Python: Introduction for Absolute Beginners

Bob Dowling

University Computing Service

Scientific computing support email address:
`scientific-computing@ucs.cam.ac.uk`

These course notes:
`www-uxsup.csx.cam.ac.uk/courses/PythonAB/`

Course outline — 1

Introduction

Who uses Python?
What is Python?
Launching Python

Using Python like a calculator

Types of value
Numbers
Text
Truth and Falsehood
Python values

Course outline — 2

Using Python like
a programming
language

We will do
lots with lists.

Variables
if…then…else…
while… loops
Comments
Lists
for… loops
Functions
Tuples
Modules

Course outline — 3

Interacting with the outside world

Built-in modules
The “sys” module
Reading input
Files

Storing data in programs

Dictionaries

What is Python used for?

Network services

Web applications

GUI applications

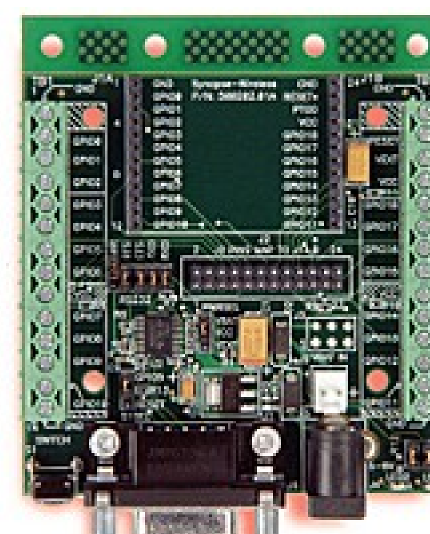
CLI applications

`/usr/bin/command-not-found`

Scientific libraries

Instrument control

Embedded systems



What is Python?

Compiled ⟷ Interpreted

Fortran, C, C++	Java, .NET	**Python**	Perl	Shell

What is Python?

Source of program?

Typed “live”

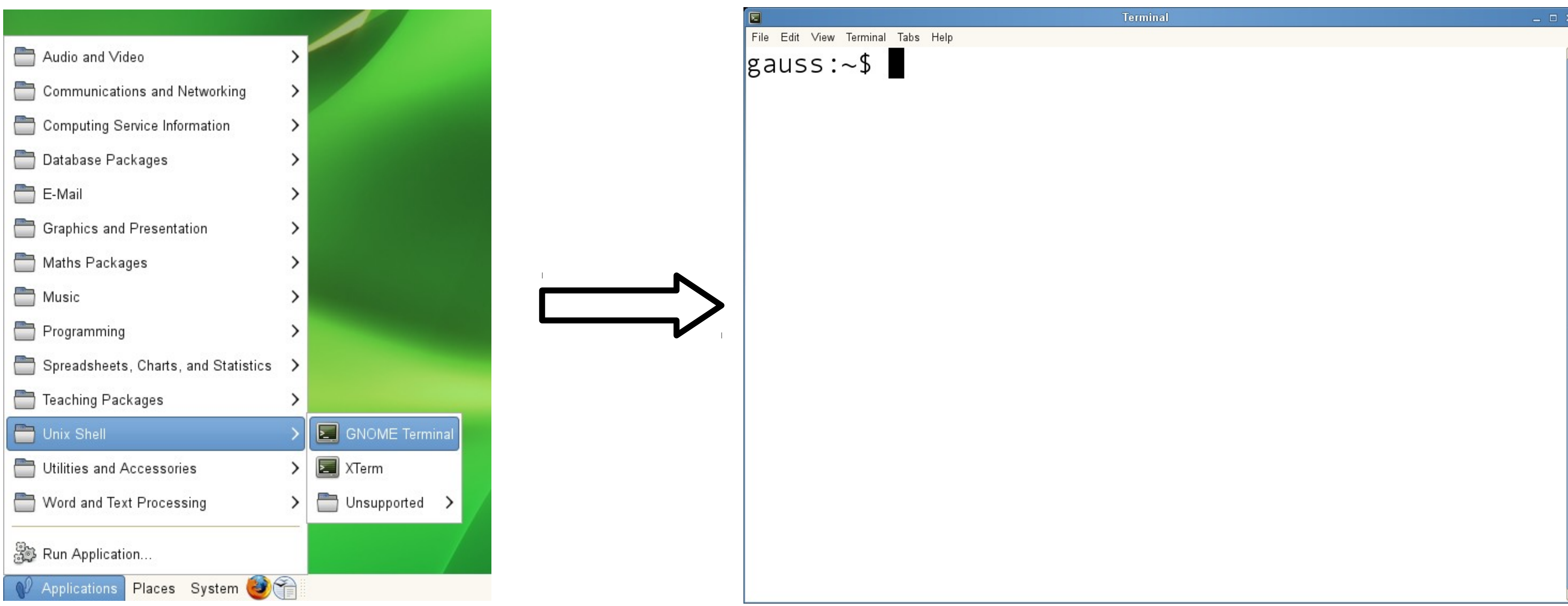
“Interactive”

Read from a file

“Batch” mode

Launching Python interactively — 1

Applications → Unix Shell → GNOME Terminal



Launching Python interactively — 2

Unix prompt

Unix command

Bold face means *you* type it.

```
$ python
Python 2.6 …
[GCC 4.3.2 …
Type "help", …
>>>
```

Introductory blurb

Python prompt

Using Python interactively

Python function

Brackets

Function *argument*

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

Function result

Python prompt

UCS

Using Python interactively

```
>>> print(3)
3

>>> 5
5
```

Instruct Python to print a 3

Python prints a 3

Give Python a literal 5

Python evaluates and displays a 5

Using Python interactively

```
>>> 5
5

>>> 2 + 3
5
```

Give Python an equivalent to 5

Python evaluates and displays a 5

UCS

Using Python interactively

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

No quotes

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

Quotes

Quitting Python interactively

Python prompt

`>>>` **`[Ctrl]+[D]`**

Unix “end of input”

`$`

Unix prompt

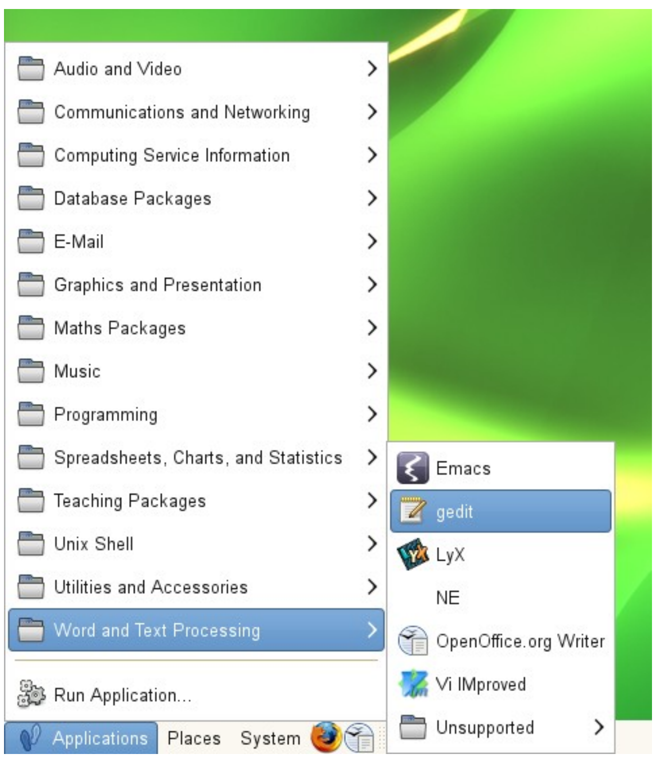
Exercise

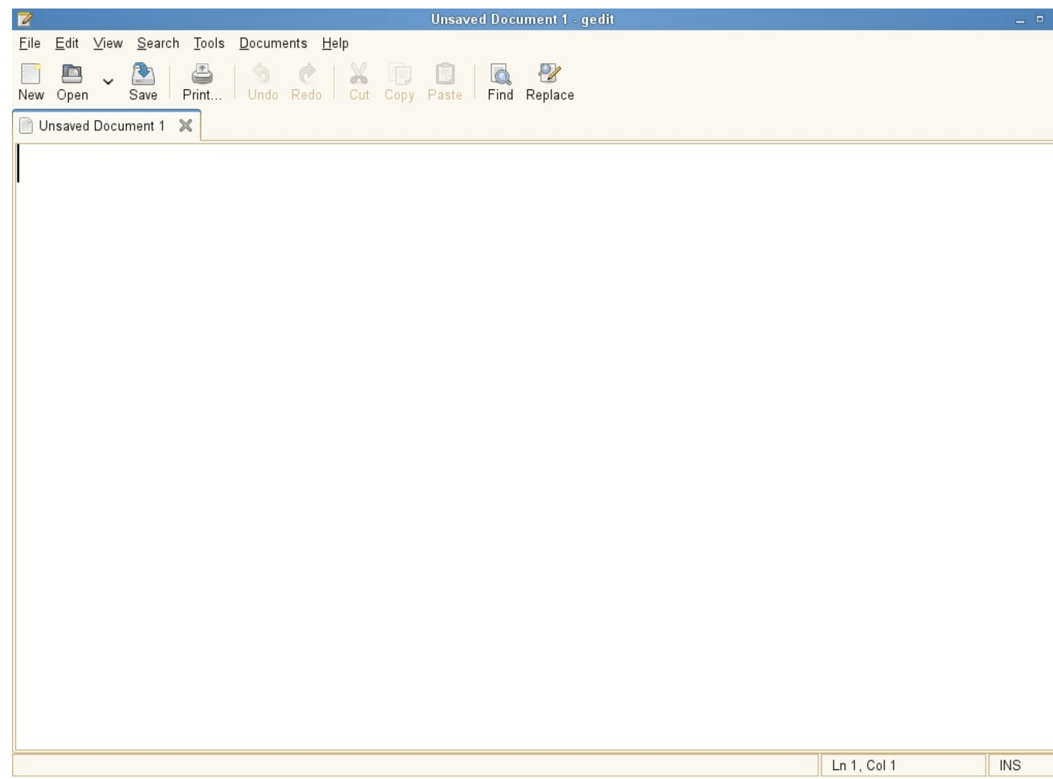
1. Launch a terminal window.
2. Launch Python.
3. Print out “Hello, world!”
4. Run these Python expressions (one per line):
   (a) 42
   (b) 26+18
   (c) 26<18
   (d) 26>18
5. Exit Python (but not the terminal window).

2 minutes

Writing Python scripts

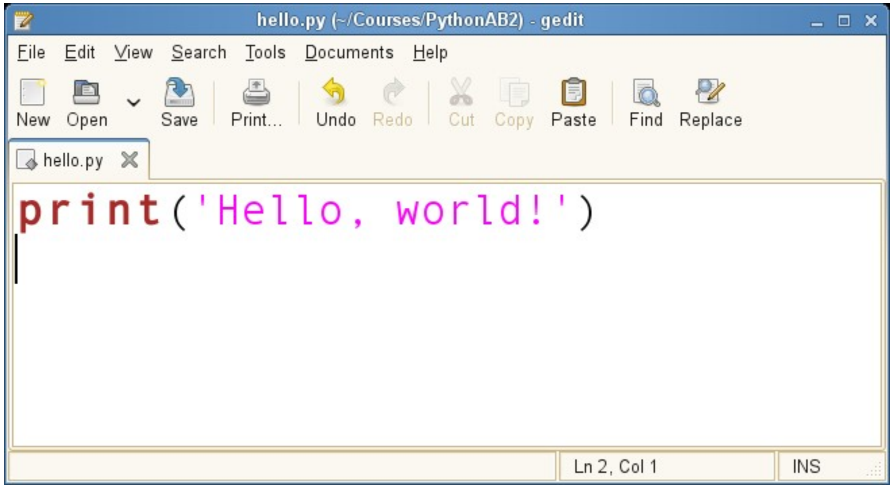
Applications → Word and Text Processing → gedit





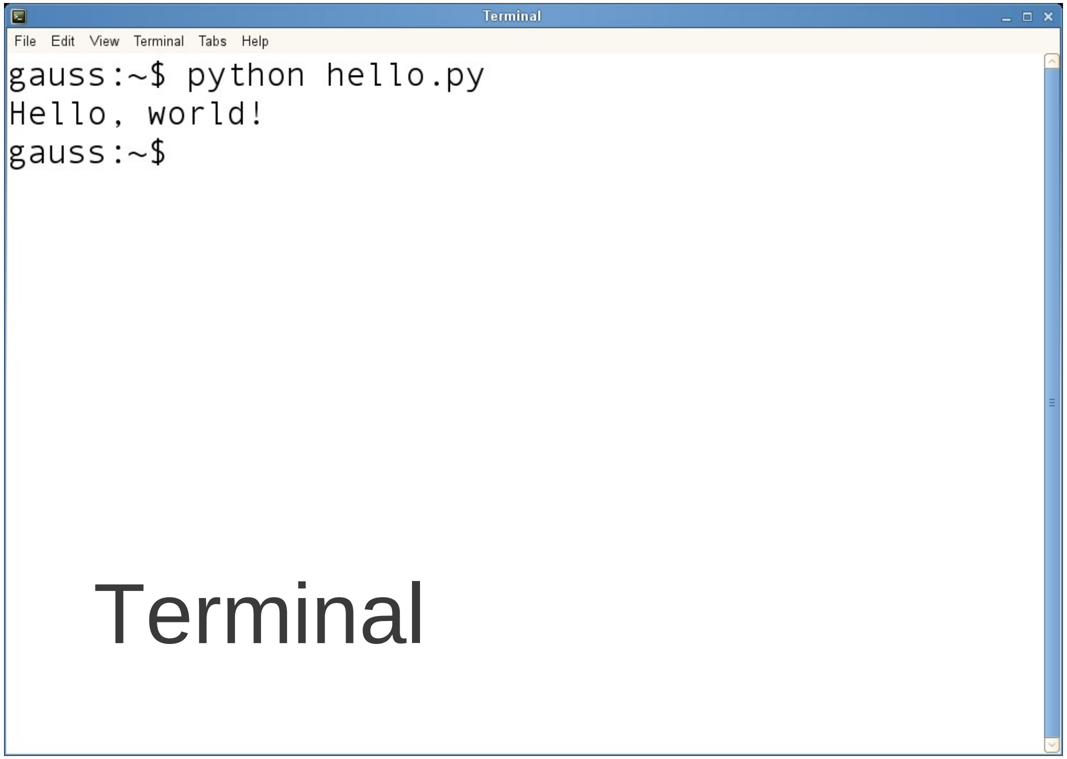
Launching Python scripts

Read / edit the script



gedit

Run the script



Terminal

UCS

Launching Python scripts

Unix prompt

```
$ python hello.py
Hello, world!

$
```

No three lines of blurb

Straight back to the Unix prompt

Launching Python scripts

```
print(3)
5
```

three.py

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

No “5” !

Interactive *vs.* Scripting

Source of program?

Typed “live”	Read from a file
“Interactive”	“Batch” mode
Introductory blurb	No blurb
Evaluations printed	Only explicit output

Progress

What Python is

Who uses Python

How to run Python interactively

How to run a Python script

Exercise

1. Launch a terminal window.
2. Run `hello.py` as a script.
3. Edit `hello.py`.
   Change “Hello” to “Goodbye”.
4. Run it again.

2 minutes

Types of values

Numbers	Whole numbers
	Decimal numbers
Text	
“Boolean”	True
	False

Integers

$\mathbb{Z}$

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

UCS

```
>>> 4+2
6
```

Addition behaves as you might expect it to.

```
>>> 3 + 5
8
```

Spaces around the “+” are ignored.

UCS

```
>>> 4-2
2
```

Subtraction also behaves as you might expect it to.

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

UCS

```
>>> 4*2
8

>>> 3 * 5
15
```

Multiplication uses a “*” instead of a “×”.

```
>>> 4/2
2
```

Division uses a "**/**" instead of a "**÷**".

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

Division rounds down.

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

Strictly down.

UCS

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

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

```
>>>   5 ** 3
125
```

Spaces around the "**" allowed, but not within it.

Remainder uses a “**%**”.

```
>>> 4%2
0
```

4 = 2×2 + **0**

```
>>> 5 % 3
2
```

5 = 1×3 + **2**

```
>>> -5 % 3
1
```

-5 = -2×3 + **1**

Always zero or positive

How far can integers go?

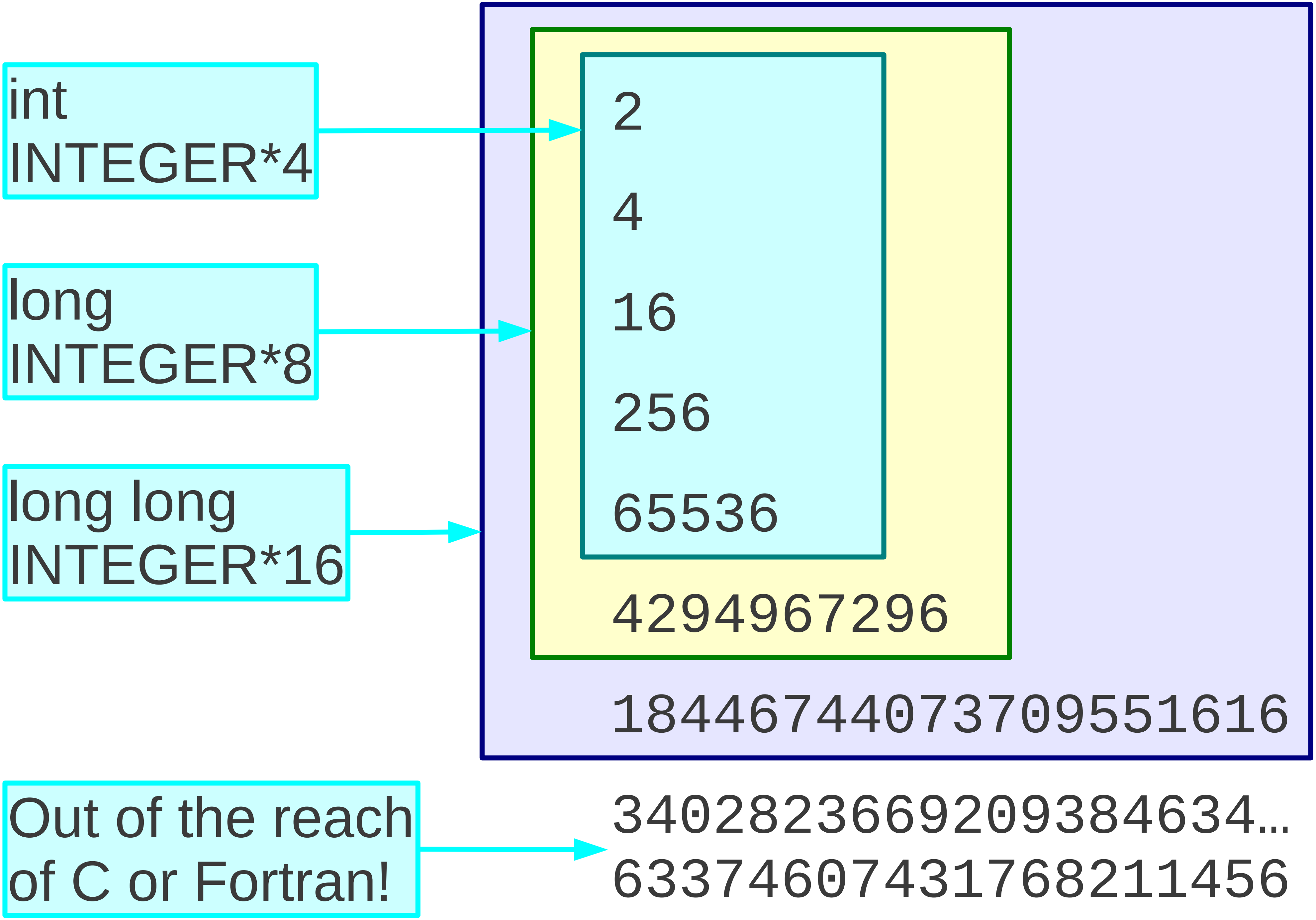
```
>>> 2 * 2
4
>>> 4 * 4
16
>>> 16 * 16
256
>>> 256 * 256
65536
```

So far, so good…

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

340282366920938463463374607431768211456L
```

Long integer

No limit to size of Python's integers!


int
INTEGER*4
long
INTEGER*8
long long
INTEGER*16
Out of the reach
of C or Fortran!
2
4
16
256
65536
4294967296
18446744073709551616
3402823669209384634…
63374607431768211456

Progress

Whole numbers …-2, -1, 0, 1, 2…

No support for fractions 1/2 → 0

Unlimited range of values

Mathematical operations

Maths:	a+b	a-b	a×b	a÷b	a^b	a mod b
Python:	`a+b`	`a-b`	`a*b`	`a/b`	`a**b`	`a%b`

Exercise

In Python, calculate:

1.	12+4	2.	12+5
3.	12−4	4.	12−5
5.	12×4	6.	12×5
7.	12÷4	7.	12÷5
9.	12^4	10.	12^5

Which of these answers is “wrong”?

2 minutes

Floating point numbers

1	1.0
1 ¼	1.25
1 ½	1.5

But…

$\cancel{\mathbb{R}}$

1 ⅓

1.3

1.33

1.333

1.3333 ?

```
>>> 1.0
1.0
>>> 0.5
0.5
>>> 0.25
0.25
>>> 0.1
0.1
```

1 is OK

½ is OK

¼ is OK

Powers of two.

1/10 is not!

Why?

UCS

```
>>> 0.1
0.1
```

1/10 is stored inaccurately.

```
>>> 0.1 + 0.1 + 0.1
0.30000000000000004
```

Floating point numbers are…

…printed in decimal

…stored in binary

17 significant figures

```
>>> 0.1 + 0.1 + 0.1
0.30000000000000004
```

If you are relying on the 17th decimal place you are doing it wrong!

Same basic operations

```
>>> 5.0 + 2.0
7.0

>>> 5.0 − 2.0
3.0

>>> 5.0 % 2.0
1.0
```

```
>>> 5.0 * 2.0
10.0

>>> 5.0 / 2.0
2.5
```

Gets it right!

```
>>> 5.0 ** 2.0
25.0
```

How far can floating point numbers go?

```
>>> 4.0 * 4.0
16.0

>>> 16.0 * 16.0
256.0

>>> 256.0 * 256.0
65536.0

>>> 65536.0 * 65536.0
4294967296.0
```

So far, so good…

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

17 significant figures

$\times 10^{19}$

$1.8446744073709552 \times 10^{19} =$

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

$4294967296 \times 4294967296 =$

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

Difference → 384

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


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

>>> 3.4028236692093846e+38 *
    3.4028236692093846e+38

1.157920892373162e+77

>>> 1.157920892373162e+77 *
    1.157920892373162e+77

1.3407807929942597e+154
```

UCS

“Overflow errors”

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

Floating point infinity

Floating point limits

$\boxed{1.2345678901234567} \times 10^{\boxed{N}}$

17 significant figures

$-325 < N < 308$

Positive values:

4.94065645841e-324 < x < 8.98846567431e+307

UCS

Progress

Floating Point numbers

$1.25 \rightarrow$ `1.25`

$1.25\times10^5 \rightarrow$ `1.25e5`

Limited accuracy

Limited range of sizes

(but typically good enough)

Mathematical operations

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

Exercise

In Python, calculate:

1. $12{\cdot}0+4.0$
2. $12{\cdot}0-4{\cdot}0$
3. $12{\cdot}0\div4.0$
4. $12\div40{\cdot}0$
5. $25{\cdot}0^{0{\cdot}5}$
6. $5{\cdot}0^{-1{\cdot}0}$
7. $1{\cdot}0\times10^{20} + 2{\cdot}0\times10^{10}$
8. $1{\cdot}5\times10^{20} + 1{\cdot}0$

Which of these answers is “wrong”?

3 minutes

Strings

“The cat sat on the mat.”

“Lorem ipsum dolor sit amet, consectetuer adipiscing elit. D
onec at purus sed magna aliquet dignissim. In rutrum libero
 non turpis. Fusce tempor, nulla sit amet pellentesque feugi
at, nibh quam dapibus dui, sit amet ultrices enim odio nec i
psum. Etiam luctus purus vehicula erat. Duis tortor lorem, c
ommodo eu, sodales a, semper id, diam. Praesent ...”

Quotes

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

The value of the text object

Quotes: Hey, this is text!

How Python represents the text object.

UCS

Why do we need quotes?

3 → It’s a number

print ? → Is it a command?

print ? → Is it a string?

'print' → It’s a string

print → It’s a command

Python command

“This is text.”

The text.

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

Hello, world!

>>>
```

print only outputs the *value* of the text

Double quotes

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

Quotes: Hey, this is text!

Single quotes

Single quotes

Double quotes

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

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

Both define the *same* text object.

Mixed quotes

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

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

Joining strings together

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

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

Repeated text

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

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

Progress

Strings

Use quotes to identify (matching single or double)

Use print to output just the value

String operations

Exercise

Predict what interactive Python will print when you type the following expressions. Then check.

1. 'Hello, ' + "world!"
2. 'Hello!' * 3
3. "" * 10000000000
4. '4' + '2'

(That's two adjacent double quote signs.)

3 minutes

UCS

Line breaks

Problem: Suppose we want to create a string that spans several lines.

```
>>> print('Hello,↵
world!')
```

✗

```
>>> print('Hello,↵
SyntaxError: EOL while scanning
string literal
```

“**end of line**”

UCS

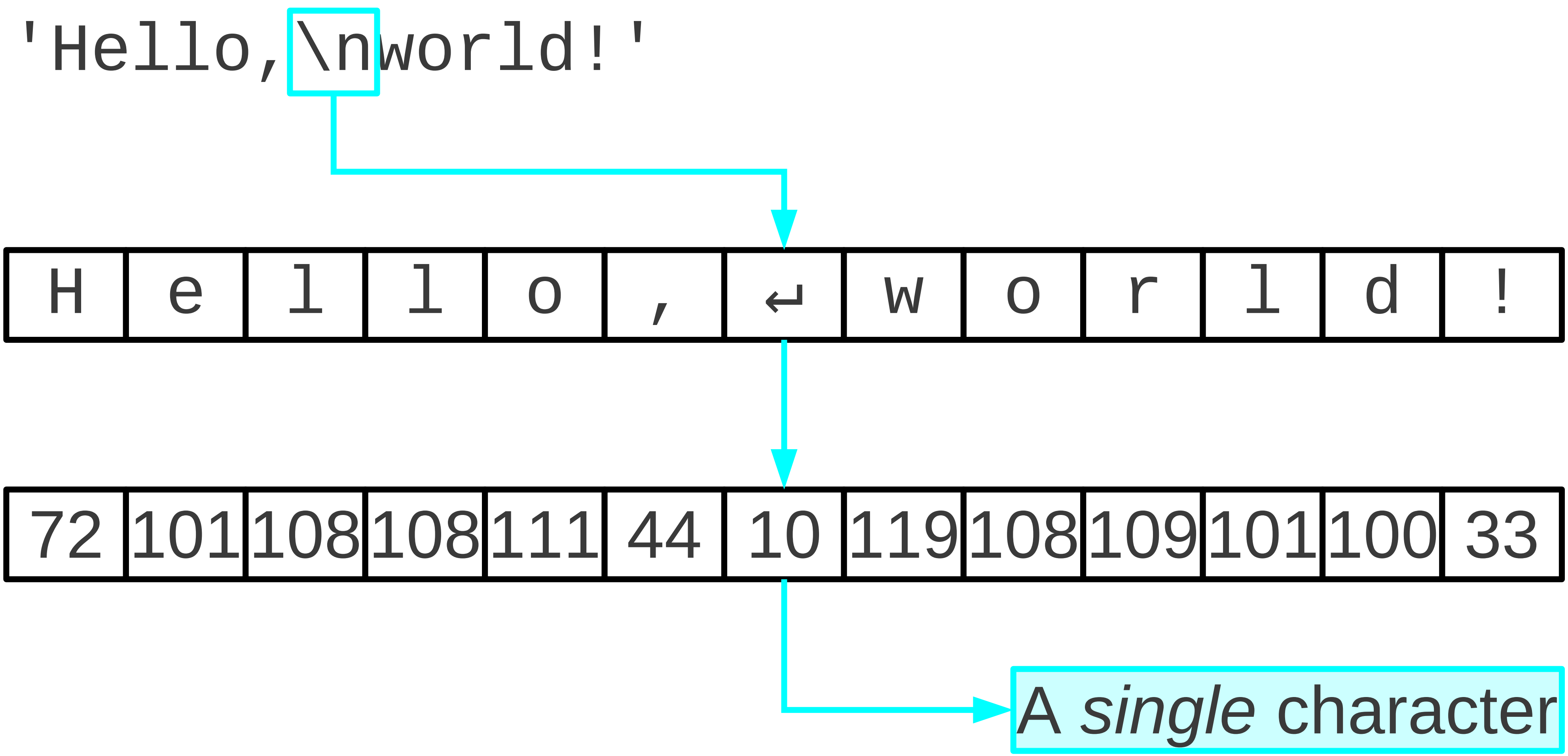
The line break character

Solution: Provide some other way to mark "line break goes here".

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

\n → **n**ew line

The line break character



Special characters

`\a`	→	♪	`\'`	→	'
`\n`	→	↵	`\"`	→	"
`\t`	→	⇥	`\\`	→	\

“Long” strings

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

'Hello,\nworld!'

>>>
```

Three single quote signs

An ordinary string

An embedded
new line character

What the string is *vs.* how the string prints

`'Hello,\nworld!'`

```
Hello,
world!
```

It's not just quotes vs. no quotes!

Single or double quotes

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

'Hello,\nworld!'

>>>
```

Three single quote signs

The same string

Long strings

```
'''Lorem ipsum dolor sit amet, consectetuer
adipiscing elit. Donec at purus sed magna aliquet
dignissim. In rutrum libero non turpis. Fusce
tempor, nulla sit amet pellentesque feugi at, nibh
quam dapibus dui, sit amet ultrices enim odio nec
ipsum. Etiam luctus purus vehicula erat. Duis
tortor lorem, commodo eu, sodales a, semper id,
diam.'''
```

Progress

Entering arbitrarily long strings	Triple quotes
Dealing with line breaks	`"""…"""` `'''…'''`
Other “special” characters	`\n` `\t` …

Exercise

Predict the results of the following instructions.
Then check.

1. `print('Goodbye, world!')`
2. `print('Goodbye,\nworld!')`
3. `print('Goodbye,\tworld!')`

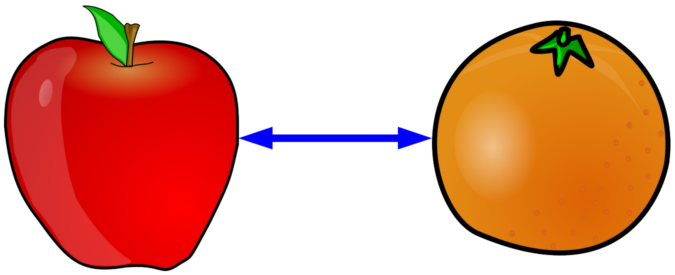
2 minutes

Comparisons

Are two values the same? $5+2 \longleftrightarrow 7$

Is one bigger than the other? $5+2 \longleftrightarrow 8$

Is “bigger” even meaningful?



Comparisons

A comparison operation

```
>>> 5 > 4
True
```

A comparison result

```
>>> 5.0 < 4.0
False
```

Only two values possible

Equality comparison

n.b. *double* equals

```
>>> 5 == 4
False
```

Useful comparisons

```
>>> (2**2)**2 == 2**(2**2)
True

>>> (3**3)**3 == 3**(3**3)
False
```

All numerical comparisons

Python	Mathematics
x == y	$x = y$
x != y	$x \neq y$
x < y	$x < y$
x <= y	$x \leq y$
x > y	$x > y$
x >= y	$x \geq y$

UCS

Comparing strings

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

>>> 'bad' < 'bud'
True

>>> 'cat' < 'cathode'
True
```

Alphabetic order…

Comparing strings

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

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

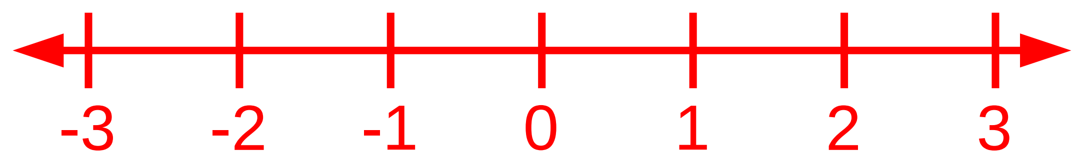
ABCDEFGHIJKLMNOPQRSTUVWXYZ…
abcdefghijklmnopqrstuvwxyz

Progress

Six comparisons:

==	!=	<	<=	>	>=
$=$	$\neq$	$<$	$\leq$	$>$	$\geq$

Numbers:
numerical order



Strings:
alphabetical order

ABCDEFGHIJKLMNOPQRSTUVWXYZ...
abcdefghijklmnopqrstuvwxyz

Exercise

Predict whether Python will print True or False when you type the following expressions.
Then check.

1. 100 < 100
2. 3*45 <= 34*5
3. 'One' < 'Zero'
4. 1 < 2.0
5. 0 < 1/10
6. 0.0 < 1.0/10.0

3 minutes

Truth and Falsehood

`True` and `False`

“Boolean” values

Same status as numbers, strings, etc.

`5 + 4`	→	`9`	Whole number
`5 > 4`	→	`True`	Boolean

Combining booleans

True True

```
>>> 1 < 2 and 5 < 6
True
```

Both True

True False

```
>>> 1 < 2 and 5 > 6
False
```

Not both True

Combining booleans

True True

```
>>> 1 < 2  or  5 < 6
True
```

Either True

True False

```
>>> 1 < 2  or  5 > 6
True
```

Either True

Combining booleans

False　　　　False

```
>>> 1 > 2  or  5 > 6
False
```

Neither True

Negating booleans

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

True → False
False → True

```
>>> not 1 > 2
True

>>> not False
True
```

Not equal to…

```
>>> 1 == 2
False

>>> 1 != 2
True

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

Progress

“Booleans”	`True`	`False`
Combination	`and`	`or`
Negation	`not`	

Exercise

Predict whether Python will print True or False when you type the following expressions.
Then check.

1. 1 > 2 or 2 > 1
2. 1 > 2 or not 2 > 1
3. not True
4. 1 > 2 or True

3 minutes

Ambiguity?

12 + 8 / 4

?

12 + (8/4)	(12 + 8)/4
12 + 2	20/4
14	5

Standard interpretation

12 + 8 / 4

✓	✗
12 + (8/4)	(12 + 8)/4
12 + 2	20/4
14	5

Division before addition

12 + 8 / 4 — Initial expression

12 + $\boxed{8 / 4}$ — Do division first

12 + $\boxed{2}$

$\boxed{12 +}$ 2 — Do addition second

$\boxed{14}$

Precedence

Division before addition

An order of execution

“Order of precedence”

Precedence

First

`**   %   /   *   -   +`   Arithmetic

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

`not   and   or`   Logical

Last

Parentheses

Avoid confusion!

`18/3*3`	“Check the precedence rules”
`18/(3*3)`	“Ah, yes!”

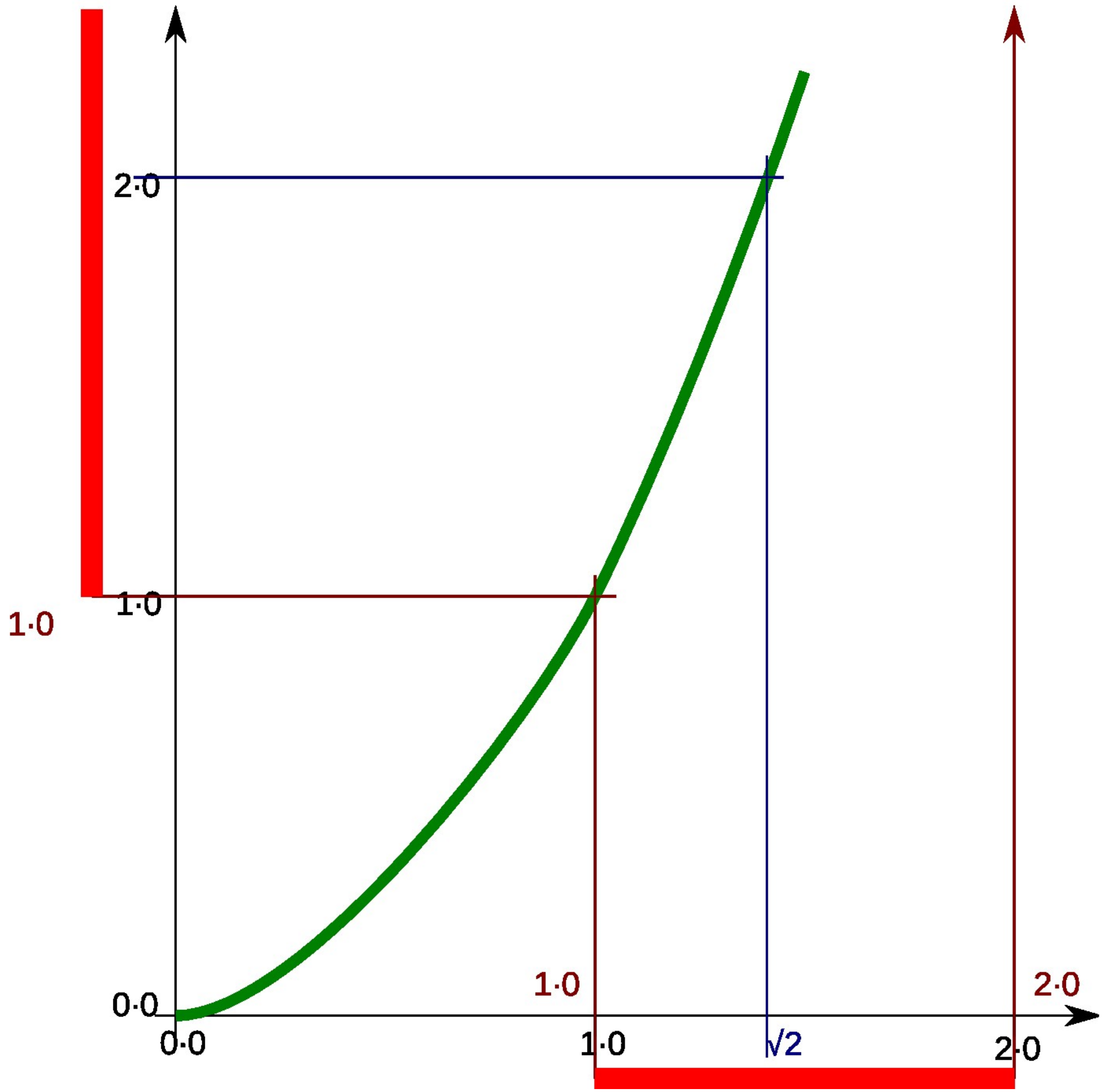
Exercise

Predict what Python will print when you type the following expressions. Then check.

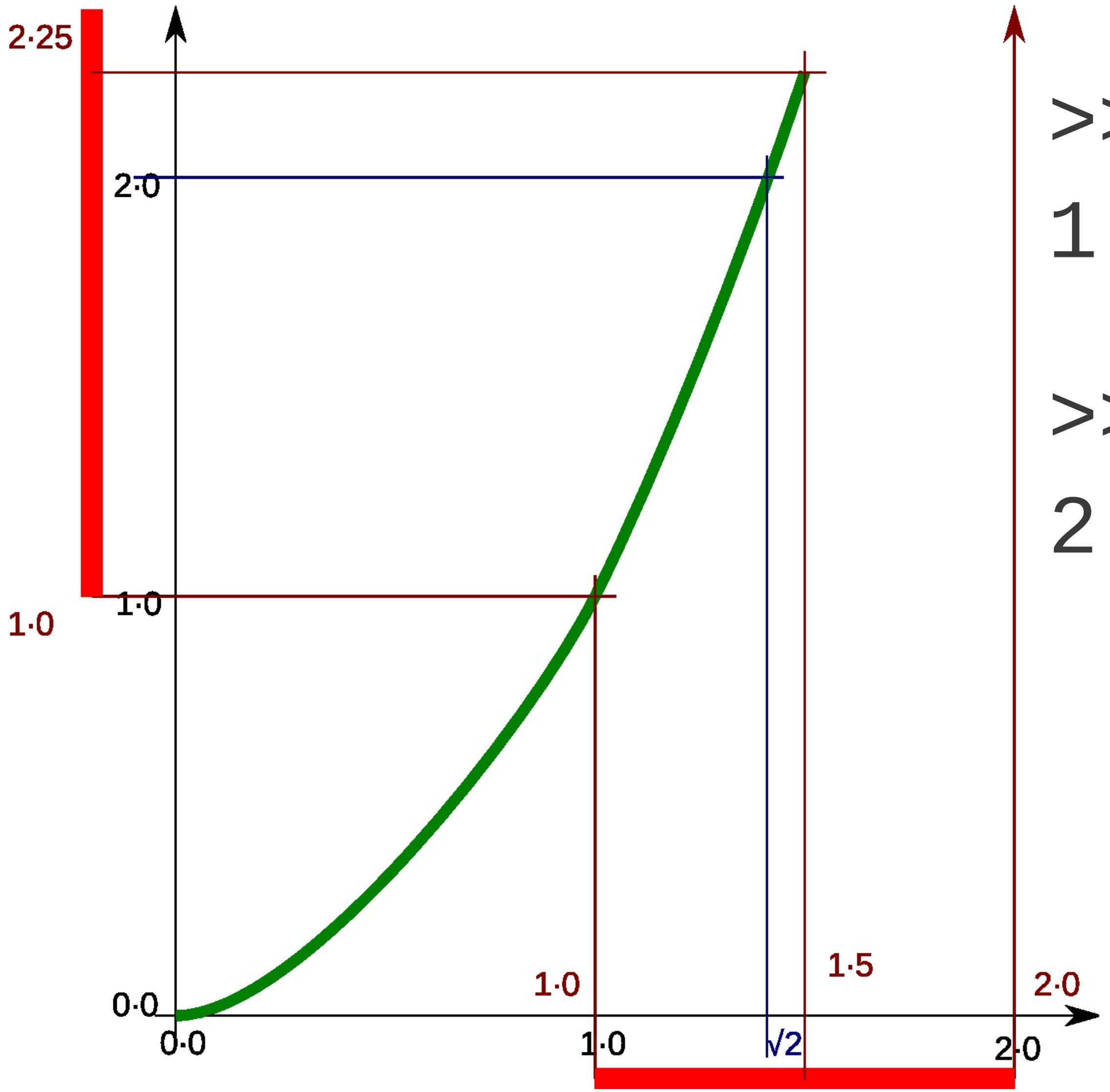
1. 12 / 3 * 4
2. 3 > 4 and 1 > 2 or 2 > 1

2 minutes

Exercise: √2 by "bisection"

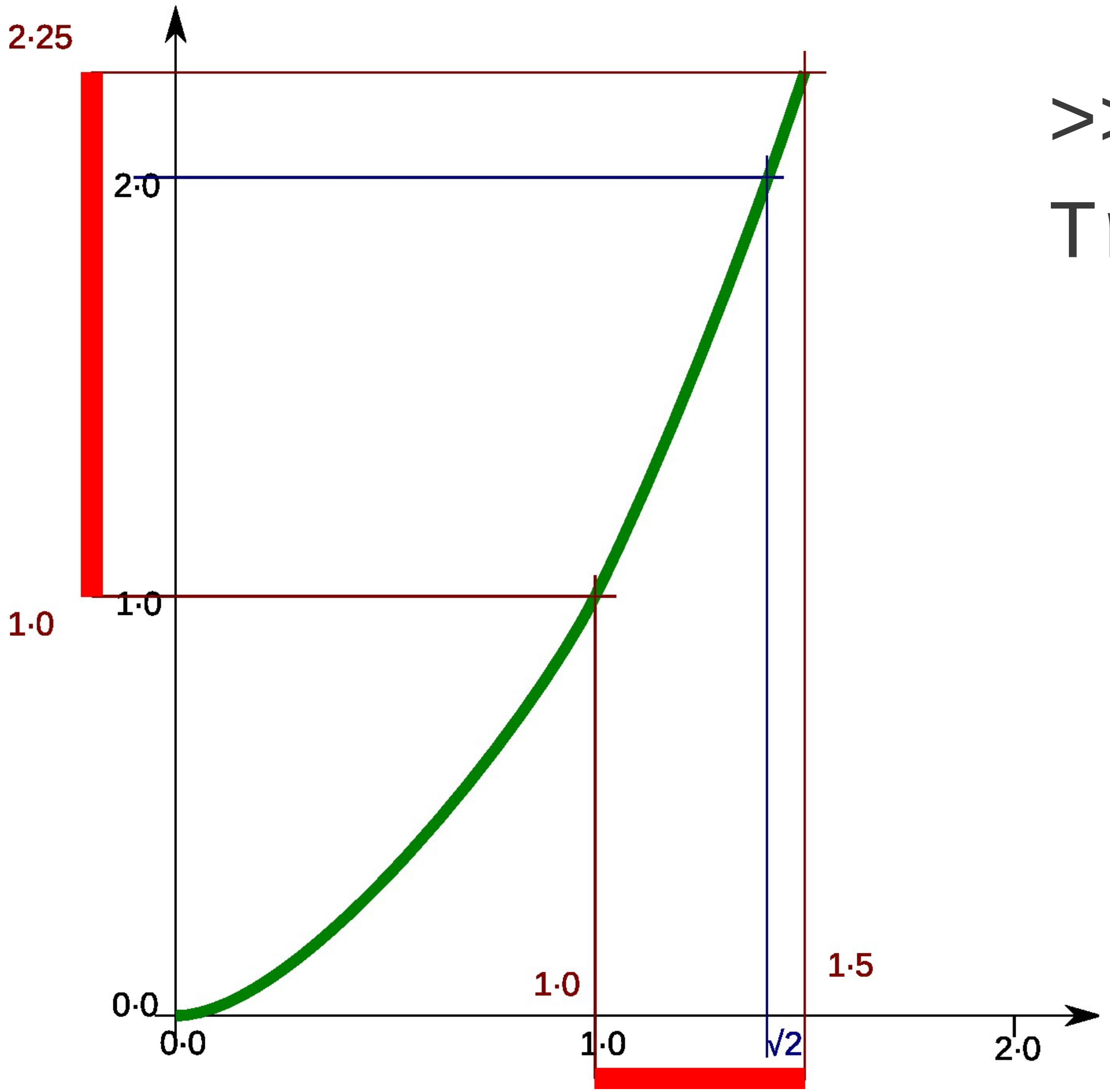


Exercise: √2 by “bisection”



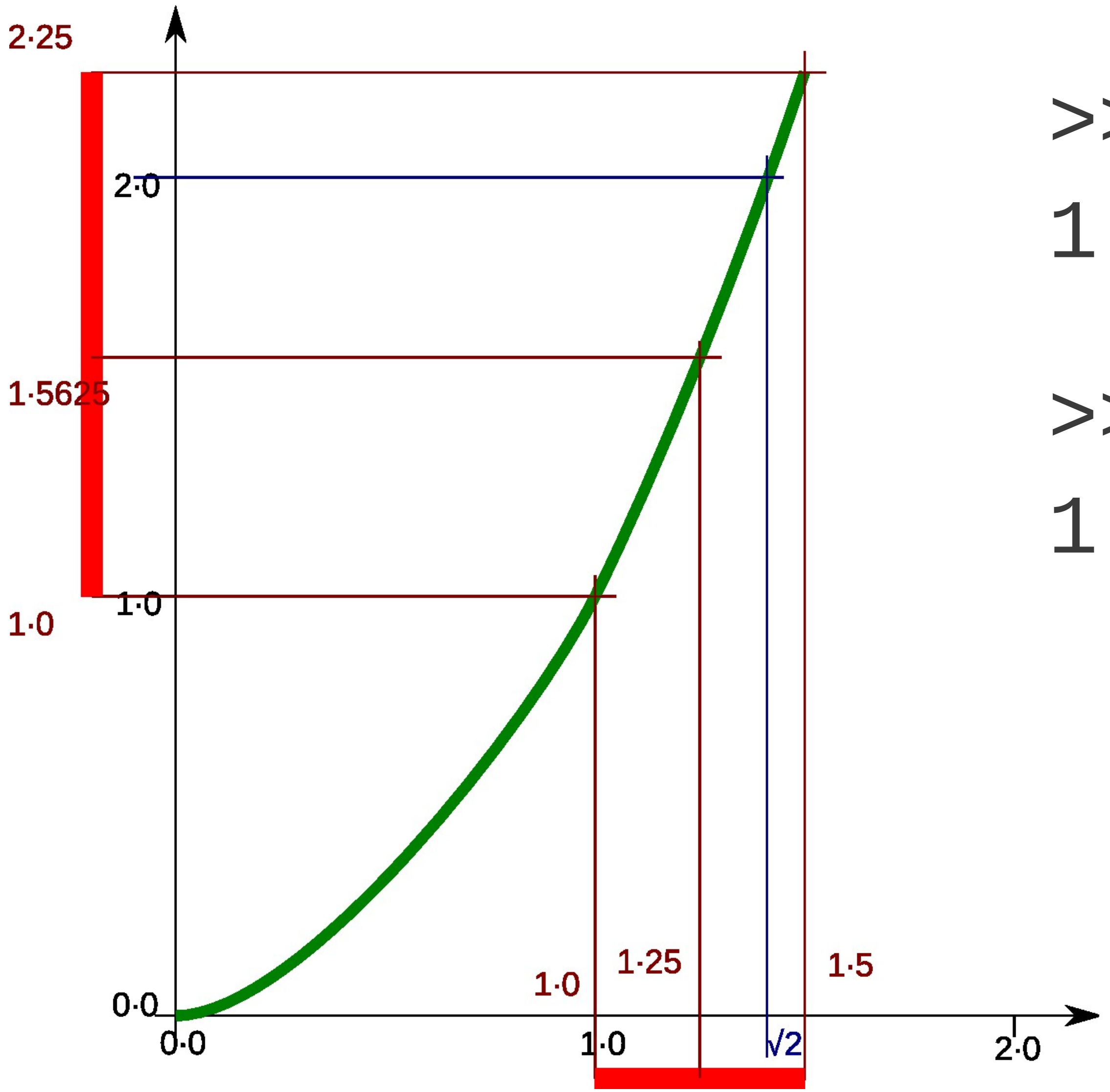

```
>>> (1.0+2.0)/2.0
1.5

>>> 1.5**2
2.25
```

Exercise: √2 by "bisection"



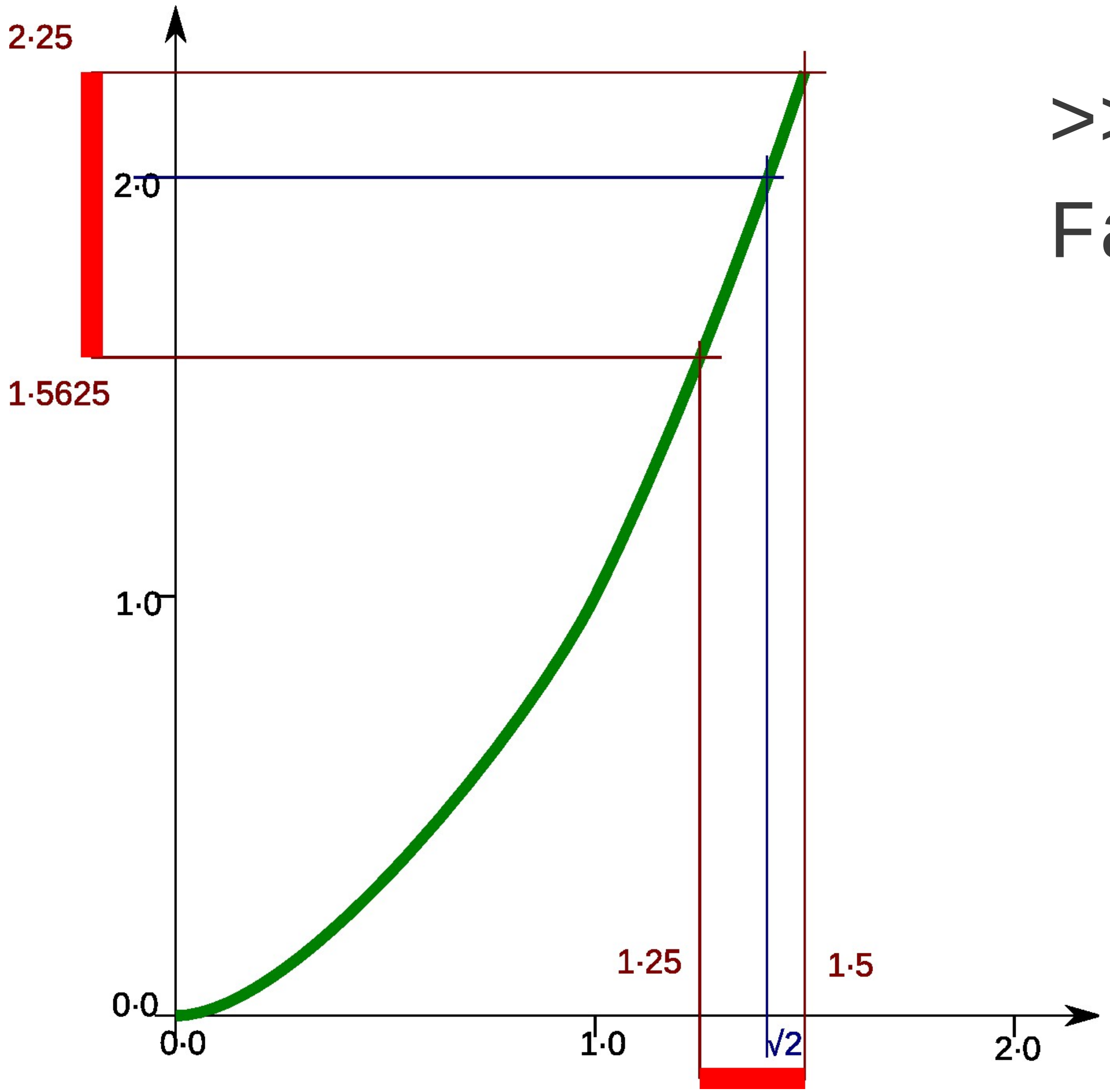

```
>>> 2.25 > 2.0
True
```

Exercise: √2 by “bisection”



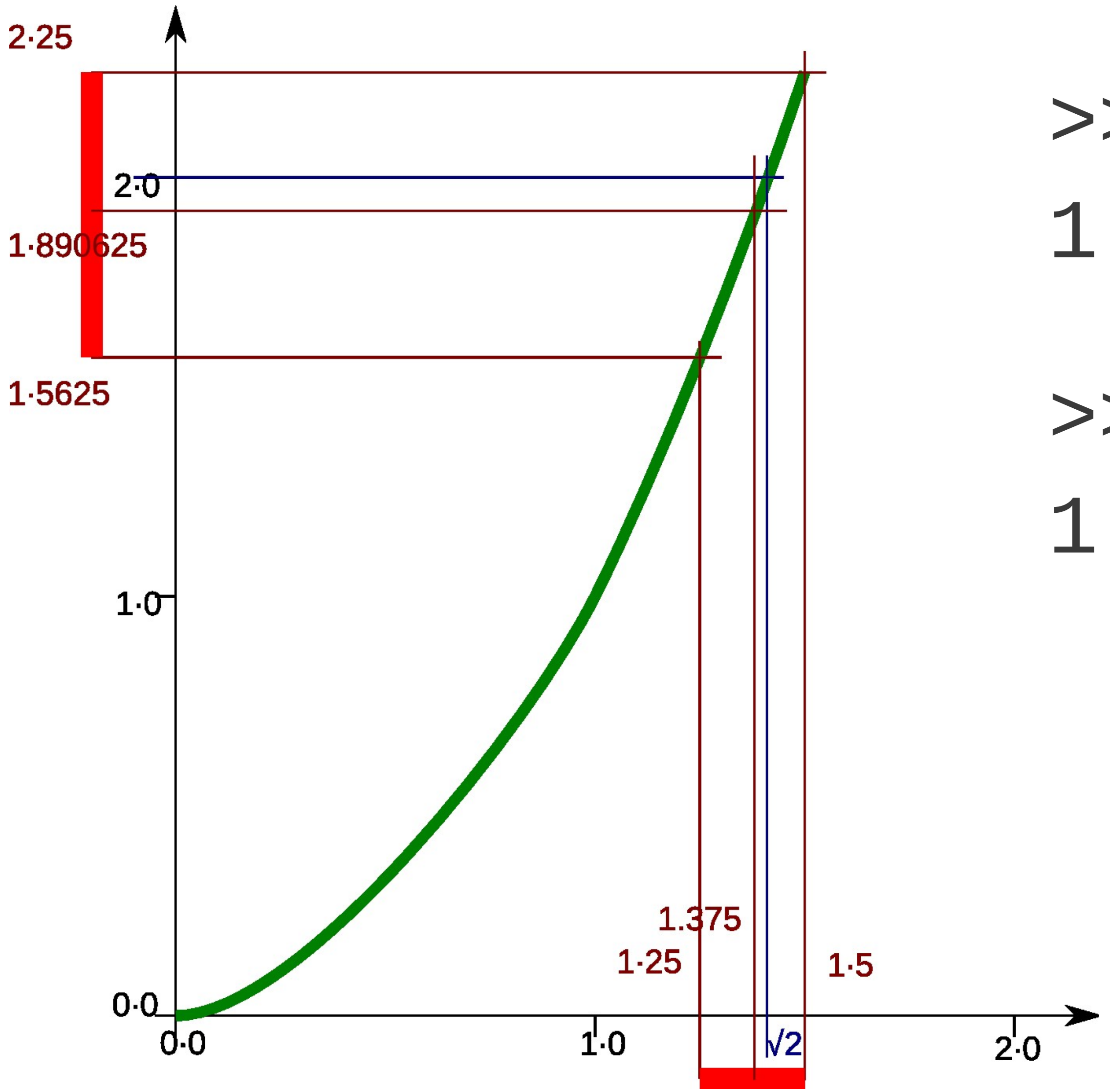

```
>>> (1.0+1.5)/2.0
1.25

>>> 1.25**2
1.5625
```

Exercise: √2 by “bisection”



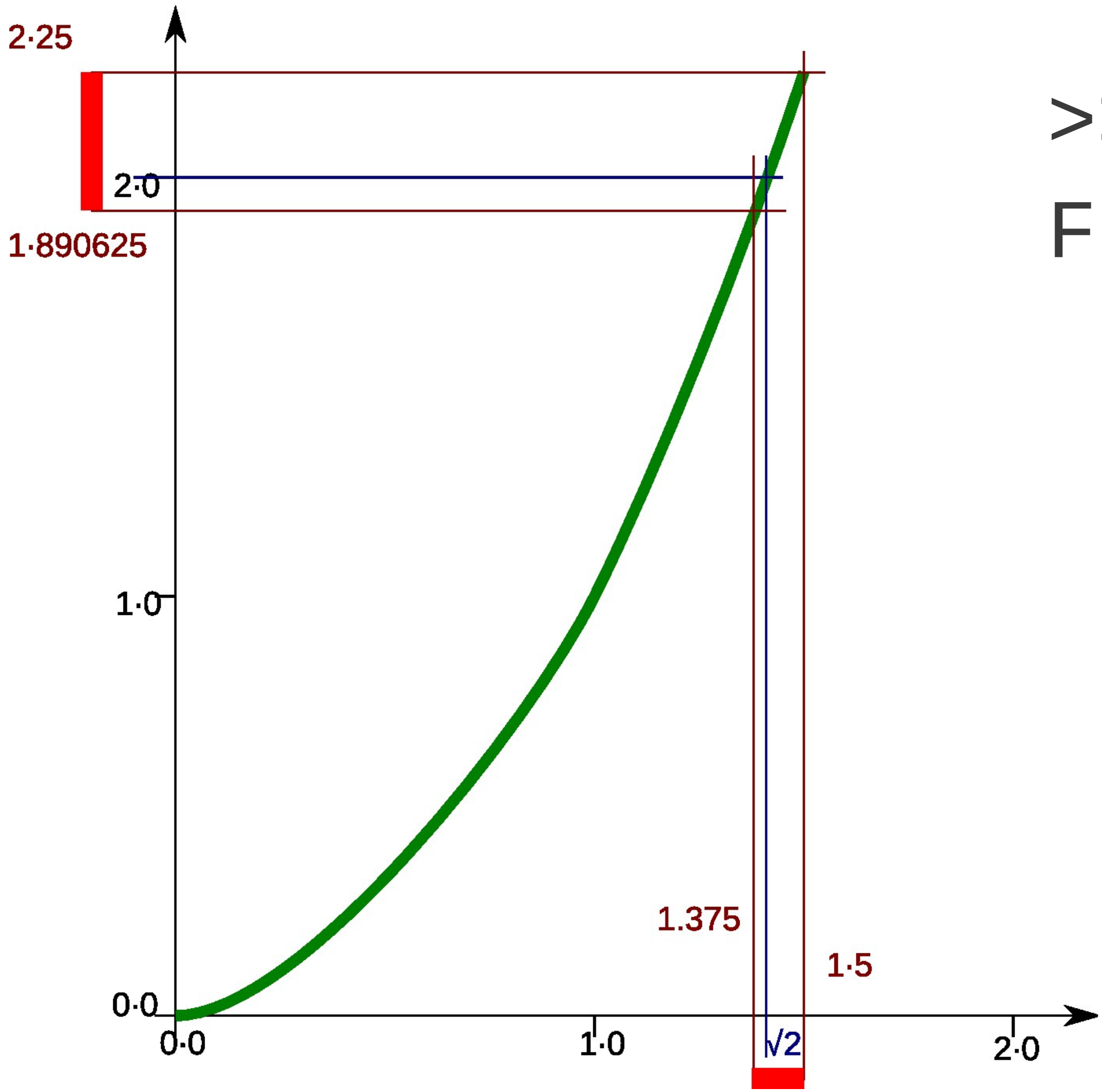

```
>>> 1.5625 > 2.0
False
```

Exercise: √2 by “bisection”




```
>>> (1.25+1.5)/2.0
1.375

>>> 1.375**2
1.890625
```

Exercise: √2 by “bisection”




```
>>> 1.890625 > 2.0
False
```

Exercise: √2 by “bisection”

Three more iterations, please.

UCS

10 minutes

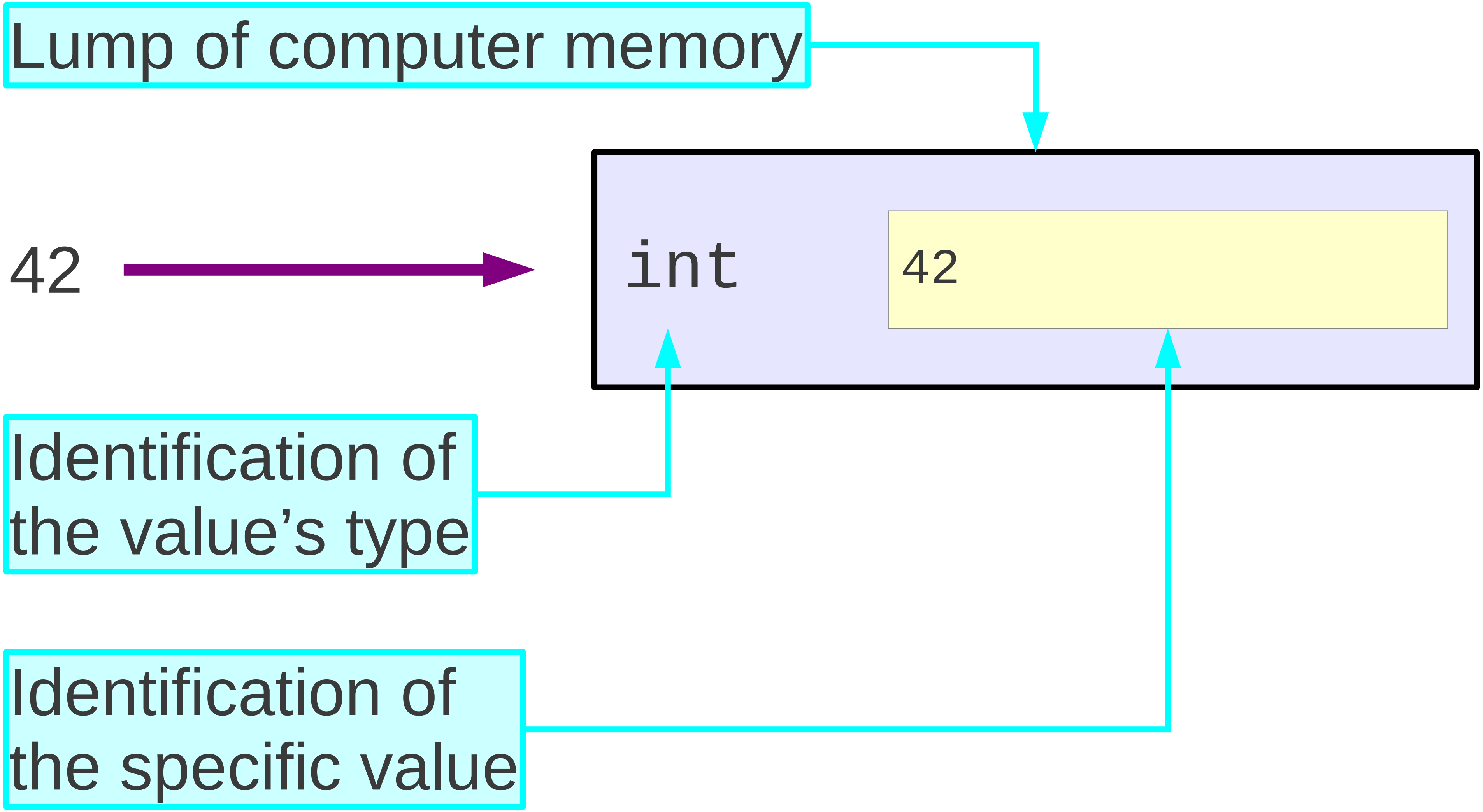
So far …

…using Python
as a *calculator*.

Now …

…use Python
as a *computer*.

How Python stores values



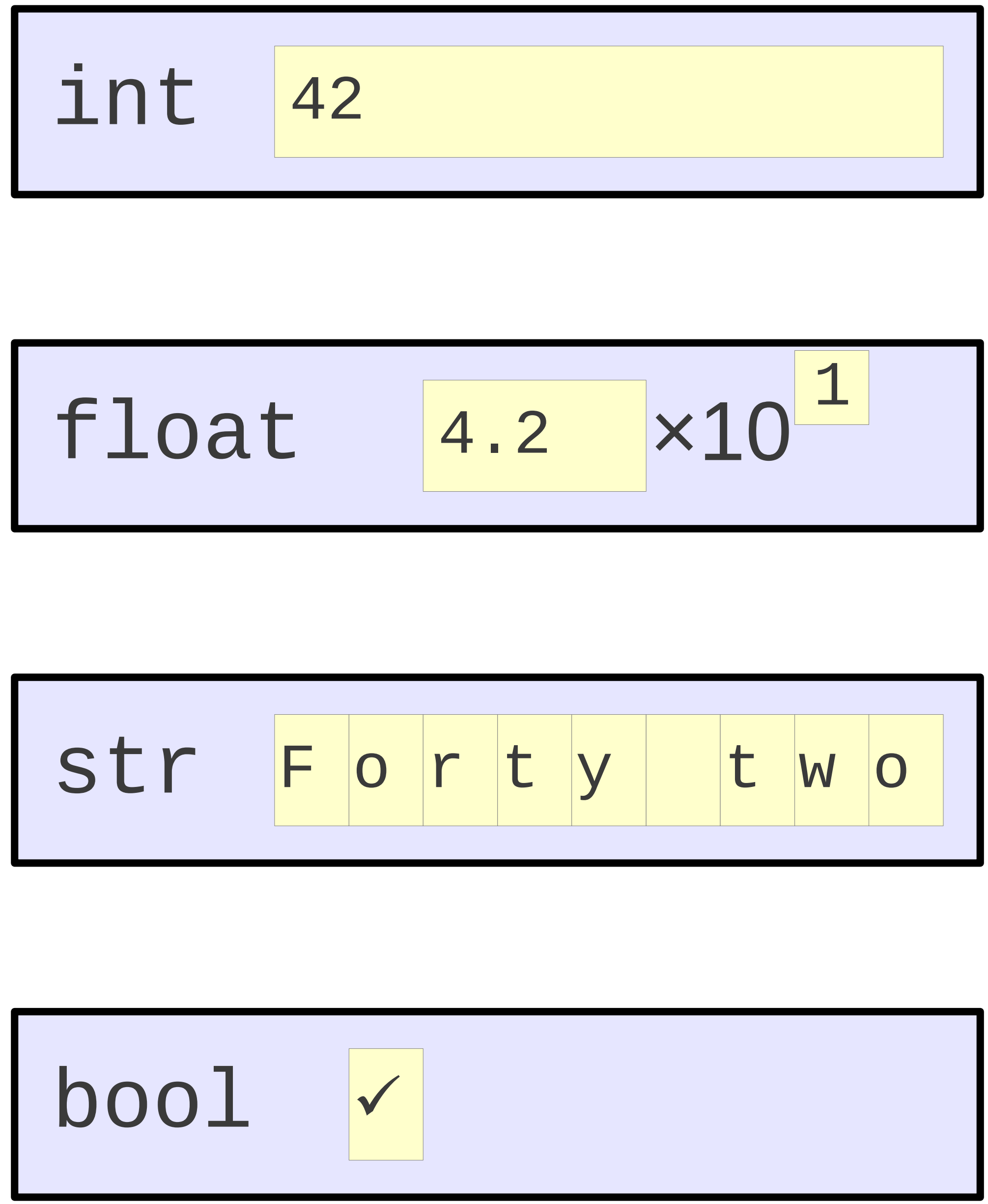
How Python stores values

42

$4{\cdot}2{\times}10^1$

'Forty two'

True



Variables

Attaching a name to a value.

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

An expression

The expression's value

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

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

```
>>> answer
42
```

The name given

The attached value returned

Variables

The name being attached

A *single* equals sign

The value being named

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

No quotes

Equals signs

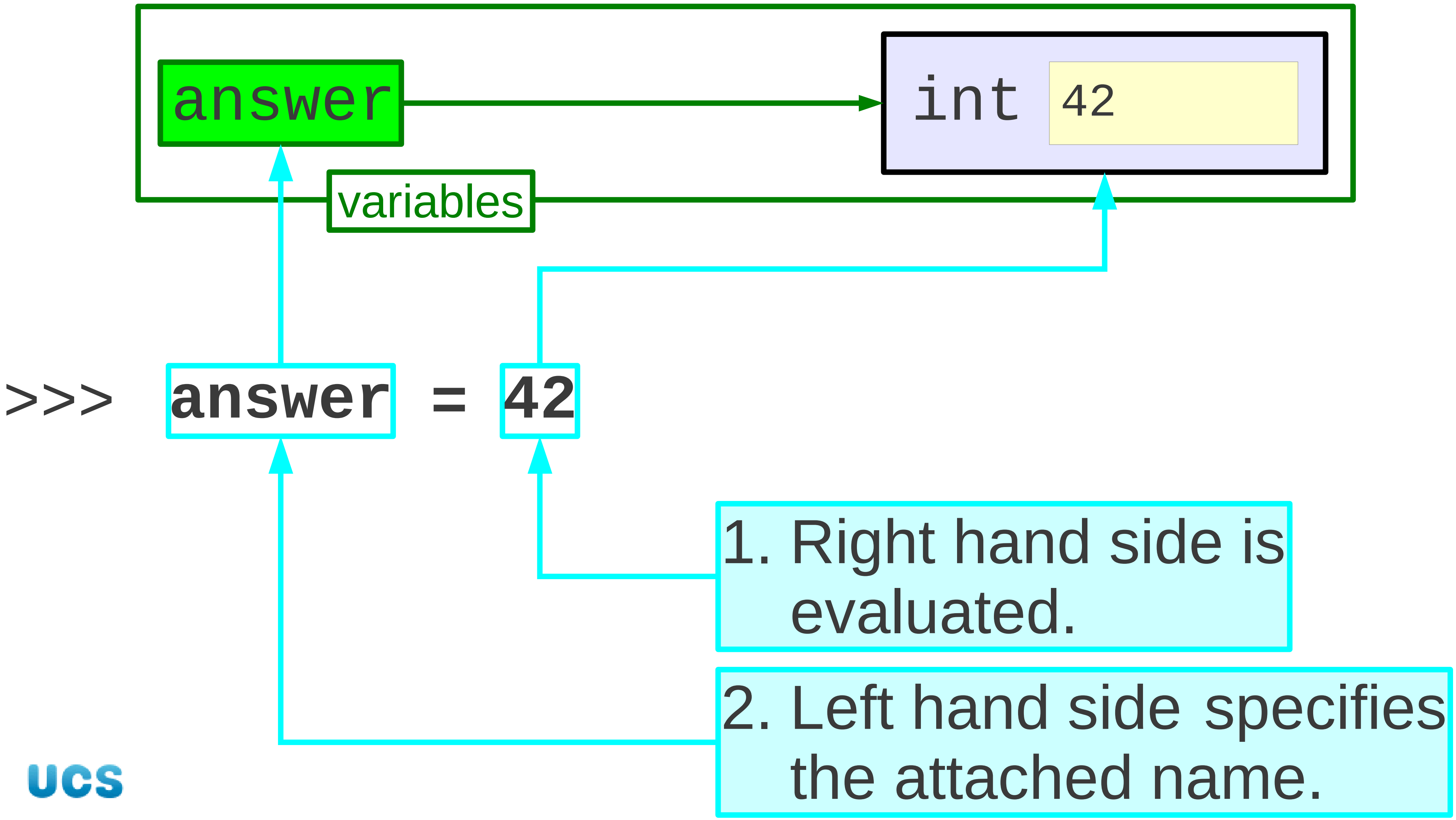
`==`

Comparison:
“are these equal?”

`=`

Assignment:
“attach the name on the left to the value on the right”

Order of activity



Example — 1

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

Simple value

UCS

Example — 2

```
>>> answer = 44 - 2
>>> answer
42
>>>
```

Calculated value

Example — 3

```
>>> answer = 42
>>> answer
42
>>> answer = answer - 2
>>> answer
40
>>>
```

“Old” value

Reattaching the name to a different value.

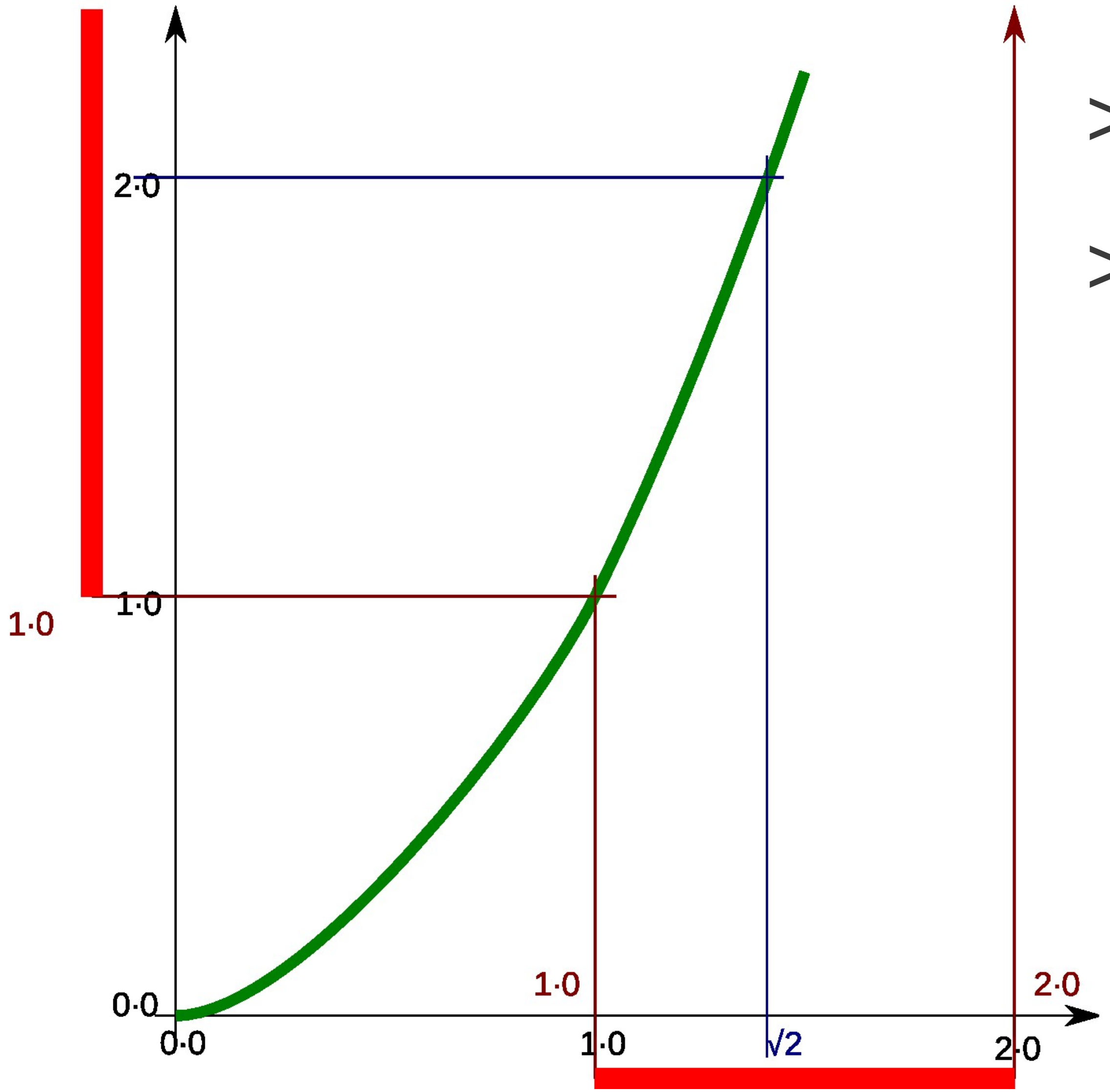
“New” value

Example — 3 in detail

`answer = answer - 2`	R.H.S. processed 1st
`answer = 42       - 2`	Old value used in R.H.S.
`answer = 40`	R.H.S. evaluated

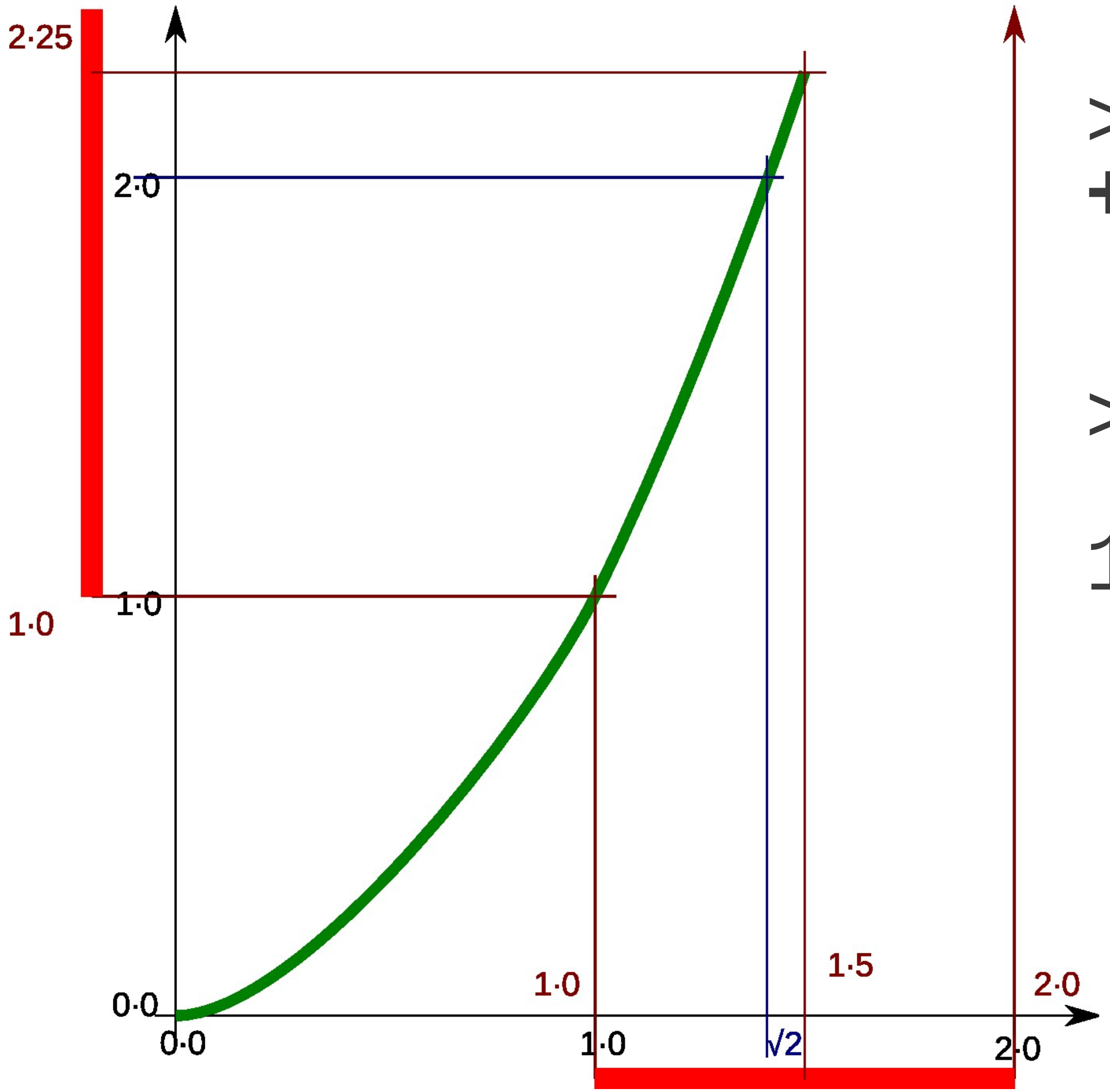
`answer = 40`	L.H.S. processed 2nd
`answer = 40`	L.H.S. name attached to value

Using named variables — 1



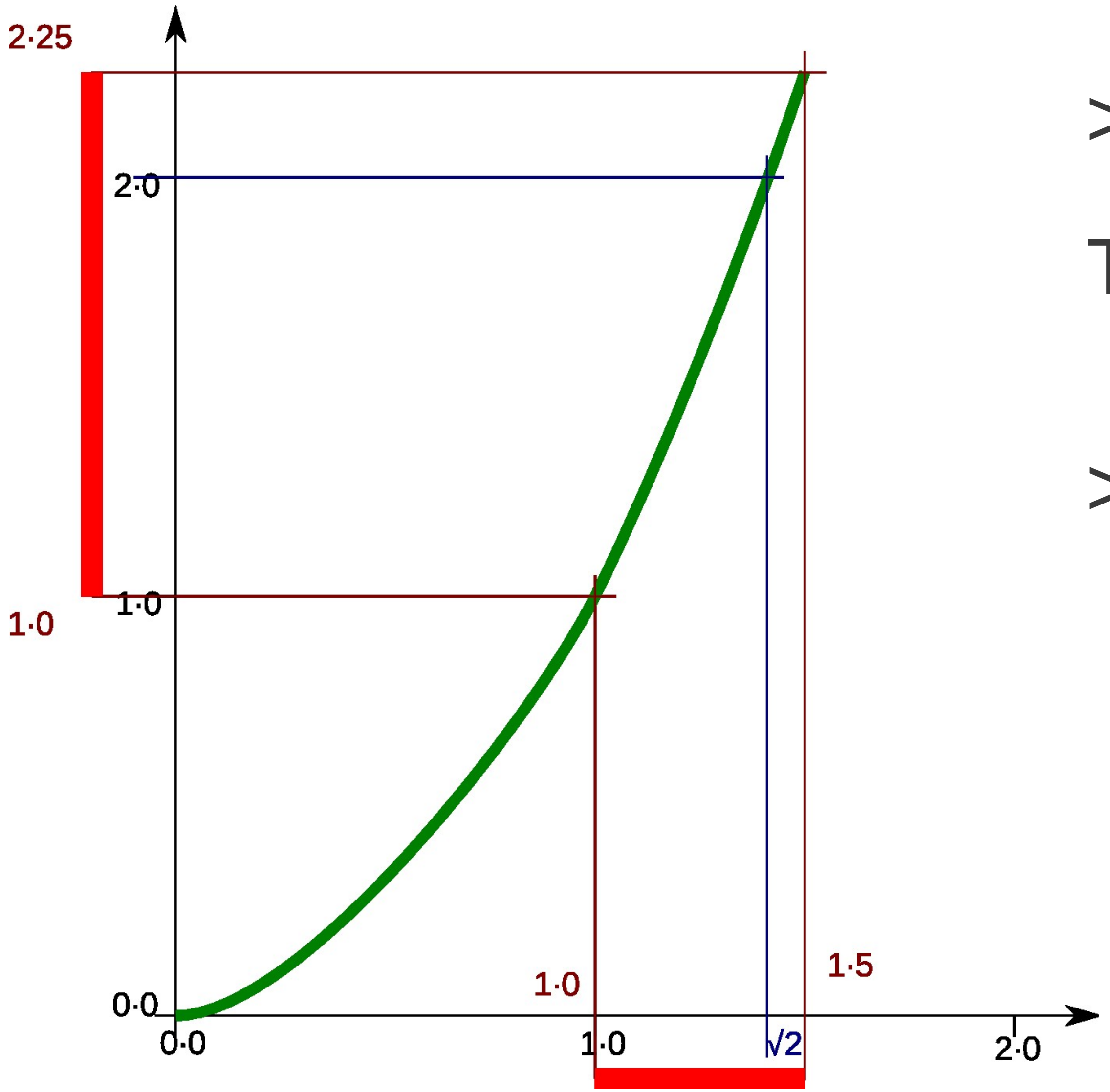

```
>>> upper = 2.0

>>> lower = 1.0
```

Using named variables — 2



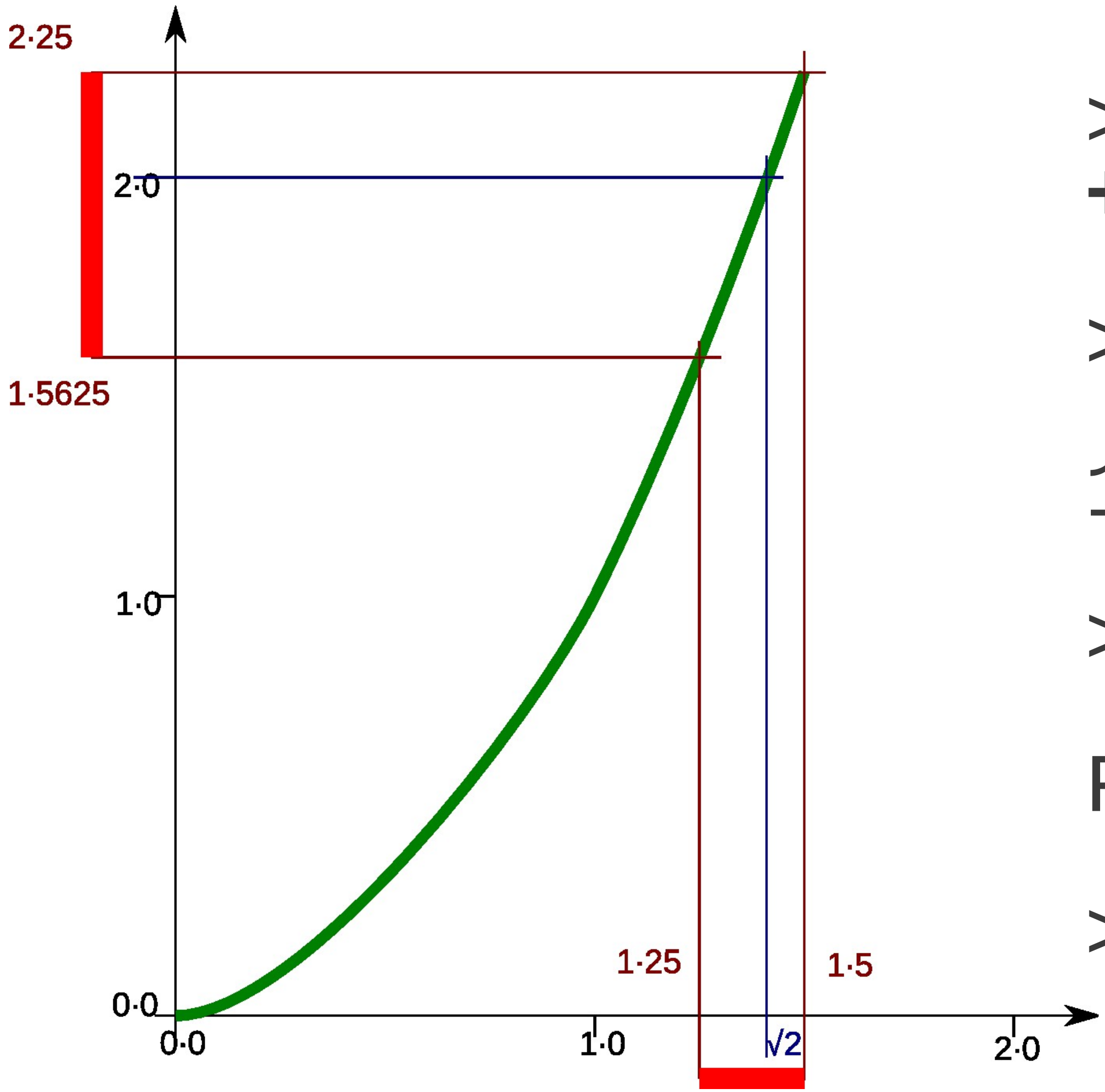

```
>>> middle = (upper
+ lower)/2.0

>>> middle

1.5
```

Using named variables — 3



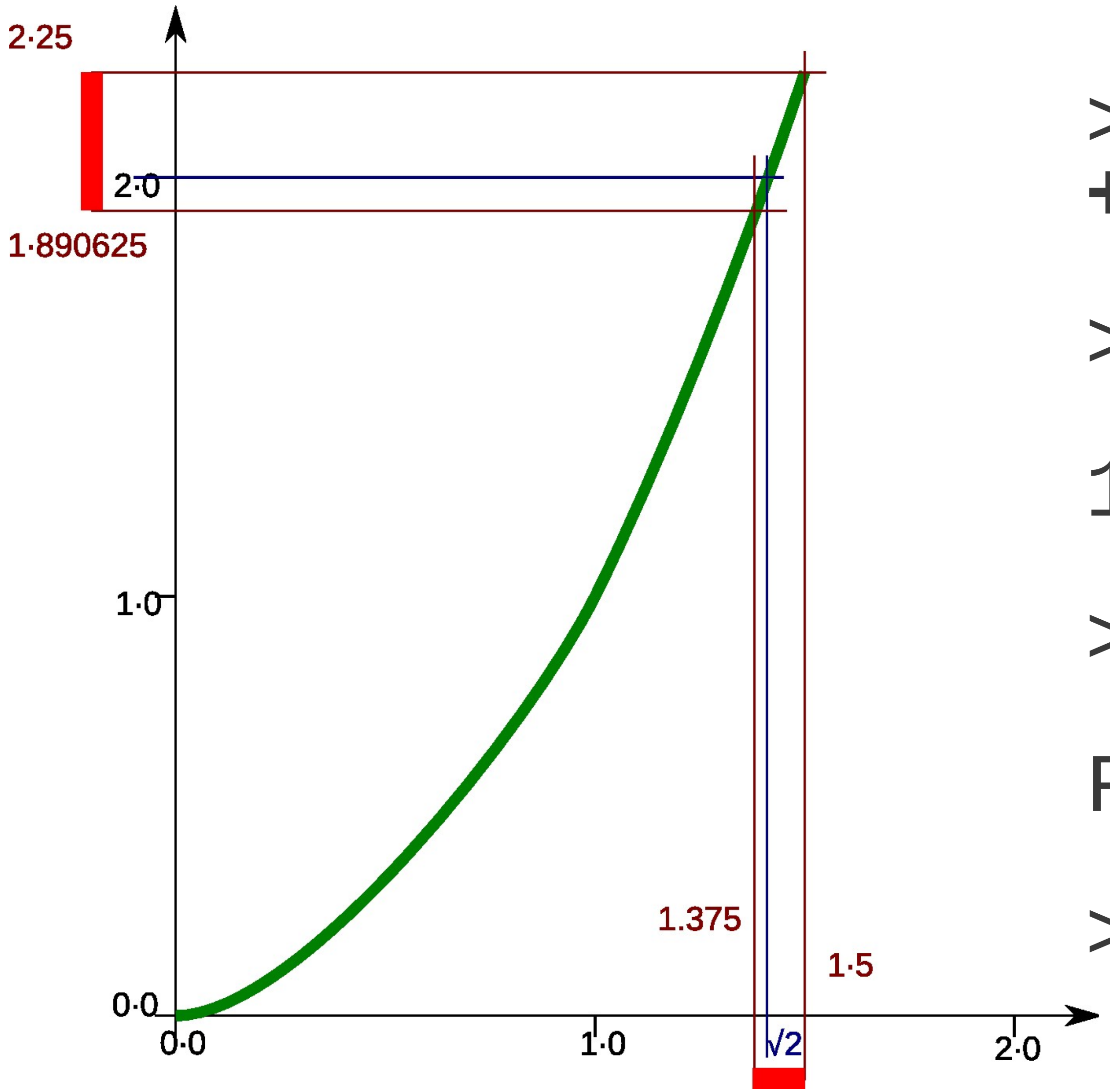

```
>>> middle**2 > 2.0
True

>>> upper = middle
```

Using named variables — 4




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

>>> middle

1.25

>>> middle**2 > 2.0

False

>>> lower = middle
```

Using named variables — 5




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

>>> middle

1.375

>>> middle**2 > 2.0

False

>>> lower = middle
```

```
upper = 2.0
lower = 1.0

middle = (upper + lower)/2.0

middle**2 > 2.0
```

?

True

False

```
upper = middle
```

```
lower = middle
```

```
print(middle)
```

UCS

Homework: √3 by bisection

Three iterations, please.

Start with

```
upper = 3.0
lower = 1.0
```

Test with

```
middle**2 > 3.0
```

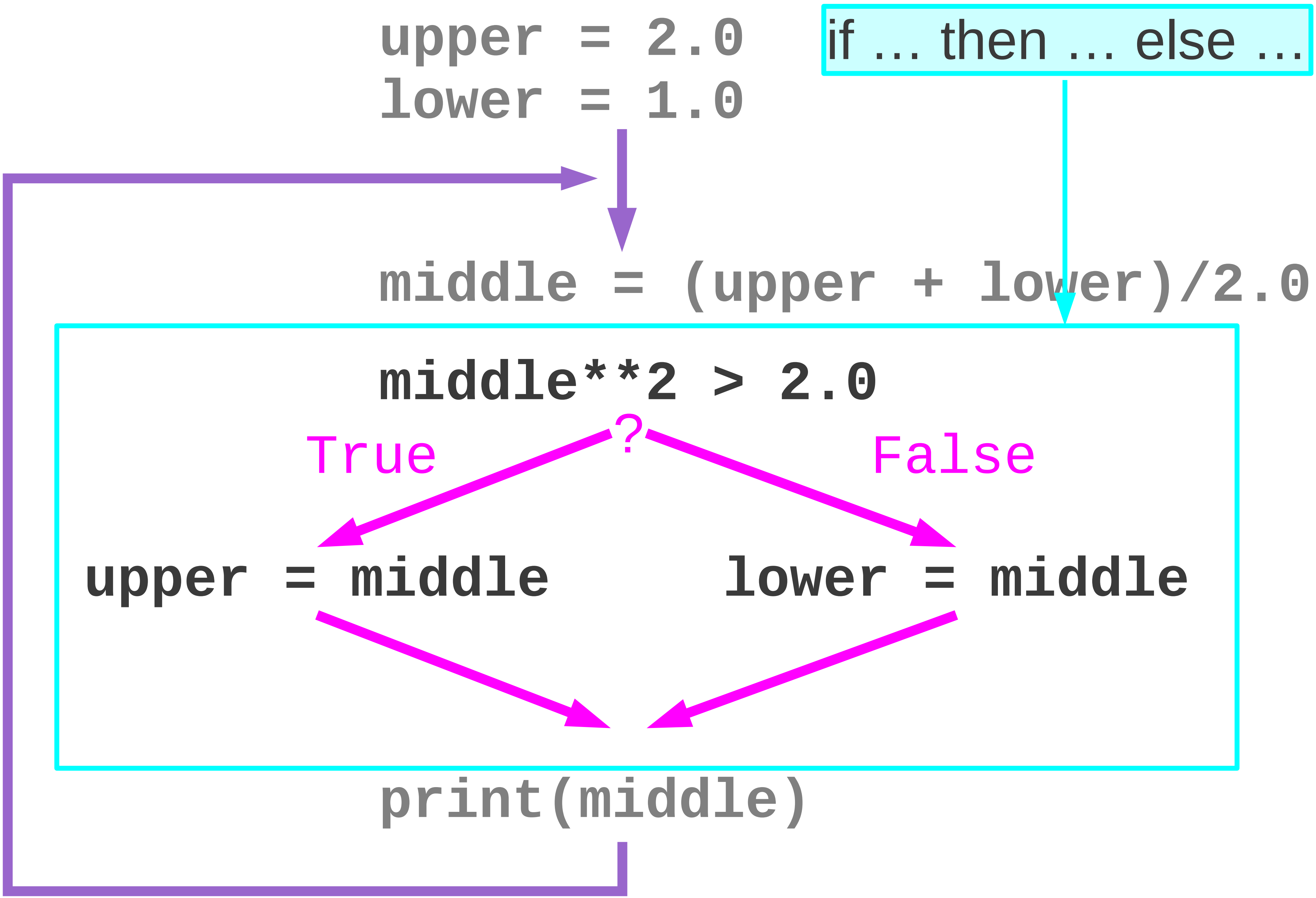
Print `middle` at the end of each stage:

```
print(middle)
```

5 minutes

UCS

Still not a computer program!

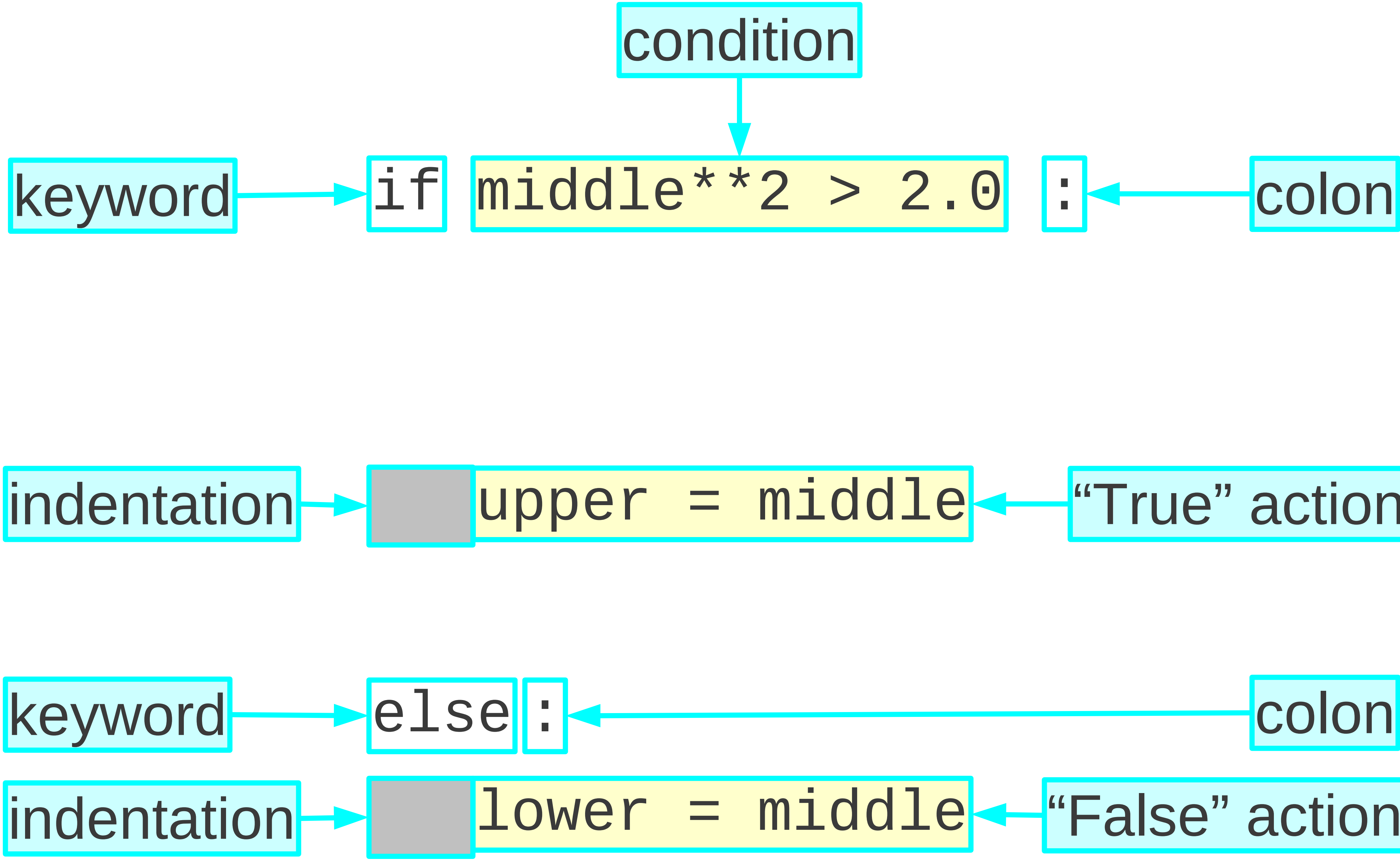

upper = 2.0
lower = 1.0
if … then … else …
middle = (upper + lower)/2.0
middle**2 > 2.0
?
True
False
upper = middle
lower = middle
print(middle)

if … then … else …

if `middle**2 > 2.0`

then `upper = middle`

else `lower = middle`


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

Example script: `middle1.py`

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

if  middle**2 > 2.0 :

    print('Moving upper')
    upper = middle

else :

    print('Moving lower')
    lower = middle

print(lower)
print(upper)
```

Example script: before

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

if middle**2 > 2.0 :

    print('Moving upper')
    upper = middle

else :

    print('Moving lower')
    lower = middle

print(lower)
print(upper)
```

Set-up prior to the test.

Example script: if…

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

if middle**2 > 2.0:

    print('Moving upper')
    upper = middle

else :

    print('Moving lower')
    lower = middle


print(lower)
print(upper)
```

keyword: “if”

condition

colon

Example script: then…

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

if middle**2 > 2.0 :

    print('Moving upper')
    upper = middle

else :

    print('Moving lower')
    lower = middle

print(lower)
print(upper)
```

Four spaces' indentation

The "True" instructions

Example script: else…

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

if  middle**2 > 2.0 :

    print('Moving upper')
    upper = middle

else :

    print('Moving lower')
    lower = middle

print(lower)
print(upper)
```

keyword: “else”

colon

Four spaces’ indentation

The “False” instructions

Example script: after

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

if  middle**2 > 2.0 :

    print('Moving upper')
    upper = middle

else :

    print('Moving lower')
    lower = middle

print(lower)
print(upper)
```

Not indented

Run regardless
of the test result.

Example script: running it

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

if  middle**2 > 2.0 :

    print('Moving upper')
    upper = middle

else :

    print('Moving lower')
    lower = middle

print(lower)
print(upper)
```

Unix prompt

```
$ python middle1.py

Moving upper
1.0
1.5

$
```

Progress

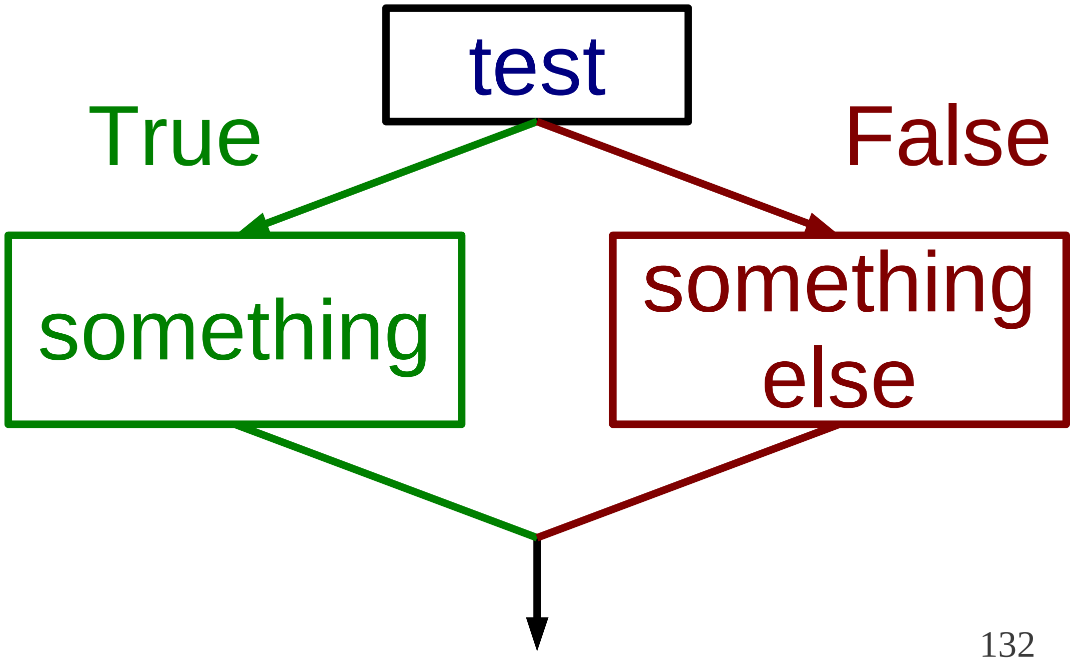
Run a test

Do something
if it succeeds.

Do something
else if it fails.

Colon…Indentation

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



Exercise

Four short Python scripts:

`ifthenelse1.py`

`ifthenelse2.py`

`ifthenelse3.py`

`ifthenelse4.py`

1. Read the file.
2. Predict what it will do.
3. Run it.

5 minutes

UCS

```
upper = 2.0
lower = 1.0
```

looping

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

```
middle**2 > 2.0
?
True                    False
upper = middle          lower = middle
```

```
print(middle)
```

Repeating ourselves

Run the script

Read the results

Edit the script

Not the way to do it!

Repeating ourselves

Looping for ever?

Keep going while…

…then stop.

```
while condition :
    action1
    action2
afterwards
```

UCS

while *vs.* until

Repeat until…

`number == 0`

`upper - lower < target`

`condition`

Repeat while…

`number != 0`

`upper - lower >= target`

`not` *`condition`*

Example script

```
number = 1
limit = 1000

while number < limit :

    print(number)
    number = number * 2

print('Finished!')
```

doubler1.py

Example script: before

```
number = 1
limit = 1000

while number < limit :

    print(number)
    number = number * 2

print('Finished!')
```

doubler1.py

Set-up prior to the loop.

Example script: while…

```
number = 1
limit = 1000

while number < limit :

    print(number)
    number = number * 2

print('Finished!')
```

keyword: “while”

condition

colon

doubler1.py

Example script: loop body

```
number = 1
limit = 1000

while number < limit :

    print(number)
    number = number * 2

print('Finished!')
```

Four spaces' indentation

loop body

doubler1.py

Example script: after

```
number = 1
limit = 1000

while number < limit :

    print(number)
    number = number * 2

print('Finished!')
```

doubler1.py

Not indented

Run after the looping is finished.

Example script: running it

```
number = 1
limit = 1000

while number < limit :

    print(number)
    number = number * 2

print('Finished!')
```

```
> python doubler1.py

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

UCS

Progress

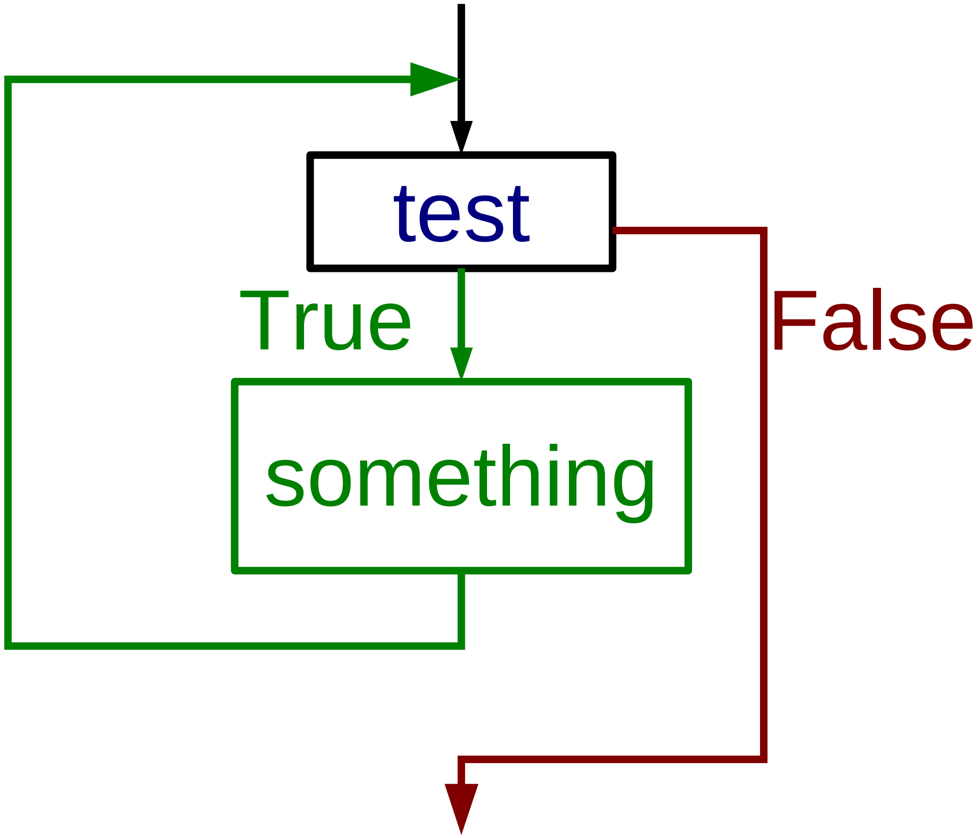
Run a test

Do something
if it succeeds.

Finish if
it fails.

Go back to the test.

```
while test :
    something
```



Exercise

Four short Python scripts:

`while1.py`

`while2.py`

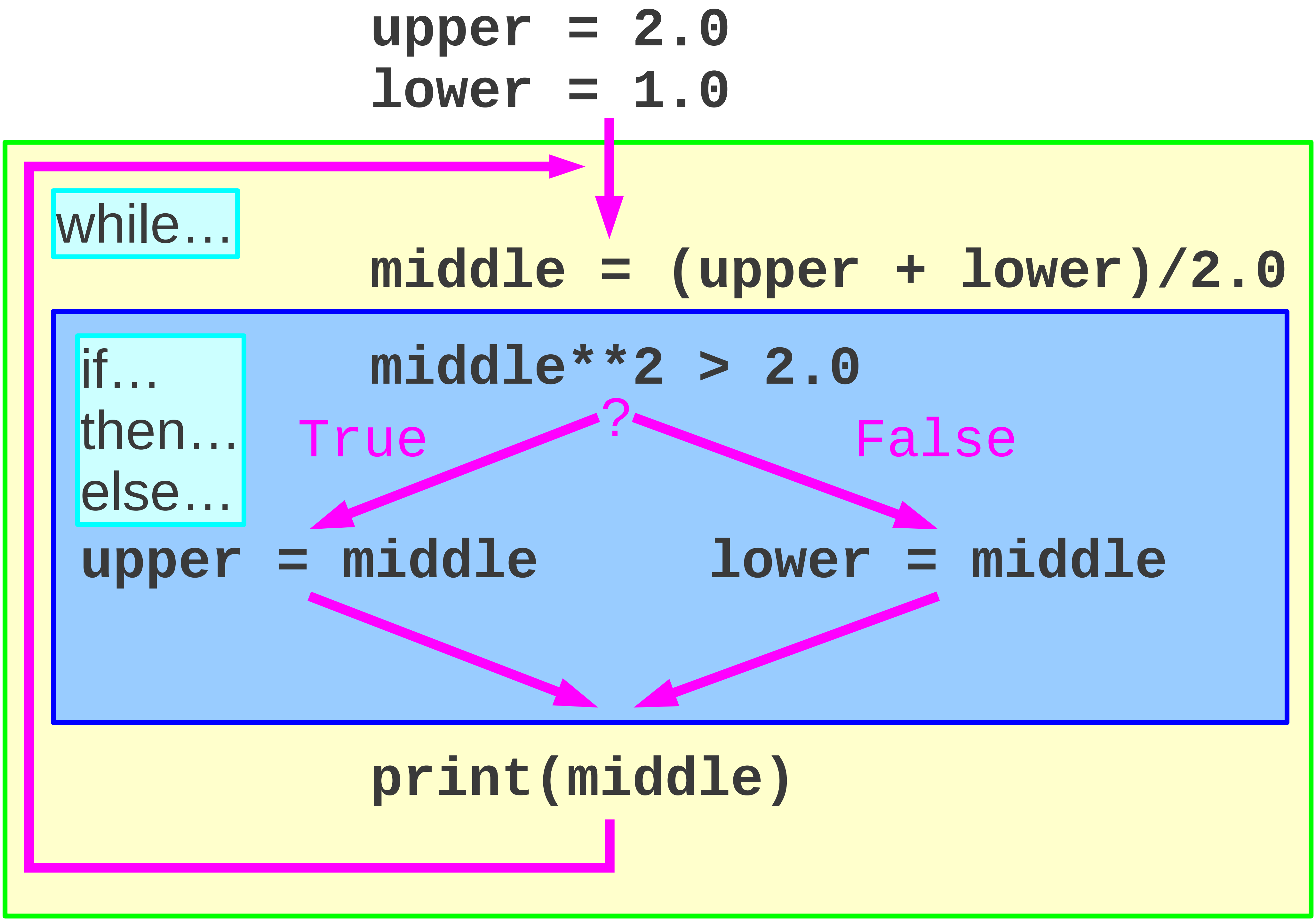
`while3.py`

`while4.py`

1. Read the file.
2. Predict what it will do.
3. Run it.

n.b. [Ctrl]+[C] will kill a script that won't stop on its own.

5 minutes


upper = 2.0
lower = 1.0
while...
middle = (upper + lower)/2.0
if...
then...
else...
middle**2 > 2.0
?
True
False
upper = middle
lower = middle
print(middle)

Combining while… and if…

if…then…else… *inside* while…

Each if…then…else… improves the approximation

How many if…then…else… repeats should we do?

What's the while… test?

Writing the while… test

Each if…then…else… improves the approximation

How much do we want it improved?

How small do we want the interval?

```
upper - lower
```

Writing the while... test

What is the interval? `upper - lower`

How small do we want the interval? `1.0e-15`

Keep going while the interval is too big:

```
while upper - lower > 1.0e-15 :
```

```
lower = 1.0
upper = 2.0

while upper - lower > 1.0e-15 :

    middle = (upper+lower)/2.0

                ?

print(middle)
```

approximation is too coarse

Single indentation

if…then…else…

No indentation

```
lower = 1.0
upper = 2.0

while upper - lower > 1.0e-15 :

    middle = (upper+lower)/2.0

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

print(middle)
```

Double indentation

Double indentation

Running the script

```
lower = 1.0
upper = 2.0

while upper - lower > 1.0e-15:


    middle = (upper+lower)/2.0

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

print(middle)
```

```
> python root2.py

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

Indentation

c.f. “legalese”

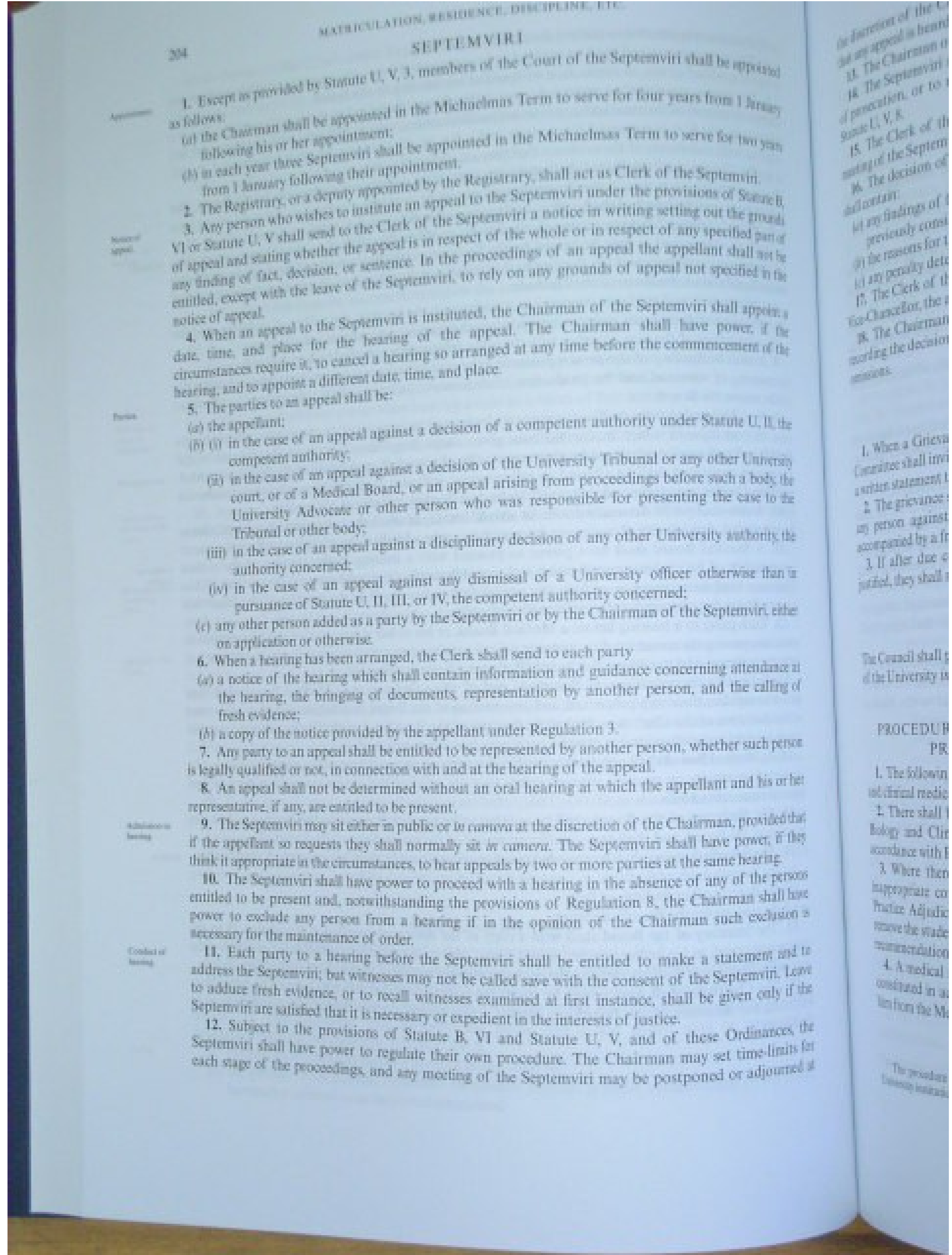
§5(b)(ii)

Other languages…

`{…}`

`IF…END IF`

`if…fi, do…done`



MATRICULATION, RESIDENCE, DISCIPLINE, ETC.

204 SEPTEMVIRI

1. Except as provided by Statute U, V, 3, members of the Court of the Septemviri shall be appointed as follows:
(a) the Chairman shall be appointed in the Michaelmas Term to serve for four years from 1 January following his or her appointment;
(b) in each year three Septemviri shall be appointed in the Michaelmas Term to serve for two years from 1 January following their appointment.

2. The Registrary, or a deputy appointed by the Registrary, shall act as Clerk of the Septemviri.

3. Any person who wishes to institute an appeal to the Septemviri under the provisions of Statute B, VI or Statute U, V shall send to the Clerk of the Septemviri a notice in writing setting out the grounds of appeal and stating whether the appeal is in respect of the whole or in respect of any specified part of any finding of fact, decision, or sentence. In the proceedings of an appeal the appellant shall not be entitled, except with the leave of the Septemviri, to rely on any grounds of appeal not specified in the notice of appeal.

4. When an appeal to the Septemviri is instituted, the Chairman of the Septemviri shall appoint a date, time, and place for the hearing of the appeal. The Chairman shall have power, if the circumstances require it, to cancel a hearing so arranged at any time before the commencement of the hearing, and to appoint a different date, time, and place.

5. The parties to an appeal shall be:
(a) the appellant;
(b) (i) in the case of an appeal against a decision of a competent authority under Statute U, II, the competent authority;
(ii) in the case of an appeal against a decision of the University Tribunal or any other University court, or of a Medical Board, or an appeal arising from proceedings before such a body, the University Advocate or other person who was responsible for presenting the case to the Tribunal or other body;
(iii) in the case of an appeal against a disciplinary decision of any other University authority, the authority concerned;
(iv) in the case of an appeal against any dismissal of a University officer otherwise than in pursuance of Statute U, II, III, or IV, the competent authority concerned;
(c) any other person added as a party by the Septemviri or by the Chairman of the Septemviri, either on application or otherwise.

6. When a hearing has been arranged, the Clerk shall send to each party
(a) a notice of the hearing which shall contain information and guidance concerning attendance at the hearing, the bringing of documents, representation by another person, and the calling of fresh evidence;
(b) a copy of the notice provided by the appellant under Regulation 3.

7. Any party to an appeal shall be entitled to be represented by another person, whether such person is legally qualified or not, in connection with and at the hearing of the appeal.

8. An appeal shall not be determined without an oral hearing at which the appellant and his or her representative, if any, are entitled to be present.

9. The Septemviri may sit either in public or *in camera* at the discretion of the Chairman, provided that if the appellant so requests they shall normally sit *in camera*. The Septemviri shall have power, if they think it appropriate in the circumstances, to hear appeals by two or more parties at the same hearing.

10. The Septemviri shall have power to proceed with a hearing in the absence of any of the persons entitled to be present and, notwithstanding the provisions of Regulation 8, the Chairman shall have power to exclude any person from a hearing if in the opinion of the Chairman such exclusion is necessary for the maintenance of order.

11. Each party to a hearing before the Septemviri shall be entitled to make a statement and to address the Septemviri; but witnesses may not be called save with the consent of the Septemviri. Leave to adduce fresh evidence, or to recall witnesses examined at first instance, shall be given only if the Septemviri are satisfied that it is necessary or expedient in the interests of justice.

12. Subject to the provisions of Statute B, VI and Statute U, V, and of these Ordinances, the Septemviri shall have power to regulate their own procedure. The Chairman may set time-limits for each stage of the proceedings, and any meeting of the Septemviri may be postponed or adjourned at

UCS

Indentation: level 1

```
lower = 1.0
upper = 2.0

while upper - lower > 1.0e-15 :

    middle = (upper+lower)/2.0

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

print(middle)
```

Colon starts the block

Indentation marks the extent of the block.

Unindented line
End of block

UCS

Indentation: level 2

```
lower = 1.0
upper = 2.0

while upper - lower > 1.0e-15 :


        middle = (upper+lower)/2.0

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

print(middle)
```

Colon…indentation

“else” unindented

Colon…indentation

Arbitrary nesting

Not just two levels deep

As deep as you want

Any combination

if… inside while…

while… inside if…

if… inside if…

while… inside while…

e.g. if... inside if...

```
number = 20

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

Progress

colon…indentation

Indented blocks

Nested constructs

Levels of indentation

Exercise

Write a script from scratch: `collatz.py`

1. Start with `number` set to `7`.

2. Repeat until `number` is `1`.

3. Each loop:

3a. If `number` is even, change it to `number/2`.

3b. If `number` is odd, change it to `3*number+1`.

3c. Print `number`.

15 minutes

Comments

Reading Python syntax

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

“What does the code do?”

Calculate the mid-point.

“*Why* does the code do that?”

Need to know the square of the mid-point’s value to compare it with 2.0 whose root we’re after.

Comments

- The “hash” character. a.k.a. “sharp”
  “pound”
  “number”
- Lines starting with “#” are ignored
  Partial lines too.

Comments — explanation

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

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

lower = 1.0 # Initial bounds.
upper = 2.0

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

UCS

Comments — authorship

```
# (c) Bob Dowling, 2010
# Released under the FSF GPL v3

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

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

lower = 1.0 # Initial bounds.
upper = 2.0
    …
```

Comments — source control

```
# (c) Bob Dowling, 2010
# Released under the FSF GPL v3

# $Id: root2.py,v 1.1 2010/05/20 10:43:43 rjd4 $

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

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

…

Comments — logging

```
# (c) Bob Dowling, 2010
# Released under the FSF GPL v3

# $Id: root2.py,v 1.2 2010/05/20 10:46:46 rjd4 $

# $Log: root2.py,v $
# Revision 1.2  2010/05/20 10:46:46  rjd4
# Removed intermediate print lines.
#

# Set the initial bounds of the interval.  Then
# refine it by a factor of two each iteration by
#   …
```

Comments

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

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

Exercise

1. Comment your script: `collatz.py`

Author	`# Bob Dowling`
Date	`# 2010-05-20`
Purpose	`# This script` `# illustrates …`

2. Then check it still works!

3 minutes

Lists

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

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

[0.0, 1.5707963267948966, 3.141592653589793]

Lists — getting it wrong

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

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

Repetition of “print”

Inflexible

Lists — getting it right

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

1. Create a list of the element names

2. Print each entry in the list

Creating a list

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

Here's a list

Yes, that's a list

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

Attaching a name to a variable.

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

Using the name

Anatomy of a list

Square brackets at end

Individual element

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

Elements separated by commas

Square brackets in Python

[…] **Defining literal lists**

e.g.

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

Order of elements

No “reordering”

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

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

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

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

Repetition

No “uniqueness”

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


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

Concatenation — 1

“+” used to join lists.

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

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

UCS

Concatenation — 2

“3” appears twice

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

“3” appears twice

Empty list

```
>>> []
[]

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

>>> [] + []
[]
```

Progress

Lists	[23, 29, 31, 37, 41]
Shown with square brackets	
Elements separated by commas	
Concatenation	[23, 29]+[31, 37, 41]
Empty list	[ ]

Exercise

Predict what these will create. Then check.

1. `[] + ['a', 'b'] + []`
2. `['c', 'd'] + ['a', 'b']`
3. `[2, 3, 5, 7] + [7, 11, 13, 17, 19]`

2 minutes

How long is the list?

Function name

```
>>>  len([10, 20, 30])
3
```

Round brackets

How long is a string?

Same function

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

!

Recall:

Quotes say “this is a string”.

They are not part of the string.

How long is a *number*?

```
>>> len(42)
```

Error message

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

Numbers don't have a "length".

Our first look at a *function*

```
>>> len([10, 20, 30])
3
```

Function name

Round brackets

Function “argument”

One argument

“Returned” value

Progress

Length of a list:	Number of elements
Length of a string:	Number of characters
Length function:	`len()`

Exercise: lengths of strings

1. Predict what these Python snippets will return.
2. Then try them.

(a) `len('Goodbye, world!')`

(b) `len('Goodbye, ' + 'world!')`

(c) `len('Goodbye, ') + len('world!')`

UCS

3 minutes for both

Exercise: lengths of lists

1. Predict what these Python snippets will return.

2. Then try them.

(d) `len(['Goodbye, world!'])`

(e) `len(['Goodbye, '] + ['world!'])`

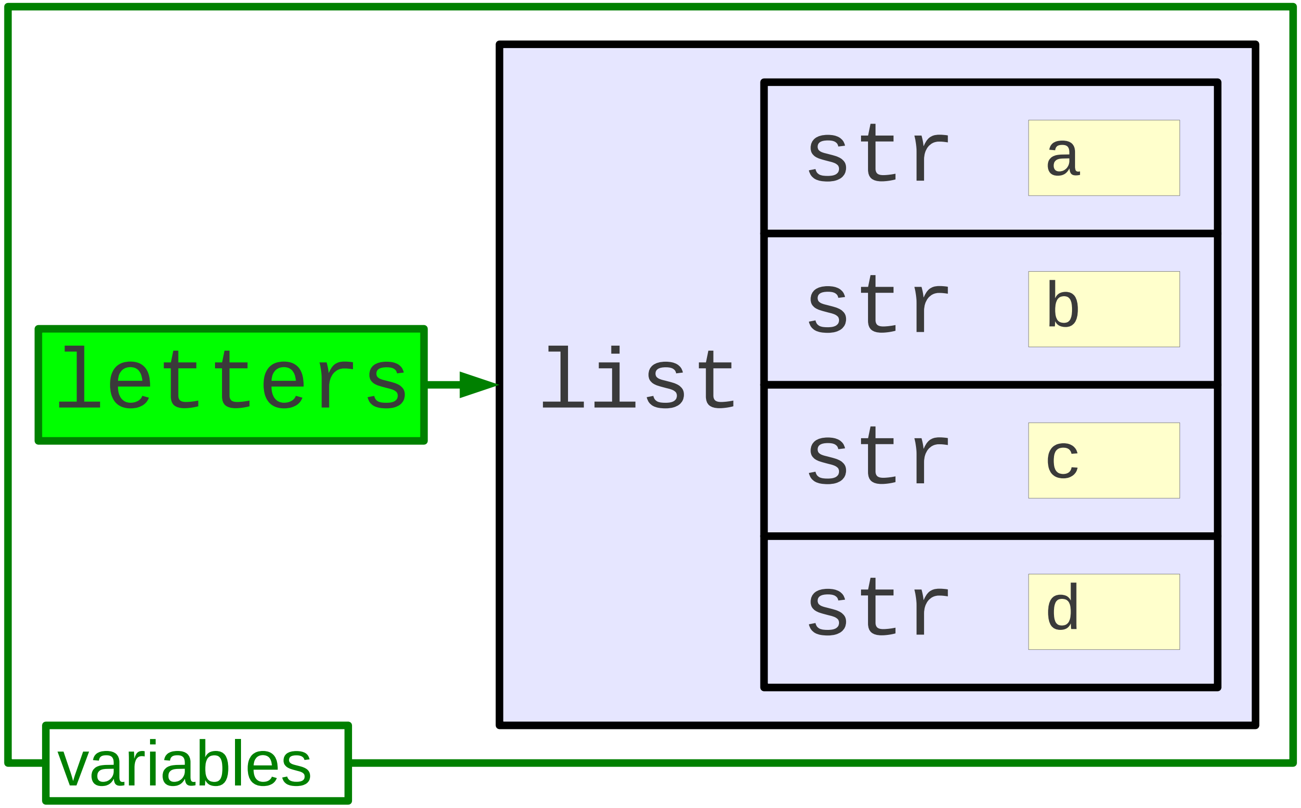
(f) `len(['Goodbye, ']) + len(['world!'])`

UCS

3 minutes
for both

Picking elements from a list

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

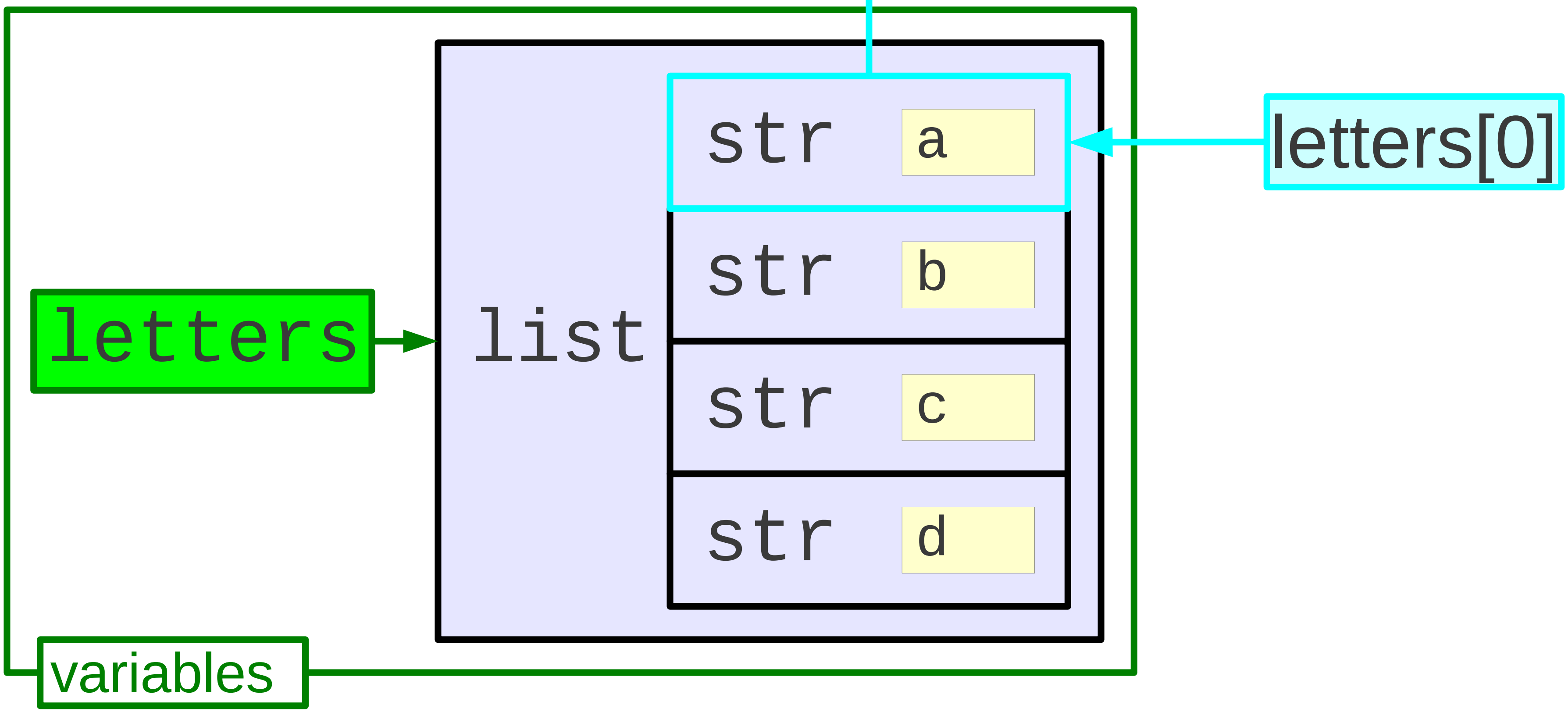


The first element in a list

```
>>> letters[0]
'a'
```

Count from zero

"Index"



Square brackets in Python

[…] Defining literal lists

`numbers[`*`N`*`]` **Indexing into a list**

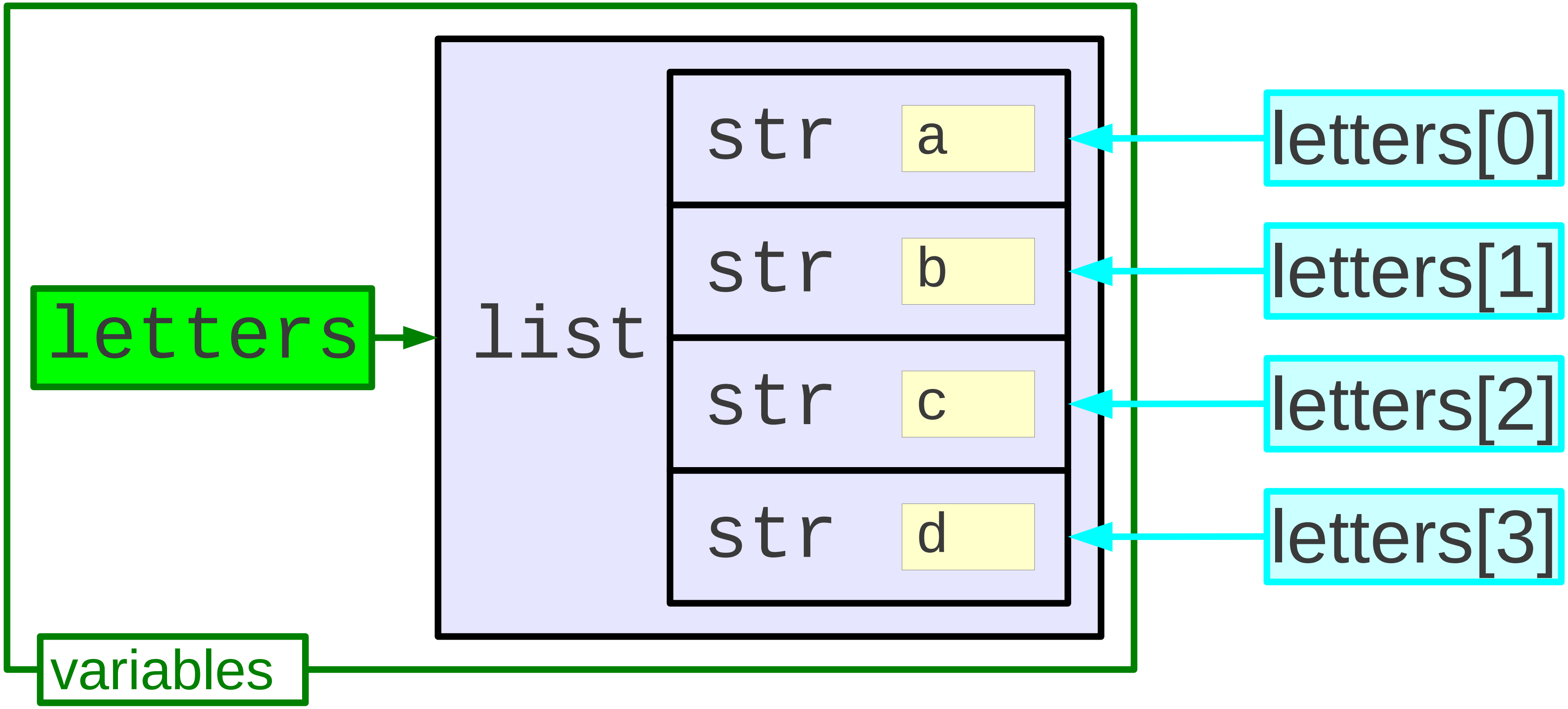
e.g.

```
>>> primes = [2,3,5,7,11,13,17,19]
>>> primes[0]
2
```

“Element number 2”

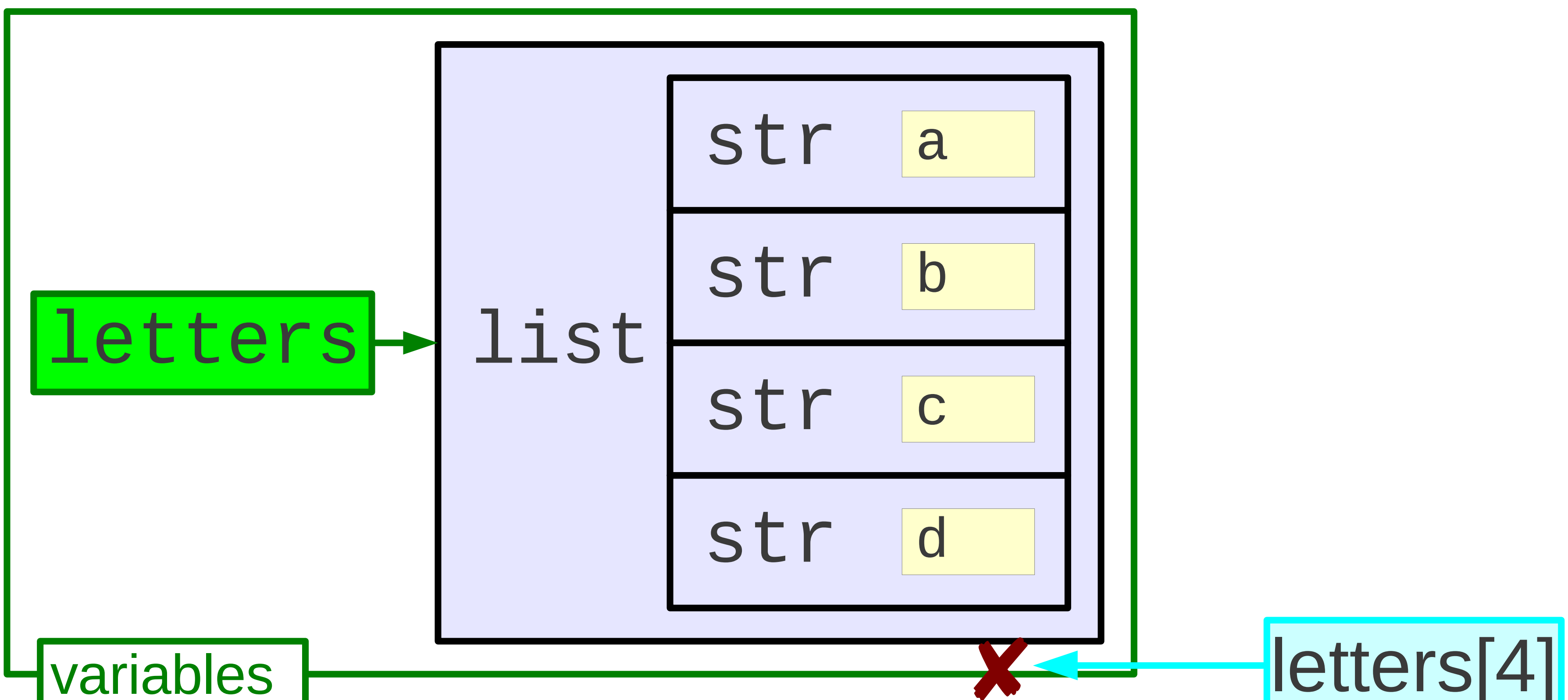
```
>>> letters[2]
'c'
```

The *third* element



Going off the end

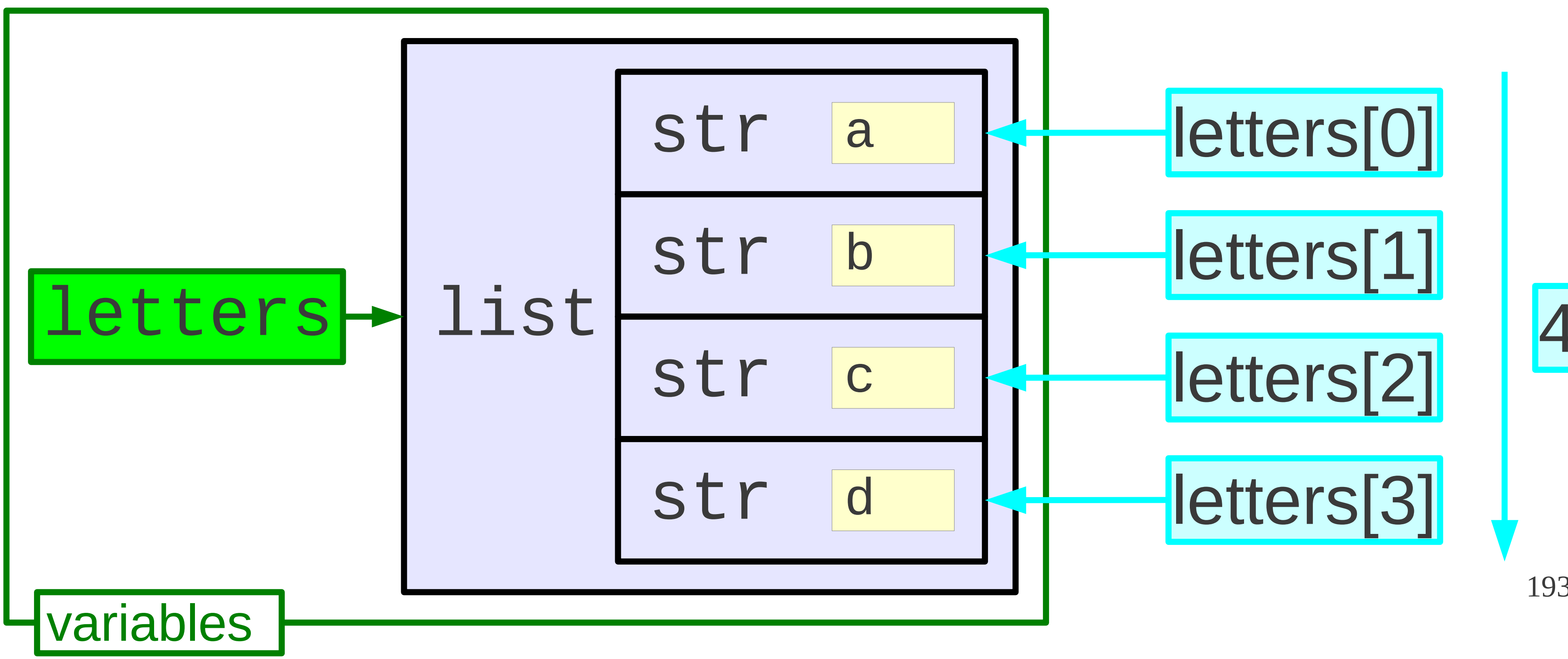
```
>>> letters[4]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```



Maximum index vs. length

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

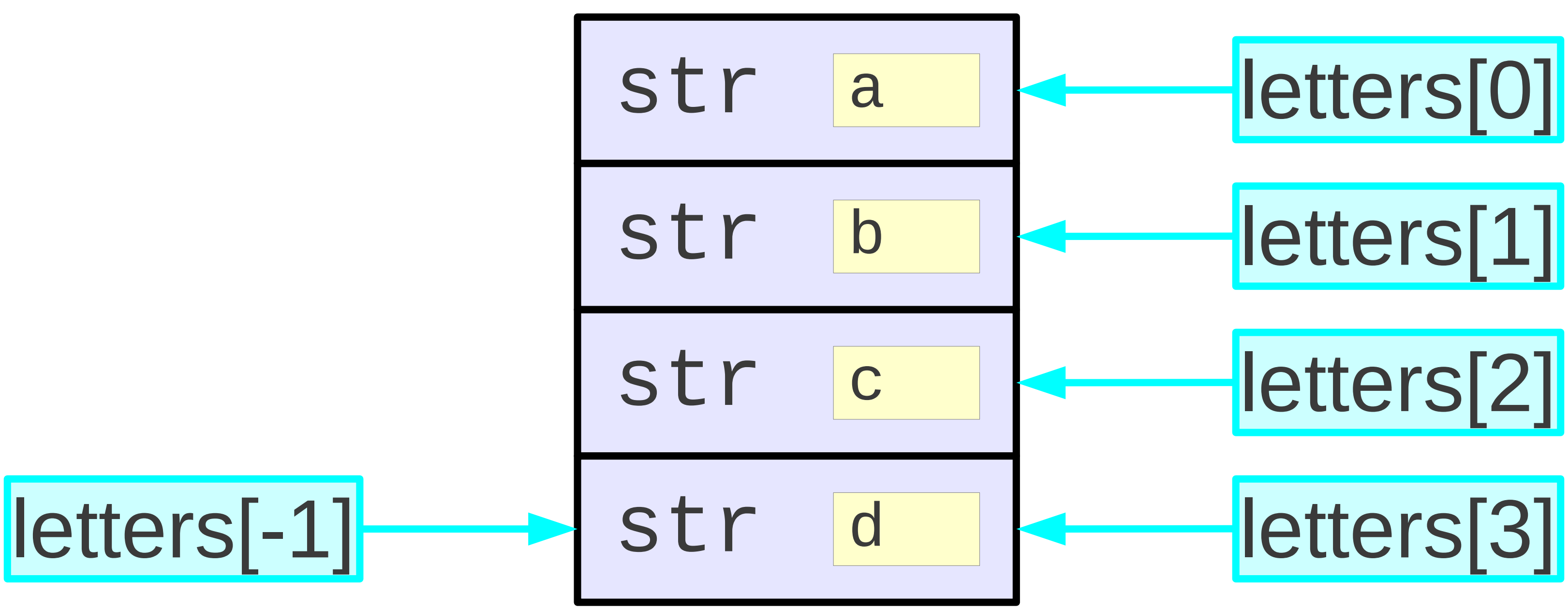
Maximum index is **3**!



“Element number -1 !”

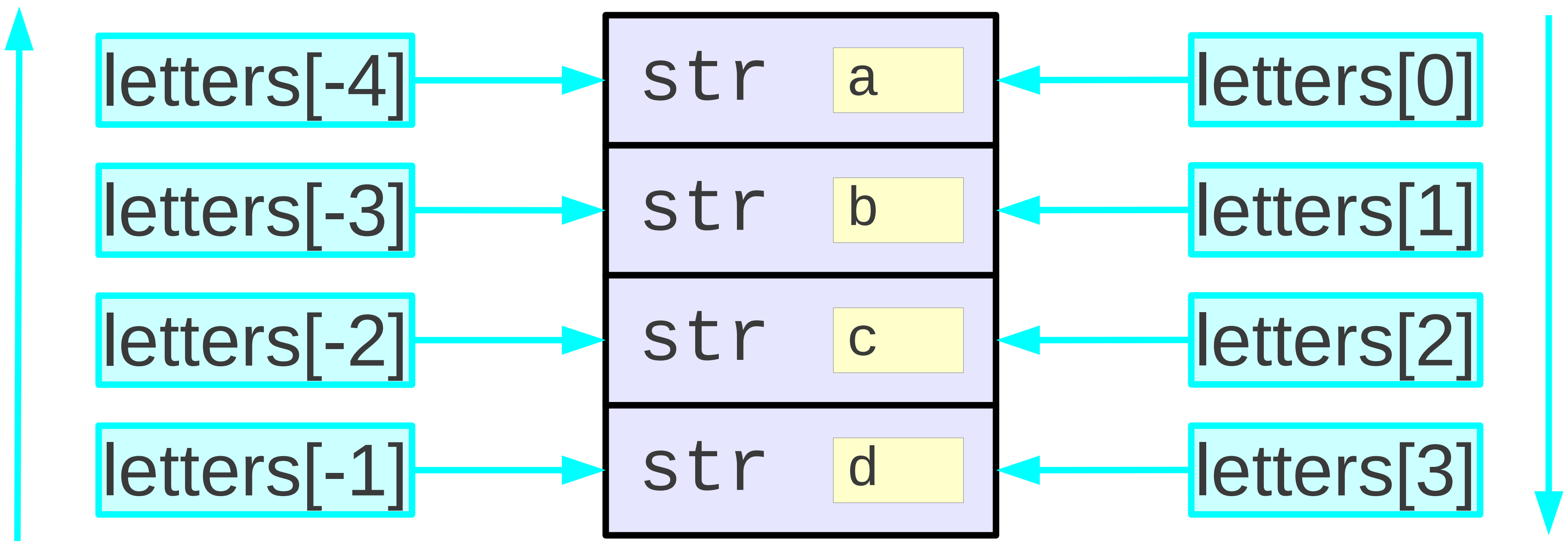
```
>>> letters[-1]
'd'
```

The *final* element



Negative indices

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

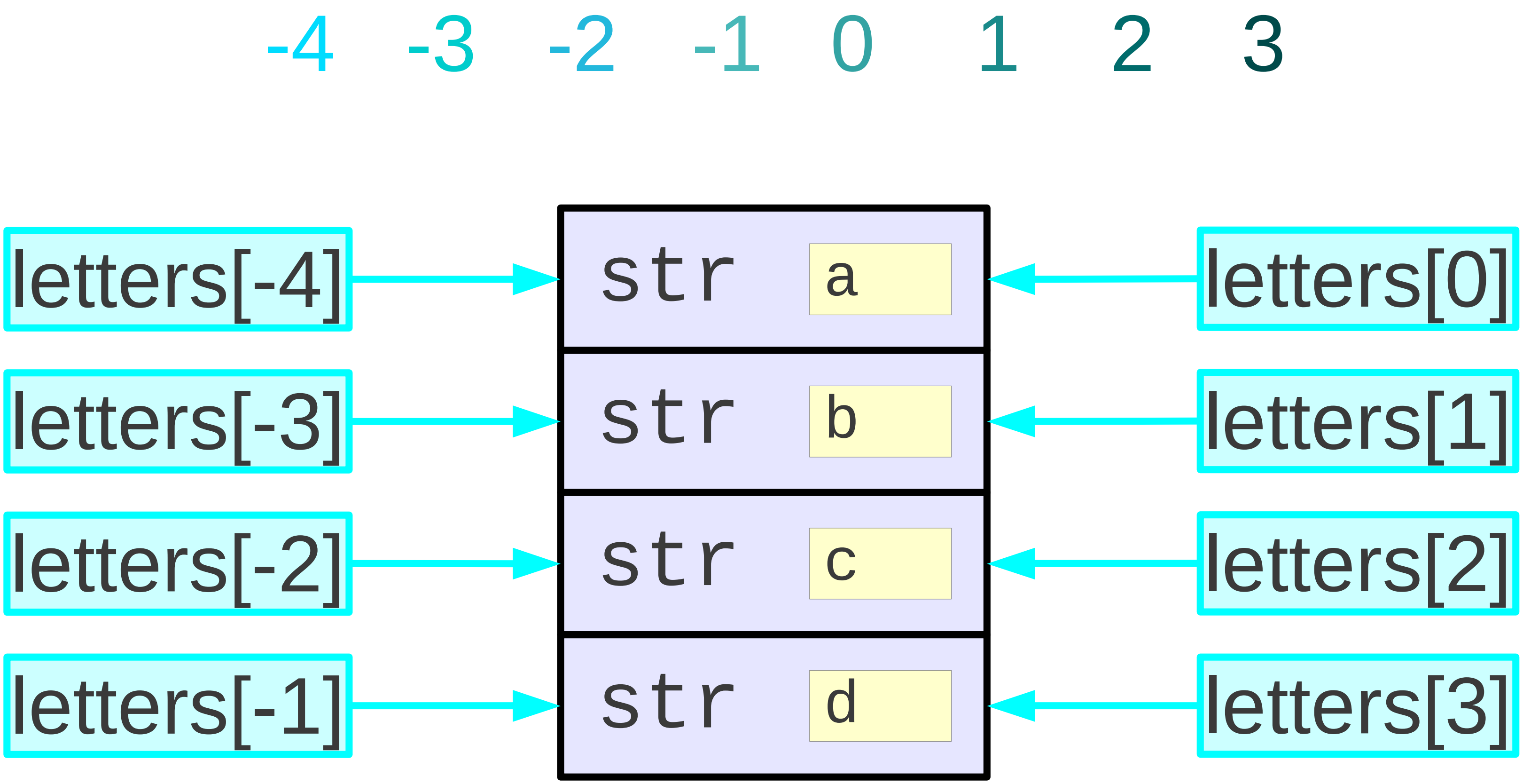


Going off the end

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

Valid range of indices

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



Indexing into literal lists

```
>>> letters = ['a', 'b', 'c', 'd']
>>> letters[3]
'd'
```

Index

Name of list

Legal, but rarely useful:

```
>>> ['a', 'b', 'c', 'd'][3]
'd'
```

Index

Literal list

Assigning list elements

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

The name attached to the list as a whole

The name attached to *one element* of the list

Assign a new value

The new value

```
>>> letters[2] = 'X'
>>> letters
['a', 'b', 'X', 'd']
```

Progress

Index into a list	`['x','y','z']`
Square brackets for index	`list[index]`
Counting from zero	`list[ 0]` → `'x'`
Negative index	`list[-1]` → `'z'`
Assignment	`list[ 1] = 'A'`

Exercise

Predict the output from the following five commands.
Then try them.

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

data[1]

data[-2]

data[2] = 'gimmel'

data
```

3 minutes

Doing something with a list

Recall our challenge:

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

1. Create a list of the element names

2. Print each entry in the list

Each list element

Given a list

Start with the first element of the list

Do something with that element

Are there any elements left? Finish

Move on to the next element

Each list element

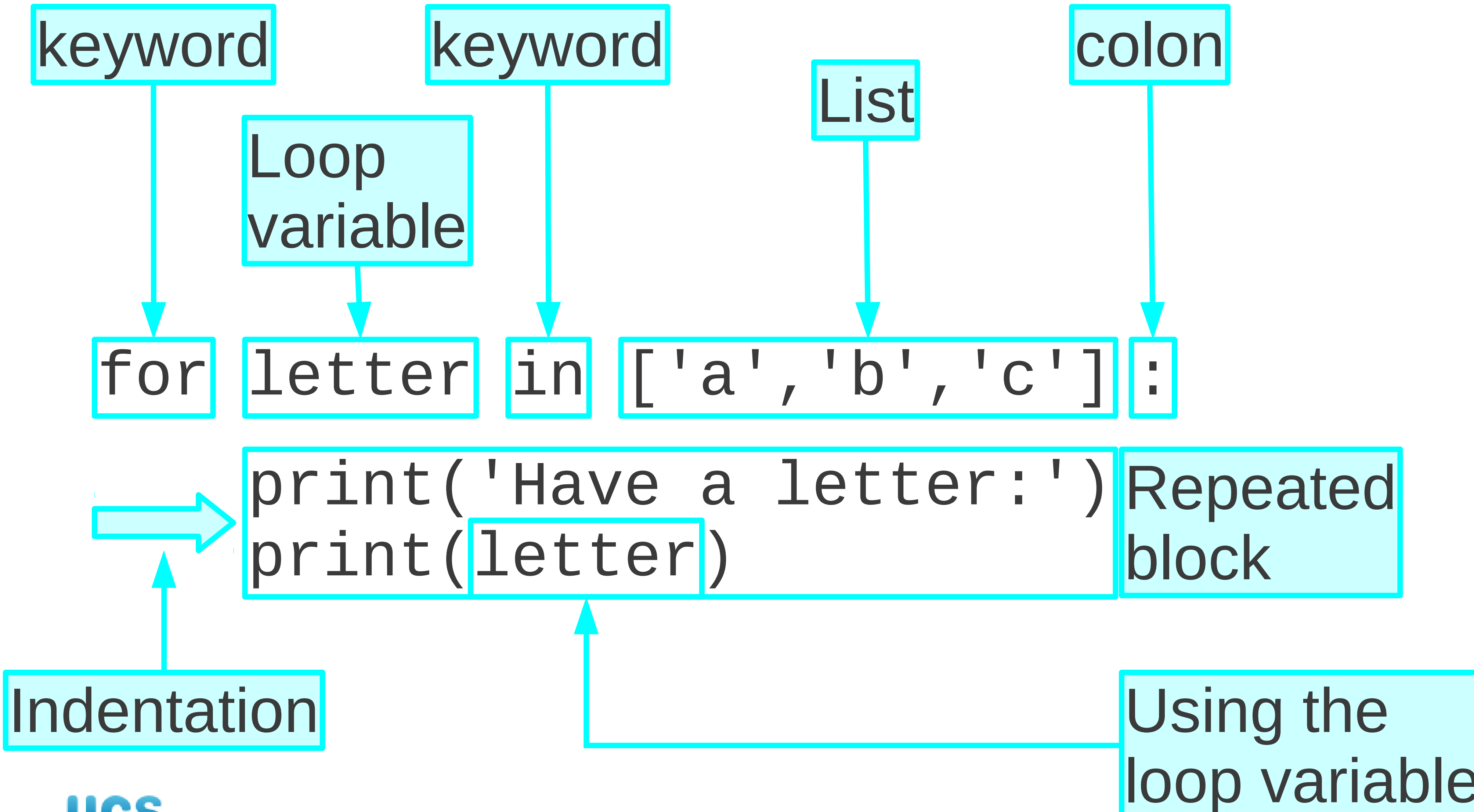
Need to identify “current” element

Do something with that element

Need to identify a block of commands

Another indented block

The “for loop”


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

The “for loop”

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

for1.py

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

UCS

Progress

The “for…” loop

Processing each element of a list

```
for item in items :
    ...item...
```

Exercise

Complete the script `elements1.py`

Print out the name of every element.

10 minutes

“Slices” of a list

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

>>>  abc[1]
'b'

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

Simple index

Single element

Slice index

A new list

“Slice”

Slice limits

“from” index

“to” index

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

abc[1]

abc[**4**]

abc[5]

Element 5
not in slice

Slice feature

Same

abc[1:3] + abc[3:5]

['b','c'] + ['d','e']

['b','c','d','e']

abc[1:5]

Open-ended slices

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

>>> abc[3:]
['d','e','f','g']

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

Open ended at the end

abc[3]

Open ended at the start

abc[4]

Open-ended slices

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

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

Open ended at *both* ends

Progress

Slices

```
data[m:n] → [ data[m], … data[n-1] ]

data[m:n]

data[:n]

data[m:]

data[:]
```

Square brackets in Python

`[…]`	Defining literal lists
`numbers[N]`	Indexing into a list
`numbers[M:N]`	Slices

Modifying lists — recap

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

>>> abc[2] = 'X'

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

abc[2]

New value

Changed

Modifying *vs.* replacing ?

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

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

Modifying the list

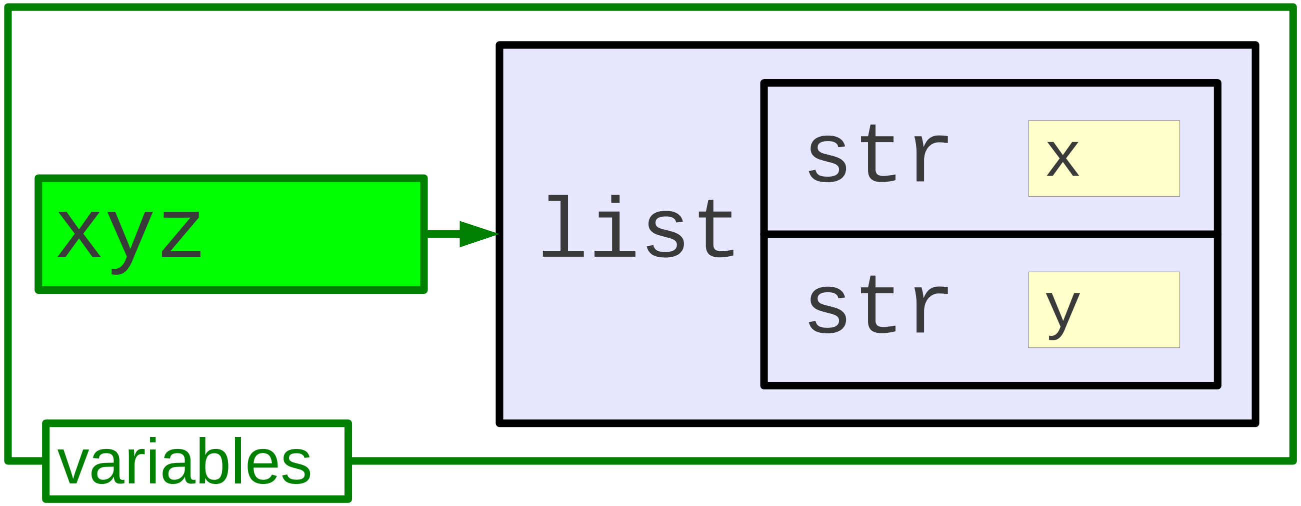
```
>>> xyz = ['A','B']
```

Replacing the list

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

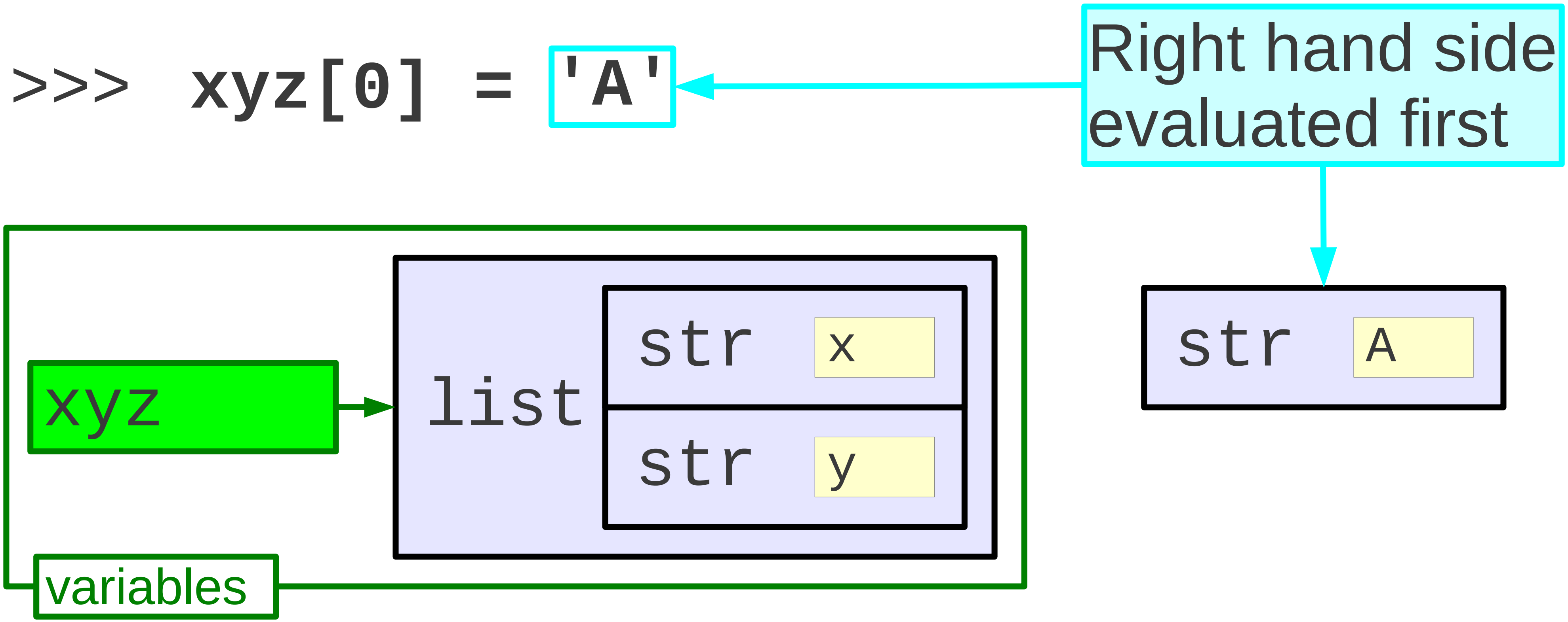
What's the difference? — 1a

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



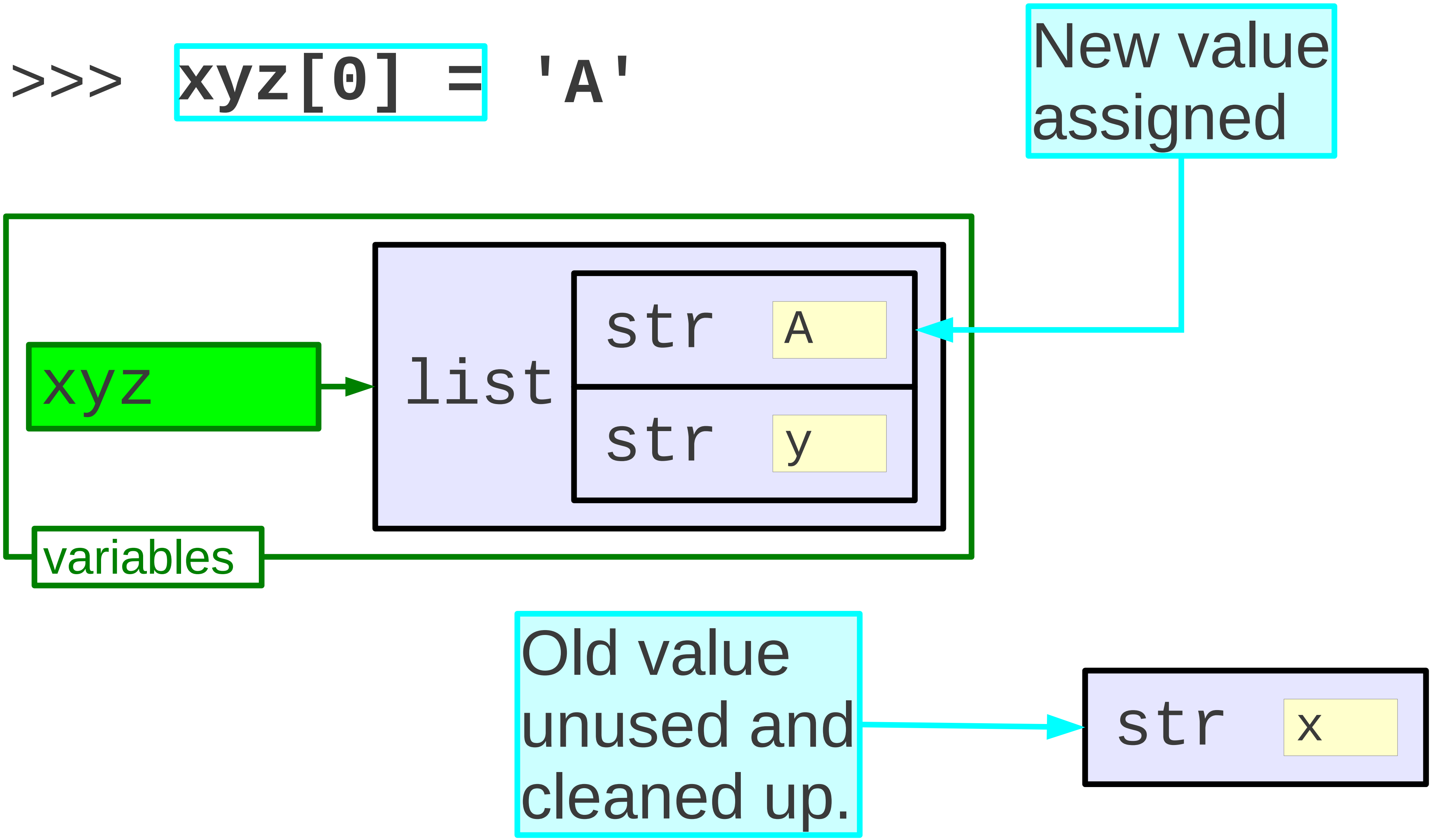
What's the difference? — 1b

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



What's the difference? — 1c

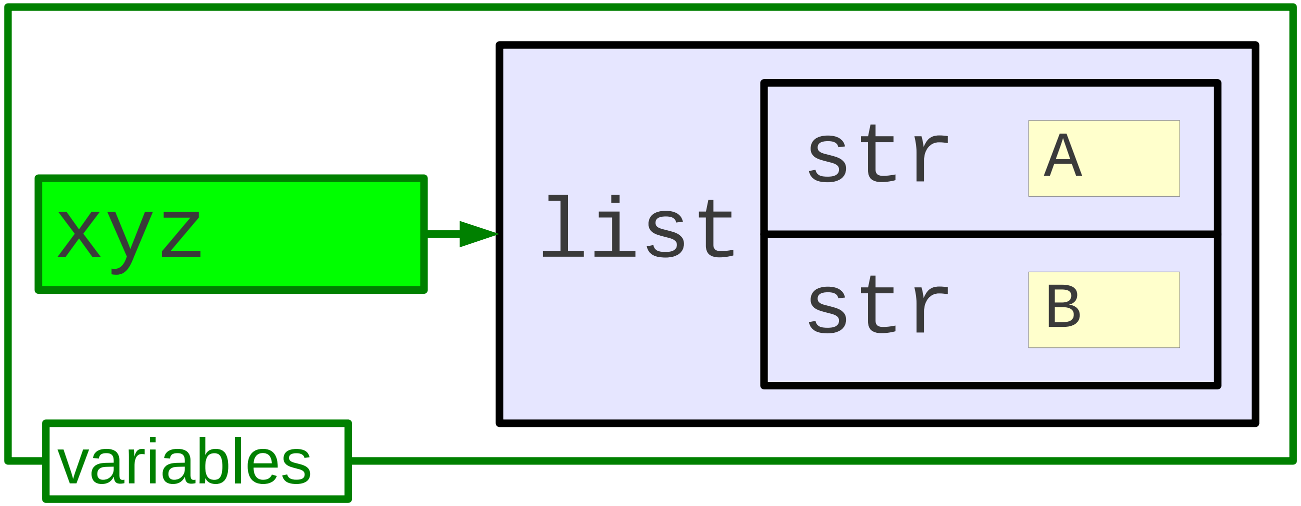
```
>>> xyz[0] = 'A'
```



What's the difference? — 1d

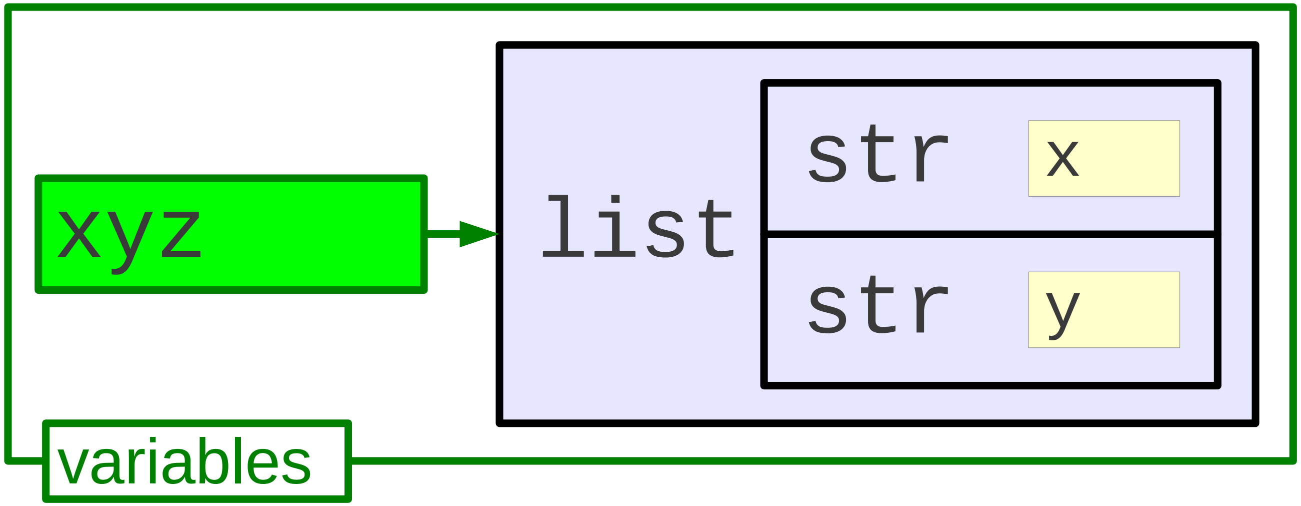
```
>>> xyz[1] = 'B'
```

Repeat for
xyz[1] = 'B'



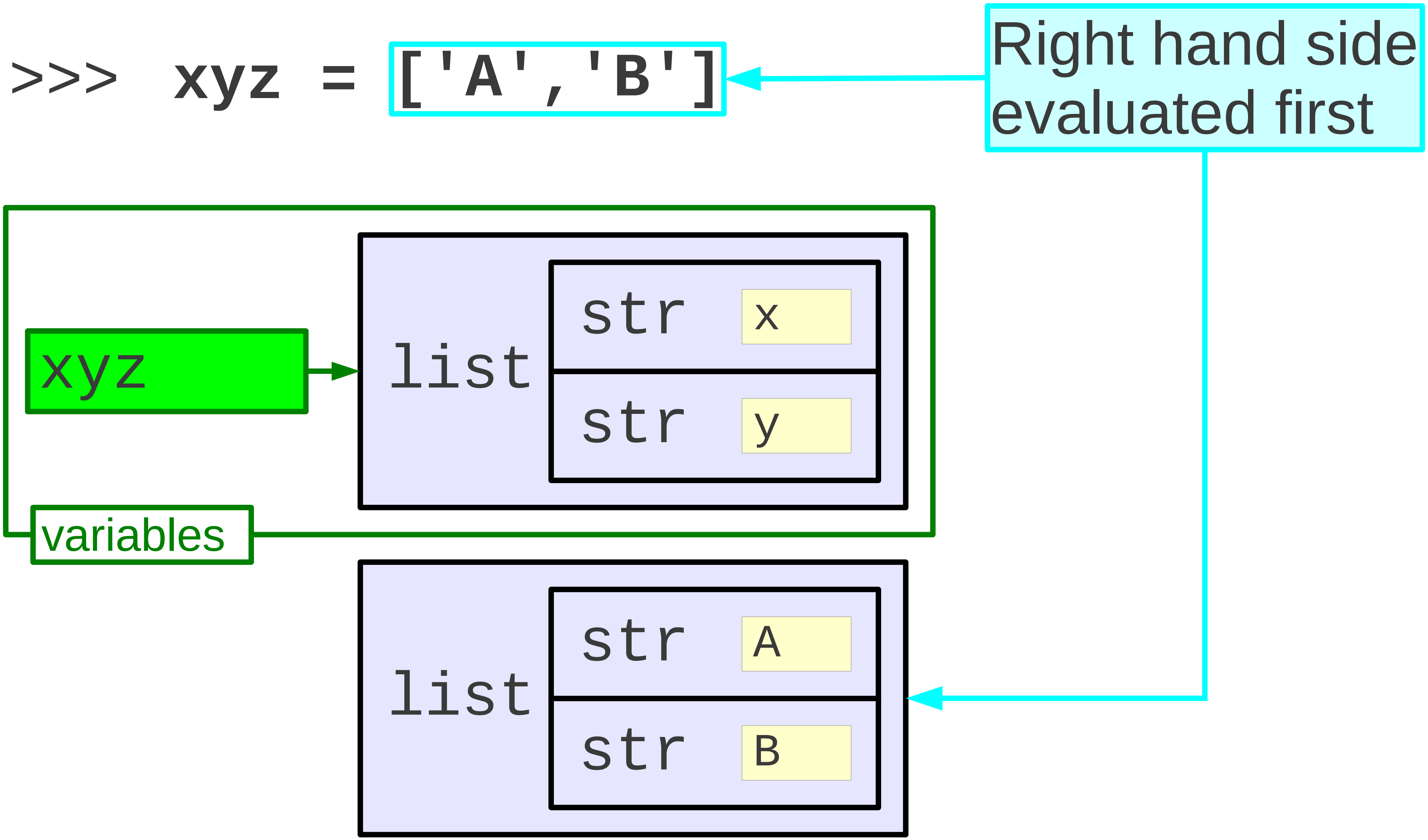
What's the difference? — 2a

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



What's the difference? — 2b

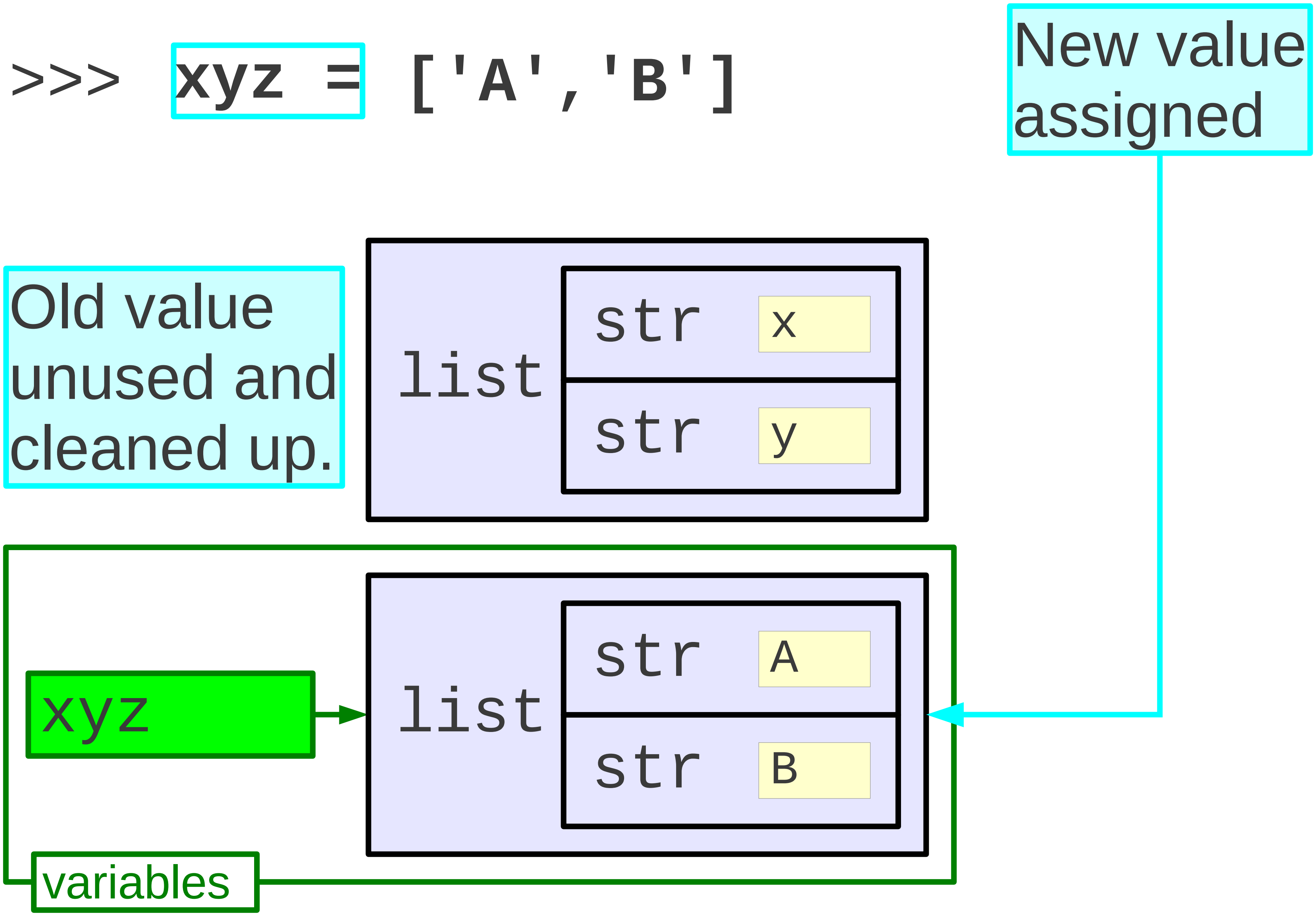
```
>>> xyz = ['A','B']
```



UCS

What's the difference? — 2c

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

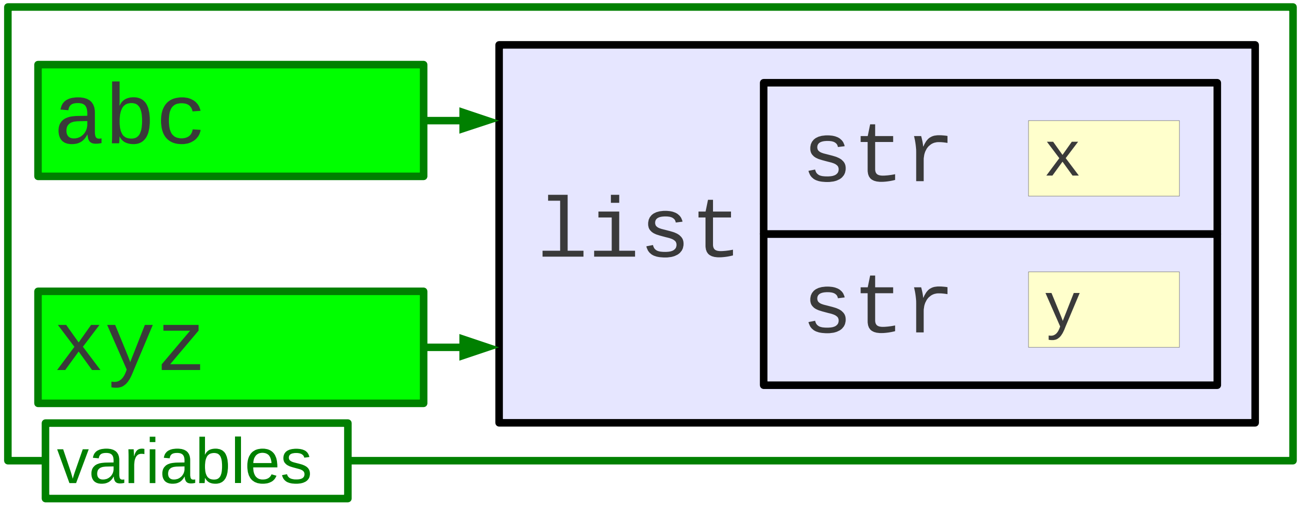


What's the difference?

Modification:	same list, different contents
Replacement:	different list
?	**Does it matter?**

Two names for the same list

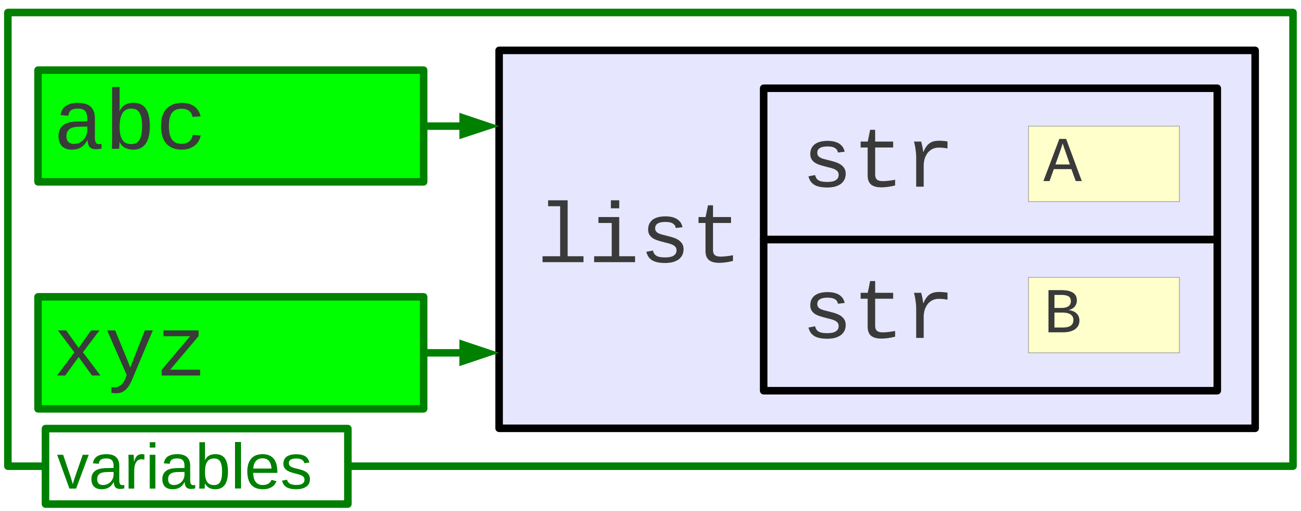
```
>>> xyz = ['x','y']
>>> abc = xyz
```



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

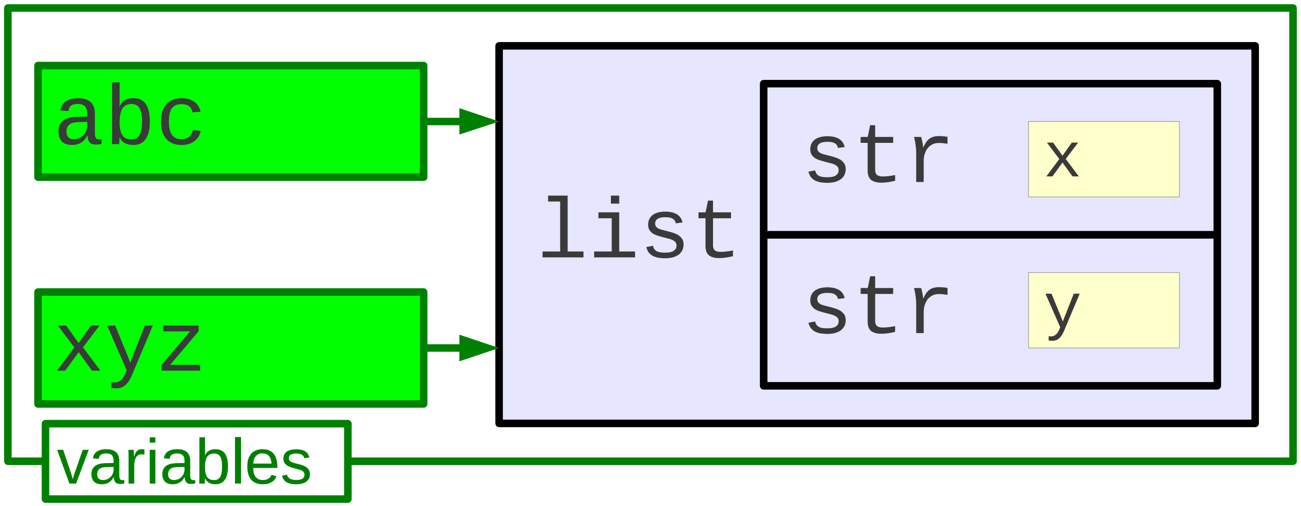
Modification

Modification



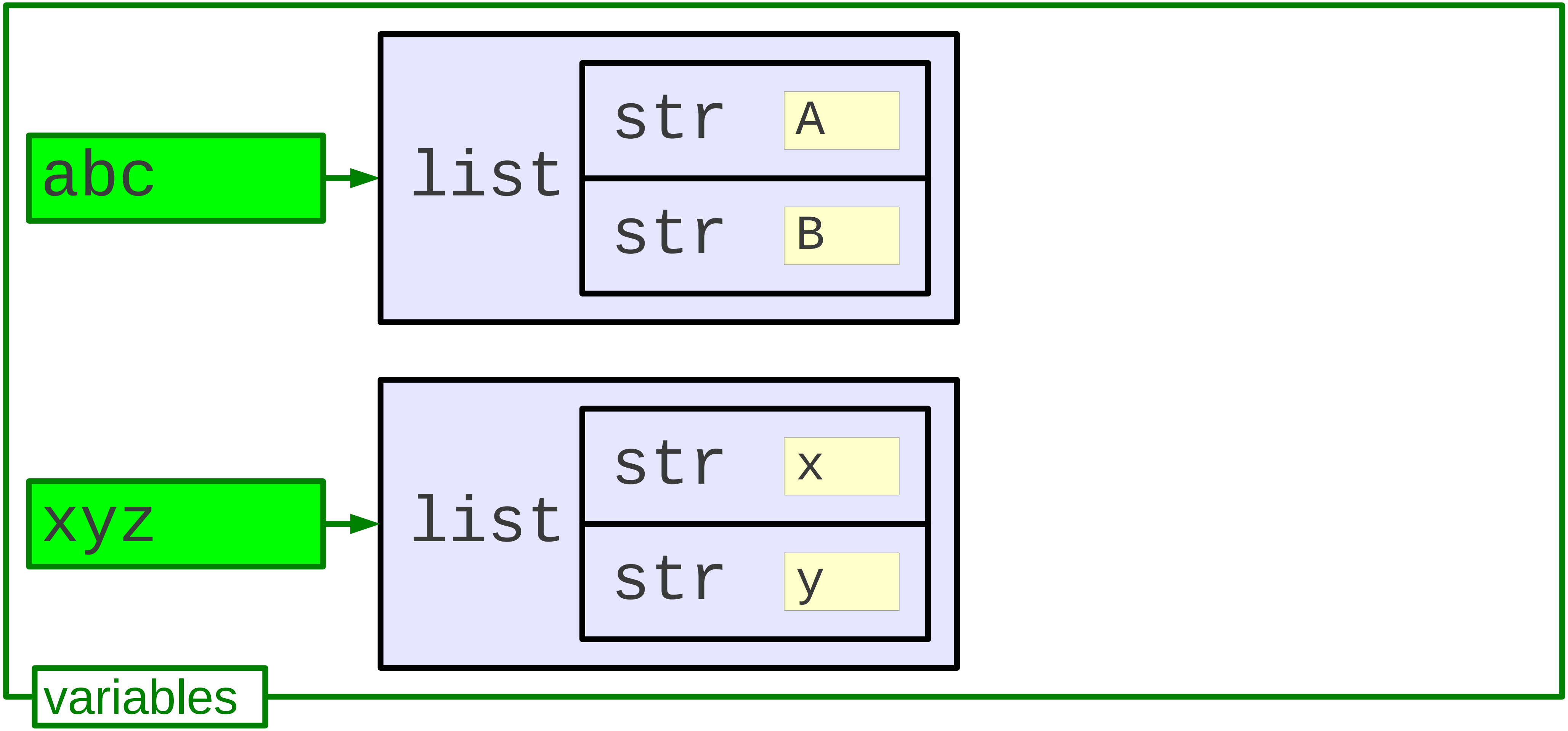
Same starting point

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



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

Replacement



One last trick with slices

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

Length 6

```
>>> abc[2:4]
['c','d']

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

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

New length

Progress

Modifying lists `values[N] = new_value`

Modification ≠ replacement

```
values[0] = 'alpha'
values[1] = 'beta'
values[2] = 'gamma'
```

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

Exercise

1. Predict what these will do.
2. Then run the commands.

```
>>> alpha = [0, 1, 2, 3, 4]
>>> beta = alpha
>>> gamma = alpha[:]
>>> delta = beta[:]

>>> beta[0] = 5

>>> alpha
>>> beta
>>> gamma
>>> delta
```

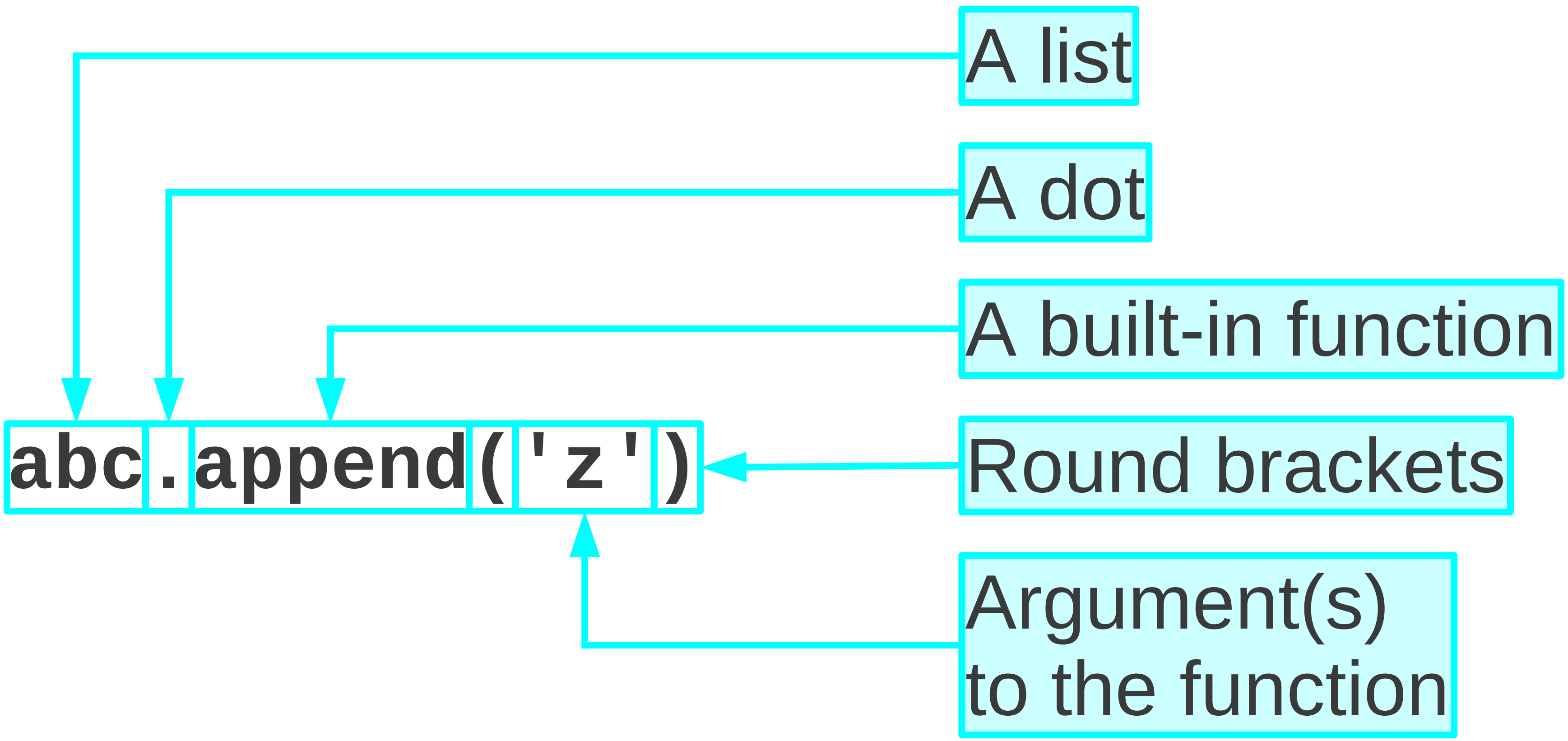
5 minutes

Appending to a list

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

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

>>> abc.append('z')

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

Add one element to the end of the list.

List “methods”



Built-in functions: “methods”

Methods

Connected by a dot

```
object.method(arguments)
```

Privileged access to object

“Object-oriented programming”

The append() method

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

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

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

One element at a time

Beware!

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

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

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

Appending a list

Get a list as the last item

“Mixed lists”

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

['x', 2, 3.0]

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

The extend() method

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

All in one go

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

Utterly unnecessary!

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

Avoiding extend()

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

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

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

Changing the list “in place”

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

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

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

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

No value returned

List itself is changed

No value returned

List itself is changed

Another list method: sort()

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

>>> abc.sort()

>>> abc

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

New method

No arguments

UCS

Any type of sortable element

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


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

Another list method: insert()

0 1 2 3

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

>>> abc.insert(2,'A')

>>> abc

['w','x','A','y','z']
```

Insert just before element number 2

“old 2”

Progress

List methods:

Change the list itself

Don't return any result

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

Exercise

1. Predict what this will do.
2. Then run the commands.

```
data = []
data.append(8)
data.extend([6,3,9])
data.sort()
data.append(1)
data.insert(3,2)
data
```

5 minutes

Creating new lists

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

Simple copying

Creating new lists

Boring!

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

Changing the value

UCS

Lists of numbers

```
>>> numbers = range(0,10)

>>> numbers

[0,1,2,3,4,5,6,7,8,9]
```

```
range(0,10)

[0,1,2,3,4,5,6,7,8,9]
```

c.f. numbers[0:5]

Creating new lists

Better!

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

UCS

Lists of words

string

method

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

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

No special handling for punctuation.

Progress

Ways to build lists:

`data[:]` slices

`for` loops appending elements

`range(m,n)` function

`split()` string method

Exercise

Write a script from scratch: `transform.py`

1. Run a variable n from 0 to 10 inclusive.
2. Create a list with the corresponding values of $n^2 + n + 41$.
3. Print the list.

UCS

10 minutes

Brief diversion

UCS

Arrays as lists of lists

0.0	-1.0	-4.0	-1.0	0.0
1.0	0.0	-1.0	0.0	1.0
4.0	1.0	0.0	1.0	4.0
1.0	0.0	-1.0	0.0	1.0
0.0	-1.0	-4.0	-1.0	0.0

```
[ [0.0, -1.0, -4.0, -1.0, 0.0] ,
  [1.0,  0.0, -1.0,  0.0, 1.0] ,
  [4.0,  1.0,  0.0,  1.0, 4.0] ,
  [1.0,  0.0, -1.0,  0.0, 1.0] ,
  [0.0, -1.0, -4.0, -1.0, 0.0] ]
```

Indexing from zero

0.0	-1.0	-4.0	-1.0	0.0
1.0	0.0	-1.0	0.0	1.0
4.0	1.0	0.0	1.0	4.0
1.0	0.0	-1.0	0.0	1.0
0.0	-1.0	-4.0	-1.0	0.0

a_{23}

```
[ [0.0, -1.0, -4.0, -1.0, 0.0] ,
  [1.0,  0.0, -1.0,  0.0, 1.0] ,
  [4.0,  1.0,  0.0,  1.0, 4.0] ,
  [1.0,  0.0, -1.0,  0.0, 1.0] ,
  [0.0, -1.0, -4.0, -1.0, 0.0] ]
```

a[2][3]

Referring to a row — easy

0.0	-1.0	-4.0	-1.0	0.0
1.0	0.0	-1.0	0.0	1.0
4.0	1.0	0.0	1.0	4.0
1.0	0.0	-1.0	0.0	1.0
0.0	-1.0	-4.0	-1.0	0.0

```
[ [0.0, -1.0, -4.0, -1.0, 0.0] ,
  [1.0,  0.0, -1.0,  0.0, 1.0] ,
  [4.0,  1.0,  0.0,  1.0, 4.0] ,      a[2]
  [1.0,  0.0, -1.0,  0.0, 1.0] ,
  [0.0, -1.0, -4.0, -1.0, 0.0] ]
```

Referring to a column

0.0	-1.0	-4.0	-1.0	0.0
1.0	0.0	-1.0	0.0	1.0
1.0	1.0	0.0	1.0	4.0
1.0	0.0	-1.0	0.0	1.0
0.0	-1.0	-4.0	-1.0	0.0

No Python construct!

```
[ [0.0, -1.0, -4.0, -1.0, 0.0] ,
  [1.0,  0.0, -1.0,  0.0, 1.0] ,
  [4.0,  1.0,  0.0,  1.0, 4.0] ,
  [1.0,  0.0, -1.0,  0.0, 1.0] ,
  [0.0, -1.0, -4.0, -1.0, 0.0] ]
```

Numerical Python?

Hold tight!

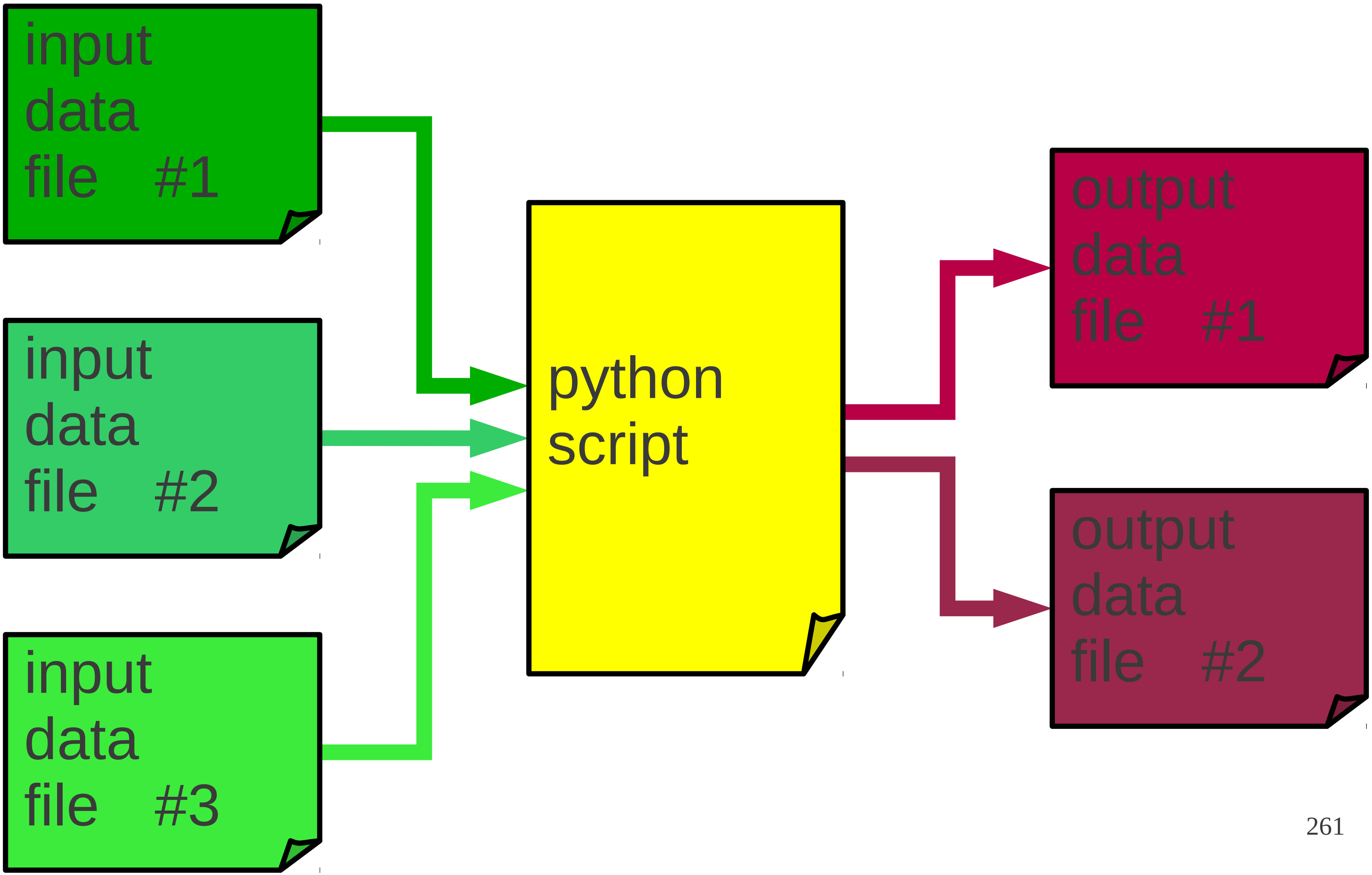
Later in this course, “numpy”
powerful support for:

numerical arrays

matrices

End of diversion

Files


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

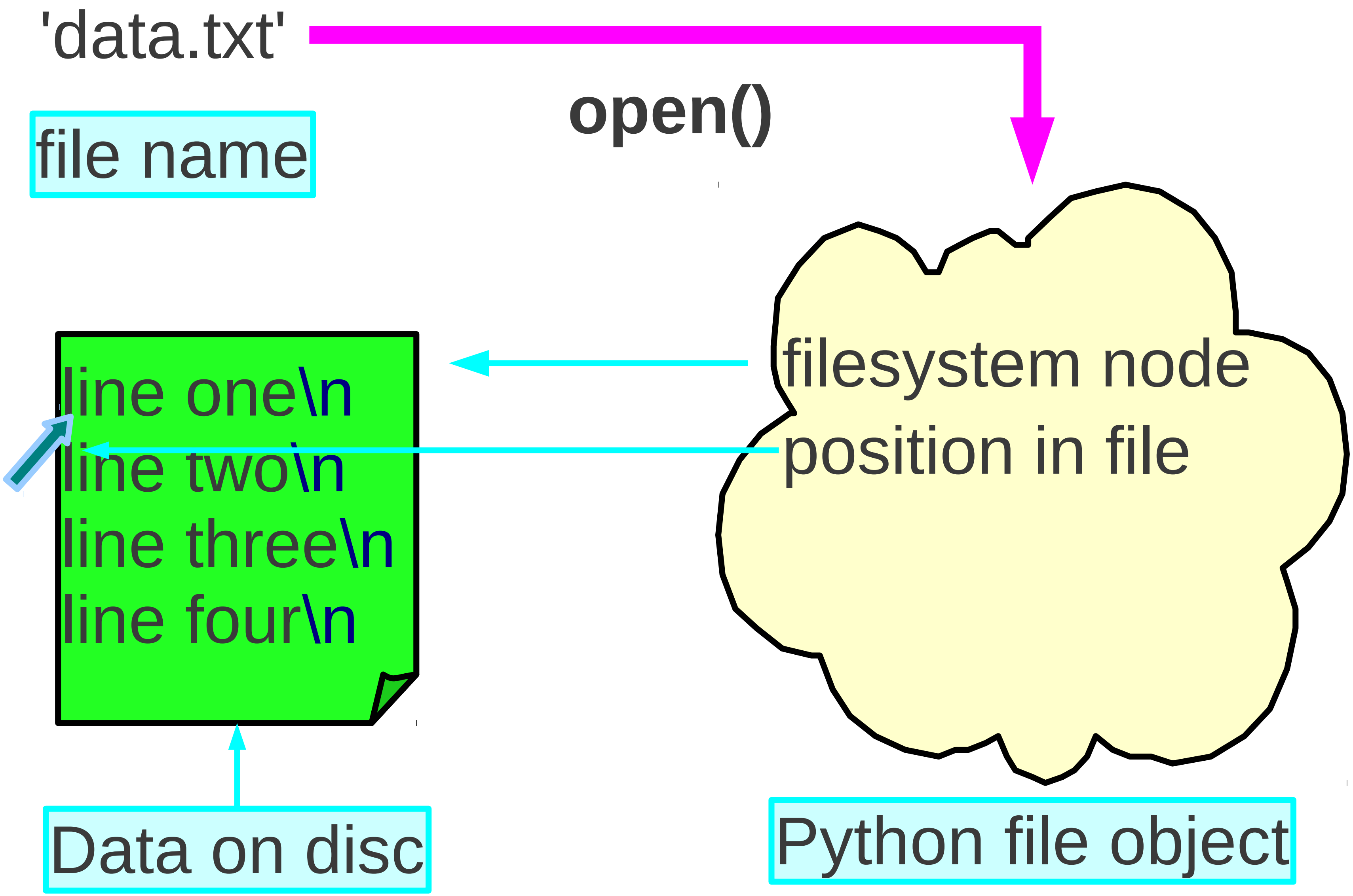
Reading a file

1. Opening a file

2. Reading from the file

3. Closing the file

Opening a file



Python command

file name

string

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

Python file object

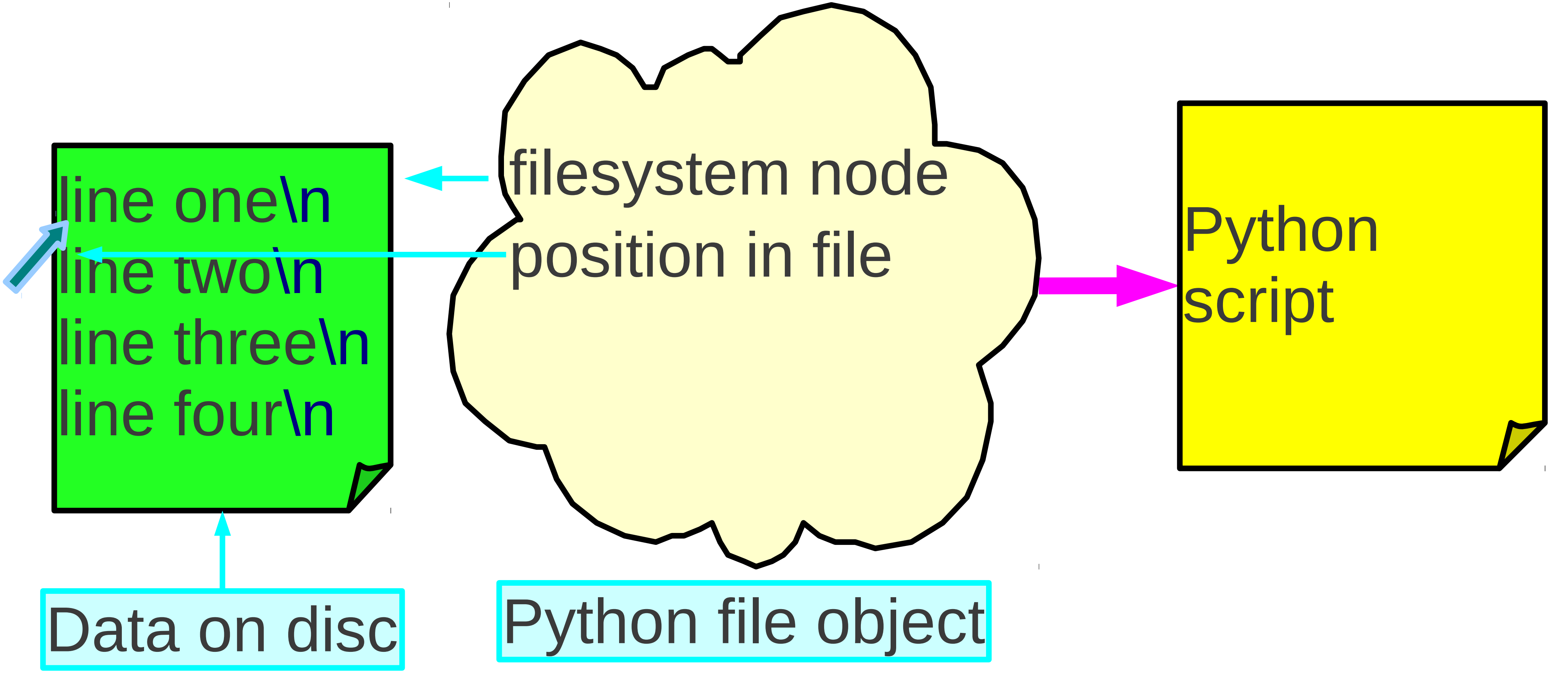
file

refers to the file with name 'data.txt'

initial position at start of data

UCS

Reading from a file



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

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

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

the Python file object

a dot

a “method”

first line of the file

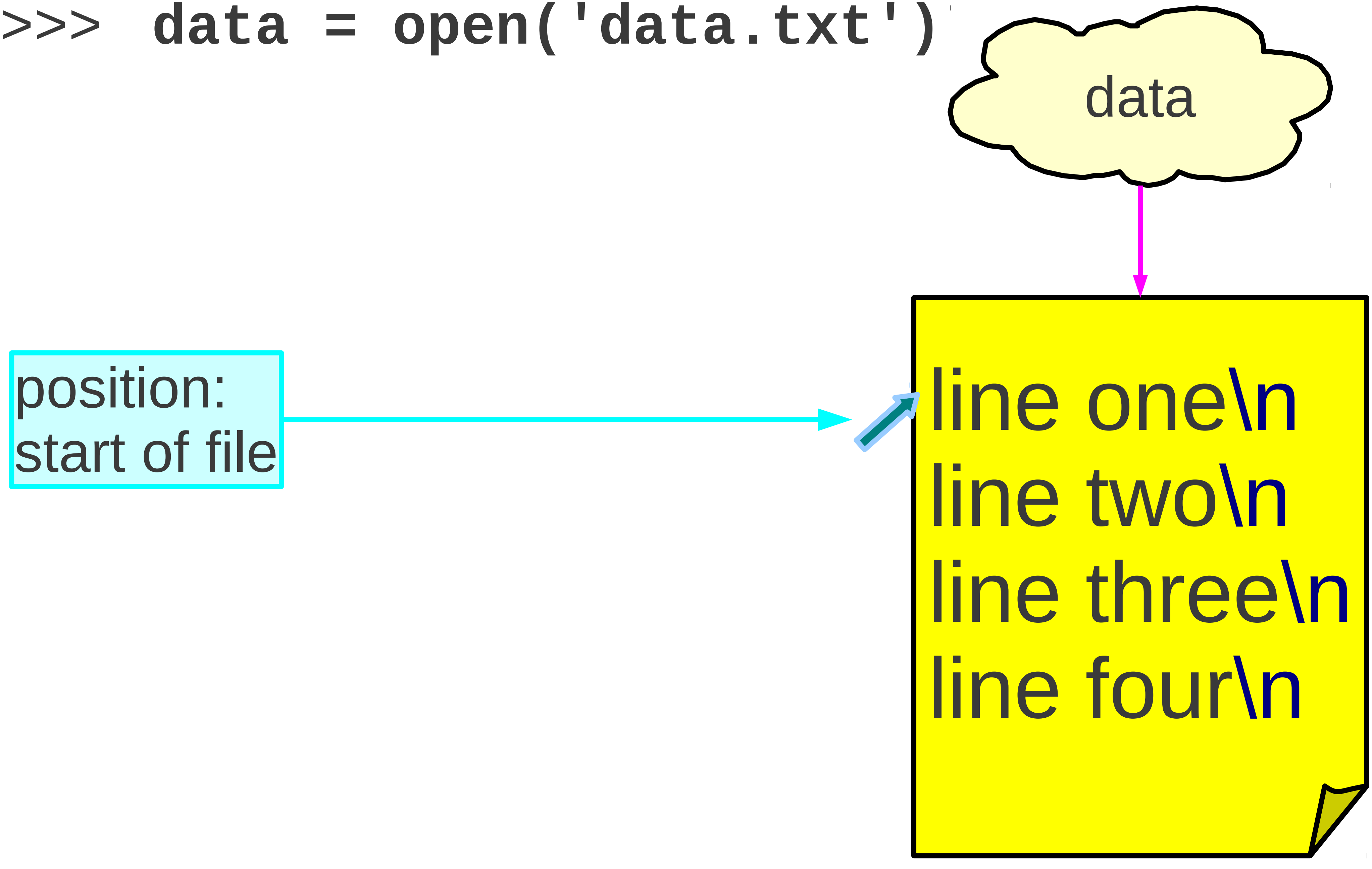
complete with “\n”

same command again

second line of file

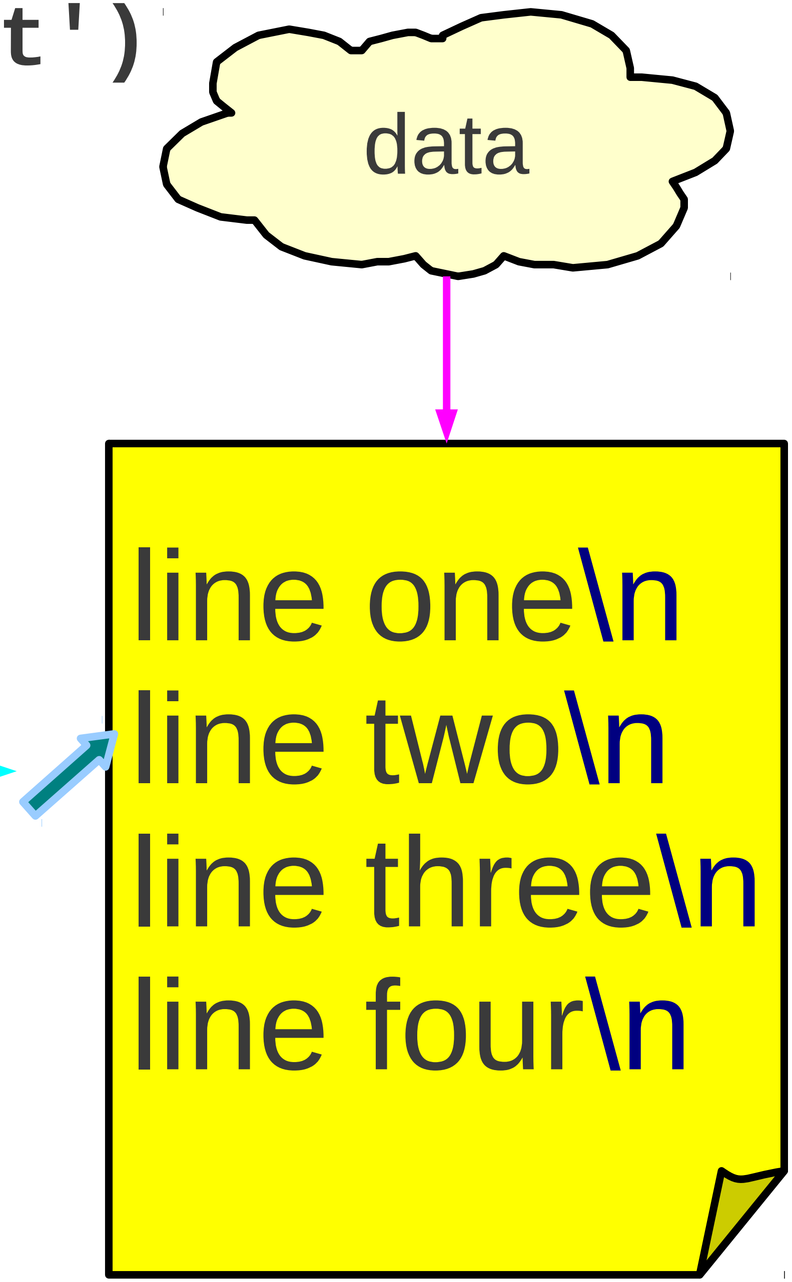
UCS

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



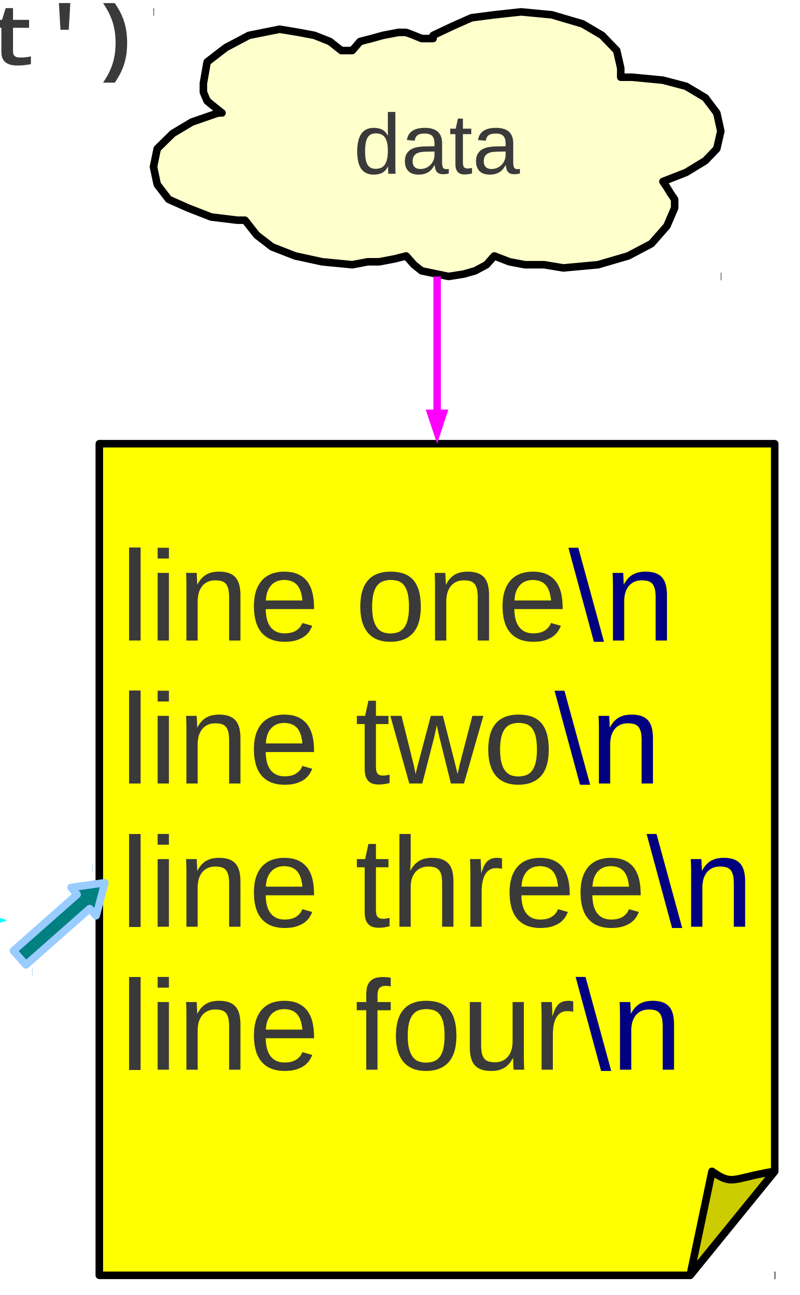
```
>>> data = open('data.txt')

>>> data.readline()

'line one\n'
```

position:
after end of first line
at start of second line

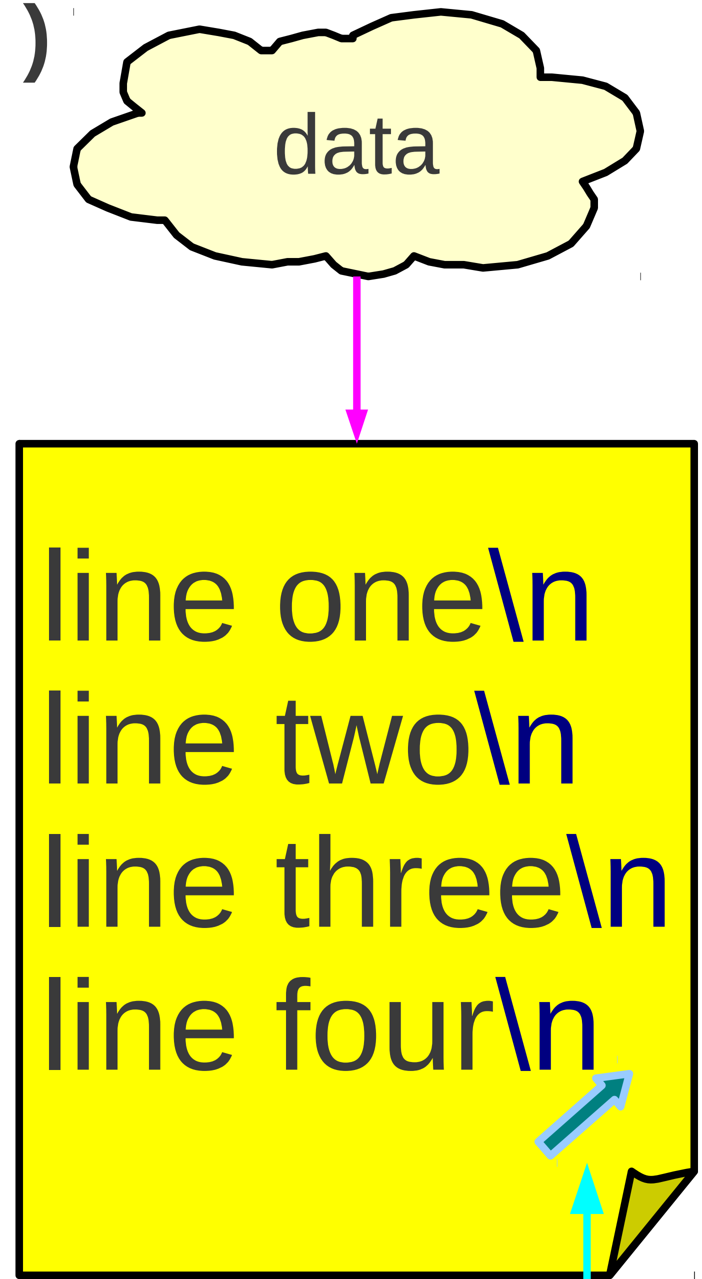


```
>>> data = open('data.txt')

>>> data.readline()

'line one\n'

>>> data.readline()

'line two\n'
```

after end of second line
at start of third line



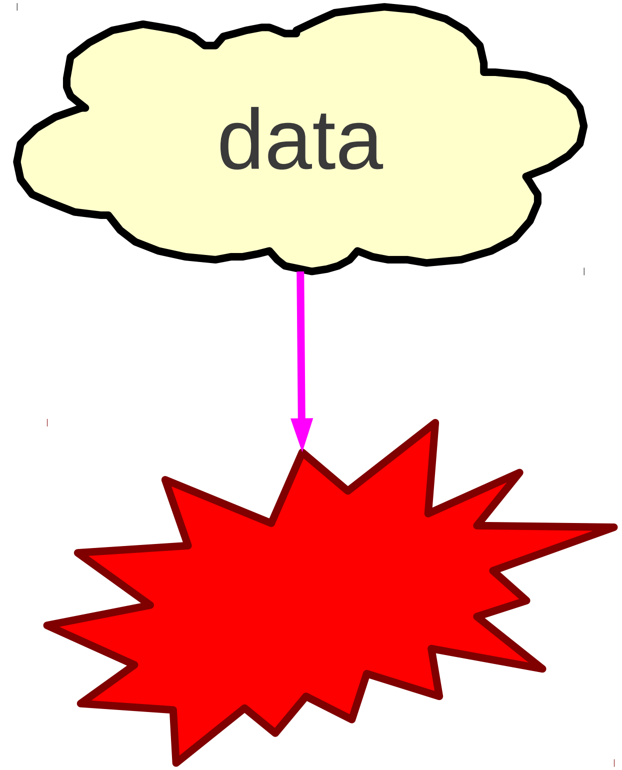
```
>>>  data = open('data.txt')

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

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

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



end of file

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

disconnect



Common trick

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

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

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

Simple example script

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

1. Open the file

2. Read the file
   One line at a time

3. Close the file

Progress

filename → `open()` → readable file object

```
data = open('input.dat')
```

```
data.readline()
```

```
for line in data:
    ...line...
```

Exercise

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

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

UCS

15 minutes

Converting the type of input

Problem:

```
1.0
2.0
3.0
4.0
5.0
6.0
7.0
8.0
9.0
10.0
11.0
```

numbers.dat

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

List of strings, not a list of numbers.

Type conversions

`>>> float('1.0\n')`	String	→	Float
`1.0`			
`>>> str(1.0)`	Float	→	String
`'1.0'`	No newline		
`>>> float(1)`	Int	→	Float
`1.0`			
`>>> int(-1.5)`	Float	→	Int
`-1`	Rounding to zero		

Type conversions to lists

```
>>> list('hello')
['h','e','l','l','o']
```

String → List

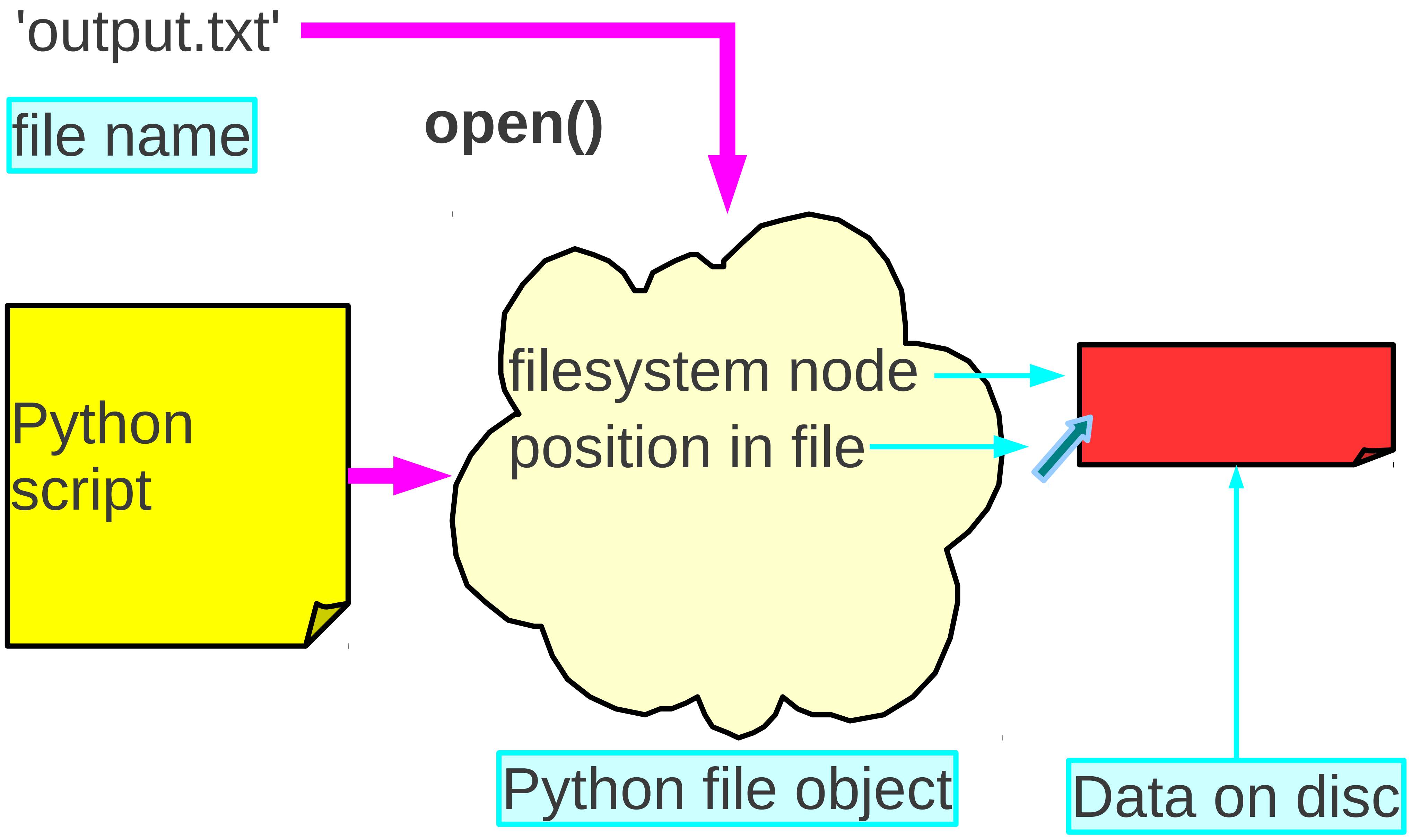
```
>>> data = open('data.txt')
>>> list(data)
['line one\n', 'line two\n',
 'line three\n', 'line four\n']
```

File → List

Example script

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

Writing to a file



Writing to a file

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

Default

Equivalent

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

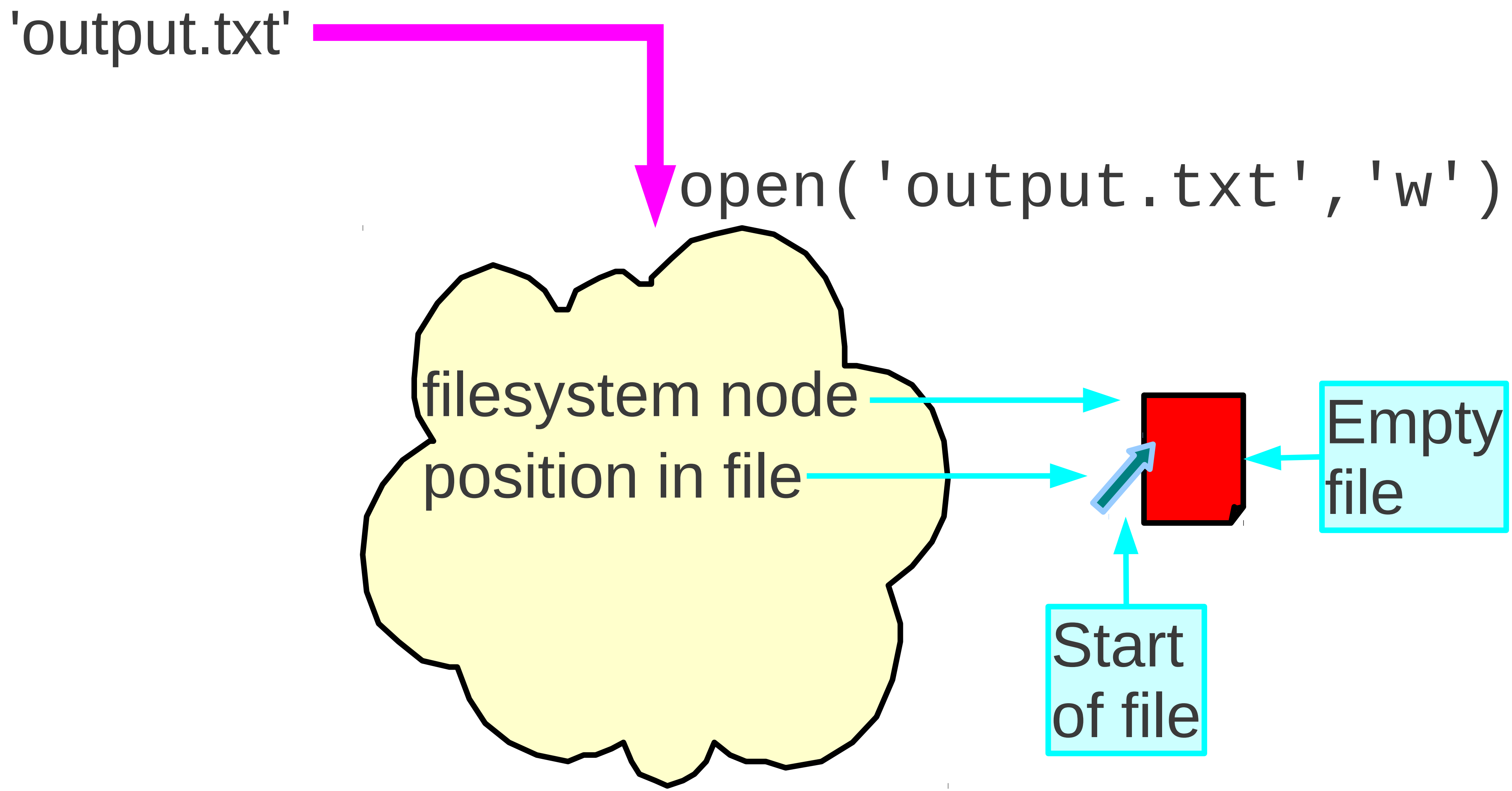
Open for reading

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

Open for writing

UCS

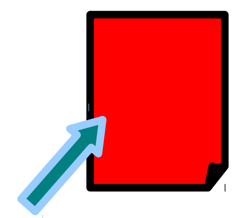
Opening a file for writing



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

file name

open for writing



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

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

Method to write a lump of data

Lump of data

“Lump”: need not be a line.

Current position changed

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

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

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

Lump of data to be written

alpha\n
bet

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

Remainder of the line

alpha\n
beta\n

UCS

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

Method to write a *list* of lumps

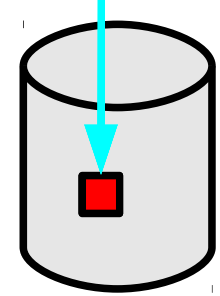
alpha\n
beta\n
gamma\n
delta\n

UCS

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

Python is done with this file.

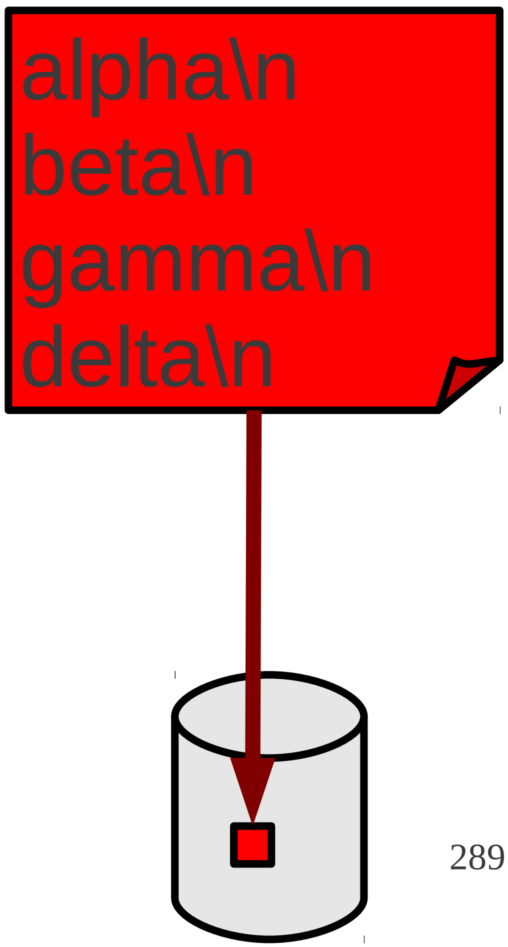
Data may not be written to disc until close()!



!

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

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



UCS

Progress

filename → `open()` → writable file object

```
data = open('input.dat', 'w')
```

`data.write(line)` line must include \n

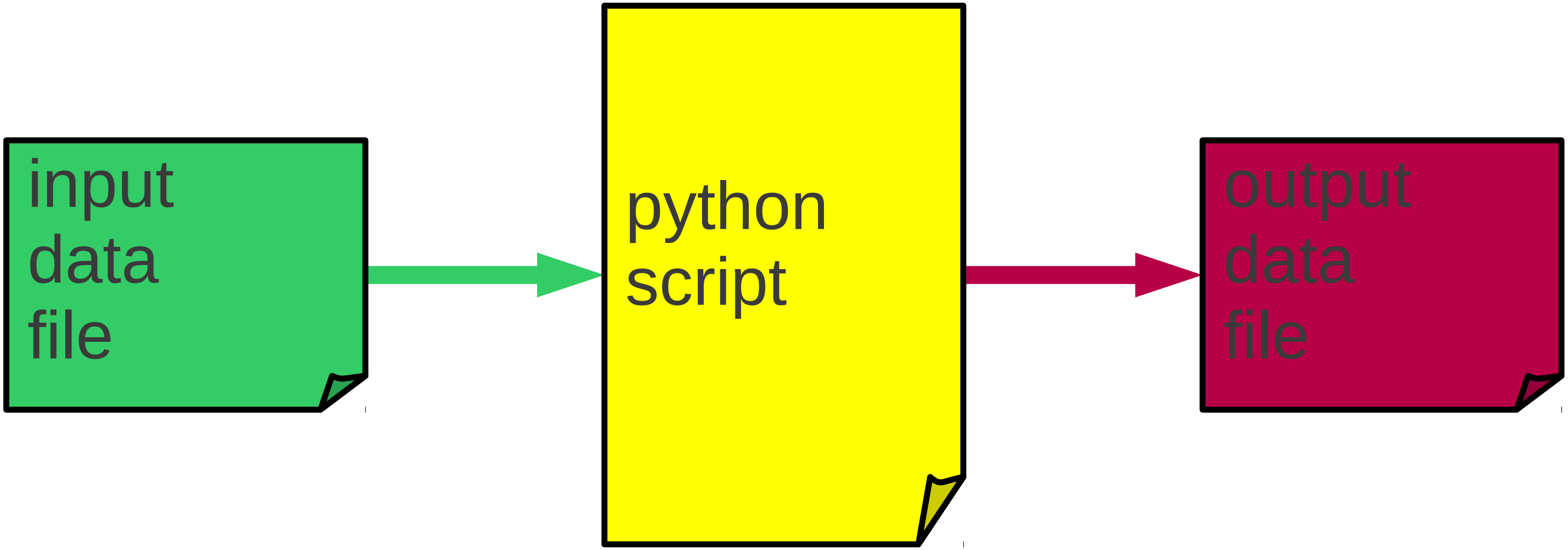
`data.close()` “flushes” to disc

Example

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

Example of a “filter”

Reads one file, writes another.



Example of a "filter"

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

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

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

Setup

Ugly!

Shutdown

filter1.py

Exercise

Change treasure.py
to do this:

Read `treasure.txt` and write `treasure.out`.
For each line write to the output:

line number

number of words on the line

number of characters in the line

separated by TABs.
At the end output a summary line

number of lines

total number of words

total number of characters

separated by TABs too.

15 minutes

Problem

…n…

```
…
results = []
for n in range(0,11):
    results.append(n**2 + n + 41)

…
```

A snippet of code using *n*

…n…

But what if *n* was already in use?

Solution in principle

...n...

```
…
results = []
for n in range(0,11):
    results.append(n**2 + n + 41)

…
```

Want to isolate this bit of code.

...n...

Keep it away from the external *n*.

Solution in principle

Pass in the *value* of the upper limit

```
…
results = []
for n in range(0,11):
    results.append(n**2 + n + 41)

…
```

The *names* used inside never get out

Pass out the calculated list's *value*.

Solution in practice

output

function

input

```
…
results = my_function( 11 )
…
```

Need to be able to define our own functions!

Defining our function

define

function name

input

colon

```
def my_function(limit):
```

indentation

Defining our function

Names are used
only in the function

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

Function definition

Defining our function

Pass back…

…this *value*

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

UCS

Using our function

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

…

results = my_function(11)
```

“answer” “limit”

Why use functions?

Reuse

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

Clarity

Clearly separated components are easier to read.

Reliability

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

A “real” worked example

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

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

Example 1

```
def norm2(values):

    sum = 0.0

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

    return sum
```

Example 1

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

↓

```
50.0
```

```
$ python norm2.py
```

```
50.0
169.0
```

[3.0, 4.0, 5.0]

[12.0, 5.0]

UCS

A second worked example

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

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

UCS

Example 2

```
def minimum(a_list):

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

    return a_min
```

? When will this go wrong?

Example 2

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

```
1.0
```

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

A third worked example

Write a function to “dot product” two vectors.

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

Example 3

```
def dot(a_vec, b_vec):

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

    return sum
```

? When will this go wrong?

Example 3

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

```
11.0
```

```
$ python dot_product.py
11.0
115
```

Example 3 — version 2

```
def dot(a_vec, b_vec):

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

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

    return sum
```

A fourth worked example

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

e.g.

[1, -2, 0, 5, -5, 3, 3, 6] → [1, 5, 3, 3, 6]

Example 4

```
def positive(a_list):

    answer = []

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

    return answer
```

Progress

Functions !

Defining them

Using them

Exercise

Write a function `list_max()` which takes two lists of the same length and returns a third list which contains, item by item the larger item from each list.

`list_max([1,5,7], [2,3,6])` → `[2,5,7]`

Hint: There is a built-in function `max(`*x*`,`*y*`)` which gives the maximum of two values.

15 minutes

How to return more than one value?

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

Returning two values

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

Pair of values

UCS

Receiving two values

```
…

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

(minval, maxval) = min_max(values)
```

Pair of variables

```
print minval
print maxval
```

UCS

Pairs, triplets, …

singles
doubles
triples
quadruples
quintuples
…

“tuples”

Tuples ≠ Lists

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

Tuple examples

Pair of measurements of a tree

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

Details about a person

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

Progress

Tuples

“not lists”

Multiple values bound together

Functions returning multiple values

Exercise

Copy the `min_max()` function.

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

10 minutes

Tuples and string substitution

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

Simple string substitution

Substitution marker

Substitution operator

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

`%s` Substitute a string.

Simple integer substitution

Substitution marker

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

%d Substitute an integer.

Tuple substitution

Two markers

A pair

```
>>> '''My name is %s and
... I am %d years old.''' % ('Bob', 46)

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

Lists of tuples

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

List of tuples

Tuple of variable names

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

Problem: ugly output

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

Columns should align

Columns of numbers should be right aligned

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

Solution: formatting

```
'%s'   % 'Bob'  →  'Bob'
'%5s'  % 'Bob'  →  '␣␣Bob'
'%-5s' % 'Bob'  →  'Bob␣␣'

'%5s'  % 'Charles'  →  'Charles'
```

Five characters

Right aligned

Left aligned

Solution: formatting

```
'%d'    % 46  ⟶  '46'

'%5d'   % 46  ⟶  '␣␣␣46'

'%-5d'  % 46  ⟶  '46␣␣␣'

'%05d'  % 46  ⟶  '00046'
```

Columnar output

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


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

Properly formatted

Floats

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

'%.4f' % 3.141592653589  →  '3.1416'

'%.4f' % 3.1             →  '3.1000'
```

Progress

Formatting operator `'%s %d' % ('Bob', 46)`

Formatting markers `%s` `%d` `%f`

Formatting modifiers `%-4s`

Exercise

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

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

5 minutes

Reusing our functions

Want to use the same function in many scripts

Copy?	Have to copy any changes.
Single instance?	Have to *import* the set of functions.

How to reuse — 0

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


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

(x, y) = min_max(vals)

print(x, y)
```

five.py

UCS

How to reuse — 1

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

(x, y) = min_max(vals)

print(x, y)
```

five.py

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

utils.py

Move the definition of the function to a separate file.

How to reuse — 2

```
import utils



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

(x, y) = min_max(vals)

print(x, y)
```

five.py

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

utils.py

Identify the file with the functions in it.

How to reuse — 3

```
import utils



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

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

print(x, y)
```

five.py

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

utils.py

Indicate that the function comes from that import.

A library of our functions

“Module”

Container

Functions

Objects

Parameters

UCS

System modules

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

Using a system module

```
>>> import math

>>> math.sqrt(2.0)
1.4142135623730951

>>>
```

Keep track of the module with the function.

Don't do this

```
>>> from math import sqrt

>>> sqrt(2.0)
1.4142135623730951

>>>
```

Really don't do this

```
>>> from math import *

>>> sqrt(2.0)
1.4142135623730951

>>>
```

Do do this

```
>>> import math

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

Progress

“Modules”

System modules

Personal modules

`import` *`module`*

`module``.function(...)`

Exercise

1. Edit your `utils.py` file.

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

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

5 minutes

Interacting with the **sys**tem

```
>>> import sys
```

Standard input and output

```
>>> import sys
```

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

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

Line-by-line copying — 1

```
import sys

for line in sys.stdin:


    sys.stdout.write(line)
```

Import module

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

Line-by-line copying — 2

```
import sys

for line in sys.stdin:

    sys.stdout.write(line)
```

Standard input

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

Line-by-line copying — 3

```
import sys

for line in sys.stdin:

    sys.stdout.write(line)
```

Standard output

An open file

The file's `write()` method

UCS

Line-by-line copying — 4

```
import sys

for line in sys.stdin:

    sys.stdout.write(line)
```

Lines in…
lines out

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

Copy

Line-by-line actions

Copying lines unchanged

Changing the lines

Gathering statistics

Only copying certain lines

Line-by-line rewriting

```
import sys

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

Define or import a function here

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

Process

Line-by-line filtering

```
import sys

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

Define or import a test function here

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

Filter

Progress

`sys` module

`sys.stdin`	Standard input
`sys.stdout`	Standard output

“Filter” scripts — process line-by-line

only output on certain input lines

Exercise

Write a script that reads from standard input.

If should generate two lines of output:

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

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

5 minutes

The command line

We are putting parameters in our scripts.

```
…
number = 1.25
…
```

We want to put them on the command line.

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

Reading the command line

```
import sys

print(sys.argv)
```

`$ python args.py 1.25`

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

sys.argv[0] — Script's name

sys.argv[1] — First argument → A string!

Command line strings

```
import sys

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

print(number)
```

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

Using the command line

```
import sys

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

print(number)
```

Enough arguments?

Valid as floats?

UCS

Better tools for the command line

`argparse` module

Very powerful parsing
Experienced scripters

General principles

1. Read in the command line
2. Convert to values of the right types
3. Feed those values into calculating functions
4. Output the calculated results

Worked example

Write a script to print points

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

Two command line arguments:

r (float) power

N (integer) number of points

General approach

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

1b. Write a script that tests that function.

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

2b. Write a script that tests that function.

3. Combine the two functions.

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

```
import sys

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

    return (pow, num)
```

curve.py

1b. Write a script that tests that function.

```
import sys

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

curve.py

1b. Write a script that tests that function.

```
$ python curve.py 0.5 5
Power:  0.500000
Points: 5
```

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

```
def power_curve(pow, num_points):

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

curve.py

2b. Write a script that tests that function.

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

power_curve(0.5, 5)
```

curve.py

UCS

2b. Write a script that tests that function.

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

3. Combine the two functions.

```
import sys

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

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

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

curve.py

Progress

Parsing the command line

`sys.argv`

Convert from strings to useful types

`int()   float()`

Exercise

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

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

5 minutes

Back to our own module

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

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

We want to do better than this.

Function help

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

```
Help on function min_max in
module utils:

min_max(numbers)
```

Annotating a function

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

Our current file

A “documentation string”

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

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

A string before the body of the function.

Annotated function

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

```
Help on function min_max in
module utils:

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

UCS

Annotating a module

```
"""A personal utility module
full of all the pythonic goodness
I have ever written.
"""
def min_max(numbers):

    """This functions takes a list
    of numbers and returns a pair
    of their minimum and maximum.
    """

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

A string before any active part of the module.

Annotated module

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

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

Progress

Annotations

…of functions

…of modules

“Doc strings”

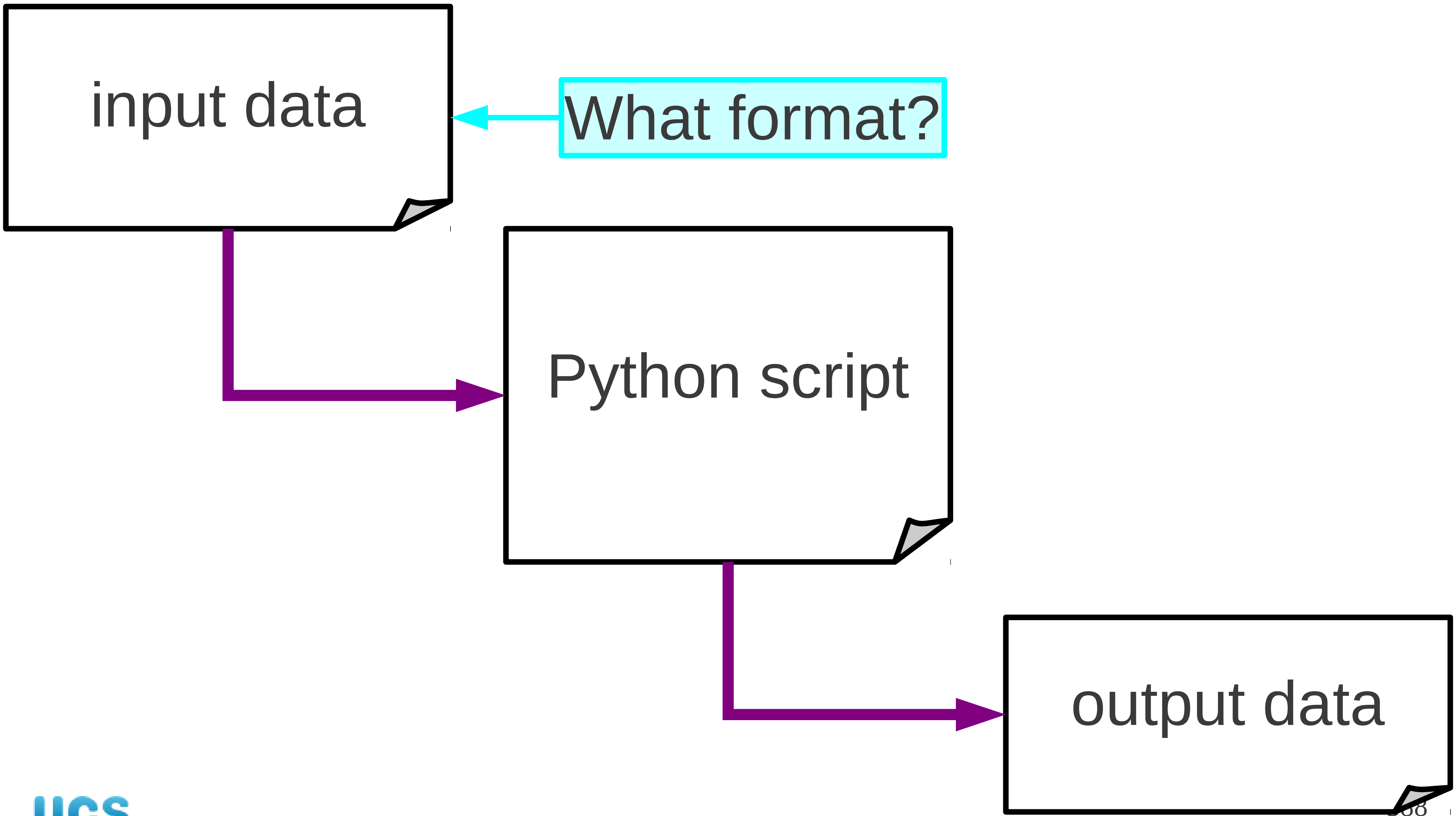
`help()`

Exercise

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

3 minutes

Simple data processing



Comma Separated Values

input data

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

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

Quick and dirty .csv — 1

CSV: “comma separated values”

More likely to have come from sys.stdin

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

>>> line.split(',')
['1.0', ' 2.0', ' 3.0', ' 4.0\n']
```

Split on commas rather than spaces.

Note the leading and trailing white space.

UCS

Quick and dirty .csv — 2

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

UCS

Quick and dirty .csv — 3

Why “quick and dirty”?

Can't cope with common cases:

Quotes `'"1.0","2.0","3.0","4.0"'`

Commas `'A,B\,C,D'`

Dedicated module: `csv`

Proper .csv

Dedicated module: `csv`

```
import csv
import sys

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

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

Much more in the “**Python: Further Topics**” course

Processing data

Storing data in the program

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

`...`

? id → (name, age, height, weight) ?

Simpler case

Storing data in the program

id	name
`A101`	`Joe`
`G042`	`Fred`
`H003`	`Bess`
`...`	

? id → name ?

Not the same as a list…

index	name
0	`Joe`
1	`Fred`
2	`Bess`
`...`	

names[1] = 'Fred'

['Joe', 'Fred', 'Bess', …]

…but similar: a “dictionary”

id	name
`A101`	`Joe`
`G042`	`Fred`
`H003`	`Bess`
`...`	

names['G042'] = 'Fred'

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

Dictionaries

Generalized look up

“key” → “value”		Python object (immutable)	→	Python object (arbitrary)
'G042'	→	'Fred'	string	→ string
1700045	→	29347565	int	→ int
'G042'	→	('Fred', 34)	string	→ tuple
(34, 56)	→	'treasure'	tuple	→ string
(5,6)	→	[5, 6, 10, 12]	tuple	→ list

Building a dictionary — 1

Curly brackets

Items

Comma

```
data = {'A101':'Joe','G042':'Fred','H003':'Bess'}
```

Key

colon

Value

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

Building a dictionary — 2

```
data = {}

data [ 'A101' ] = 'Joe'

data [ 'G042' ] = 'Fred'

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

Empty dictionary

Square brackets

Key

Value

A101	→	Joe
G042	→	Fred
H003	→	Bess

UCS

Example — 1

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

>>> data

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

Order is not preserved!

Example — 2

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

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

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

Square brackets in Python

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

Example — 3

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

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

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

Progress

Dictionaries

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

```
data['G042']  ⟶  ('Fred',34)
```

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

UCS

Exercise

Write a script that:

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

10 minutes

Worked example — 1

Reading a file to populate a dictionary

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

`elements.txt` File

↓

`symbol_to_name` Dictionary

UCS

Worked example — 2

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

Open file

```
symbol_to_name = {}
```

Empty dictionary

Read data

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

Populate dictionary

```
data.close()
```

Close file

Now ready to use the dictionary

Worked example — 3

Reading a file to populate a dictionary

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

`names.txt`

↓

`key_to_name`

Worked example — 4

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

key_to_name = {}


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

data.close()
```

Make it a function!

```
symbol_to_name = {}

data = open('elements.txt')

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

data.close()
```

UCS

Make it a function!

```
symbol_to_name = {}

data = open('elements.txt')

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

   symbol_to_name[symbol] = name

data.close()
```

Input

Make it a function!

```
def filename_to_dict(filename):

    symbol_to_name = {}

    data = open(filename)

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

       symbol_to_name[symbol] = name

    data.close()
```

Input

Make it a function!

```
def filename_to_dict(filename):

    symbol_to_name = {}

    data = open(filename )

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

    data.close()
```

Output

Make it a function!

```
def filename_to_dict(filename):

    x_to_y  = {}

    data = open(filename )

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

    data.close()
```

Output

Make it a function!

```
def filename_to_dict(filename):

    x_to_y  = {}

    data = open(filename )

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

    data.close()

    return(x_to_y)
```

Output

Exercise

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

2. Write a script that does this:

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

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

   c. For each line, prints out the element name

UCS

10 minutes

Keys in a dictionary?

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

“Treat it like a list”

“Treat it like a list”

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

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

“Treat it like a list”

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

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

Missing key?

```
>>> data = {'a':'alpha', 'b':'beta'}

>>> data['g']

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

Dictionary equivalent of “index out of range”

“Treat it like a list”

```
if item in list:
    blah blah
    …item…
    blah blah
```

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

UCS

Convert to a list

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

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

Progress

Keys in a dictionary

"Treat it like a list"

```
list(dictionary) ⟶ [keys]

for key in dictionary:
    ...

if key in dictionary:
    ...
```

Exercise

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

`symbol_to_name` 'Li' → 'Lithium'

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

`name_to_symbol` 'Lithium' → 'Li'

10 minutes

One last example

Word counting

Given a text, what words appear and how often?

Word counting algorithm

Run through file line-by-line

Run through line word-by-word

Clean up word

Is word in dictionary?

If not: add word as key with value 0

Increment the counter for that word

Output words alphabetically

Word counting in Python: 1

```
# Set up
import sys
```

Need sys for sys.argv

```
count = {}
```

Empty dictionary

```
data = open(sys.argv[1])
```

Filename on command line

Word counting in Python: 2

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

Lines

Words

We need to write this function.

Word counting in Python: 3

Insert at *start* of script

“Placeholder” function

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

UCS

Word counting in Python: 4

```
clean_word = cleanup(word)

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

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

Two levels indented

Create new entry in dictionary?

Increment count for word

Word counting in Python: 5

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

data.close()

words = list(count)

words.sort()
```

Be tidy!

All the words

Alphabetical order

Word counting in Python: 6

```
words.sort()

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

Alphabetical order

Run it!

```
$  python counter.py treasure.txt
```

What changes would you make to the script?

And we're done!

Python types

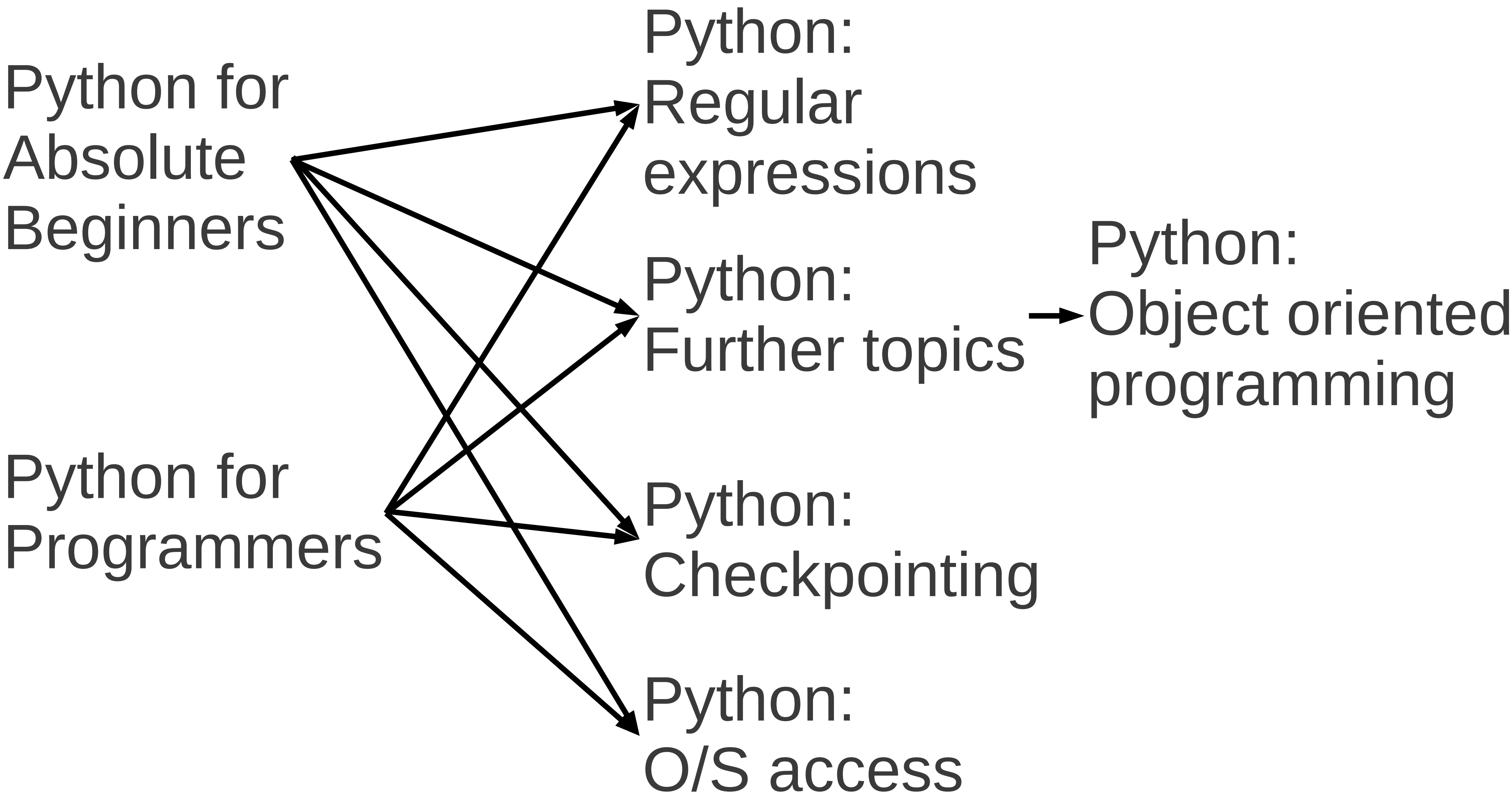
Python control structures

Python functions

Python modules

and now you are ready
to do things with Python!

More Python


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