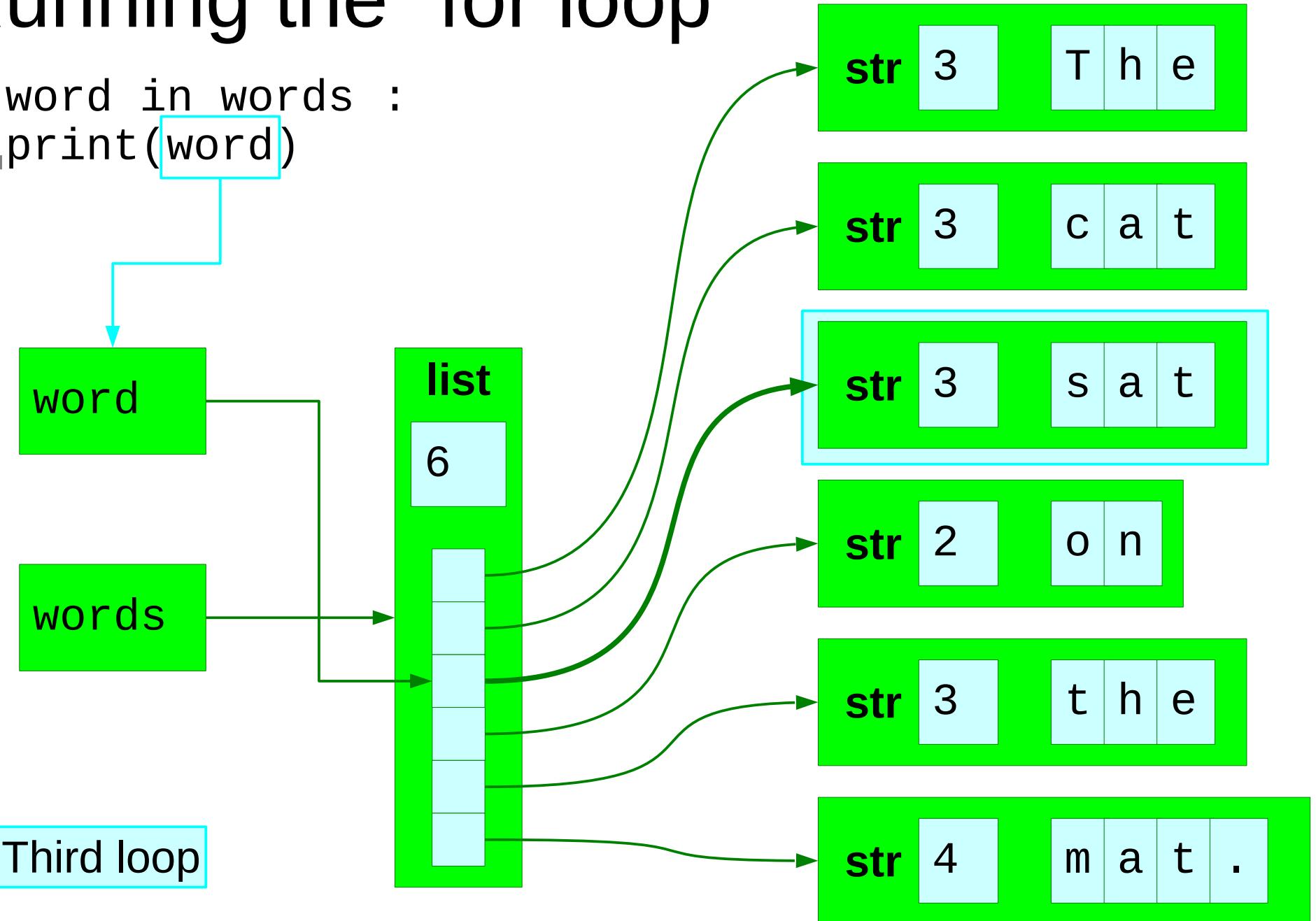
6

Second loop



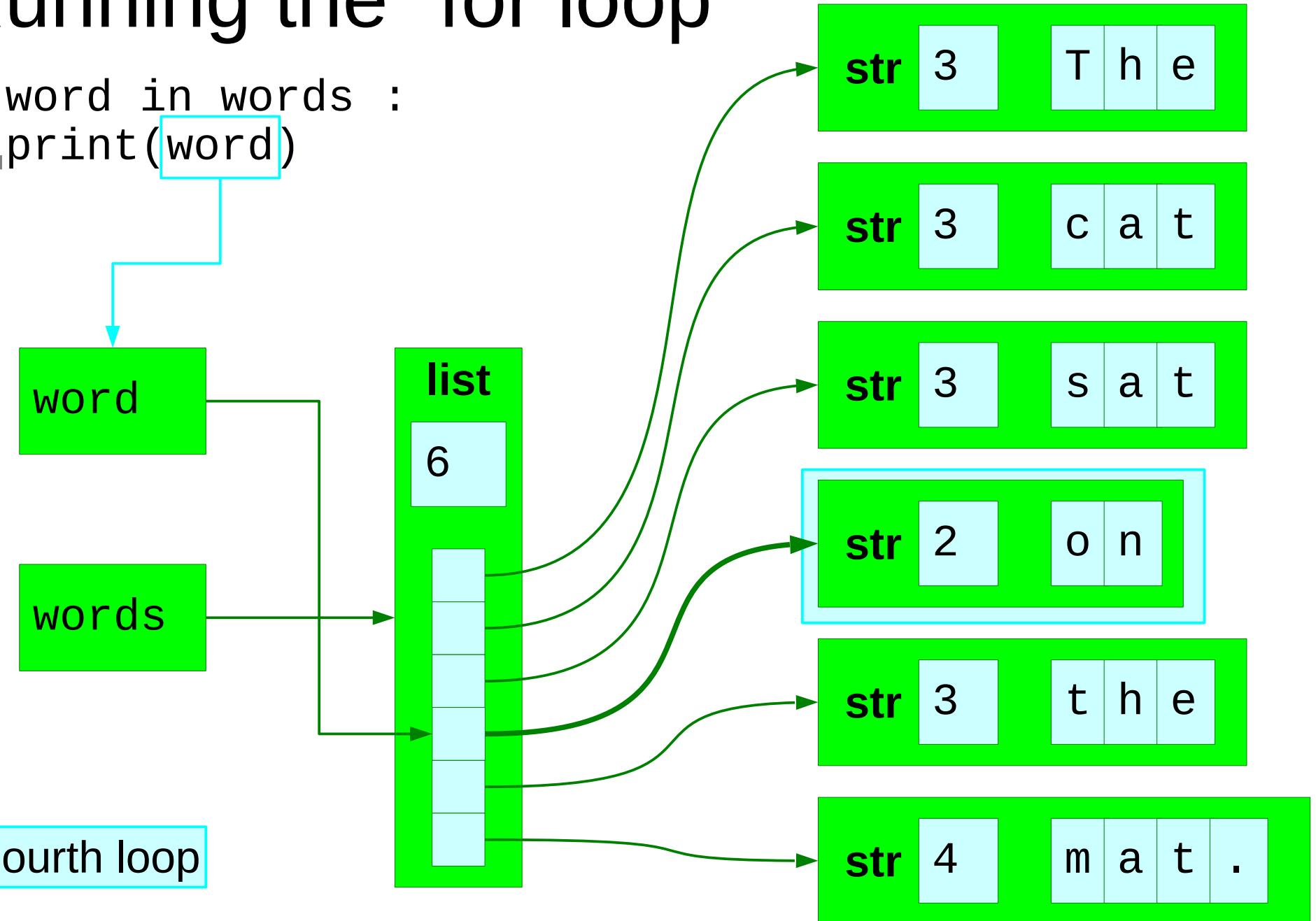
Running the “for loop”

```
for word in words :  
    print(word)
```



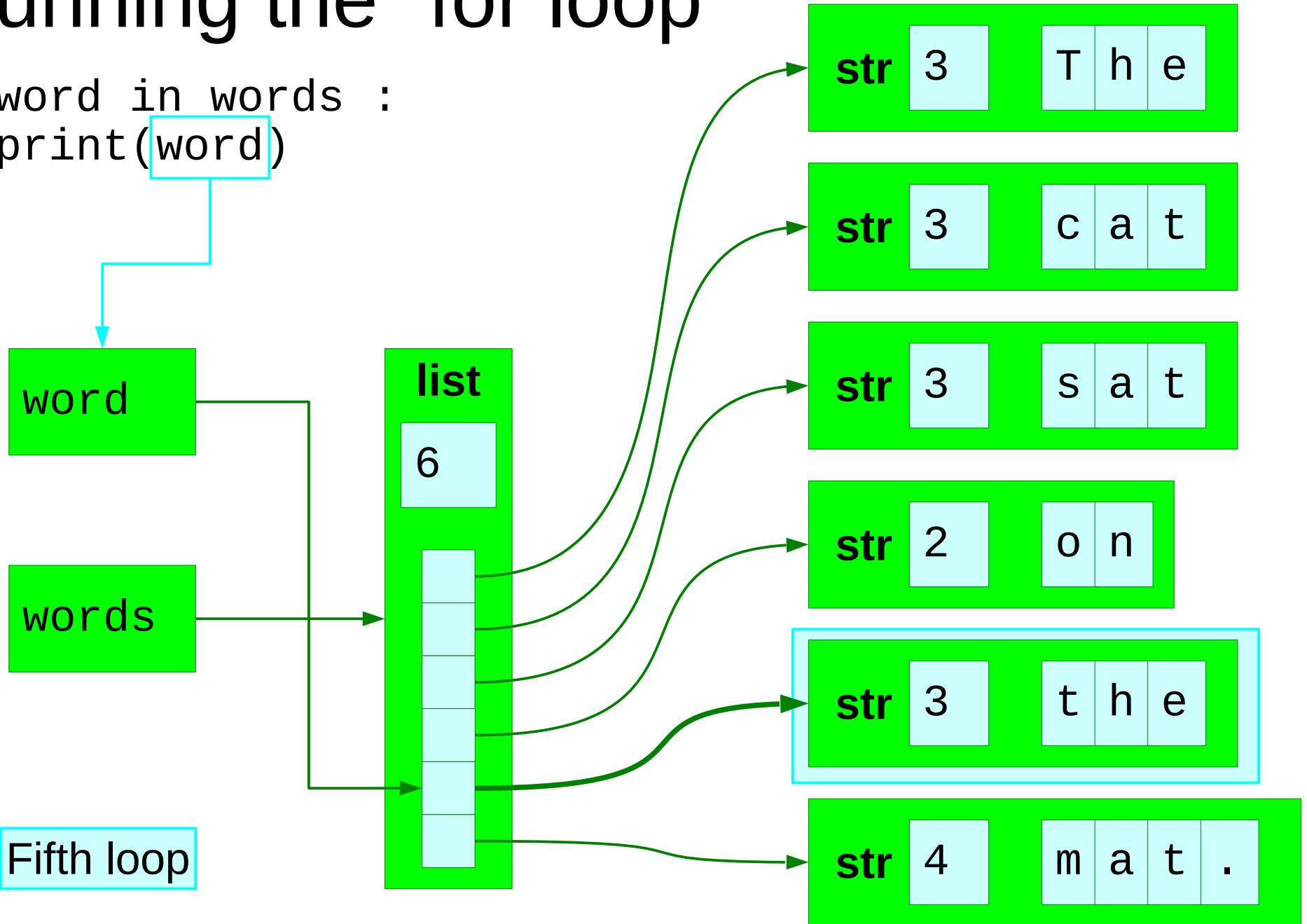
Running the “for loop”

```
for word in words :  
    print(word)
```



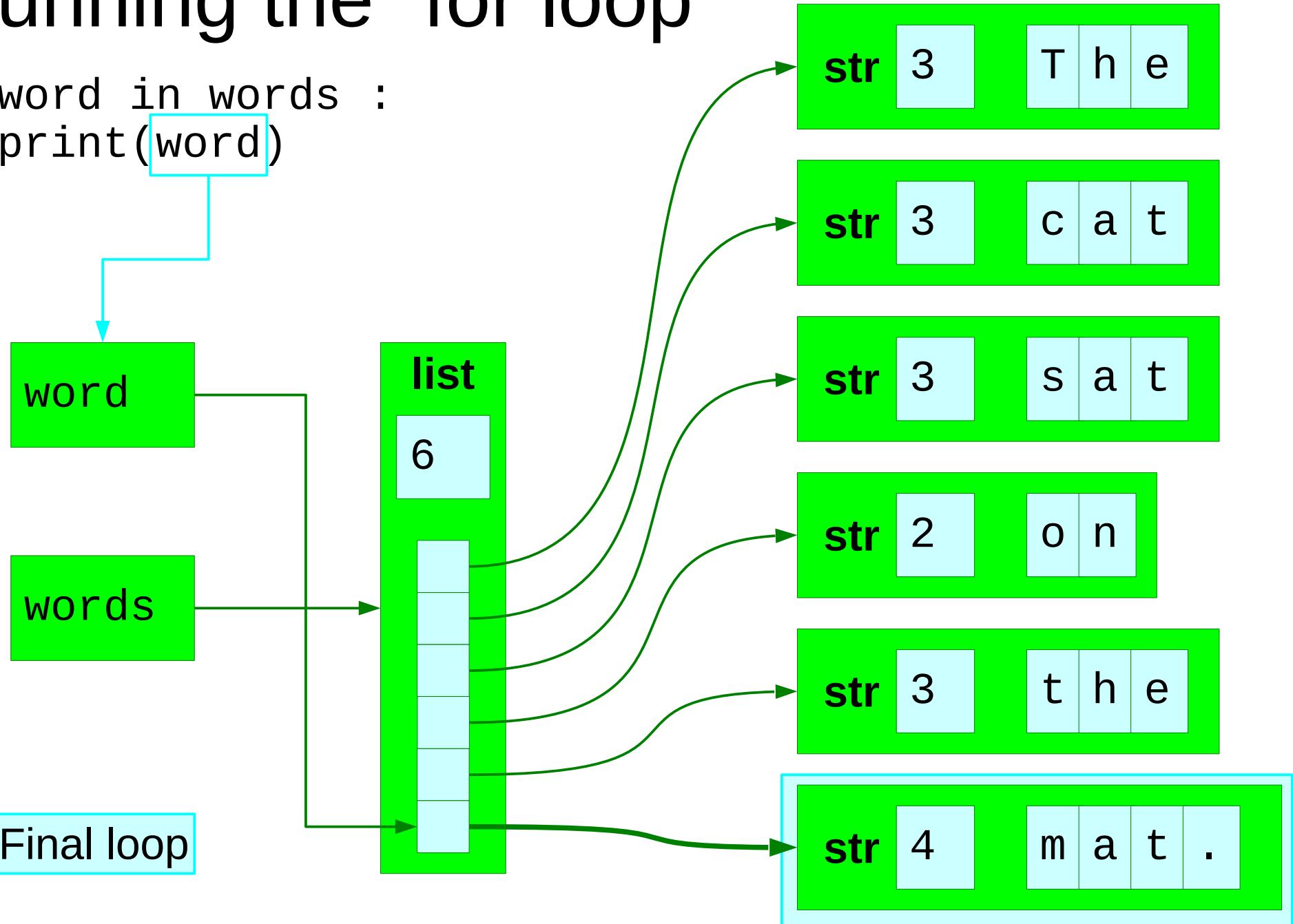
Running the “for loop”

```
for word in words :  
    print(word)
```



Running the “for loop”

```
for word in words :  
    print(word)
```



The “for loop” for printing

```
words = ['The', 'cat', 'sat', 'on', 'the', 'mat.']

for word in words :
    print(word)
```

for1.py

The “for loop” for adding

```
numbers = [45, 76, -23, 90, 15]
```

```
total = 0
```

Set up before the loop

```
for number in numbers :
```

```
    total += number
```

Processing in the loop

```
print(total)
```

for2.py

Results after the loop

The “for loop” for creating a new list

```
numbers = [4, 7, -2, 9, 1]
```

```
squares = [ ]
```

Set up before the loop

```
for number in numbers :
```

```
    squares.append(number**2)
```

Processing in the loop

```
print(squares)
```

Results after the loop

for3.py

The loop variable persists!

```
numbers = [4, 7, -2, 9, 1]
```

```
squares = [ ]
```

```
for number in numbers :
```

```
    squares.append(number**2)
```

```
print(number)
```

Loop variable only
meant for use in loop!

But it persists!

“for loop hygiene”

```
numbers = [4, 7, -2, 9, 1]
```

```
squares = [ ]
```

```
for number in numbers :
```

```
    squares.append(number**2)
```

```
del number
```

Delete it after use

Progress

Testing items in lists

3 in [1, 2, 3, 4] → True

for loops

```
total = 0
for number in [1, 2, 3, 4]:
    total += number
del number
```

loop variables

```
for number in [1, 2, 3, 4]:
    total += number
del number
```

Exercise 13

What does this print?

```
numbers = [0, 1, 2, 3, 4, 5]  
  
total = 0  
total_so_far = []  
  
for number in numbers:  
    total += number  
    total_so_far.append(total)  
  
print(total_so_far)
```



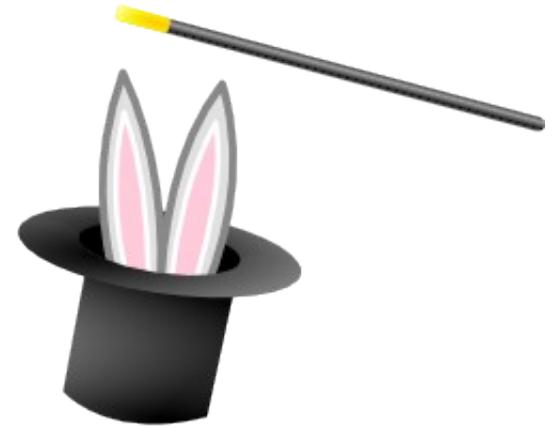
5 minutes

“Sort-of-lists”

Python “magic”:

Treat it like a list and it will
behave like a *useful* list

What can “it” be?



Strings as lists

Recall:

```
list('Hello') → ['H', 'e', 'l', 'l', 'o']
```

```
for letter in 'Hello' :  
    print(letter)
```

Gets turned
into a list.

```
H  
e  
l  
l  
o
```

for4.py

Creating lists of numbers

Built in to Python:

```
range(start, limit)
```

```
for number in range(3, 8):  
    print(number)
```

3
4
5
6
7

8 not included

ranges of numbers

Not actually lists:

```
>>> range(0,5)
```

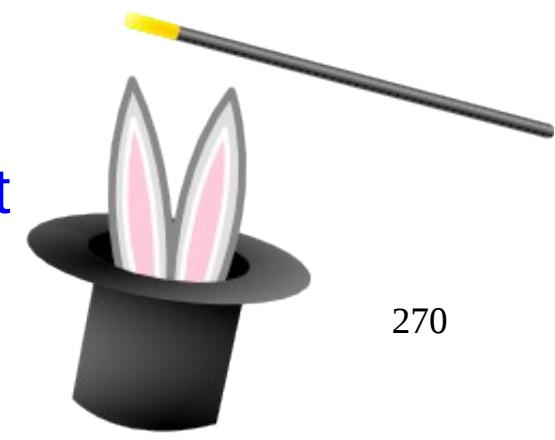
```
range(0,5)
```

But close enough:

```
>>> list(range(0,5))
```

```
[0, 1, 2, 3, 4]
```

Treat it like a list and
it will behave like a list



Why not just a list?

Most common use:

```
for number in range(0, 10000):
```

...

Inefficient to make a huge list just for this

“**iterator**” : anything that can be treated like a list

`list(iterator)` → Explicit list

Ranges of numbers again

via `list()`

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

Start at 0

`range(3, 10)` → [3, 4, 5, 6, 7, 8, 9]

`range(3, 10, 2)` → [3, 5, 7, 9] Every n^{th} number

`range(10, 3, -2)` → [10, 8, 6, 4] Negative steps

Indices of lists

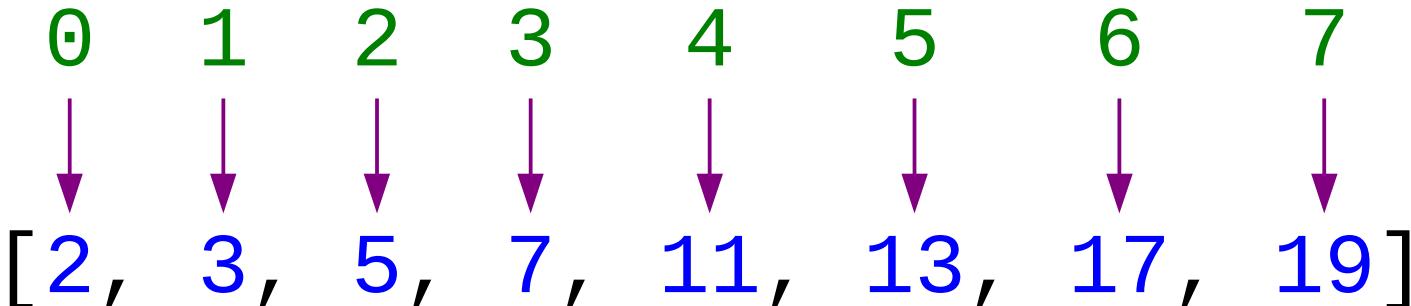
```
>>> primes = [ 2, 3, 5, 7, 11, 13, 17, 19]
```

```
>>> len(primes)
```

8

```
>>> list(range(8))
```

```
[0, 1, 2, 3, 4, 5, 6, 7] ← valid indices
```



Direct value or via the index?

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

```
for prime in primes:  
    print(prime)
```

Simpler

```
for index in range(len(primes)):  
    print(primes[index])
```

Equivalent

Working with two lists: “dot product”

```
list1 = [ 0.3,      0.0,      0.4]
```

▪ × × ×

```
list2 = [ 0.2,      0.5,      0.6]
```

$$0.06 + 0.0 + 0.24 \longrightarrow 0.3$$

Working with two lists: indices

0 1 2

A diagram showing three light blue boxes containing the numbers 0, 1, and 2 respectively. A horizontal arrow points from the right side of the number 2 towards the code below.

```
list1 = [0.3, 0.0, 0.4]  
list2 = [0.2, 0.5, 0.6]
```

```
sum = 0.0
```

```
for index in range(len(list1)):
```

```
    sum += list1[index]*list2[index]
```

```
print(sum)
```

indices

Dealing with
values from
both lists at
the same time.

A little more about iterators — 1

```
>>> greek = ['alpha', 'beta', 'gamma', 'delta']
```

```
>>> greek_i = iter(greek)
```

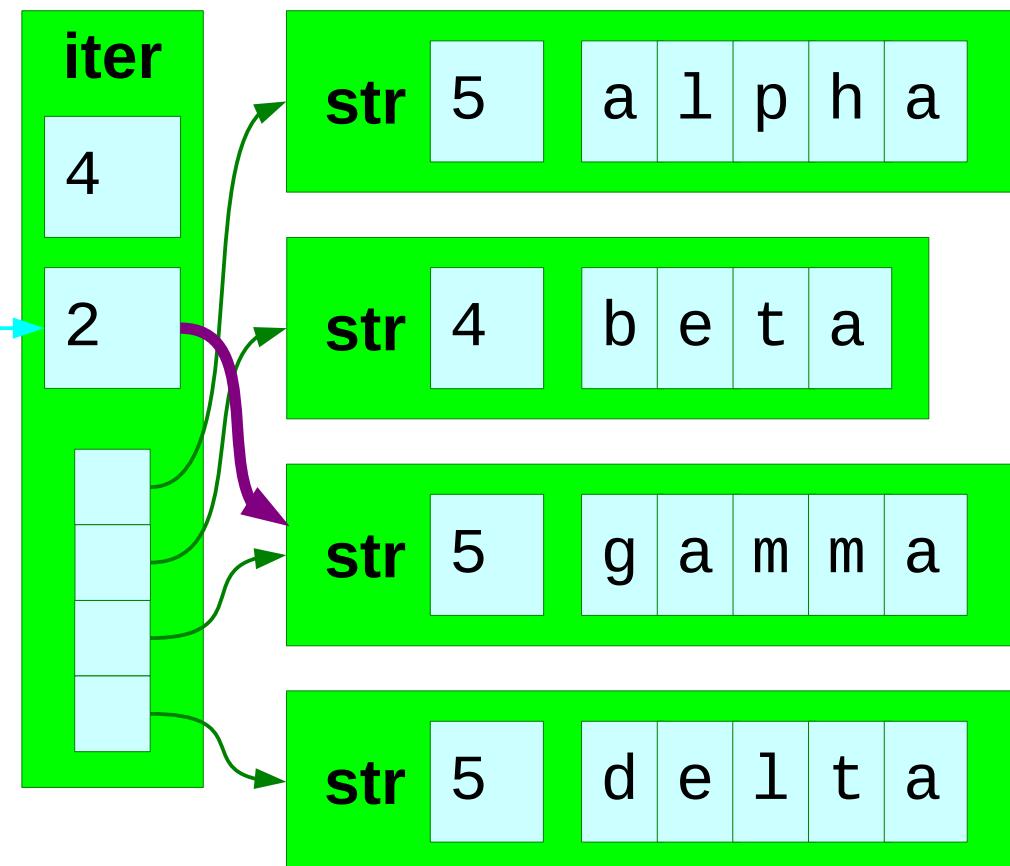
```
>>> next(greek_i)
```

'alpha'

Offset

```
>>> next(greek_i)
```

'beta'



A little more about iterators — 2

```
>>> next(greek_i)
```

```
'gamma'
```

```
>>> next(greek_i)
```

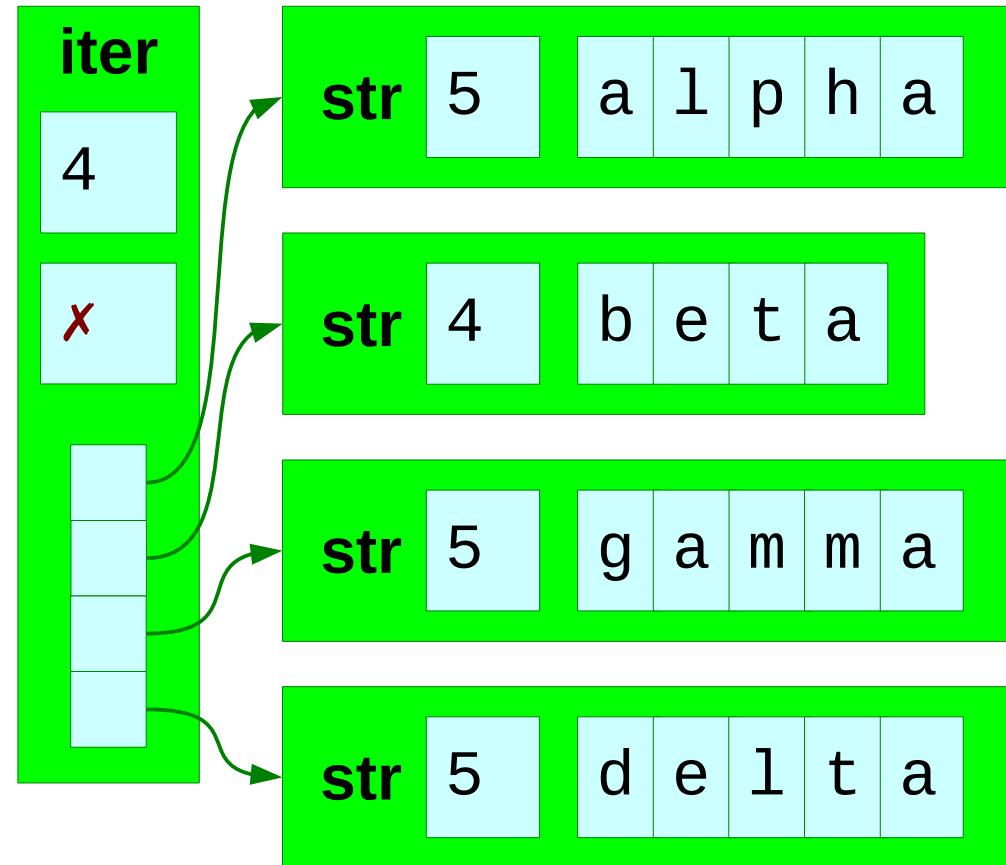
```
'delta'
```

```
>>> next(greek_i)
```

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

StopIteration



Progress

Non-lists as lists

```
for letter in 'Hello':  
    ...
```

range()

```
range(limit)  
range(start, limit)  
range(start, limit, step)  
range(3, 7) → [3, 4, 5, 6]
```

“Iterators”

```
greek_i = iter(greek)  
next(greek_i)
```

Indices of lists

```
for index in range(len(things)):
```

Parallel lists

Exercise 14

Complete `exercise14.py`

```
list1 = [ 0.3,      0.0,      0.4]
```

```
list2 = [ 0.2,      0.5,      0.6]
```

$$\begin{array}{ccc} & \downarrow & \downarrow & \downarrow \\ 0.1 & -0.5 & -0.2 \end{array}$$

Difference

$$\begin{array}{c} \downarrow \\ 0.01 + 0.25 + 0.04 \end{array} \xrightarrow{\text{Add}} 0.3$$

Square

Add



5 minutes

List “slices”

```
>>> primes = [2, 3, 5, 7, 11, 13, 17, 19, 23, 29]
```

```
>>> primes
```

```
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29] ← The list
```

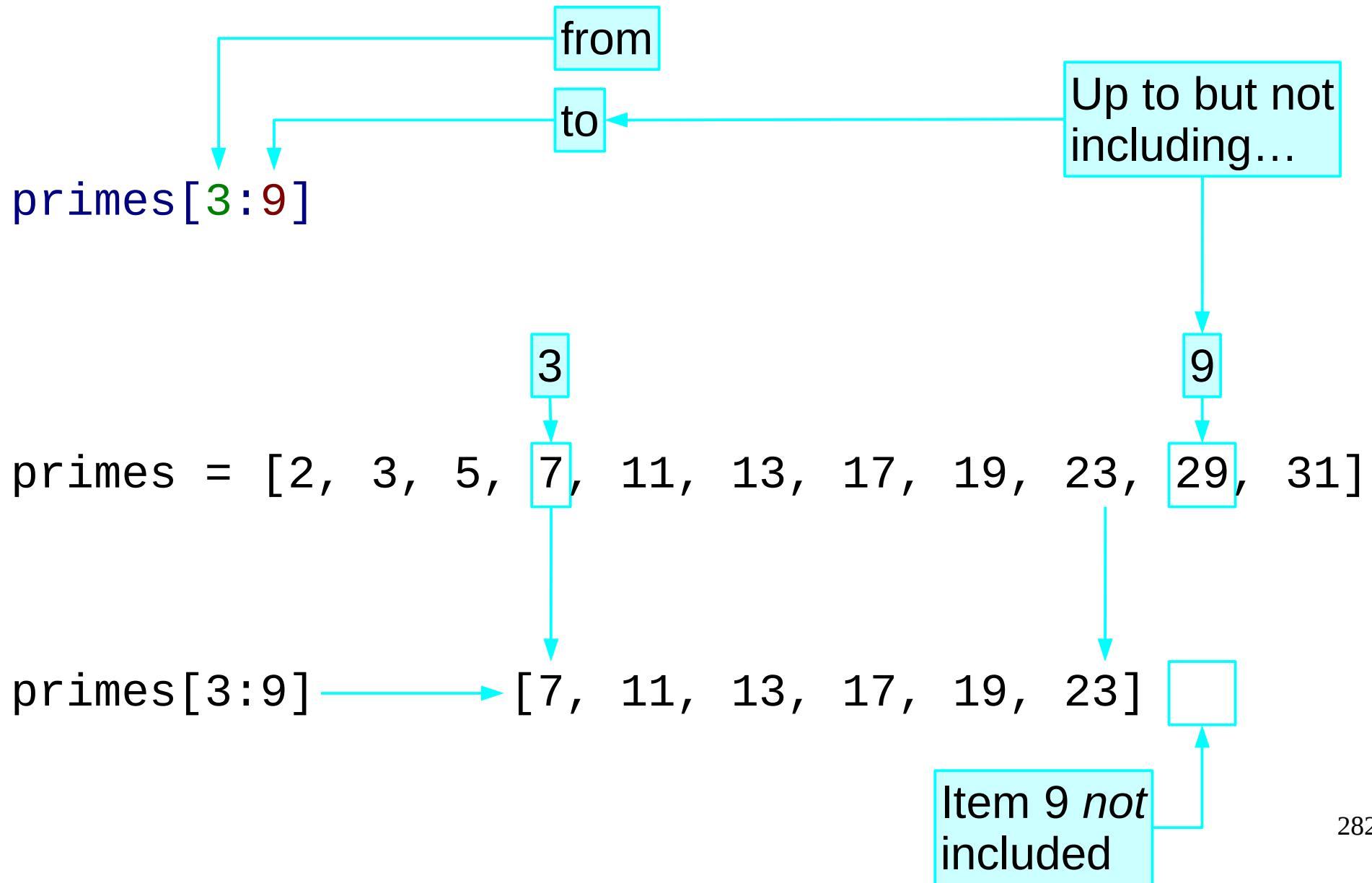
```
>>> primes[3]
```

```
7 ← An item
```

```
>>> primes[3:9]
```

```
[7, 11, 13, 17, 19, 23] ← Part of the list
```

Slices — 1



Slices — 2

primes [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31]

primes[3:9] [7, 11, 13, 17, 19, 23]

primes[:9] [2, 3, 5, 7, 11, 13, 17, 19, 23]

primes[3:] [7, 11, 13, 17, 19, 23, 29, 31]

primes[:] [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31]

Slices — 3

```
primes [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31]
```

```
primes[3:9] [7, 11, 13, 17, 19, 23]
```

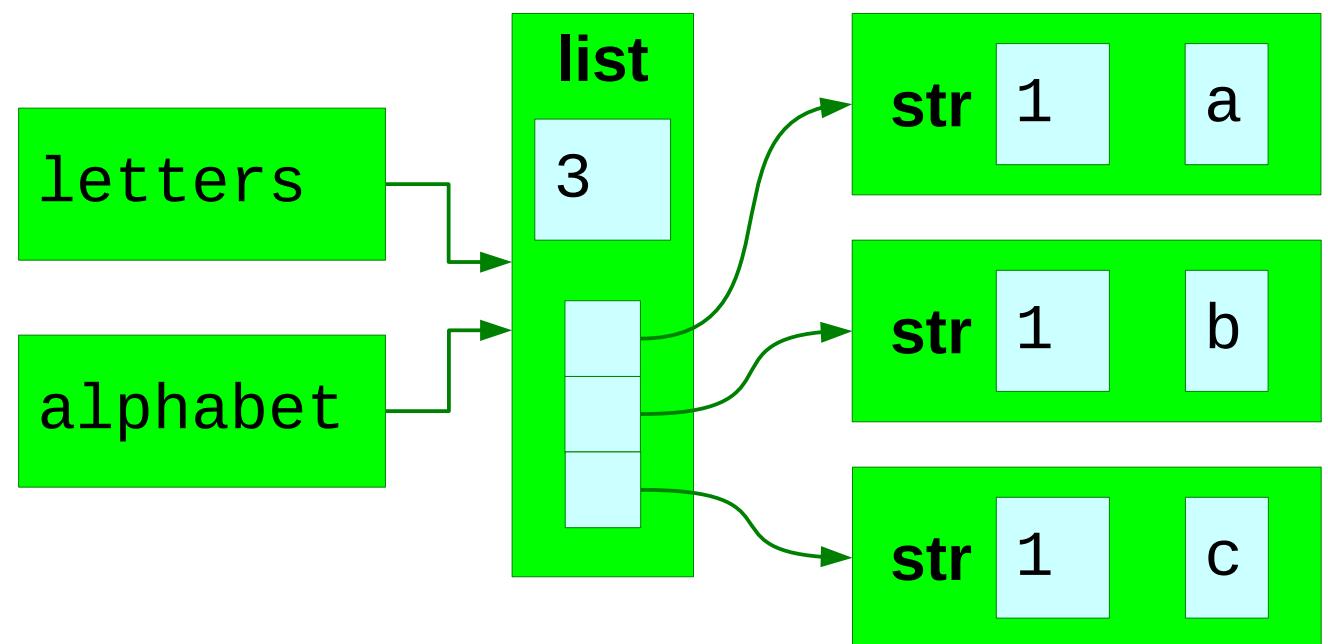
```
primes[3:9:2] [7, 13, 19 ]
```

```
primes[3:9:3] [7, 17 ]
```

Copies and slices — 1

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

```
>>> alphabet = letters
```

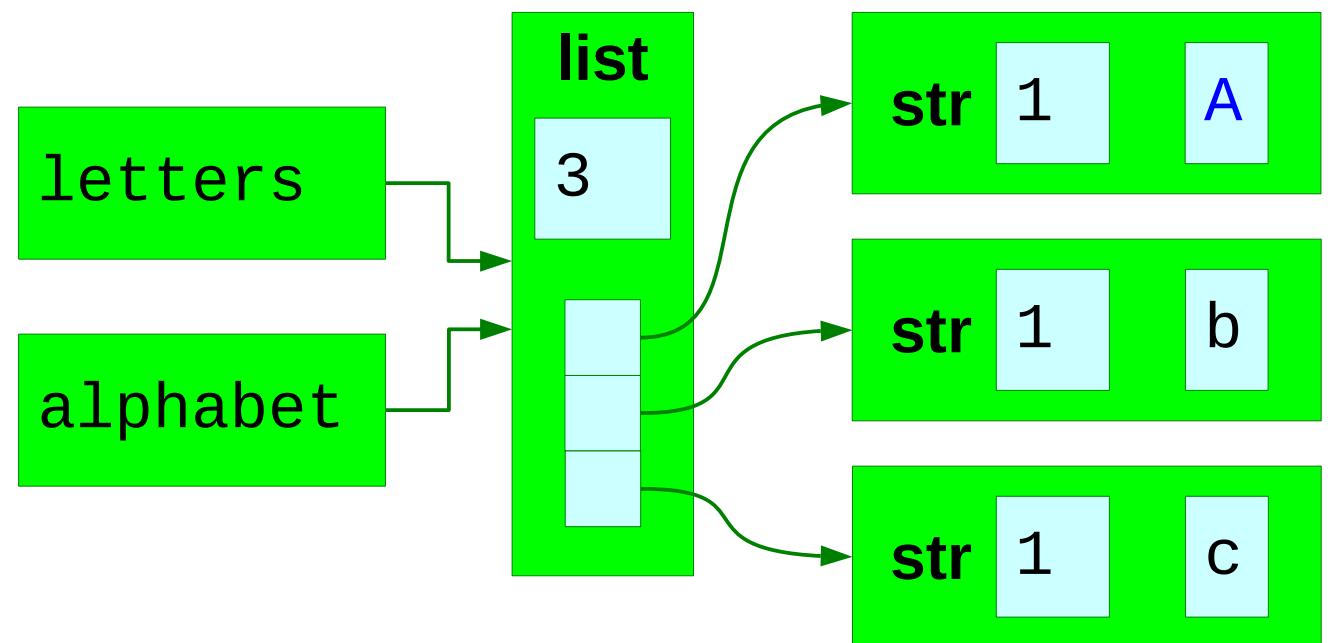


Copies and slices — 2

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

```
>>> print(alphabet)
```

```
[ 'A', 'b', 'c' ]
```

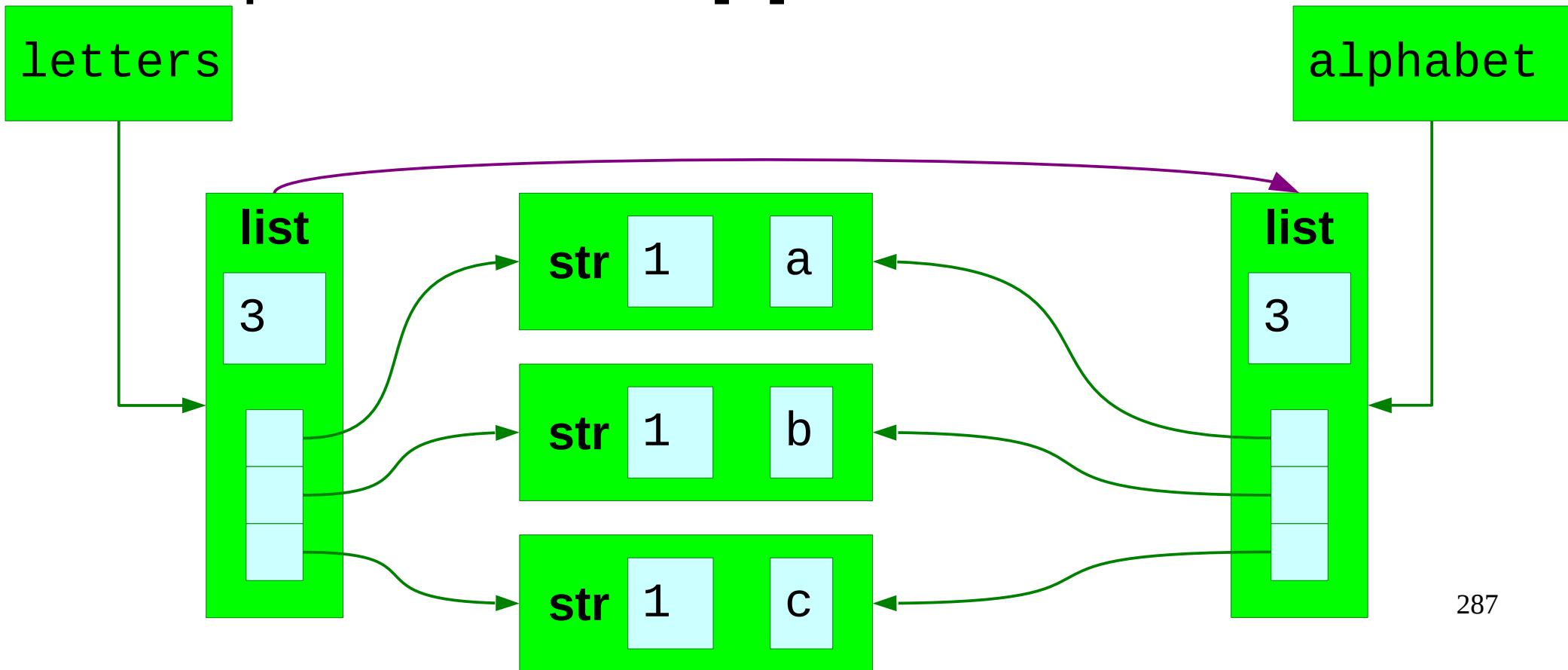


Copies and slices — 3

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

Slices are copies.

```
>>> alphabet = letters[:]
```



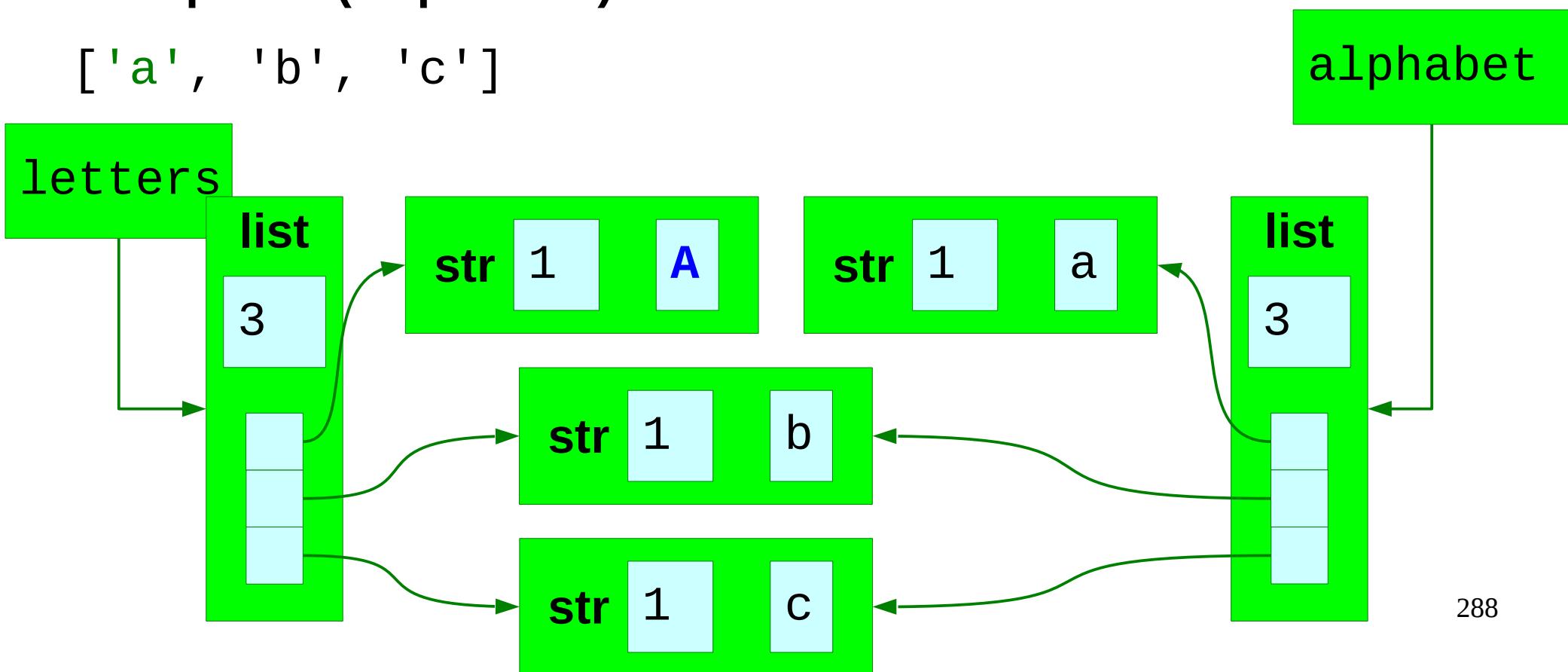
Copies and slices — 4

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

Slices are copies.

```
>>> print(alphabet)
```

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



Progress

Slices

End-limit excluded

Slices are copies

`items[from:to]`

`items[from:to:stride]`

`items[:to]`

`items[:to:stride]`

`items[from:]`

`items[from::stride]`

`items[:]`

`items[::stride]`

Exercise 15

Predict what this Python will do.
Then run it.
Were you right?

`exercise15.py`

```
foo = [4, 6, 2, 7, 3, 1, 9, 4, 2, 7, 4, 6, 0, 2]  
  
bar = foo[3:12:3]  
  
bar[2] += foo[4]  
  
foo[0] = bar[1]  
  
print(bar)
```

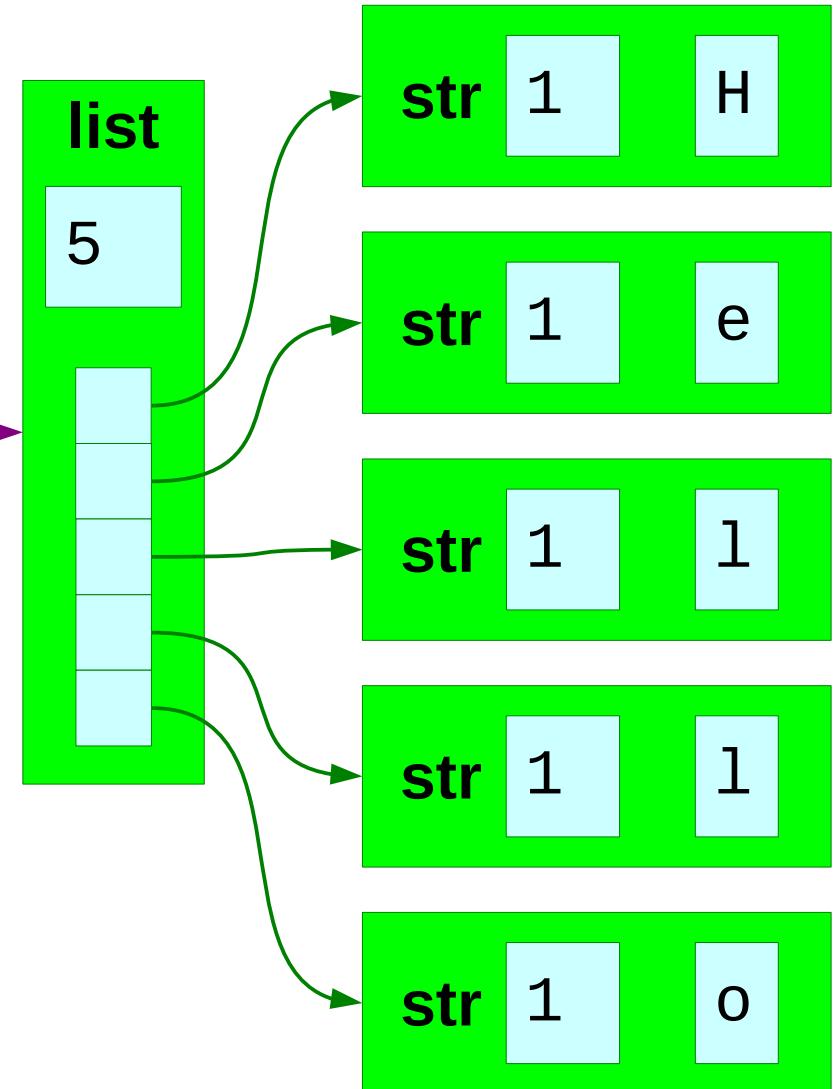
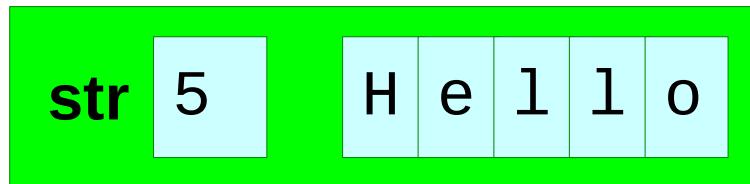


3 minutes

Diversion: Lists and strings

Text: “a **string** of characters”

`list()`



Indexing into strings

```
>>> 'Hello, world!'[0]
```

```
'H'
```

Simple indexing

```
>>> 'Hello, world!'[:4]
```

```
'Hell'
```

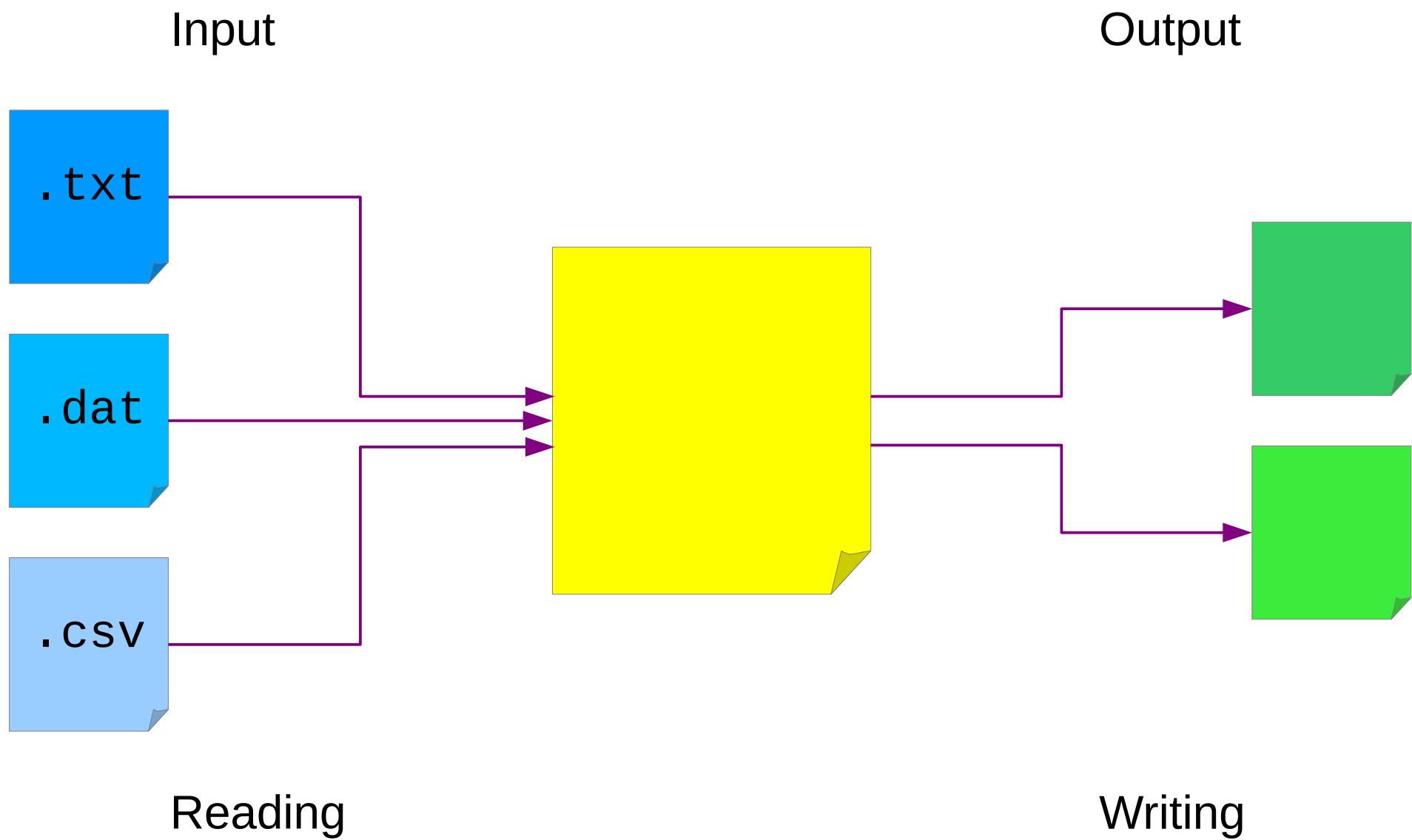
Slicing

Strings are immutable

```
>>> 'Hello, world!'[0] = 'c'
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support
           item assignment
```

Files



Reading a text file

File name

'treasure.txt'

string

“opening” the file

File object

book

file

reading from the file

File contents

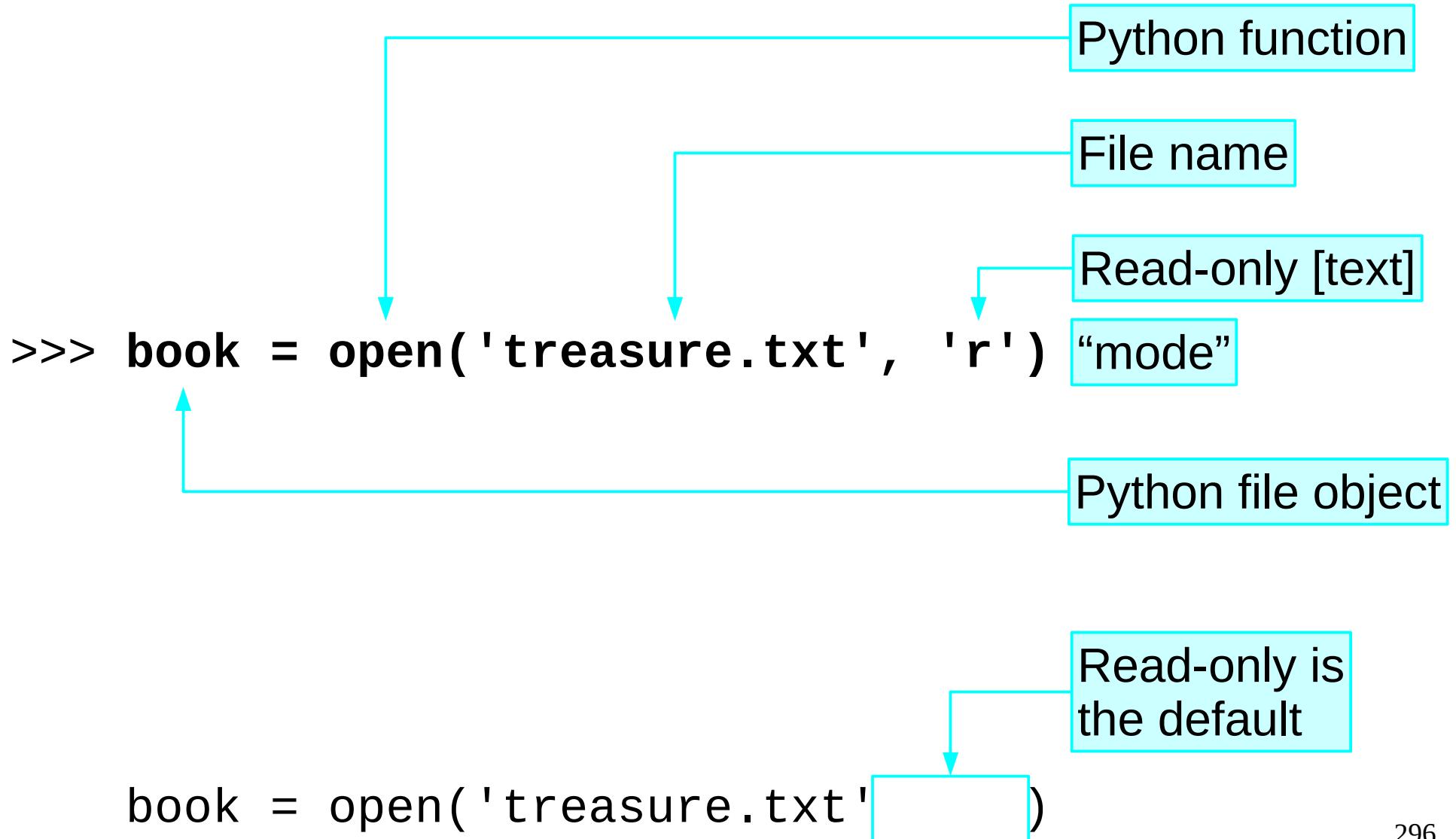
'TREASURE ISLAND'

string

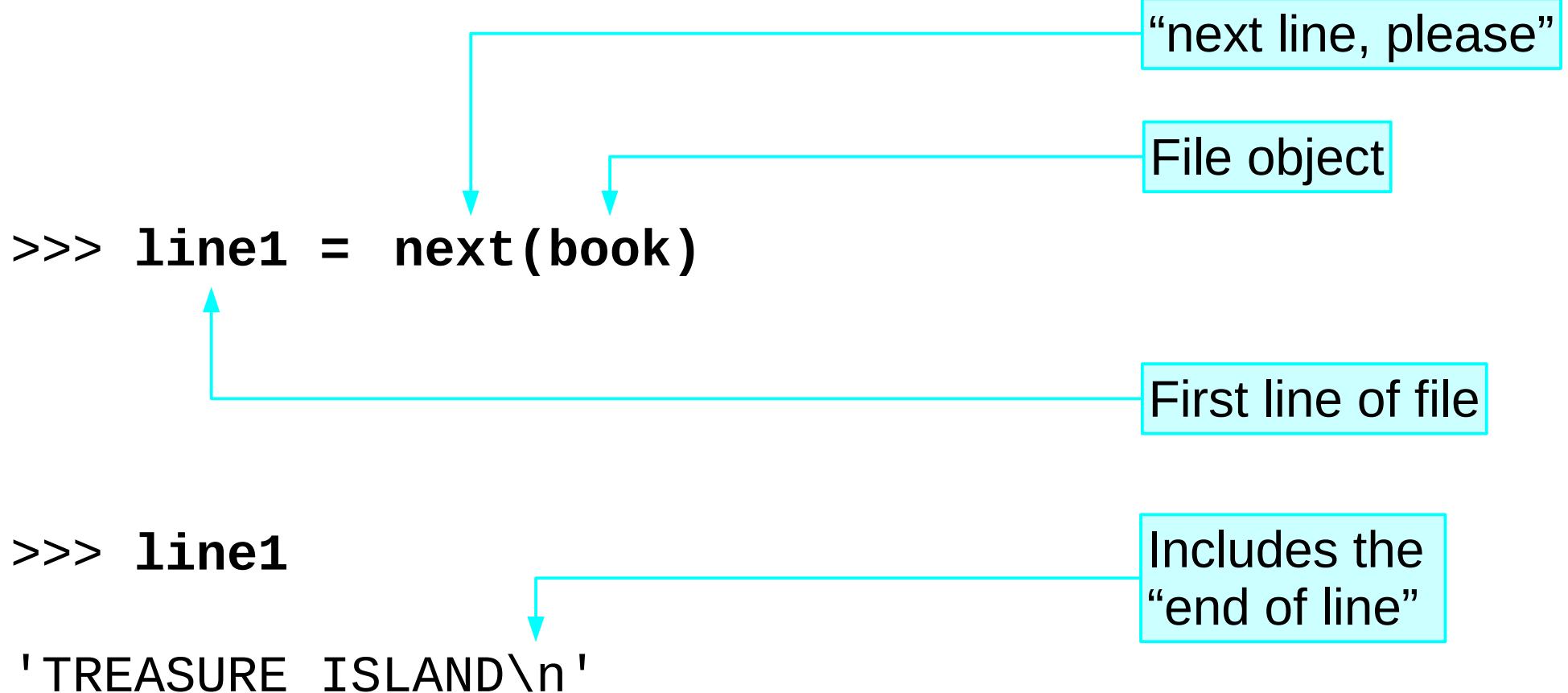
“closing” the file

Finished with the file

Opening a text file



Reading from a file object



File object are iterable

```
>>> line2 = next(book)
```



Second line of file

```
>>> line2
```

```
'\n'
```

A blank line

```
>>> line3 = next(book)
```

```
>>> line3
```

```
'PART ONE\n'
```

Third line of file

Closing the file

```
>>> book.close()
```

Method built in to file object

Frees the file for other
programs to write to it.

The file object — 1

```
>>> book
```

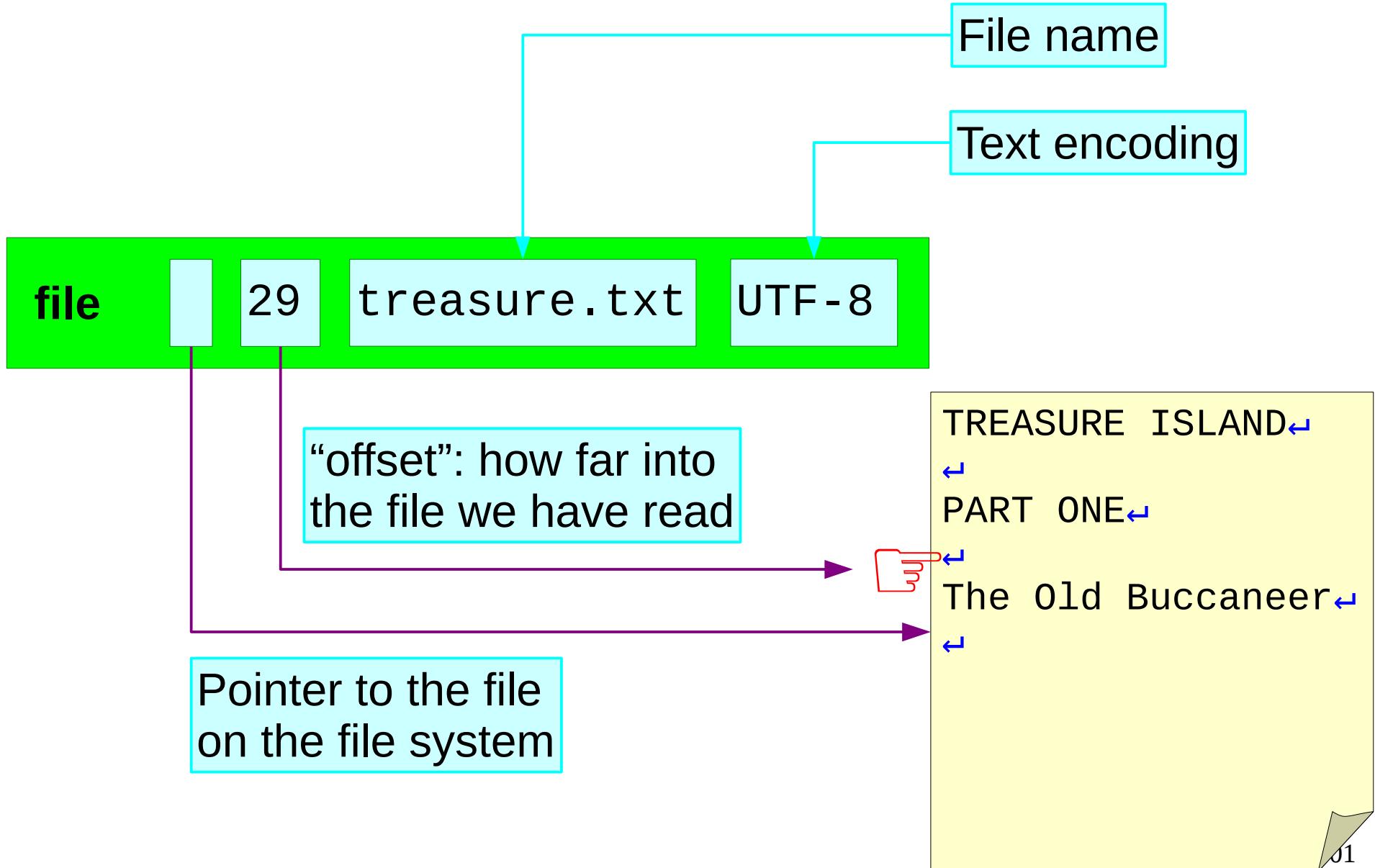
```
<_io.TextIOWrapper  
  name='treasure.txt'  
  encoding='UTF-8'>
```

Text Input/Output

File name

Character encoding:
how to represent
letters as numbers.

The file object — 2



Reading through a file

Treat it like a list and
it will behave like a list



`list(file_object)` → List of lines in the file

```
>>> book = open('treasure.txt', 'r')
```

```
>>> lines = list(book)
```

```
>>> print(lines)
```

Reading a file moves the offset



```
>>> book = open('treasure.txt', 'r')
```

```
>>> lines_a = list(book)
```

```
>>> print(lines_a)
```

... ← Huge output

```
>>> lines_b = list(book)
```

```
>>> print(lines_b)
```

[] ← Empty list

Reading a file moves the offset

```
>>> book = open('treasure.txt', 'r')
```

File object starts with offset at **start**.

```
>>> lines_a = list(book)
```

Operation reads entire file from offset.

```
>>> print(lines_a)
```

Offset changed to **end** of file.

...

```
>>> lines_b = list(book)
```

Operation reads entire file from offset.

```
>>> print(lines_b)
```

So there's nothing left to read.

[]

Resetting the offset

```
>>> book = open('treasure.txt', 'r')
```

```
>>> lines_a = list(book)
```

```
>>> print(lines_a)
```

...

```
>>> book.seek(0) ← Set the offset explicitly
```

```
>>> lines_b = list(book)
```

```
>>> print(lines_b)
```

Typical way to process a file

```
book = open('treasure.txt', 'r')
```

```
for line in book : ← Treat it like a list...
```

```
...
```

Example: lines in a file

```
book = open('treasure.txt', 'r')
```

```
n_lines = 0
```

Line count

```
for line in book :
```

Read each line of the file

```
n_lines += 1
```

Increment the count

```
print(n_lines)
```

Print out the count

```
book.close()
```

Example: characters in a file

```
book = open('treasure.txt', 'r')
```

```
n_chars = 0
```

```
for line in book :  
    n_chars += len(line)  
print(n_chars)
```

Number of characters on the line

Increase the count by that many

```
book.close()
```

Progress

Opening file to read

```
book = open(filename, 'r')
```

Reading file

```
for line in book:  
    ...
```

Closing file

```
book.close()
```

File offset

```
book.seek(0)
```

Exercise 16

Complete a script to count
lines, words and characters
of a file.

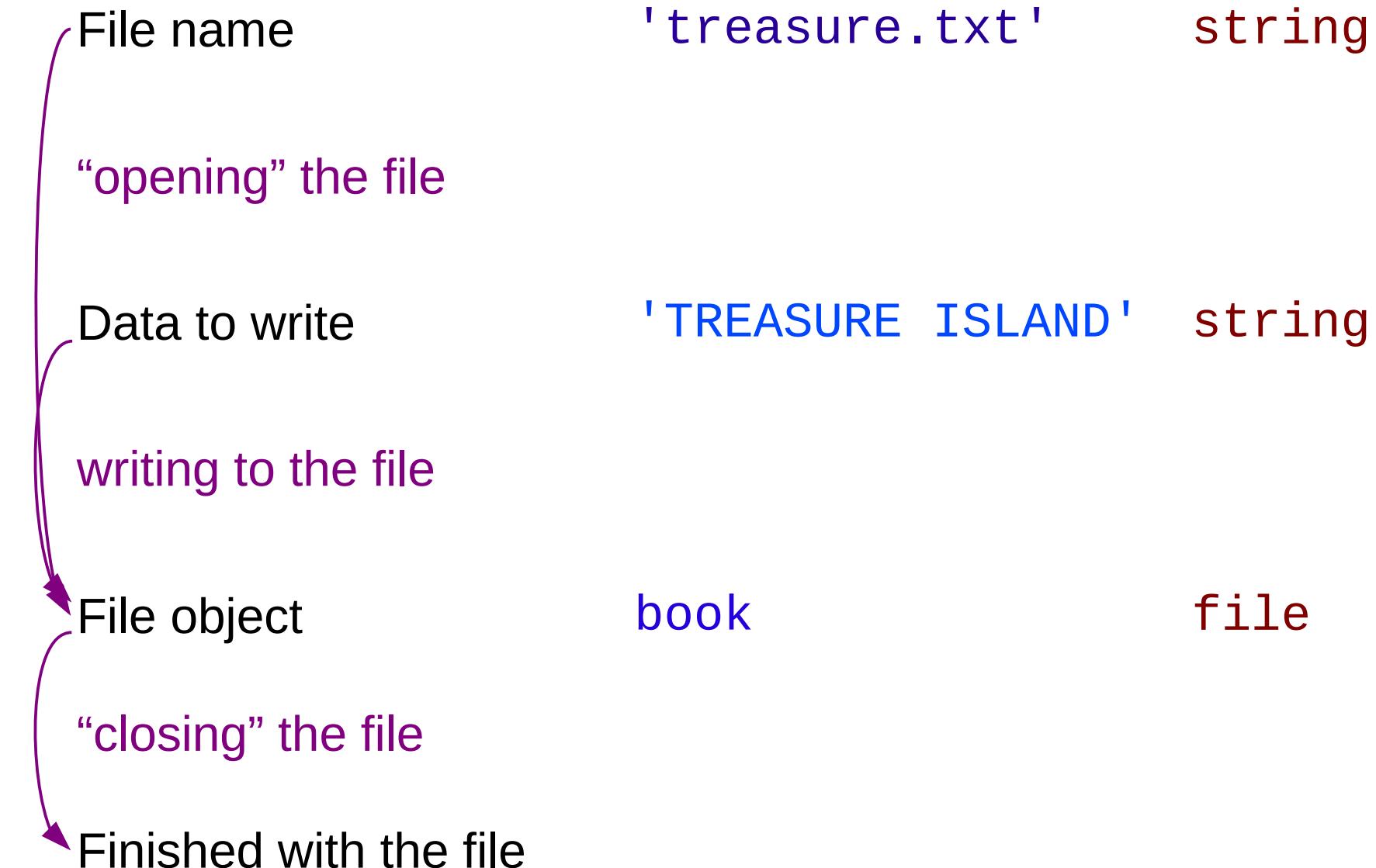
`exercise16.py`



5 minutes

310

Writing files



Opening a text file for writing

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

Open for
writing [text]

'w'

This will *truncate*
an existing file



Opening a text file for *appending*

```
>>> output = open('output.txt', 'a')
```

Open for
appending [text]

Writing to a file object — 1

```
>>> output.write(line1)
```

6

```
>>> len(line1)
```

File object

Method built in to file object

Data being written

Number of characters actually written

Writing to a file object — 2

```
>>> output.write('alpha\n')
```

6

Typical use: whole line.

Includes “end of line”

```
>>> output.write('be')
```

2

Doesn't have
to be whole lines

```
>>> output.write('ta\n')
```

3

```
>>> output.write('gamma\ndelta\n')
```

12

Can be multiple lines

Closing the file object

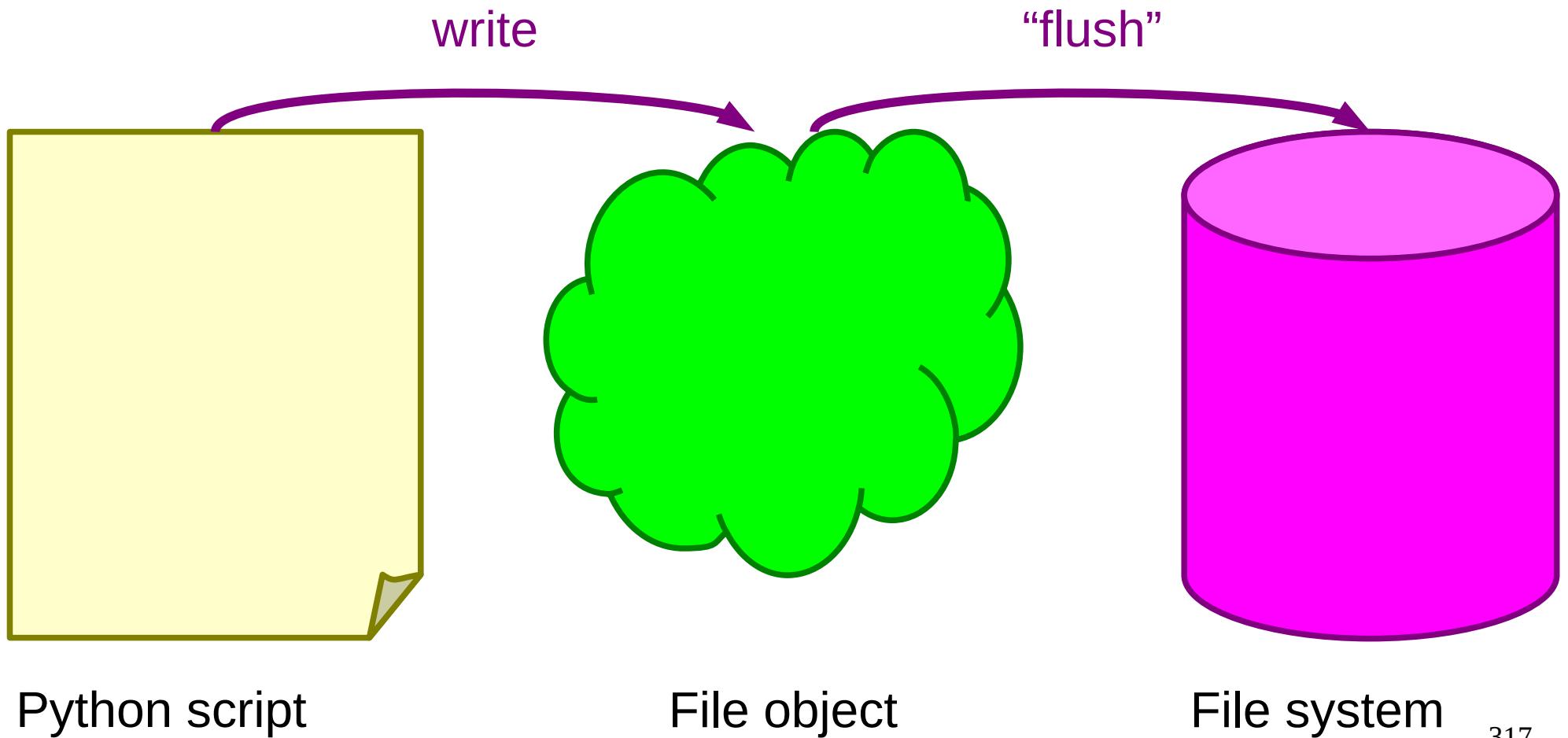
```
>>> output.close()
```

Vital for written files

Importance of closing



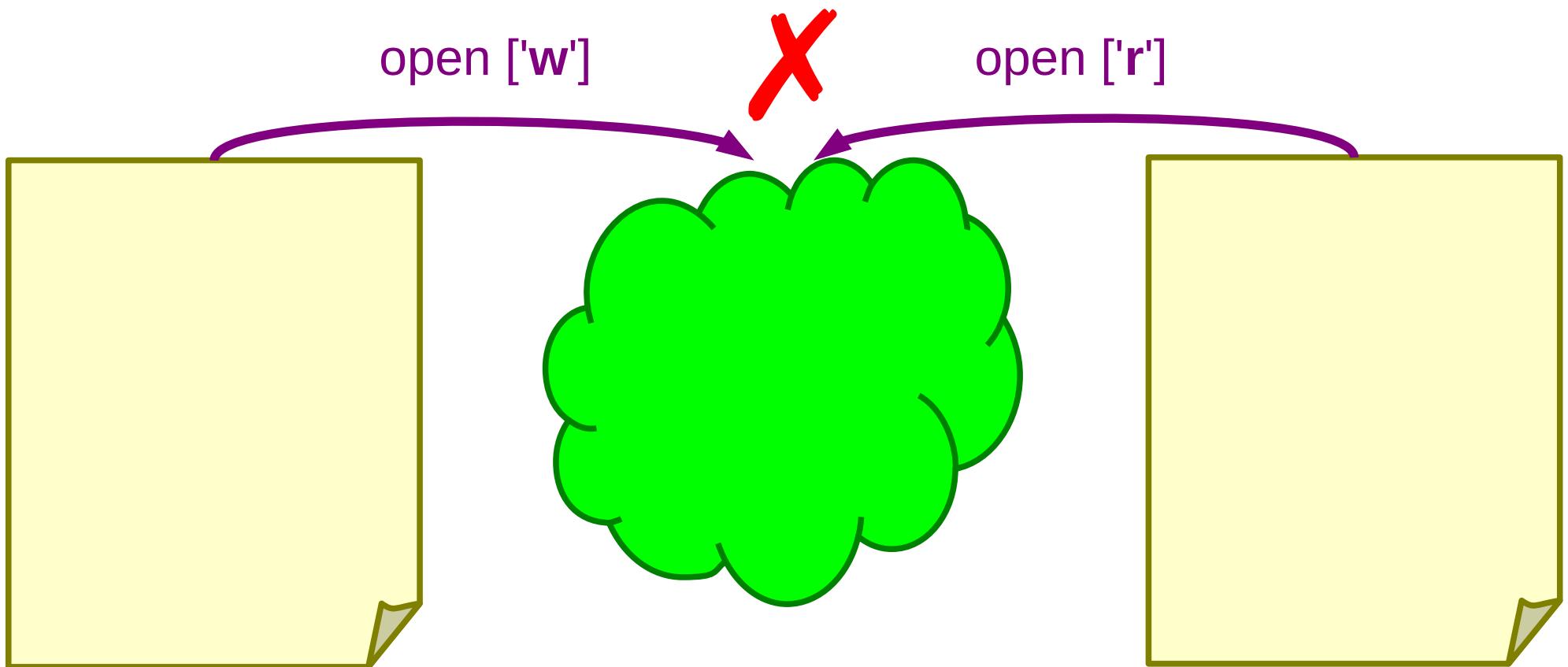
Data “flushed” to disc on closure.



Importance of closing *promptly*



Files locked for other access



(More a problem for Windows than Unix)

Writing non-text values

```
>>> output.write('Boo!\n') ← Writing text (str)
```

5

```
>>> output.write(42) ← Writing non-text (int)
```

Traceback (most recent call last):

 File "<stdin>", line 1, in <module>

 TypeError: must be str, not int

X

write() only accepts text

Writing non-text values

```
>>> output.write(str(42))
```

Convert to text: str()

2

```
>>> output.write('\n')
```

Explicit end-of-line

1

Text formatting (later in the course) provides a more elegant solution.

Progress

Opening files for writing

```
book = open(filename, 'w')
```

Writing text

```
book.write(text)
```

Writing data

```
book.write(str(data))
```

Explicit ends of lines

```
book.write('\n')
```

Closing the file is *important*

```
book.close()
```

Exercise 17

The script `exercise17.py` prints
a series of numbers ending in 1.

Change the script to write to a file called
`output.txt` instead of printing to the console.



5 minutes

Functions

$$y = f(x)$$



Functions we have met

`input(prompt)`

`bool(thing)`

`len(thing)`

`float(thing)`

`open(filename, mode)`

`int(thing)`

`print(line)`

`iter(list)`

`type(thing)`

`list(thing)`

`ord(char)`

`range(from, to, stride)`

`chr(number)`

`str(thing)`

Not that many!

“The Python Way”:
If it is appropriate to an object,
make it a method of that object.

Why write our own functions?

Easier to ...

... read

... write

... test

... fix

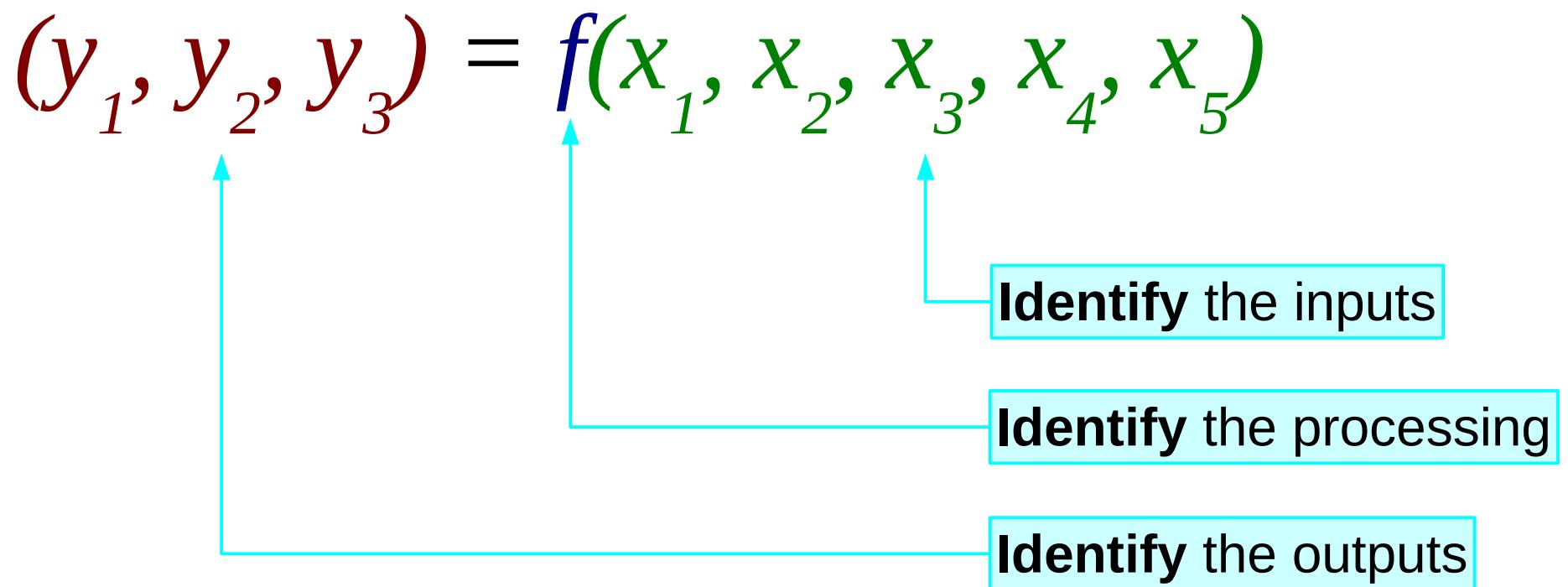
... improve

... add to

... develop

“Structured
programming”

Defining a function



A function to define: `total()`

Sum a list

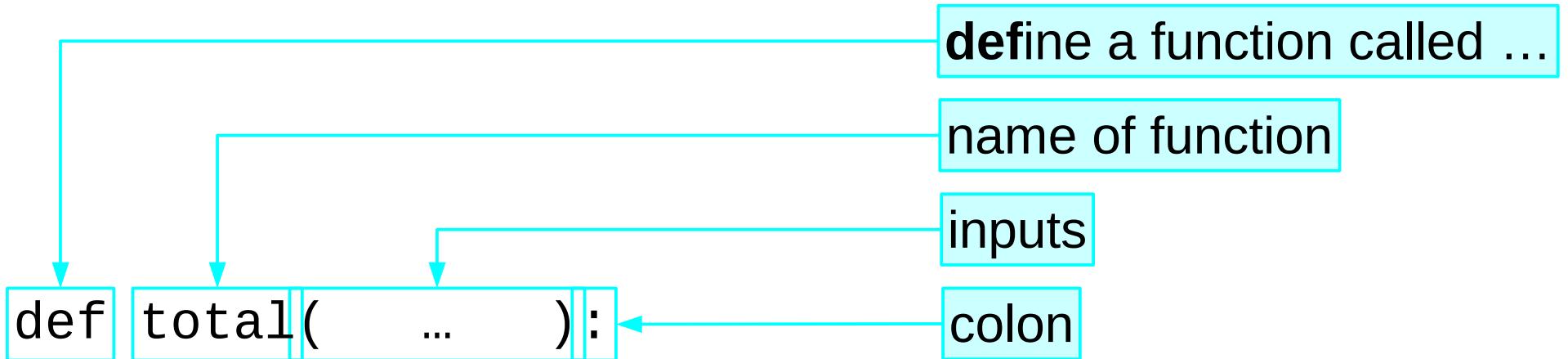
[1, 2, 3] → 6

[7, -4, 1, 6, 0] → 10

[] → 0

“Edge case”

Defining a Python function — 1



Defining a Python function — 2

```
def total(numbers):
```

name for the input

This name is
internal to
the function.

Defining a Python function — 3

```
def total(numbers):  
    Colon followed by indentation
```



Defining a Python function — 4

```
def total(numbers):
```

```
    sum_so_far = 0  
  
    for number in numbers:  
        sum_so_far += number
```

“Body” of function

Defining a Python function — 4

```
def total(numbers):  
    sum_so_far = 0  
    for number in numbers:  
        sum_so_far += number
```

These variables exist *only* within the function's body.

Defining a Python function — 5

```
def total(numbers):  
  
    sum_so_far = 0  
  
    for number in numbers:  
        sum_so_far += number
```

return

sum_so_far

This value
is returned

return this value

Defining a Python function — 6

And that's it!

```
def total(numbers):  
  
    sum_so_far = 0  
  
    for number in numbers:  
        sum_so_far += number  
  
    return sum_so_far
```


Unindented
after this

Defining a Python function — 7

And that's it!

All internal names *internal* → No need to avoid reusing names

All internal names cleaned up → No need for `del`

Using a Python function — 1

```
def total(numbers):  
  
    sum_so_far = 0  
  
    for number in numbers:  
        sum_so_far += number  
  
    return sum_so_far
```

```
print(total([1, 2, 3]))
```

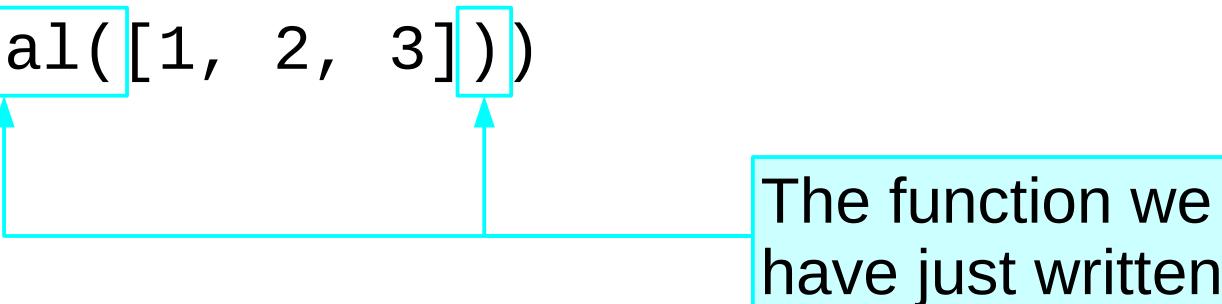


The list we
want to add up

Using a Python function — 2

```
def total(numbers):  
  
    sum_so_far = 0  
  
    for number in numbers:  
        sum_so_far += number  
  
    return sum_so_far
```

```
print(total([1, 2, 3]))
```



The function we
have just written

Using a Python function — 3

```
def total(numbers):  
  
    sum_so_far = 0  
  
    for number in numbers:  
        sum_so_far += number  
  
    return sum_so_far
```

```
print(total([1, 2, 3]))
```

Printing out
the answer

Using a Python function — 4

```
def total(numbers):  
    sum_so_far = 0  
  
    for number in numbers:  
        sum_so_far += number  
  
    return sum_so_far  
  
print(total([1, 2, 3]))
```

total1.py

nb: Unix prompt

\$ **python3 total1.py**

6

Using a Python function — 5

```
def total(numbers):  
  
    sum_so_far = 0  
  
    for number in numbers:  
        sum_so_far += number  
  
    return sum_so_far
```

```
print(total([1, 2, 3]))  
print(total([7, -4, 1, 6, 0]))  
print(total([]))
```

total2.py

```
$ python3 total2.py  
6  
10  
0
```

Use the function
multiple times

Functions' private names — 1

```
def total(numbers):  
  
    sum_so_far = 0  
  
    for number in numbers:  
        sum_so_far += number  
  
    return sum_so_far
```

Function definition

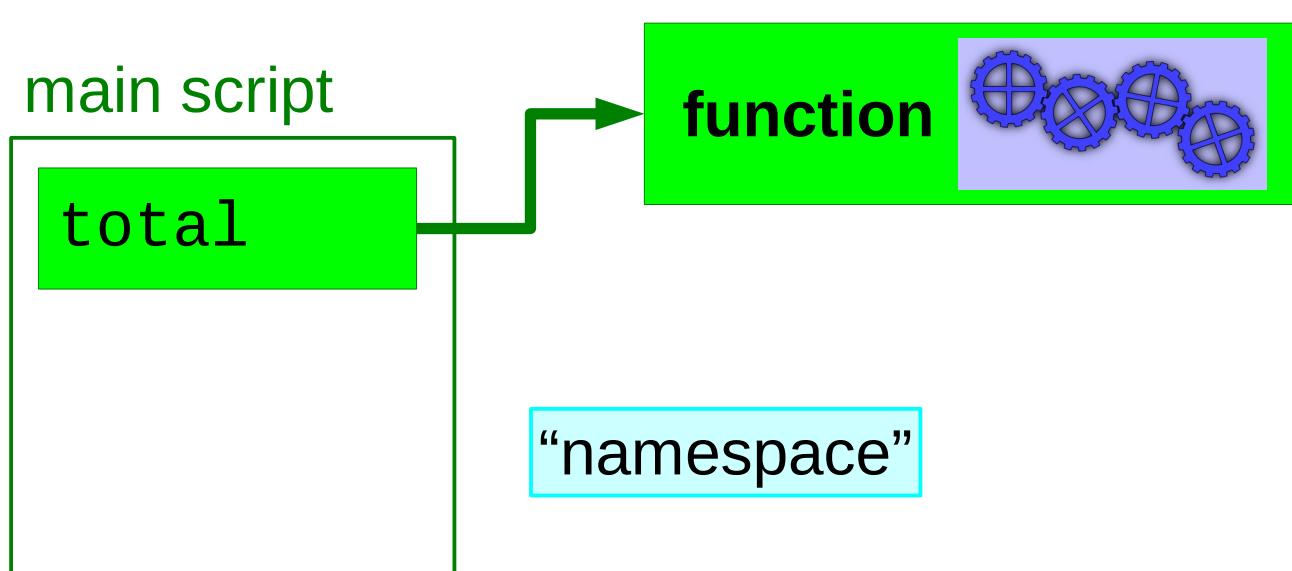
```
data = [1, 2, 3]  
data_sum = total(data)  
print(data_sum)
```

Main script

Functions' private names — 2

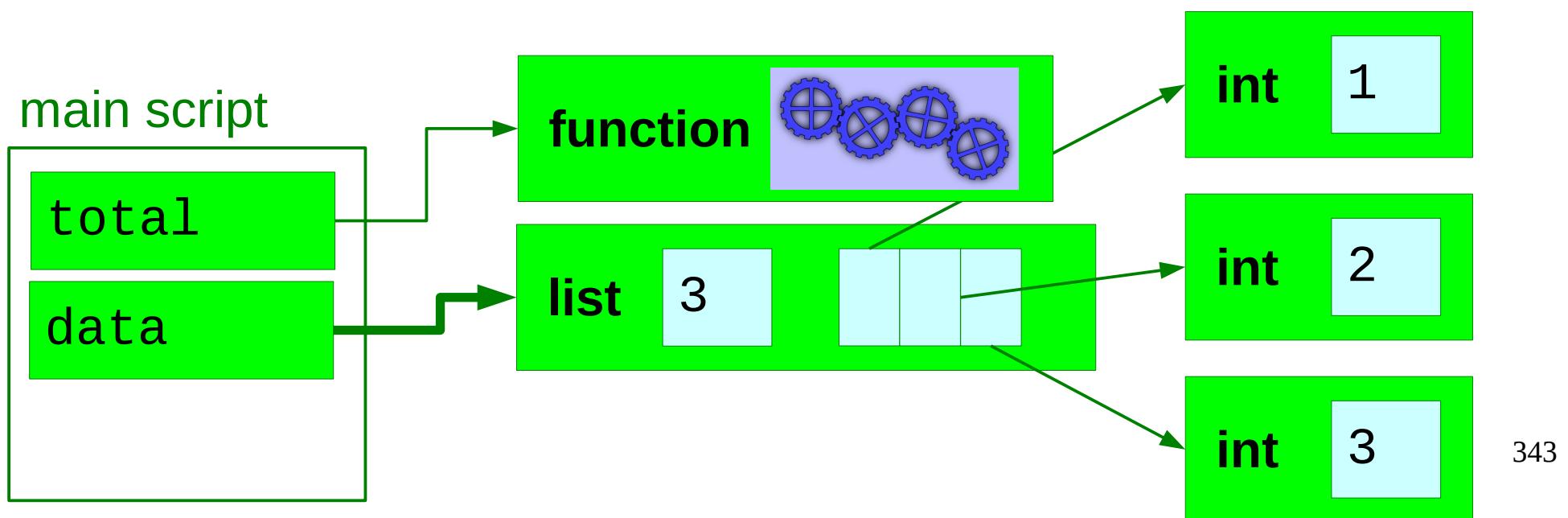
```
def total(numbers):
```

```
    ...
```



Functions' private names — 3

```
data = [1, 2, 3]
```

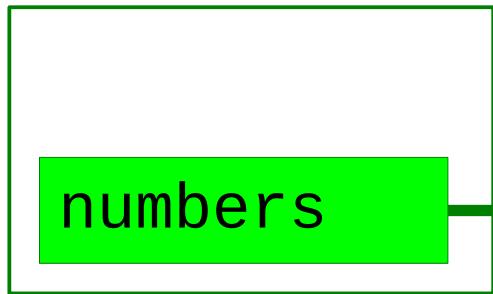


Functions' private names — 4

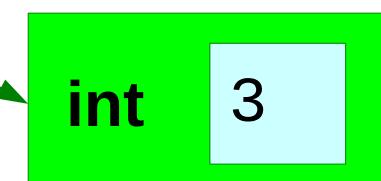
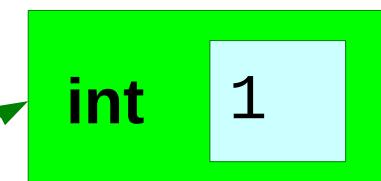
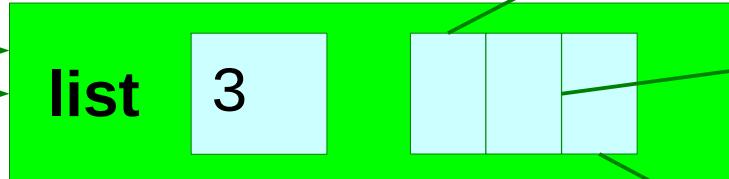
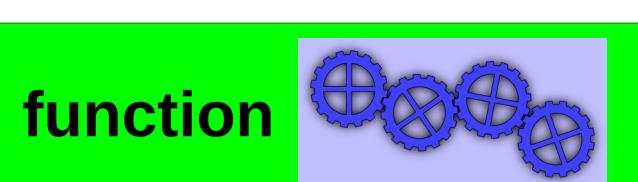
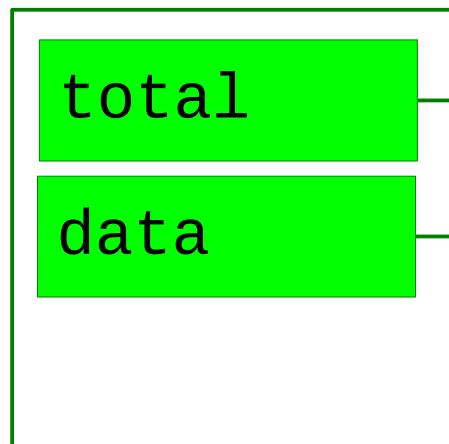
```
... = total(data)
```

```
def total(numbers):
```

total



main script

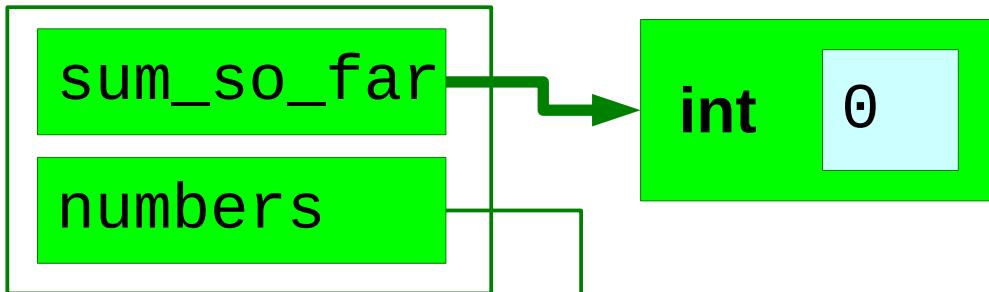


Functions' private names — 5

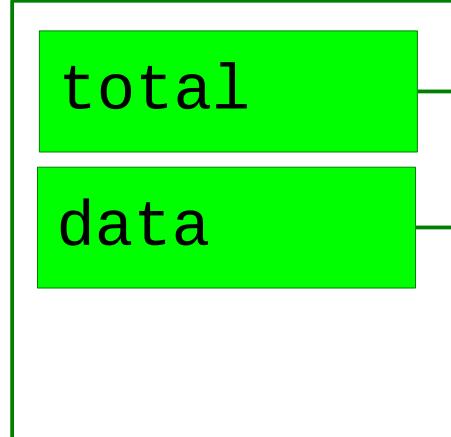
```
... = total(data)
```

```
sum_so_far = 0
```

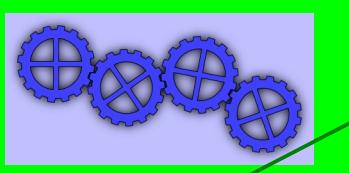
total



main script

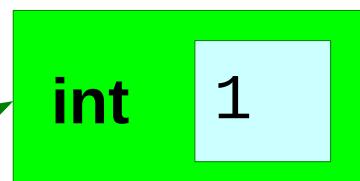


function



list

3

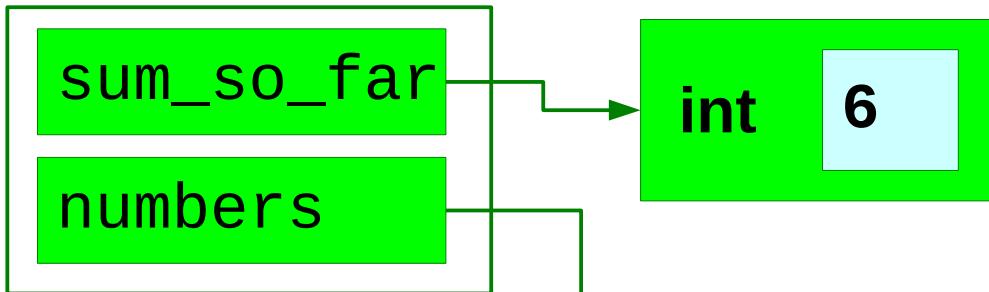


Functions' private names — 6

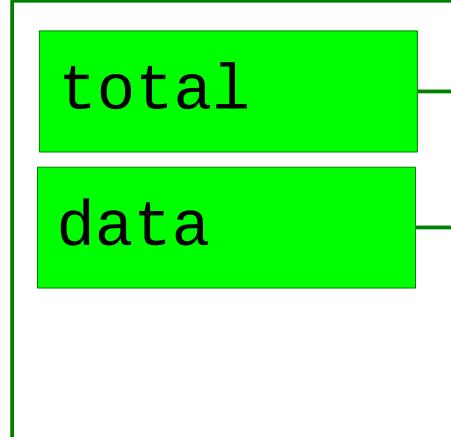
... = total(data)

return sum_so_far

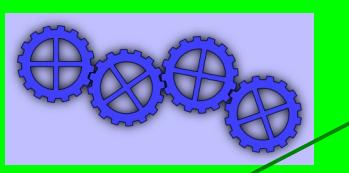
total



main script

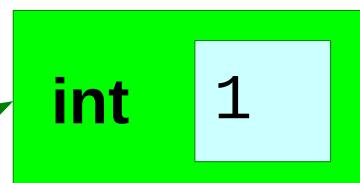


function



list

3

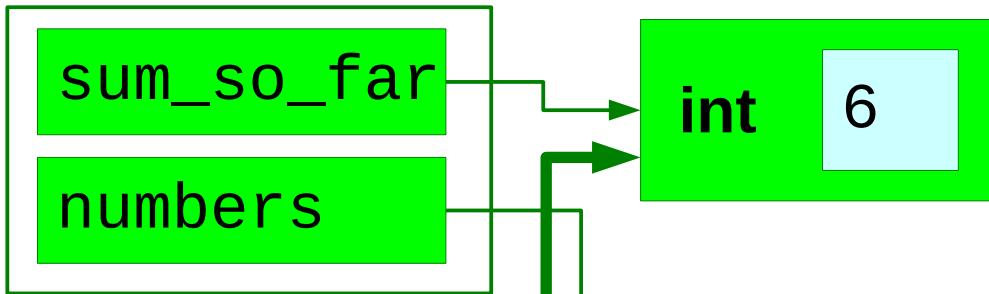


Functions' private names — 7

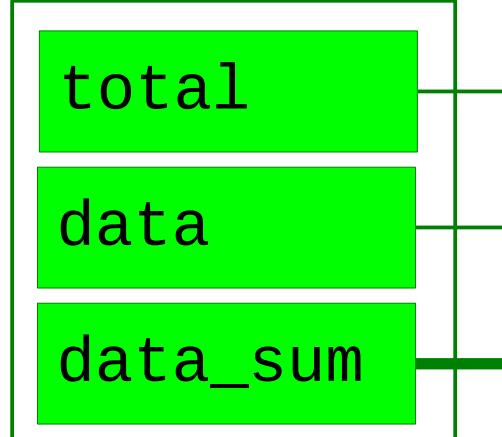
```
data_sum = total(data)
```

```
return sum_so_far
```

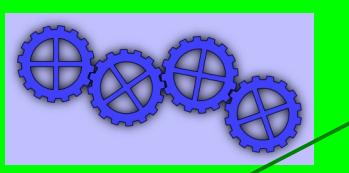
total



main script

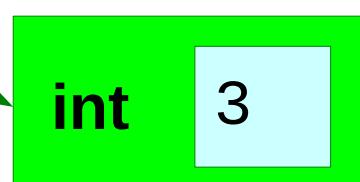
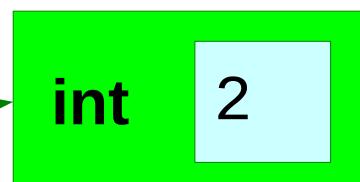
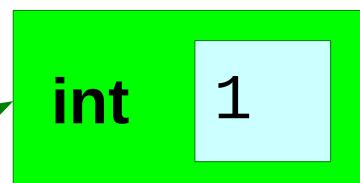


function



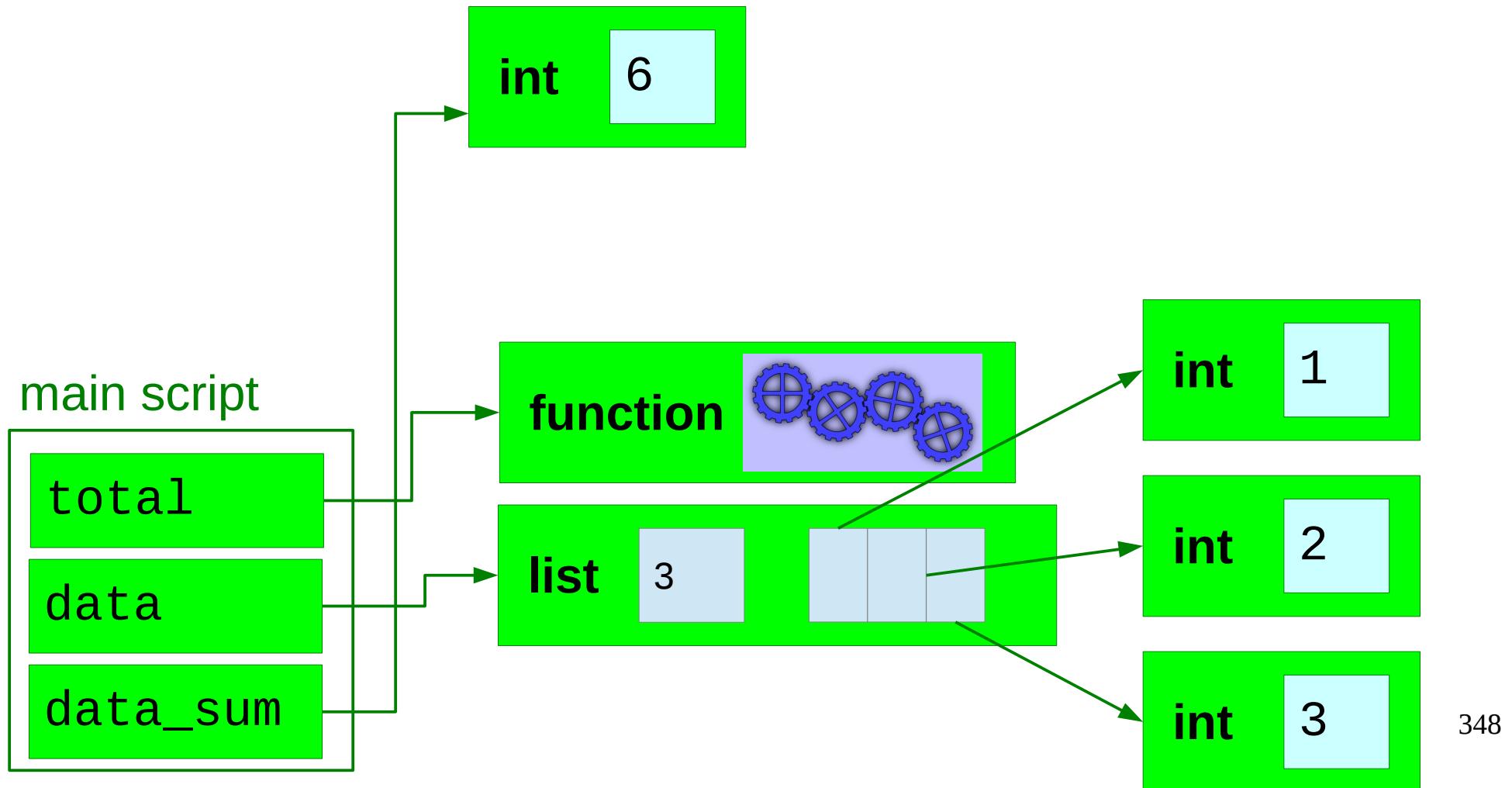
list

3



Functions' private names — 8

```
data_sum = total(data)
```



Progress

Functions

“Structured programming”

Defining a function

```
def function(input):  
    ...
```

Returning a value

```
return output
```

Private name spaces

Exercise 18

Edit the script `exercise18.py`.

It currently defines and uses a function `total()` which adds the elements of a list.

Change it to create a function `product()` which multiplies them.

Three examples are included in the script.

Running it should produce the following output:

```
$ python3 exercise18.py  
6  
0  
1
```



5 minutes

Reminder about indices

```
def total(numbers):
    sum_so_far = 0
    for number in numbers:
        sum_so_far += number
    return sum_so_far
```

total2.py

Equivalent

```
def total(numbers):
    sum_so_far = 0
    for index in range(len(numbers)):
        sum_so_far += numbers[index]
    return sum_so_far
```

total3.py

Example of multiple inputs

Want a function to add two lists of the same length term-by-term:

$$[1, 2, 3] \quad \& \quad [5, 7, 4] \longrightarrow [6, 9, 7]$$

$$[10, -5] \quad \& \quad [15, 14] \longrightarrow [25, 9]$$

$$[3, 7, 4, 1, 7] \quad \& \quad [8, 4, -6, 1, 0] \longrightarrow [11, 11, -2, 2, 7]$$



Two inputs

Functions with multiple inputs

```
def add_lists(a_list, b_list):  
    sum_list = []  
  
    for index in range(len(a_list)):  
        sum = a_list[index] + b_list[index]  
  
        sum_list.append(sum)  
  
    return sum_list
```

Multiple inputs are separated by commas



Functions with multiple inputs

```
def add_lists(a_list, b_list):  
    sum_list = []  
    for index in range(len(a_list)):  
        sum = a_list[index] + b_list[index]  
        sum_list.append(sum)  
    return sum_list
```

We have two lists...

...so we have to use indexing

Multiple outputs

Write a function to find minimum *and* maximum value in a list

[1, 2, 3] → 1 & 3

[10, -5] → -5 & 10

[3, 7, 4, 1, 7] → 1 & 7

A diagram illustrating the concept of multiple outputs. It shows three examples of lists and their corresponding minimum and maximum values. In the third example, blue arrows point from the numbers 1 and 7 in the list [3, 7, 4, 1, 7] to a light blue box containing the text "Two outputs".

Two outputs

Finding just the minimum

```
def min_list(a_list):
```

minlist.py

```
    min_so_far = a_list[0]
```

List cannot be empty!

```
    for a in a_list:
```

```
        if a < min_so_far:  
            min_so_far = a
```

```
    return min_so_far
```

Returning a single value

Finding just the maximum

```
def max_list(a_list):
```

maxlist.py

```
    max_so_far = a_list[0]
```

```
    for a in a_list:
```

```
        if a > max_so_far: ← Only real change
            max_so_far = a
```

```
    return max_so_far
```

Returning a single value

Finding both

```
def minmax_list(a_list):  
  
    min_so_far = a_list[0]  
    max_so_far = a_list[0]  
  
    for a in a_list:  
  
        if a < min_so_far:  
            min_so_far = a  
  
        if a > max_so_far:  
            max_so_far = a
```

return *what?*

This is the real question

Returning both

```
def minmax_list(a_list):
```

minmaxlist.py

```
    min_so_far = a_list[0]  
    max_so_far = a_list[0]
```

```
    for a in a_list:
```

```
        if a < min_so_far:  
            min_so_far = a
```

```
        if a > max_so_far:  
            max_so_far = a
```

```
    return min_so_far, max_so_far
```

A pair of values

“Tuples”

e.g.

min_value, max_value

Pair

min_value, avg_value, max_value

Triple

Commas

Often written with parentheses:

(min_value, max_value)

(min_value, avg_value, max_value)

Using tuples to return values

```
def ...
```

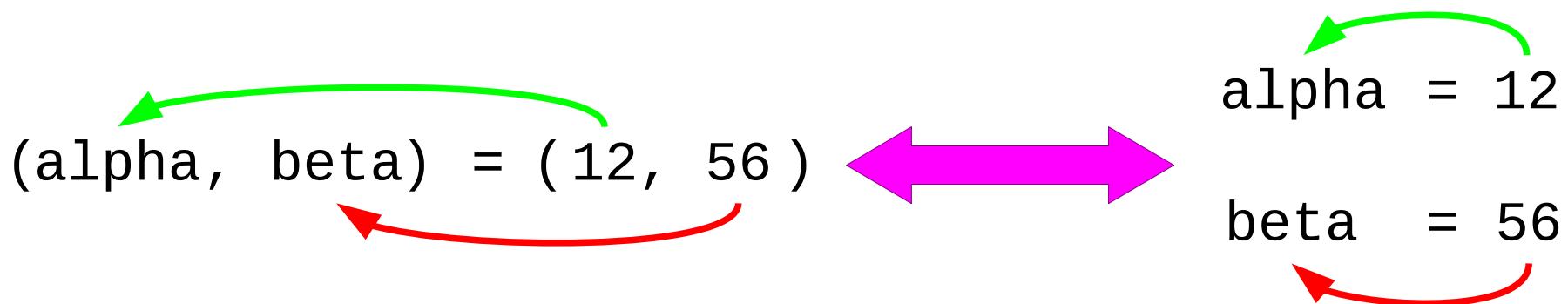
In the function definition

```
    return min_value , max_value
```

```
minimum , maximum = minmax_list(values)
```

Using the function

Using tuples to attach names



Swapping values

```
>>> alpha = 12
```

```
>>> beta = 56
```

```
>>> (alpha, beta) = (beta, alpha)
```

Swapping values

```
>>> print(alpha)
```

56

```
>>> print(beta)
```

12

Assignment works right to left

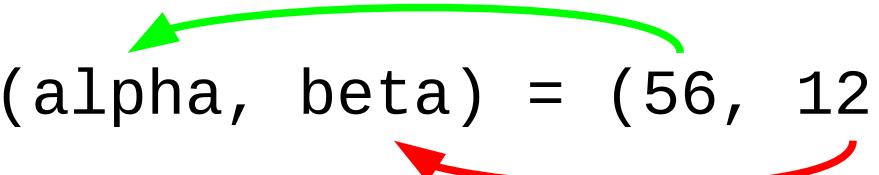
alpha = 12

beta = 56

(alpha, beta) = (beta, alpha)

Stage 1: $(\text{beta}, \text{alpha}) \xrightarrow{\text{purple arrow}} (56, 12)$

Stage 2: $(\text{alpha}, \text{beta}) = (56, 12)$



Progress

Multiple inputs

```
def thing(in1, in2, in3):
```

Multiple outputs

```
return (out1, out2, out3)
```

“Tuples”

```
(a, b, c)
```

Simultaneous assignment

```
(a, b) = (a+b, a-b)
```

Exercise 19

The script `exercise19.py` is an answer to [exercise 16](#).

Edit it to:

1. define a function `file_stats()` that takes a file name and returns a triple `(n_lines, n_words, n_chars)`
2. use `input()` to read in a file name
3. use `file_stats()` with that file name
4. end with `print(filename, file_stats(filename))`



10 minutes

Tuples and lists: similarities

```
>>> x = ['alpha', 'beta']
```

```
>>> x[1]  
'beta'
```

Indexing

```
>>> len(x)  
2
```

Length

```
>>> (a, b) = (1, 2)  
>>> a  
1
```

Simultaneous
assignment

```
>>> y = ('alpha', 'beta')
```

```
>>> y[1]  
'beta'
```

```
>>> len(y)  
2
```

```
>>> [c, d] = [1, 2]  
>>> c  
1
```

Tuples and lists: differences

```
>>> x = ['alpha', 'beta']
```

```
>>> x[1] = 'B'
```

```
>>> x
```

```
['alpha', 'B']
```

Lists are
mutable

```
>>> y = ('alpha', 'beta')
```

```
>>> y[1] = 'B'
```

TypeError:
'tuple' object does not
support item assignment

Tuples are
immutable

Tuples and lists: philosophy

Lists

Sequential:
Concept of “next item”

Best with all items
of the same type

Serial

```
[2, 3, 5, 7, 11, 13,  
 17, 19, 23, 29, 31]
```

Sequence of prime numbers

Tuples

Simultaneous:
All items “at once”

Safe to have
multiple types

Parallel

```
('Dowling', 'Bob',  
 50, 105.0, 'rjd4')
```

Surname, forename,
age, weight, user id

Functions we have written so far

`total(list)`

`add_lists(list1, list2)`

`minmax_list(list)`

Reusing functions within a script

```
def square(limit):  
    ...  
    ...  
    squares_a = square(34)  
    ...  
    five_squares = squares(5)  
    ...  
    squares_b = squares(56)  
    ...
```

One definition

Multiple uses in
the same file

Easy!

Reusing functions between scripts?

```
def squares(limit):  
    ...
```

One definition

```
...  
squares_a = squares(34)  
...
```

```
...  
squares_b = squares(56)  
...
```

Multiple uses in
multiple files

```
five_squares = squares(5)  
...
```

How?

“Modules”

```
def square(limit):
```

```
    ...
```

```
def cubes(limit):
```

```
    ...
```

```
...
```

```
five = squares(5)
```

```
...
```

Definition → Use

Module: a container of functions

Modules: a worked example — 1a

utils.py

Starts empty

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

```
def total(numbers):  
    sum_so_far = 0  
    for number in numbers:  
        sum_so_far += number  
    return sum_so_far
```

```
text = input('Number? ')  
number = int(text)  
squares_n = squares(number)  
total_n = total(squares_n)  
print(total_n)
```

sum_squares.py

Modules: a worked example — 1b

```
$ python3 sum_squares.py
```

Number? 5

30 = $0 + 1 + 4 + 9 + 16$

```
$ python3 sum_squares.py
```

Number? 7

91 = $0 + 1 + 4 + 9 + 16 + 25 + 36$

Modules: a worked example — 2a

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

```
def total(numbers):  
    sum_so_far = 0  
    for number in numbers:  
        sum_so_far += number  
    return sum_so_far
```

utils.py

```
text = input('Number? ')  
number = int(text)  
squares_n = squares(number)  
total_n = total(squares_n)  
print(total_n)
```

sum_squares.py

Move the definitions
into the other file.

Modules: a worked example — 2b

```
$ python3 sum_squares.py
```

Number? 5

```
Traceback (most recent call last):
  File "sum_squares.py", line 4, in <module>
    squares_n = squares(number)
NameError: name 'squares' is not defined
```

Because we have (re)moved its definition.

Modules: a worked example — 3a

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

def total(numbers):
    sum_so_far = 0
    for number in numbers:
        sum_so_far += number
    return sum_so_far
```

utils.py

```
import utils

text = input('Number? ')
number = int(text)
squares_n = squares(number)
total_n = total(squares_n)
print(total_n)
```

sum_squares.py

import utils
and *not*
import utils.py

import: Make
a reference to
the other file.

Modules: a worked example — 3b

```
$ python3 sum_squares.py
```

Number? 5

```
Traceback (most recent call last):
  File "sum_squares.py", line 4, in <module>
    squares_n = squares(number)
NameError: name 'squares' is not defined
```

Still can't find the function(s).

Modules: a worked example — 4a

squares() →
utils.squares()

total() →
utils.total()

```
import utils

text = input('Number? ')
number = int(text)
squares_n = utils.squares(number)
total_n = utils.total(squares_n)
print(total_n)
```

sum_squares.py

utils....: Identify
the functions
as coming from
the module.

Modules: a worked example — 4b

```
$ python3 sum_squares.py
```

Number? 5

30

```
$ python3 sum_squares.py
```

Number? 7

91

Working again!



Progress

Sharing functions between scripts

“Modules”

Importing modules

`import module`

Using functions from modules

`module.function()`

Exercise 20

The script `exercise20.py` is an answer to `exercise19`.

Move the function `file_stats()` from `exercise19.py` into `utils.py` and edit `exercise19.py` so that it still works.



5 minutes

The Python philosophy

A small core
language ...

... plus lots
of modules



**“Batteries
included”**

Example: the “math” module

```
>>> import math
```

Load in the “math” module.

```
>>> math.sqrt(2.0)
```

Run the `sqrt()` function...

... from the `math` module.

```
1.4142135623730951
```

`import module as alias`

```
>>> import math
```

Too long to keep typing?

```
>>> math.sqrt(2.0)
```

```
1.4142135623730951
```

```
>>> import math as m
```

“Alias”

```
>>> m.sqrt(2.0)
```

```
1.4142135623730951
```

Don't do these

```
>>> from math import sqrt
```

```
>>> sqrt(2.0)
```

```
1.4142135623730951
```

*Much better
to track the
module.*



```
>>> from math import *
```

```
>>> sqrt(2.0)
```

```
1.4142135623730951
```



What system modules are there?

Python 3.2.3 comes with over **250** modules.

glob

math

argparse

csv

io

cmath

datetime

html

os

random

getpass

json

signal

colorsys

logging

re

subprocess

email

pickle

string

sys

http

sqlite3

unicodedata

tempfile

webbrowser

unittest

xml

“Batteries included”

```
>>> help('modules')
```

Please wait a moment while I gather a list
of all available modules...

CDROM	binascii	inspect	shlex
bdb	importlib	shelve	263 modules

Enter any module name to get more help. Or,
type "modules spam" to search for modules whose
descriptions contain the word "spam".

Not quite this simple

Additional downloadable modules

Numerical

numpy

scipy

Graphics

matplotlib

Databases

pyodbc

psycopg2

PostgreSQL

MySQLdb

MySQL

cx_oracle

Oracle

ibm_db

DB2

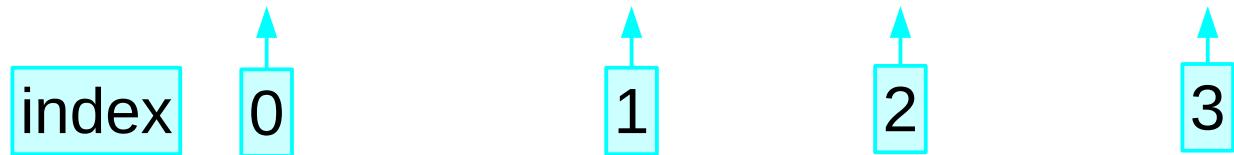
pymssql

SQL Server

An example system module: `sys`

```
import sys  
  
print(sys.argv)  
  
argv.py
```

```
$ python3 argv.py one two three  
['argv.py', 'one', 'two', 'three']
```



```
$ python3 argv.py 1 2 3
```

```
['argv.py', '1', '2', '3']
```



Always strings

`sys.exit()`

`exit()`

What we have been using

`sys.exit(rc)`

What we should use

“Return Code”: an integer

0: Everything was OK

$\neq 0$: Something went wrong

An example system module: `sys`

But also...

`sys.modules`

All modules currently imported

`sys.path`

Directories Python searches for modules

`sys.version`

Version of Python

`sys.stdin`

Where input inputs from

`sys.stdout`

Where print prints to

`sys.stderr`

Where errors print to

`sys.float_info`

All the floating point limits

...and there's more!

Modules in Python

“How do I do
X in Python?”



“What’s the Python
module to do X?”



“Where do I find
Python modules?”³⁹⁴

Finding modules



Python: Built-in modules



SciPy: **Scientific Python** modules



PyPI: **Python Package Index**



Search: “Python3 module for X”

Help with modules

```
>>> import math
```

```
>>> help(math)
```

NAME

math

DESCRIPTION

This module is always available. It provides access to the mathematical functions defined by the C standard.

...

Help with module functions

...

FUNCTIONS

`acos(x)`

Return the arc cosine (measured in radians) of `x`.

...

```
>>> math.acos(1.0)
```

```
0.0
```

Help with module constants

...

DATA

```
e = 2.718281828459045  
pi = 3.141592653589793
```

...

```
>>> math.pi
```

```
3.141592653589793
```

Help for our own modules?

```
def squares(limit):  
    answer = []  
    for n in range(0,limit):  
        answer.append(n**2)  
    return answer  
  
def total(numbers):  
    sum_so_far = 0  
    for number in numbers:  
        sum_so_far += number  
    return sum_so_far
```

utils.py

Basic help already
provided by Python

```
>>> import utils  
  
>>> help(utils)  
  
NAME  
    utils  
  
FUNCTIONS  
    squares(limit)  
  
    total(numbers)  
  
FILE  
    /home/y550/utils.py
```

Adding extra help text

```
"""Some utility functions  
from the Python for  
Absolute Beginners course  
"""
```

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

```
def total(numbers):  
    sum_so_far = 0  
    for number in numbers:  
        sum_so_far += number  
    return sum_so_far
```

utils.py

```
>>> import utils
```

Fresh start

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

NAME
utils

DESCRIPTION
Some utility functions
from the Python for
Absolute Beginners course

FUNCTIONS
squares(limit)
total(numbers)

Adding extra help text to functions

```
"""Some utility functions  
from the Python for  
Absolute Beginners course  
"""
```

```
def squares(limit):  
    """Returns a list of  
    squares from zero to  
    limit**2.  
    """  
  
    answer = []  
    for n in range(0,limit):  
        answer.append(n**2)  
    return answer
```

utils.py

```
>>> import utils
```

Fresh start

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

NAME

utils

DESCRIPTION

...

FUNCTIONS

squares(limit)

```
Returns a list of  
squares from zero to  
limit**2.
```

Adding extra help text to functions

```
"""Some utility functions  
from the Python for  
Absolute Beginners course  
"""  
  
def squares(limit):  
    """Returns a list of  
    squares from zero to  
    limit**2.  
    """  
  
    answer = []  
    for n in range(0,limit):  
        answer.append(n**2)  
    return answer
```

utils.py

```
>>> import utils Fresh start  
  
>>> help(utils.squares)  
  
squares(limit)  
    Returns a list of  
    squares from zero to  
    limit**2.
```

Progress

Python a small language...

Functionality → Module

...with many, many modules

System modules

Foreign modules

Modules provide help

`help(module)`

Doc strings

`help(module.function)`

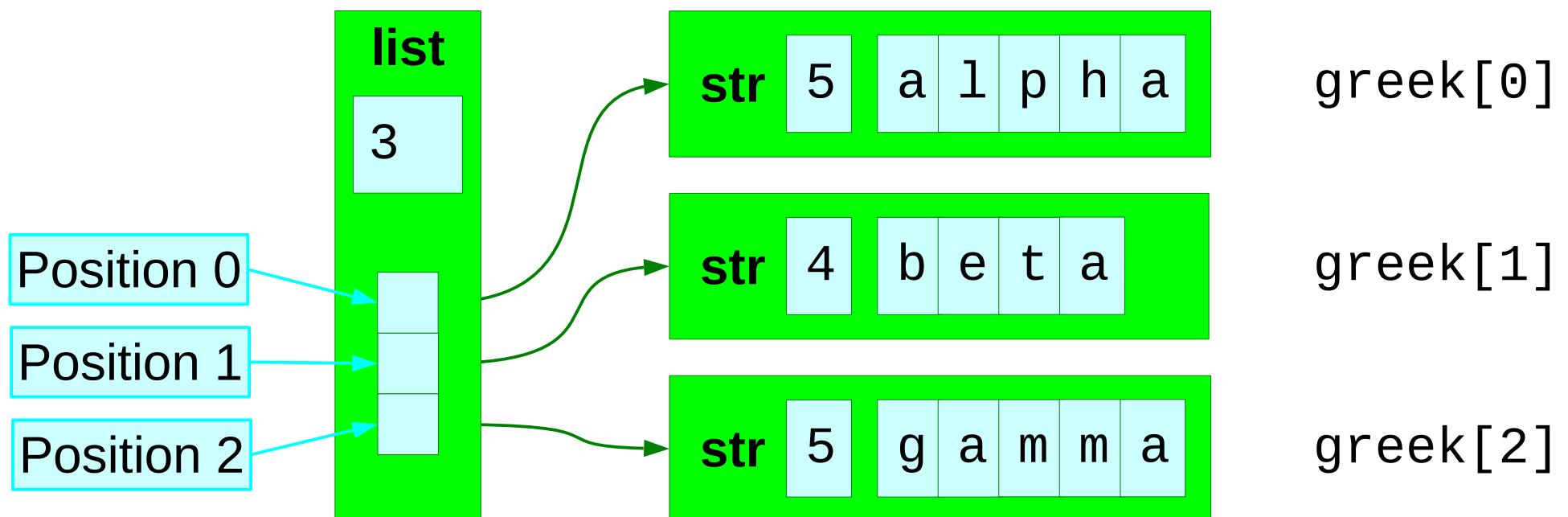
Exercise 21

Add help text to your `utils.py` file.

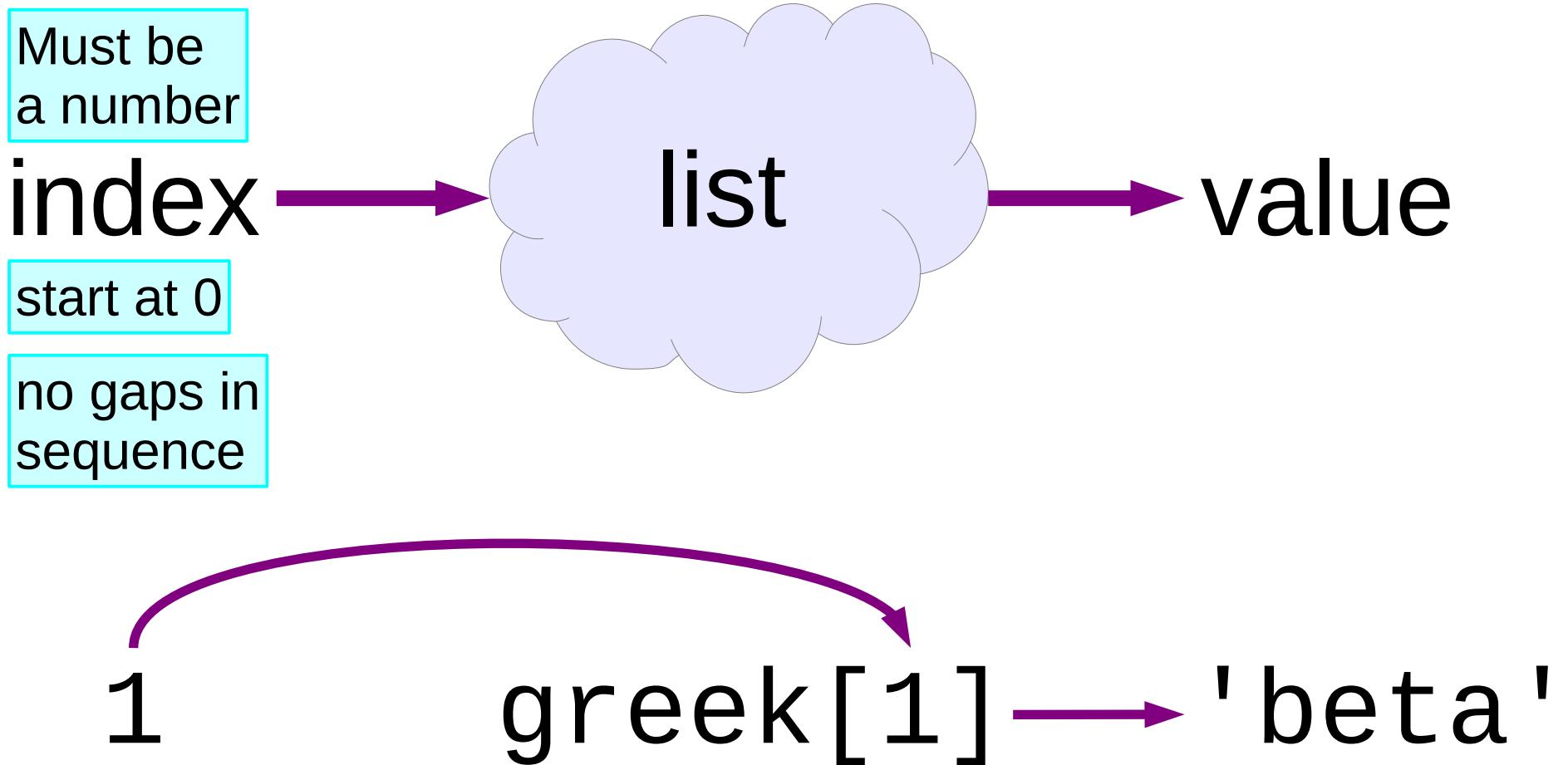


Recap: Lists & Indices

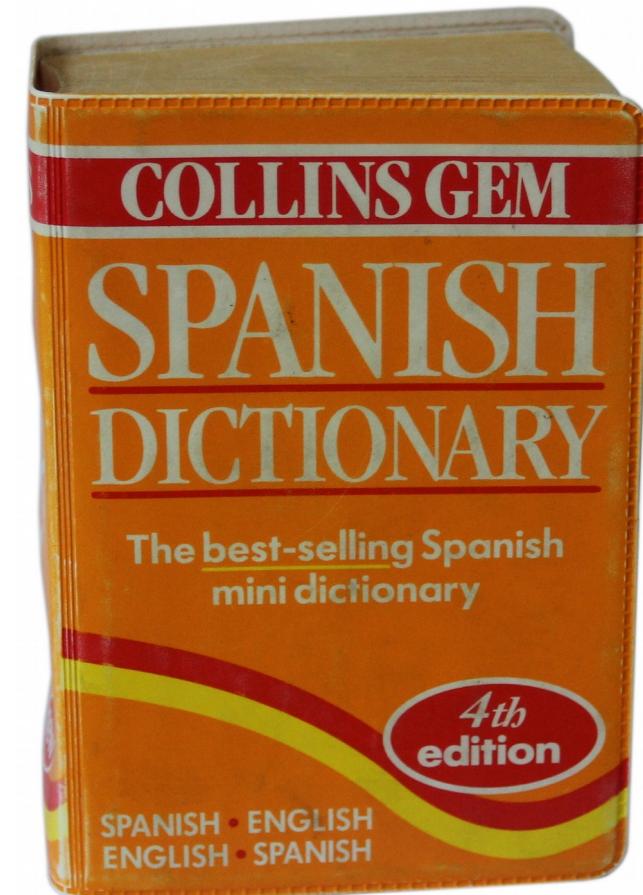
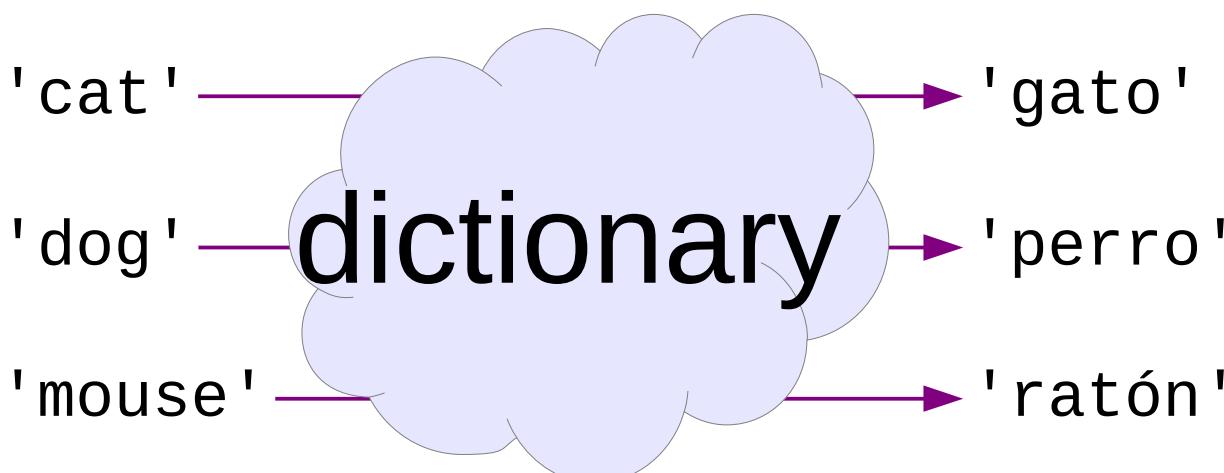
```
greek = ['alpha', 'beta', 'gamma']
```



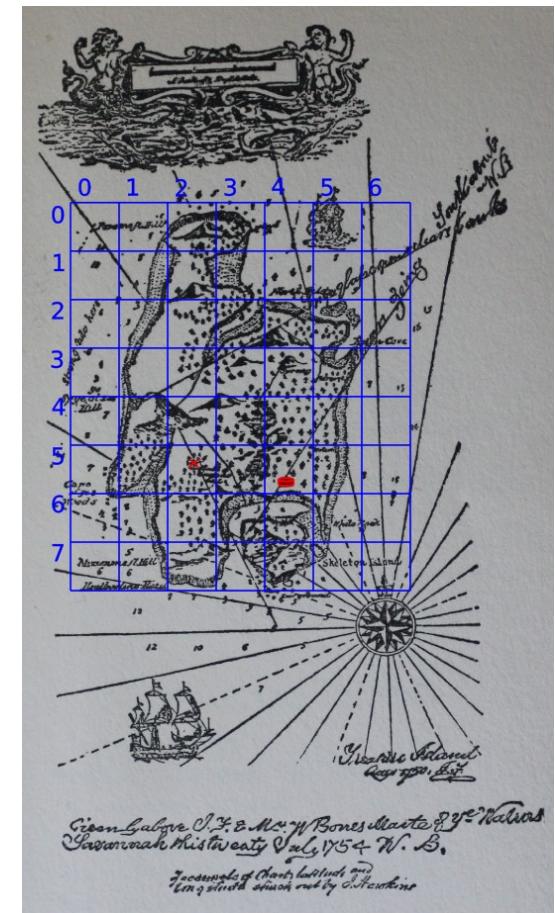
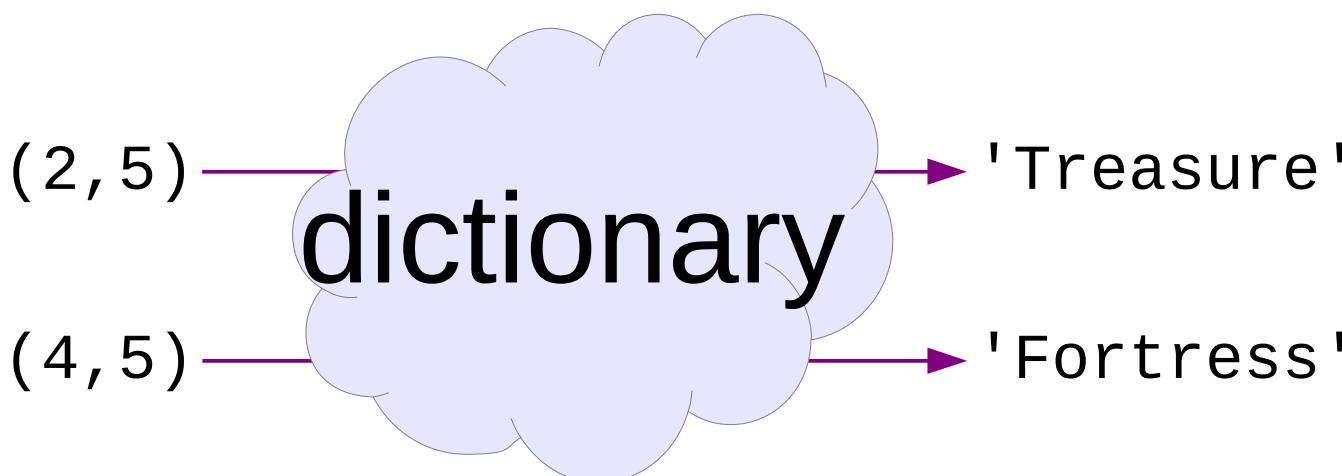
“Index in — Value out”



Other “indices”: Strings?



Other “indices”: Tuples?



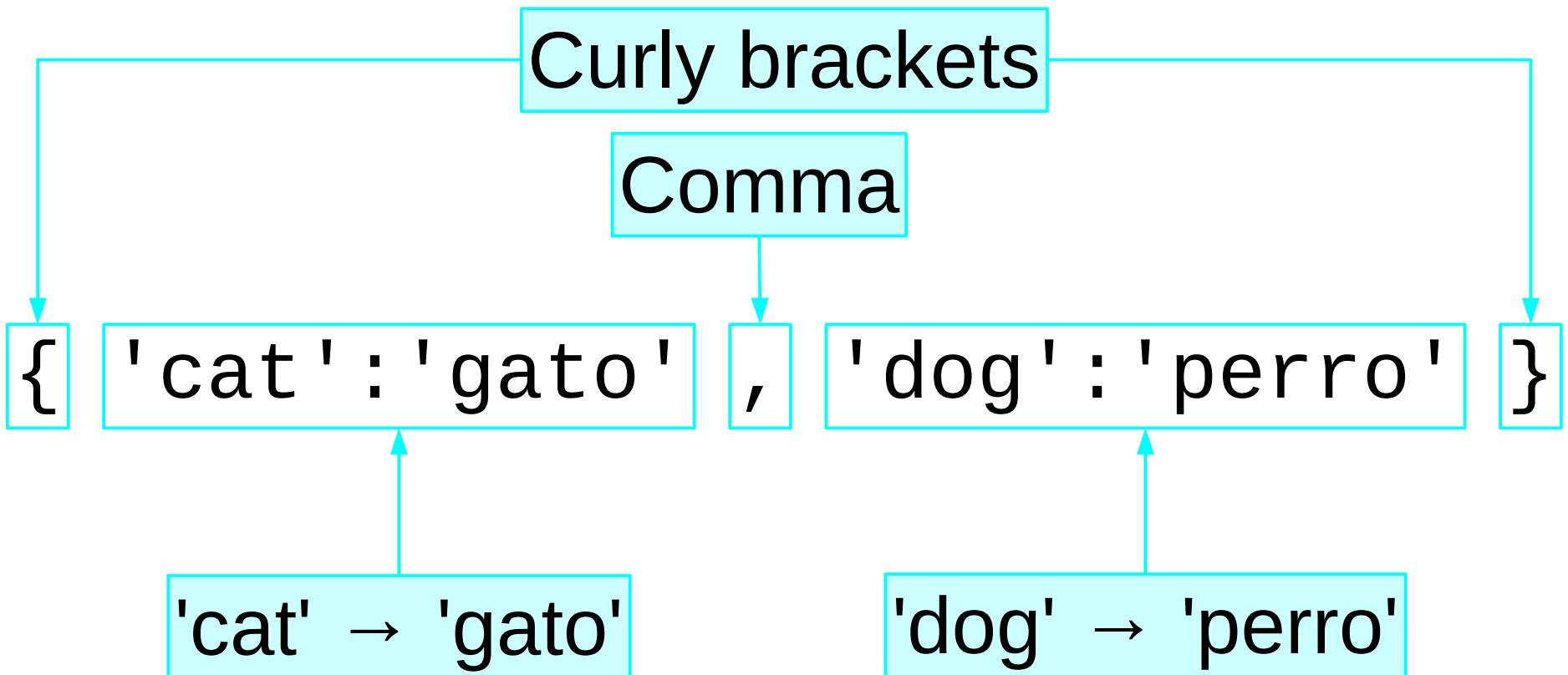
Python “dictionaries”

```
>>> en_to_es = { 'cat':'gato' , 'dog':'perro' }
```

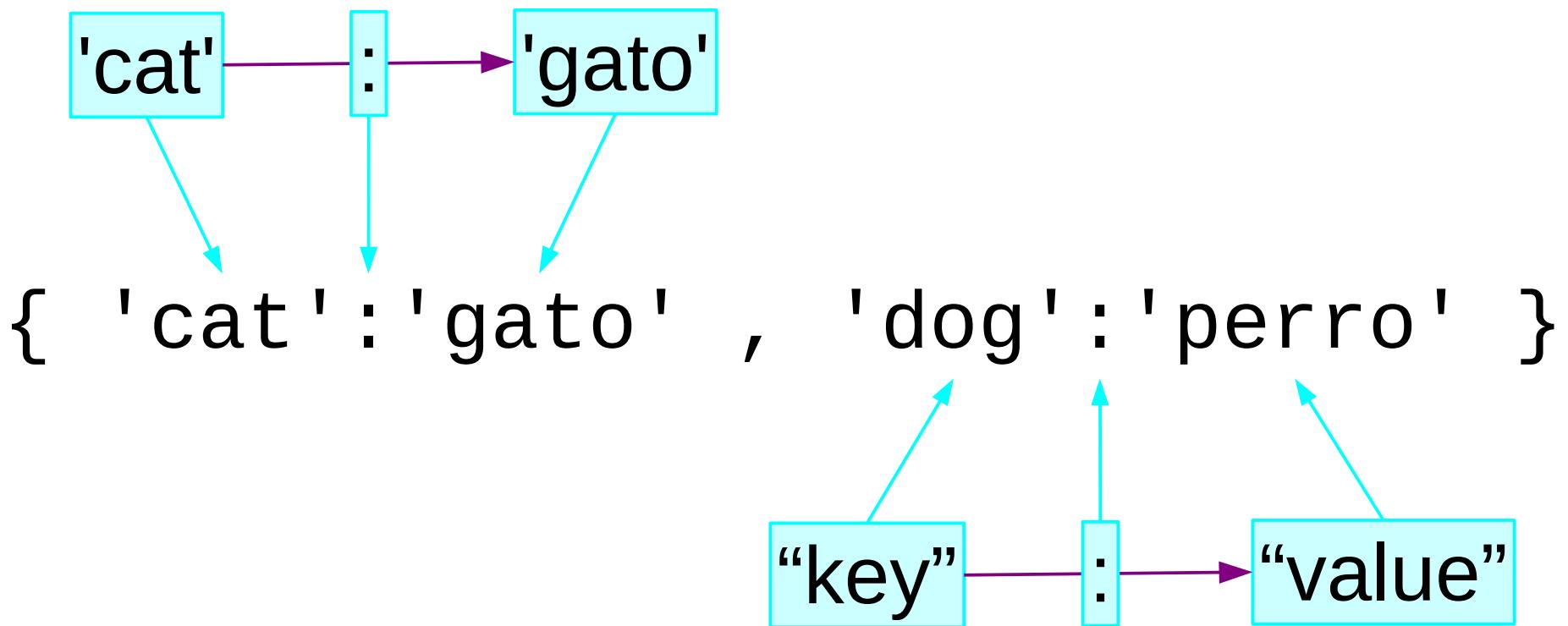
```
>>> en_to_es['cat']
```

```
'gato'
```

Creating a dictionary — 1



Creating a dictionary — 2



Using a dictionary — 1

```
>>> en_to_es = { 'cat':'gato' , 'dog':'perro' }
```

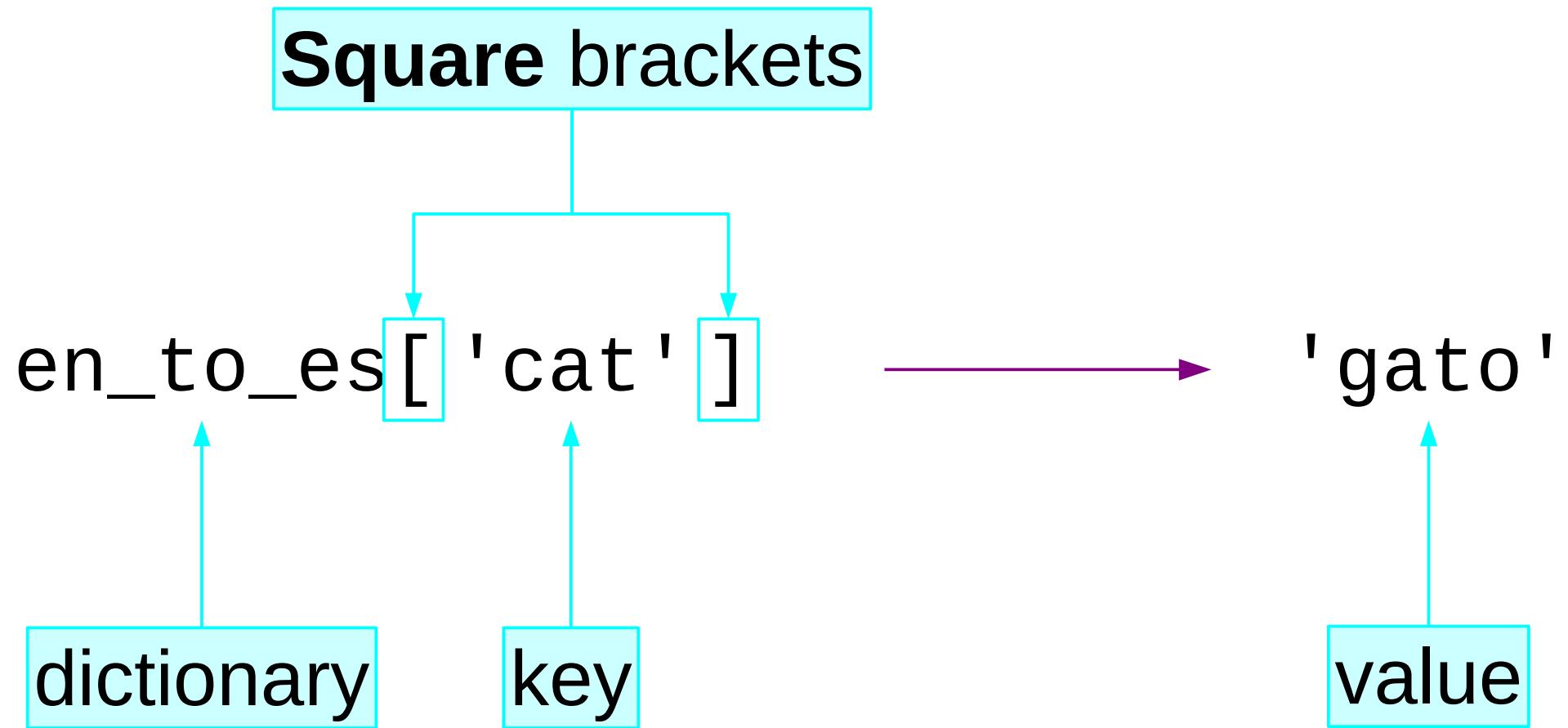
Creating the dictionary

```
>>> en_to_es['cat']
```

Using the dictionary

```
'gato'
```

Using a dictionary — 2



Missing keys

```
>>> en_to_es = { 'cat':'gato' , 'dog':'perro' }
```

```
>>> en_to_es['dog']
```



```
'perro'
```

```
>>> en_to_es['mouse']
```



```
Traceback (most recent call last):
```

```
  File "<stdin>", line 1, in <module>
```

```
    KeyError: 'mouse'
```

Error message

Dictionaries are one-way



```
>>> en_to_es = { 'cat':'gato' , 'dog':'perro' }
```



```
>>> en_to_es['dog']
```



```
'perro'
```

```
>>> en_to_es['perro']
```



```
Traceback (most recent call last):
```

```
  File "<stdin>", line 1, in <module>
```

```
KeyError: 'perro'
```



Looking for a **key**

Adding to a dictionary

```
>>> en_to_es = { 'cat':'gato' , 'dog':'perro' }
```

Initial dictionary has no 'mouse'

```
>>> en_to_es['mouse'] = 'ratón'
```

Adding 'mouse' to the dictionary

```
>>> en_to_es['mouse']
```

'ratón'

Removing from a dictionary

```
>>> print(en_to_es)
```

```
{'mouse': 'ratón', 'dog': 'perro', 'cat': 'gato'}
```

```
>>> del en_to_es['dog']
```

```
>>> print(en_to_es)
```

```
{'mouse': 'ratón', 'cat': 'gato'}
```

Progress

Dictionaries

Key → Value

{ key₁:value₁ , key₂:value₂, key₃:value₃ }

Looking up values

dictionary[key] → value

Setting values

dictionary[key] = value

Removing keys

del dictionary[key]

Exercise 22

Complete [exercise22.py](#) to create an English to French dictionary.

cat → chat

dog → chien

mouse → souris

snake → serpent



5 minutes

What's in a dictionary? — 1

```
>>> en_to_es
```

```
{'mouse': 'ratón', 'dog': 'perro', 'cat': 'gato'}
```

```
>>> en_to_es.keys()
```

```
dict_keys(['mouse', 'dog', 'cat'])
```

Orders
match

```
>>> en_to_es.values()
```

```
dict_values(['ratón', 'perro', 'gato'])
```

Just treat them like lists

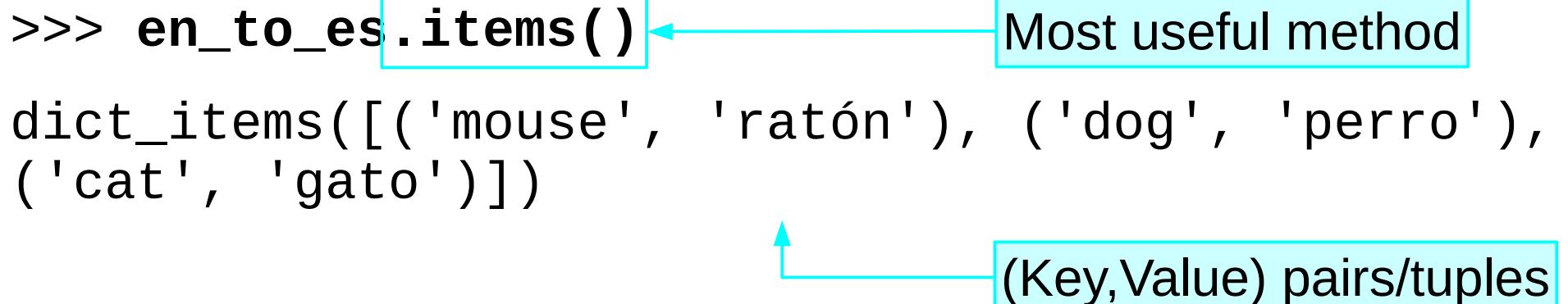
(or convert them to lists)

What's in a dictionary? — 2

```
>>> en_to_es.items()  
dict_items([('mouse', 'ratón'), ('dog', 'perro'),  
           ('cat', 'gato')])
```

Most useful method

(Key,Value) pairs/tuples



```
>>> for (english, spanish) in en_to_es.items():  
...     print(spanish, english)
```

...
ratón mouse

perro dog

gato cat

What's in a dictionary? — 3

Common simplification

```
>>> list(en_to_es.items())
```

```
[('mouse', 'ratón'), ('dog', 'perro'), ('cat', 'gato')]
```

Getting the list of keys



```
{'the': 2, 'cat': 1, 'sat': 1, 'on': 1, 'mat': 1}
```



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

Is a key in a dictionary?

```
>>> en_to_es['snake']
```

Traceback (most recent call last):

 File "<stdin>", line 1, in <module>

 KeyError: 'snake'

Want to avoid this

```
>>> 'snake' in en_to_es
```

False

We can test for it

Example: Counting words — 1

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



```
counts = {'the':2, 'cat':1, 'sat':1, 'on':1, 'mat':1}
```

Example: Counting words — 2

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

```
counts = {}
```

Start with an empty dictionary

```
for word in words:
```

Work through all the words

Do something

Example: Counting words — 3

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

```
counts = {}
```

```
for word in words:
```

 counts[word] += 1

This will not work

counter1.py

Why doesn't it work?

```
counts = {'the':1, 'cat':1}
```

✓ counts['the'] += 1

The key must already
be in the dictionary.

counts['the'] = counts['the'] + 1

✗ counts['sat'] += 1

Key is not in
the dictionary!

counts['sat'] = counts['sat'] + 1

Example: Counting words — 4

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

```
counts = {}
```

```
for word in words:
```

```
    if word in counts:
```

```
        counts[word] += 1
```

```
    else:
```

Do something

Need to add the key

Example: Counting words — 5

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

counts = {}

for word in words:
    if word in counts:
        counts[word] += 1
    else:
        counts[word] = 1

print(counts)
```

counter2.py

Example: Counting words — 6

```
$ python3 counter2.py
```

```
{'on': 1, 'the': 2, 'sat': 1, 'mat': 1, 'cat': 1}
```



You cannot predict the order of the keys when a dictionary prints out.

Example: Counting words — 7

```
print(counts)
```



Too ugly

```
items = list(dictionary.items())
```

Better

```
items.sort()
```

```
for (key, value) in items:
```

```
    print(key, value)
```

counter3.py

Example: Counting words — 8

```
$ python3 counter3.py
```

```
cat 1
mat 1
on 1
sat 1
the 2
```

Progress

Inspection methods

```
dictionary.keys()  
dictionary.values()  
dictionary.items()
```

Testing keys in dictionaries

```
if key in dictionary:  
    ...
```

Creating a list of keys

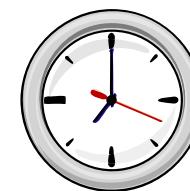
```
keys = list(dictionary)
```

Exercise 23

Complete [exercise23.py](#) to write a script that reverses a dictionary.

{'mouse': 'ratón', 'cat': 'gato', 'dog': 'perro'}

{'ratón': 'mouse', 'gato': 'cat', 'perro': 'dog'}

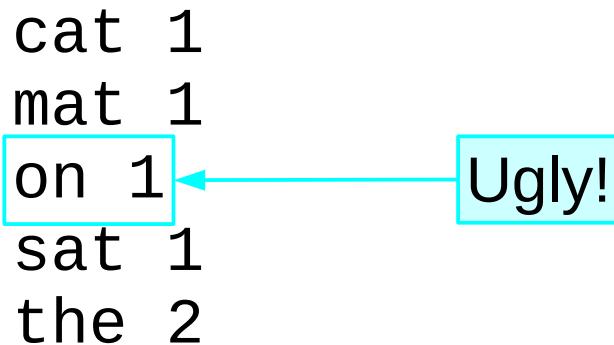


5 minutes

Formatted output

```
$ python3 counter3.py
```

```
cat 1  
mat 1  
on 1  
sat 1  
the 2
```



Ugly!

```
cat 1  
mat 1  
on 1  
sat 1  
the 2
```

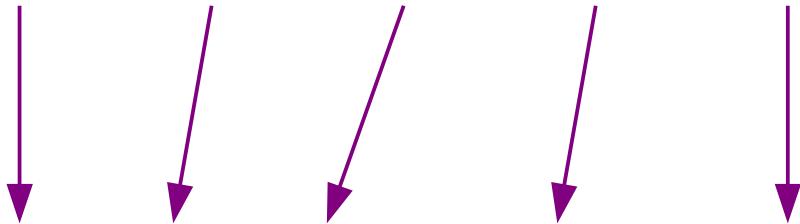
We want data
nicely aligned

The format() method

```
>>> 'xxx{}yyy{}zzz'.format('A', 100)
```

```
'xxxAyyy100zzz'
```

'xxx{}yyy{}zzz'



'xxxAyyy100zzz'

String formatting — 1

```
>>> 'xxx{:5s}yyy'.format('A')
```

```
'xxxA_____yyy'
```

'**xxx{:5s}yyy**'

{:5s}

↓ ↓ ↓

```
'xxxA_____yyy'
```

s — substitute a string

5 — pad to **5** spaces
(left aligns by default) 438

String formatting — 2

```
>>> 'xxx{ :<5s}yyy'.format('A')
```

```
'xxxA\u2022\u2022\u2022yyy'
```

'**xxx{ :<5s}yyy**'

{ :<5s}

↓ ↓ ↓

```
'xxxA\u2022\u2022\u2022yyy'
```

< — align to the left (←)

String formatting — 3

```
>>> 'xxx{:>5s}yyy'.format('A')
```

```
'xxx      Ayyy'
```

'**xxx**{:>5s}yyy'

{:>5s}

```
'xxx      Ayyy'
```

> — align to the right (→)

Integer formatting — 1

```
>>> 'xxx{:5d}yyy'.format(123)
```

```
'xxx  123yyy'
```

'**xxx**{:**5d**}**yyy**'

The diagram shows two strings. The first string is 'xxx{:5d}yyy'. The second string is 'xxx 123yyy'. Three purple arrows point from the '{:**5d**' part of the first string to the ' 1' part of the second string, indicating that the integer 123 is being formatted with a width of 5 digits and a leading space.

```
'xxx  123yyy'
```

{:**5d**}

d — substitute an integer
(**digits**)
5 — pad to **5** spaces
(right aligns by default)

Integer formatting — 2

```
>>> 'xxx{:>5d}yyy'.format(123)
```

```
'xxx  123yyy'
```

'xxx{:>5d}yyy' {:>5d}

```
'xxx  123yyy'
```

> — align to the right (→)

Integer formatting — 3

```
>>> 'xxx{:>5d}yyy'.format(123)
```

```
'xxxuu123yyy'
```

'xxx{:<5d}yyy' {:<5d}

```
'xxx123yyy'
```

< — align to the left (←)

Integer formatting — 4

```
>>> 'xxx{:05d}yyy'.format(123)
```

```
'xxx00123yyy'
```

'xxx{:05d}yyy' {:05d}

0 — pad with zeroes

'xxx00123yyy'

Integer formatting — 5

```
>>> 'xxx{:+5d}yyy'.format(123)
```

```
'xxx↳+123yyy'
```

'xxx{:+5d}yyy' { :05d}

The diagram illustrates how the format specifiers in the first string map to the output. Three purple arrows point from the specifiers to the output: one from the green 'xxx' to the green 'xxx' in the output, one from the red '{:+5d}' to the red '+123' in the output, and one from the red '}' to the red 'y' in the output.

'xxx_↳+123yyy'

+ — always show sign

Integer formatting — 6

```
>>> 'xxx{:+05d}yyy'.format(123)
```

```
'xxx+0123yyy'
```

'xxx{:+05d}yyy' {:05d}

+ — always show sign
0 — pad with zeroes

Integer formatting — 7

```
>>> 'xxx{:5,d}yyy'.format(1234)
```

```
'xxx1,234yyy'
```

'xxx{:5,d}yyy' {:5,d}

↓ ↓ ↓

'xxx1,234yyy'

, — 1,000s

Floating point formatting — 1

```
>>> 'xxx{:5.2f}yyy'.format(1.2)
```

```
'xxx 1.20yyy'
```

'xxx{:5.2f}yyy' {:5.2f}

'xxx_{1.20}yyy'

f — substitute a float

5 — 5 places *in total*

.2 — 2 places after the point

Floating point formatting — 2

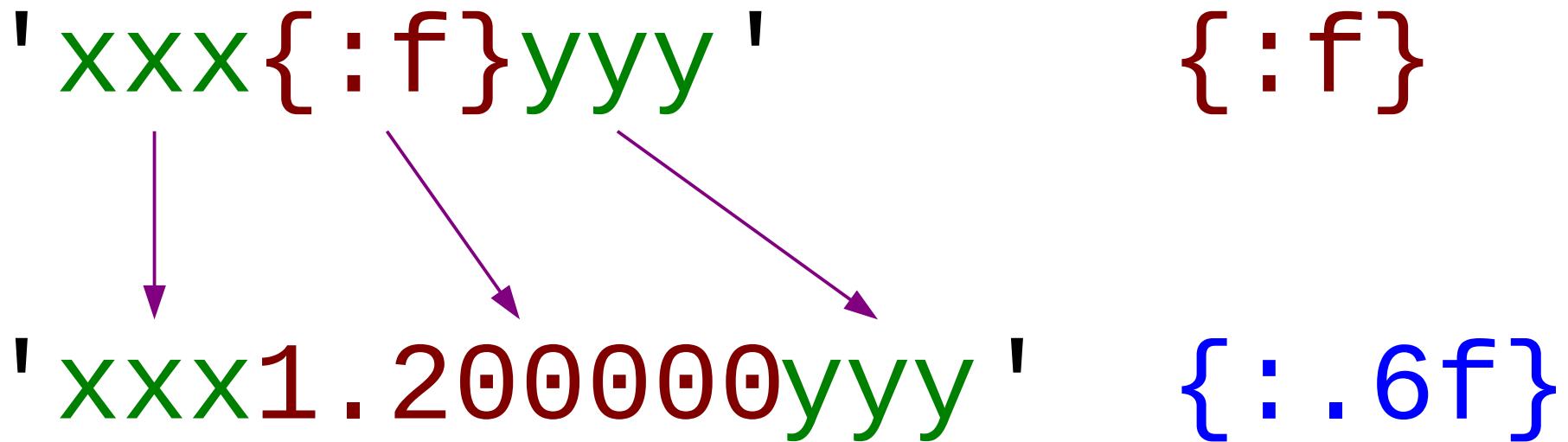
```
>>> 'xxx{:f}yyy'.format(1.2)
```

```
'xxx1.200000yyy'
```

'xxx{:f}yyy' {:f}

↓

'xxx1.200000yyy' {: .6f}



Ordering and repeats — 1

0 1 2

```
>>> 'x{:s}x{:d}x{:f}x'.format('abc', 123, 1.23)
```

'xabcx123x1.230000x'

0 1 2

0 1 2



Equivalent

```
>>> 'x{0:s}x{1:d}x{2:f}x'.format('abc', 123, 1.23)
```

'xabcx123x1.230000x'

0 1 2

0 1 2

Ordering and repeats — 2

```
>>> 'X{0:s}X{2:f}X{1:d}X'.format('abc', 123, 1.23)
```

```
'XabcX1.230000X123X'
```

0

2

1

0

1

2

```
>>> 'X{0:s}X{1:d}X{1:d}X'.format('abc', 123, 1.23)
```

```
'XabcX123X123X'
```

0

1

1

0

1

2

Formatting in practice

```
...
formatting = '{:3} {:1}'
for (word, number) in items:
    print(formatting.format(word, number))
```

```
$ python3 counter4.py
```

```
cat 1
mat 1
on 1
sat 1
the 2
```

Progress

Formatting *string.format(args)*

Numbered parameters {0} {1} {2}

Substitutions {:>6s}

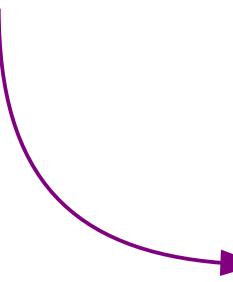
{:+06d}

{:+012.7f}

Exercise 24

Complete [exercise24.py](#)
to format the output as shown:

```
[ ( Joe, 9 ), ( Samantha, 45 ), ( Methuselah, 969 ) ]
```



Joe	9
Samantha	45
Methuselah	969



5 minutes

And that's it! (And “it” is a lot!)

Text	“if” test	Reading files
Prompting	Indented blocks	Writing files
Numbers	Lists	Functions
Arithmetic	Indices	“Structured programming”
Comparisons	Object methods	Tuples
Booleans	Built-in help	“for” loops
Variables	“for” loops	Modules
Deleting names	“Treat it like a list...”	Dictionaries
“while” loop	Values direct/via index	Formatting

But wait! There's more...

Advanced topics: Self-paced introductions to modules

Object-oriented programming in Python

Congratulations!

Text

Programs

Names

Annotations

Comparisons

Booleans

Variables

Deleting names

“while” loop

“if” test

Indented blocks

Lists

Indices

Object methods

Built-in help

“for” loops

“Treat it like a list...”

Values direct/via index

Reading files

Writing files

Functions

“Structured programming”

Tuples

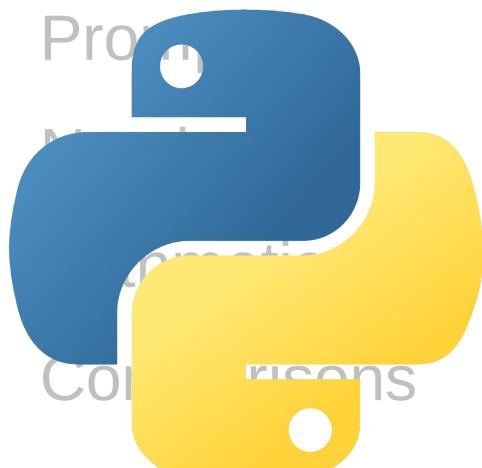
“for” loops

Modules

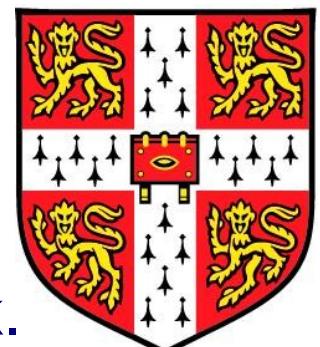
Dictionaries

Formatting

TM



python



Please don't forget the feedback.