

Python: Introduction for Absolute Beginners

Bruce Beckles
Bob Dowling

Day three

University Computing Service

Scientific Computing Support e-mail address:

escience-support@ucs.cam.ac.uk

Homework 3

'GCU' → 'Ala'

'GCC' → 'Ala'

'AUG' → 'Met'



'Ala' → ['GCU', 'GCC']

'Met' → ['AUG']

Reverse a “many-to-one” mapping.
Apply it to `amino.py`.

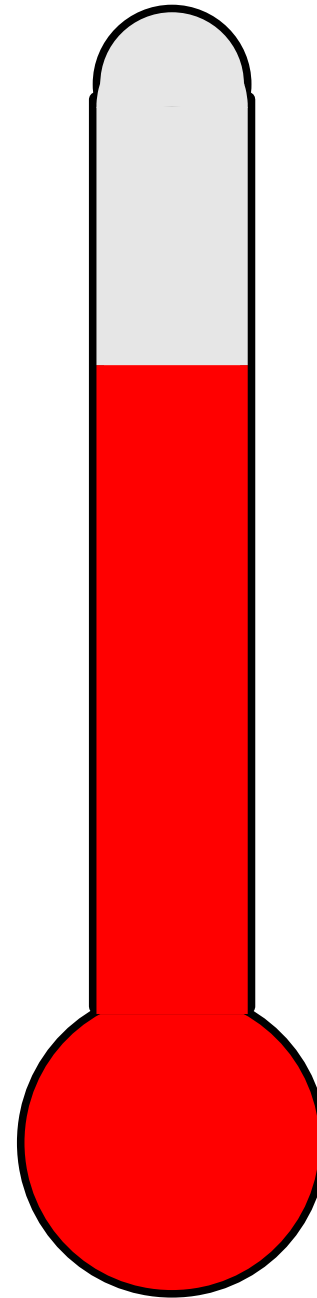
Function to
reverse a
many-to-one
dictionary

```
def reverse2(a_to_b):  
    b_to_list_of_a = {}  
  
    for a in a_to_b:  
        b = a_to_b[a]  
  
        if not b in b_to_list_of_a:  
            b_to_list_of_a[b] = []  
  
        b_to_list_of_a[b].append(a)  
  
    return b_to_list_of_a
```

...

```
triples = reverse2(acids)  
print_dict(triples)
```

- Launching Python
- Values and types
- Variables
- Tests
- Control structures
- Lists
- Dictionaries
- Functions



Any questions?

Function
to reverse a
dictionary

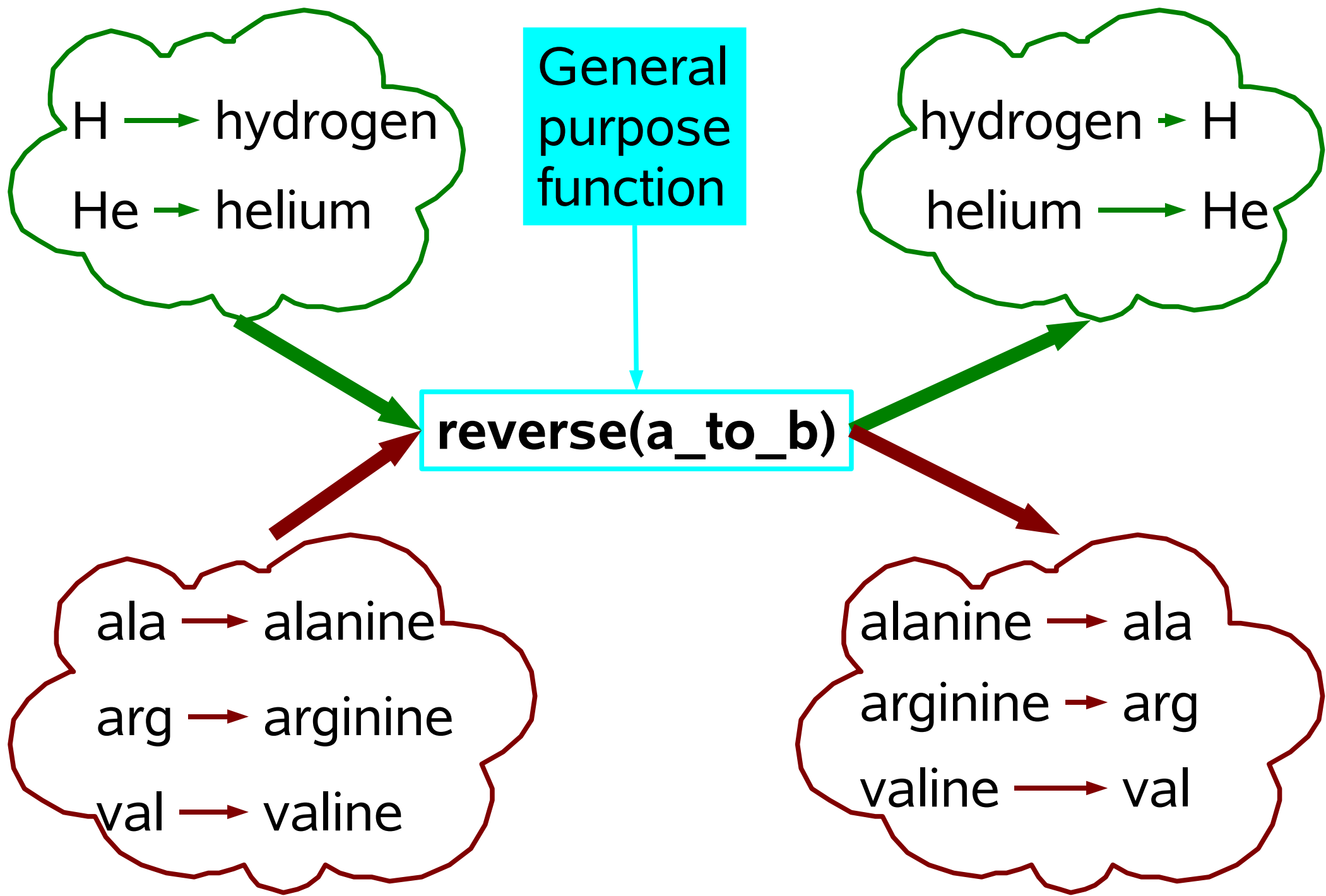
```
def reverse(a_to_b):  
    b_to_a = {}  
    for a in a_to_b:  
        b = a_to_b[a]  
        b_to_a[b] = a  
    return b_to_a
```

Function
to print a
dictionary

```
def print_dict(a_to_b):  
    for a in a_to_b:  
        print a, a_to_b[a]
```

Main body
of script

```
english_to_spanish = {...}  
  
spanish_to_english =  
    reverse(english_to_spanish)  
print_dict(spanish_to_english)
```



Re-using functions in multiple scripts

```
reverse()  
reverse2()  
print_dict()  
...
```

(1) Move the function definitions into a file, `utils.py`.

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

```
def print_dict(a_to_b):  
    for a in a_to_b:  
        print a, a_to_b[a]
```

`utils.py`

```
english_to_spanish = {...}  
spanish_to_english =  
    reverse(english_to_spanish)  
print_dict(spanish_to_english)  
translation1.py
```

```
import utils
```

(2) Replace them with the line
`import utils`
to get their definitions back again.

```
english_to_spanish = {...}
```

```
spanish_to_english =  
    reverse(english_to_spanish)  
print_dict(spanish_to_english)
```

```
translation1.py
```

```
import utils
```

(3) Put “utils.” at the start of the function names.

```
english_to_spanish = {...}
```

```
spanish_to_english =  
    utils.reverse(english_to_spanish)  
utils.print_dict(spanish_to_english)
```

```
translation1.py
```

```
import utils
...
long_names=utils.reverse(acids)
utils.print_dict(long_names)
```

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

```
import utils
...
chemicals=utils.reverse(numbers)
utils.print_dict(chemicals)
```

```
def reverse(a_to_b):
    ...
def print_dict(dict):
    ...
utils.py
```

Easy to reuse
functions in
`utils.py` in
many scripts.

The utils “module”

```
def reverse(a_to_b):  
    ...  
  
def print_dict(dict):  
    ...  
  
utils.py
```

Exercise

1. Take the homework answer, `amino.py`.
2. Move `reverse2()` to `utils.py`.
3. Modify `amino.py` to use `utils`.

```
...
def reverse2(a_to_b):
    b_to_list_of_a = {}

    for a in a_to_b:
        b = a_to_b[a]

        if not b in b_to_list_of_a:
            b_to_list_of_a[b] = []

        b_to_list_of_a[b].append(a)

    return b_to_list_of_a

... amino_reversed.py
```

profile getpass re bisect os gzip pickle time
anydbm bz2
calendar
unittest atexit shelve datetime
cgi csv asyncore
optparse asynchat webbrowser mmap hmac
BaseHTTPServer math sched
SimpleHTTPServer cmath heapq
CGIHTTPServer imageop
Cookie logging sys unicodedata base64 sets
codecs stringprep tempfile hashlib mutex
select string chunk code ConfigParser
locale cmd linecache glob
colorsys gettext collections

System modules

Functions again

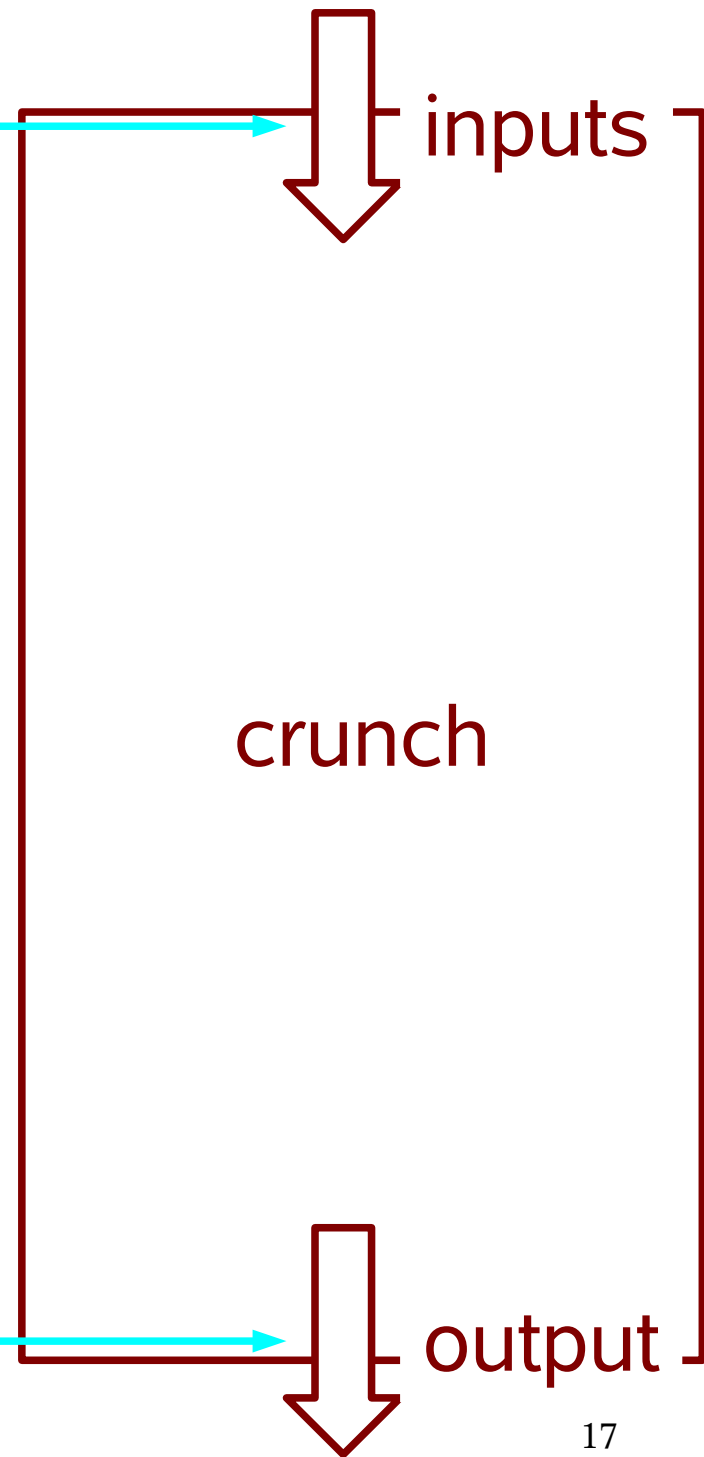
More complex
input and output.

def crunch(input):

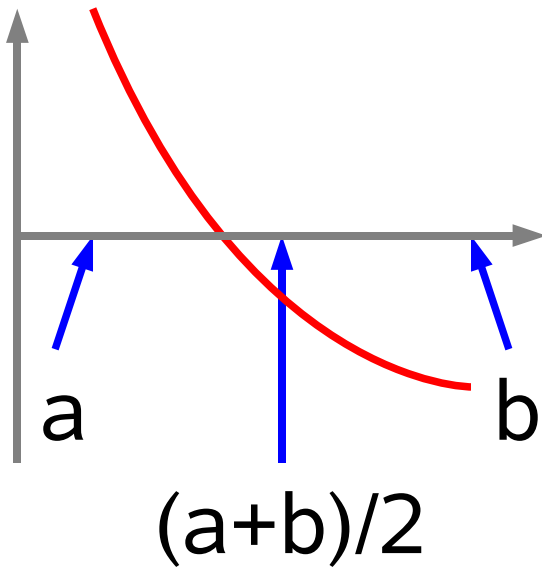
multiple inputs?

multiple outputs?

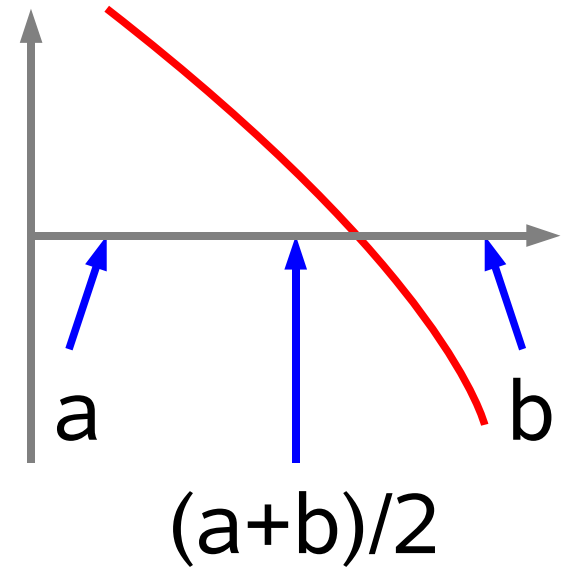
return output



Root-finding by bisection of intervals



$$y=f(x)$$



a, b



$a, (a+b)/2$

until $b-a < \delta$

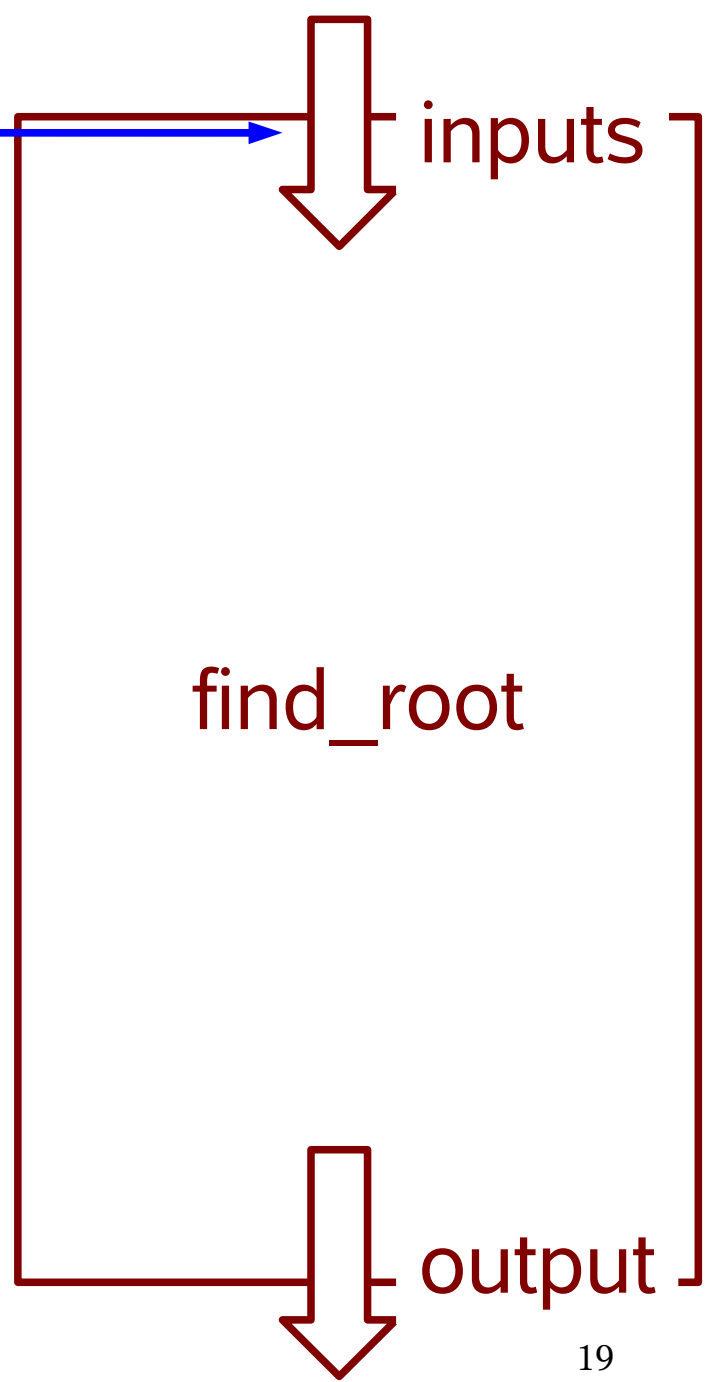
a, b



$(a+b)/2, b$

function
lower bound
upper bound
tolerance

f
a
b
 δ



multiple arguments separated by commas

```
def find_root ( f , a , b , d ):
```

function having its root found

lower bound

upper bound

tolerance (“ δ ”)

really bad variable names

```
def find_root ( f , a , b , d ):
```

function
having
root found



bound

tolerance
("δ")

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

decent
variable
names

function body

```
return ...
```

```
def find_root(
```

```
function,  
lower,  
upper,  
tolerance
```

```
):
```

```
while upper - lower > tolerance:
```

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

```
    if function(upper)*function(middle) > 0.0:
```

```
        upper = middle
```

```
    else:
```

```
        lower = middle
```

```
return upper
```

inputs

n.b. float

function
body

output

utils.py

```
#!/usr/bin/python
```

```
import utils
```

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

```
print utils.find_root(poly, 0.0, 2.0, 0.0001)
```

poly.py

import
module

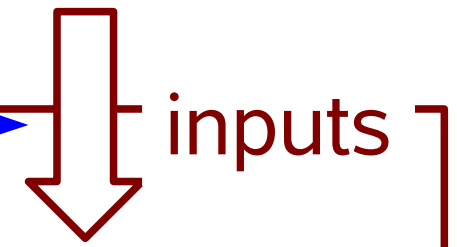
function
whose
root we
want

use the
module
function

```
> python poly.py
```

```
1.41424560547
```

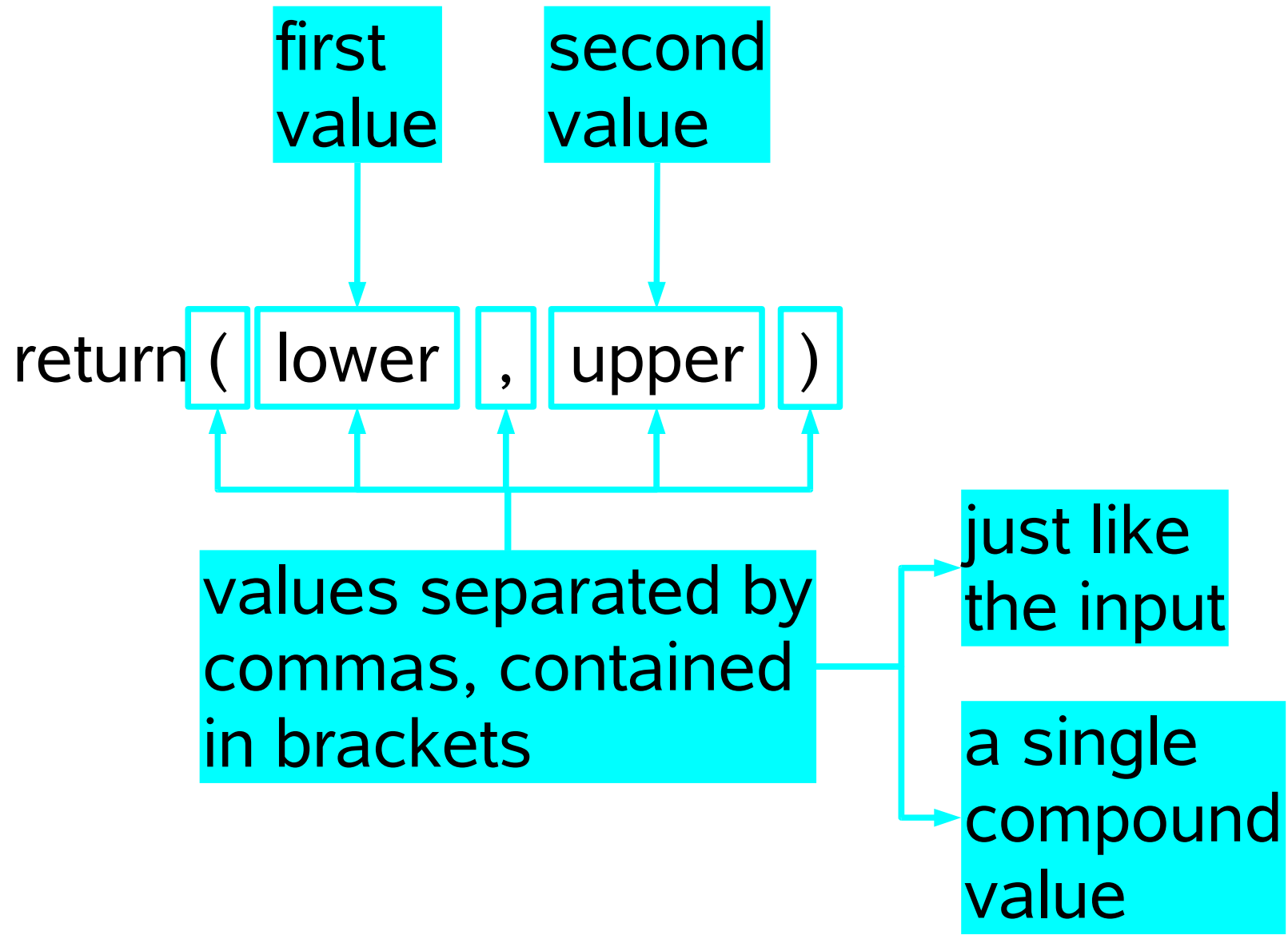
function f
lower bound a
upper bound b
tolerance δ



find_root

new lower bound a'
new upper bound b'





```
def find_root(
    function,
    lower,
    upper,
    tolerance
):
    while upper - lower > tolerance:
        middle = (lower + upper)/ 2.0
        if function(upper)*function(middle) > 0.0:
            upper = middle
        else:
            lower = middle
    return (lower, upper)
```

return a *pair*
of values

utils.py

```
#!/usr/bin/python
import utils
```

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

```
print utils.find_root(poly, 0.0, 2.0, 0.0001)
```

poly.py

> **python poly.py**

(1.4141845703125, 1.41424560546875)

```
#!/usr/bin/python
import utils
```

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

```
(below, above) = utils.find_root(poly, 0.0, 2.0, 0.0001)
```

```
print below
print above
```

poly.py

Assign the pair
of values to a
pair of variables.

Tuples

Singles
Doubles
Triples
Quadruples
Quintets

(42 , 1.90 , 'Bob')

(-1 , +1)

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

Brackets in Python

[item₁, item₂, ...]

Start and end
of explicit list

listname [index]

Surround the index

listname [index₁ : index₂]

Slices

{ key₁ : value₁, key₂ : value₂, ... }

Start and end
of explicit dictionary

dictionaryname [key]

Surround the key

(item₁, item₂, ...)

Start and end of tuple

Tuples

“not the same as lists”

“tuples are not the same as lists”

(minimum, maximum)

(height, age, name)

(age, height, name)

(height, age, name, weight)

Independent,
grouped
components

Related,
sequential
items

[2, 3, 5, 7, 11]

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

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

...

“tuples are not the same as lists”

(min, max) = (0.001, 0.002)

x = (0.001, 0.002)

(min, max) = x

(age, height, name)
= (42, 1.95, 'Bob')

Simultaneous access

Sequential access

for item in list:

...

“tuples are not the same as lists”

[1, 2, 3, 4, 6, 6, 7]

5



('Bob', 1.95, 42)

43



“immutable”

Brackets in Python

[item₁, item₂, ...]

Start and end
of explicit list

listname [index]

Surround the index

listname [index₁ : index₂]

Slices

{ key₁ : value₁, key₂ : value₂, ... }

Start and end
of explicit dictionary

dictionaryname [key]

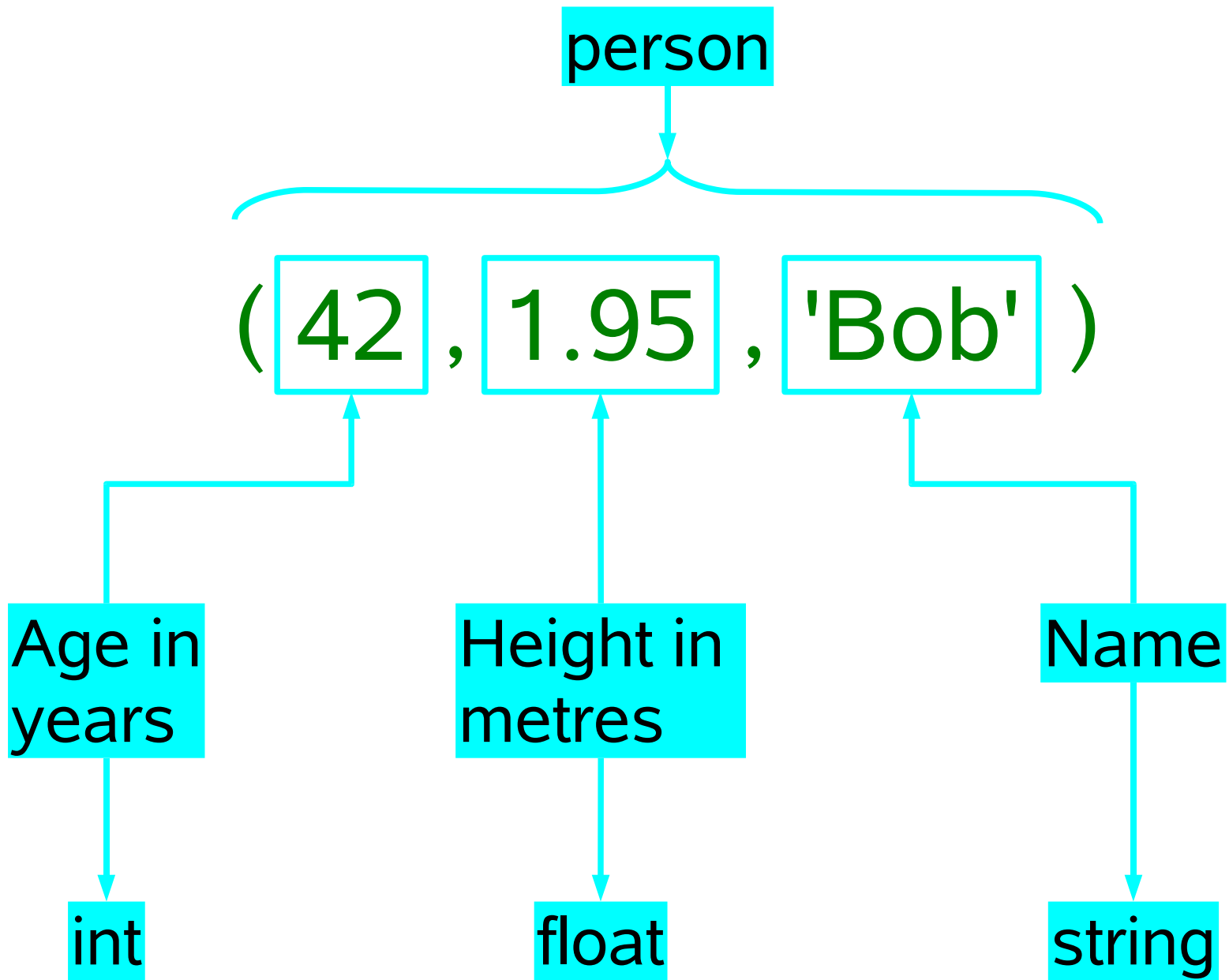
Surround the key

(item₁, item₂, ...)

Start and end of tuple

tuplename [index]

Surround the index



Is a tuple a thing or a collection of things?

Thing

Yes

Collection
of things

```
>>> x = (42, 1.90, 'Bob')
```

```
>>> x  
(42, 1.9, 'Bob')
```

```
>>> type(x)  
<type 'tuple'>
```

```
>>> (a,b,c) = (42, 1.90, 'Bob')
```

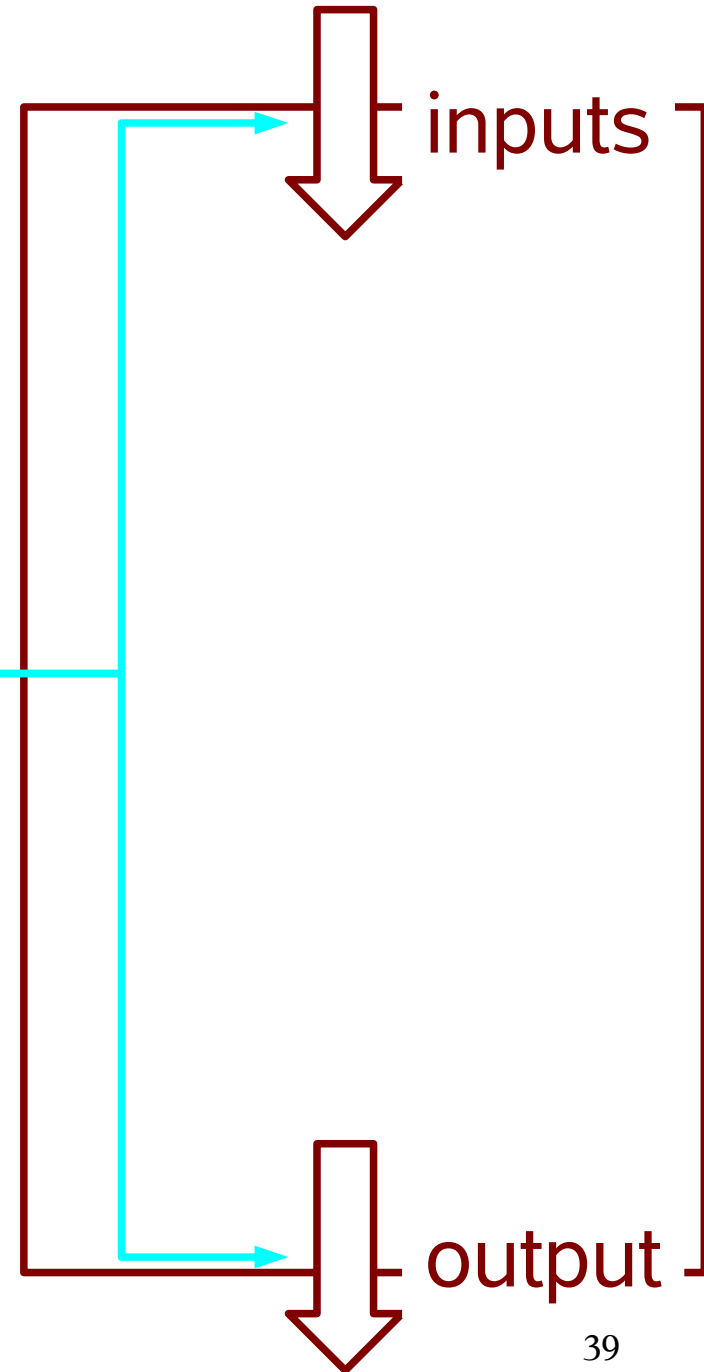
```
>>> a  
42
```

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

Using tuples

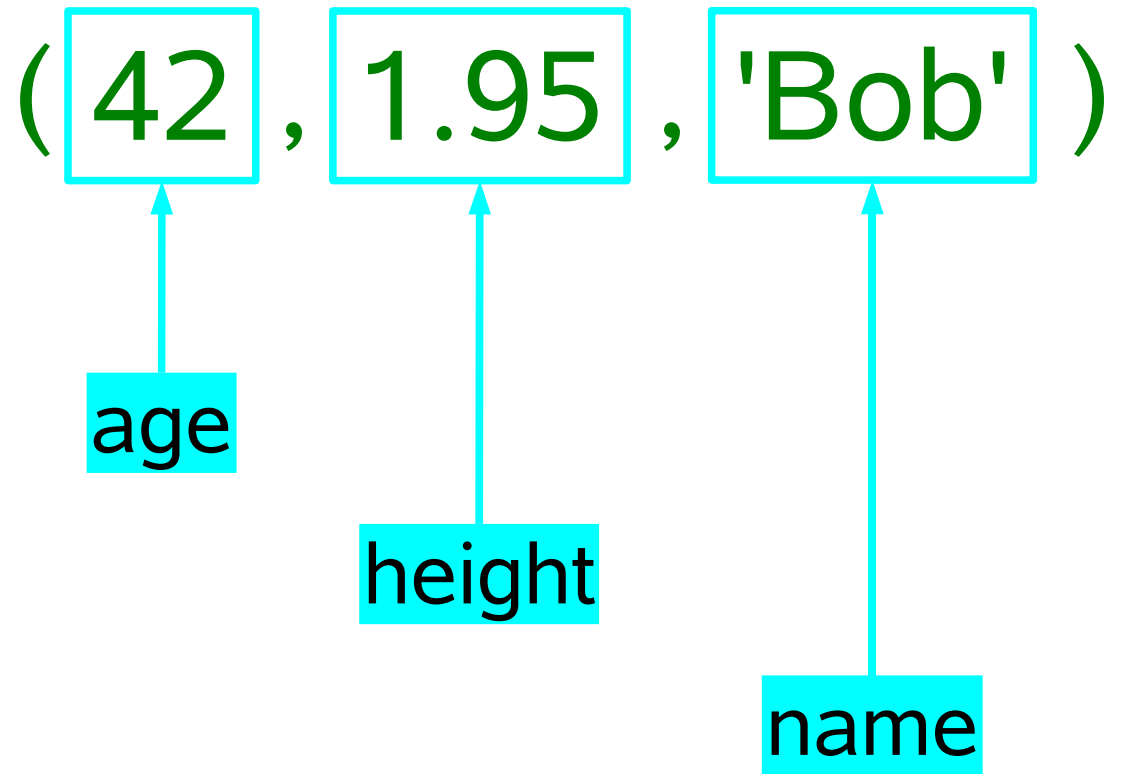
1. Functions

Multiple inputs
Multiple outputs



Using tuples

1. Functions
2. Related data



e.g. `#!/usr/bin/python`

`# symbol -> (name, atomic number, boiling pt.)`

```
symbol_to_properties = {  
    'H': ('hydrogen', 1, 20.28),  
    'He': ('helium', 2, 4.22),  
    'Li': ('lithium', 3, 1615.0),  
    ...  
    'Th': ('thorium', 90, 5061.0),  
    'Pa': ('protactinium', 91, 4300.0),  
    'U': ('uranium', 92, 4404.0),  
}
```

`(name, number, boil) = symbol_to_properties['He']`

`chemicals4.py`

Using tuples

1. Functions
2. Related data
3. String substitution

'Hi, I'm 42 years old,
1.95 metres tall and my
name is Bob.'

(42 , 1.95 , 'Bob')

A diagram illustrating the mapping between a tuple and a string. The tuple (42, 1.95, 'Bob') is shown at the bottom, with each element enclosed in a cyan box. Three cyan arrows originate from these boxes: one from '42' pointing to the number '42' in the string, one from '1.95' pointing to '1.95', and one from ''Bob'' pointing to 'Bob'.

'Hi, I am %d years old,
%f metres tall, and
my name is %s.'

%

%
The “substitution
operator”

(42 , 1.95 , 'Bob')

'Hi, I am 42 years old,
1.950000 metres tall, and
my name is Bob.'

String substitution

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

substitution
operator

%f

substitute a float

String substitution

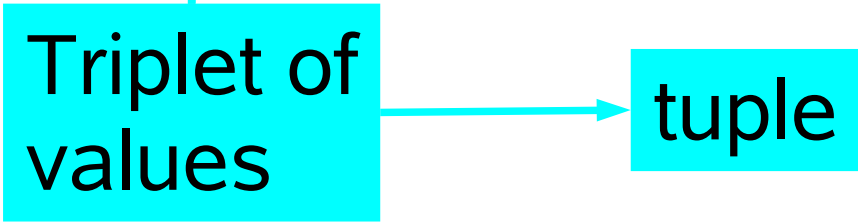
```
>>> '%d years old, %f metres tall, called %s .' %  
( 42, 1.95, 'Bob' )  
'42 years old, 1.950000 metres tall, called Bob .'
```

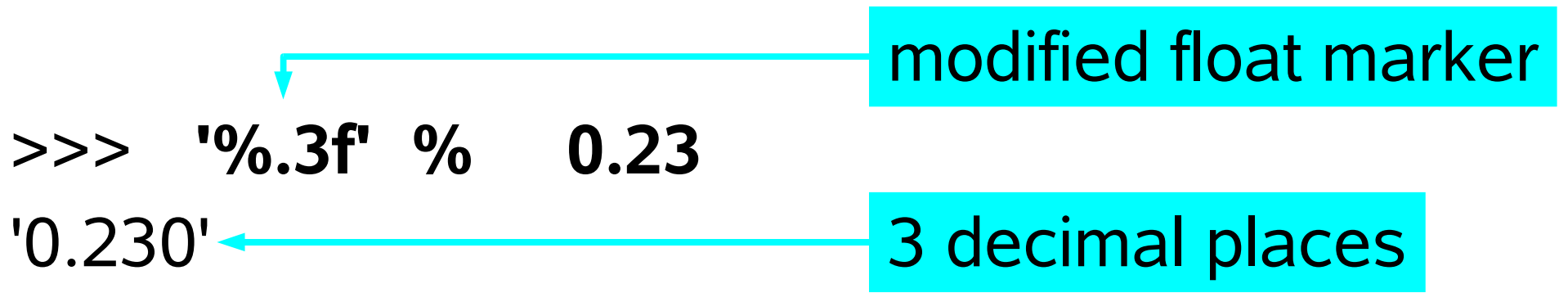
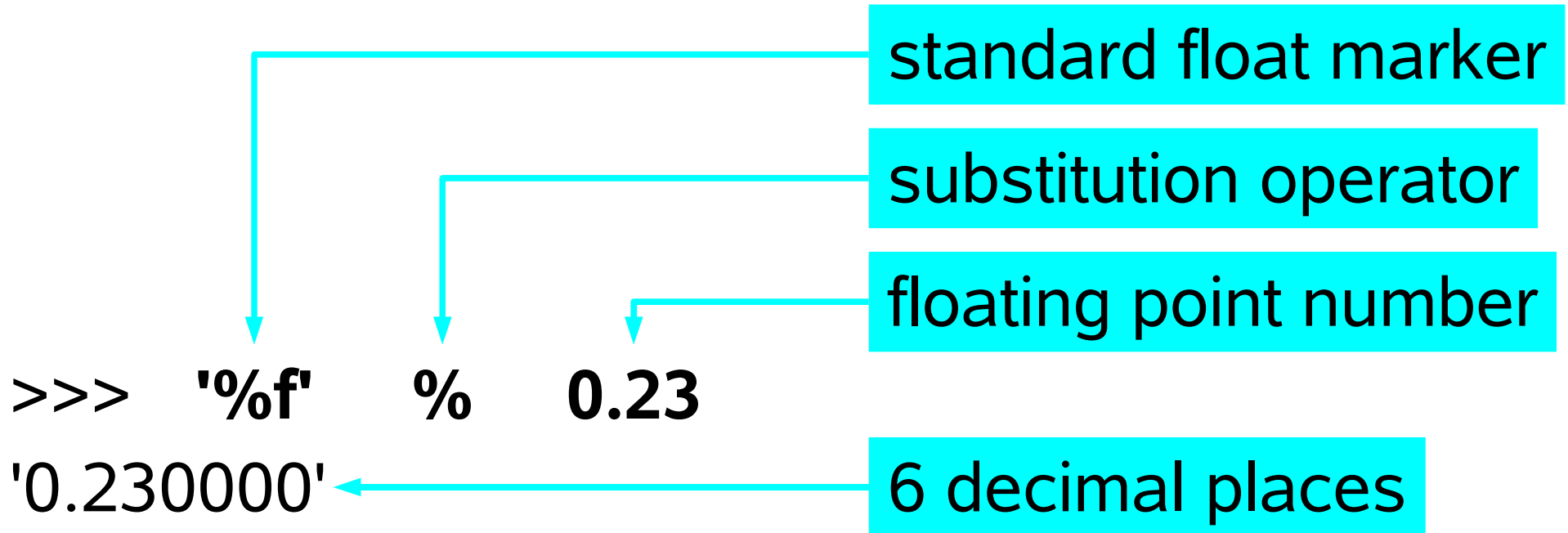
The diagram illustrates the string substitution process. It shows a format string with three placeholders: a blue box around '%d', a red box around '%f', and a green box around '%s'. Below the format string, a tuple of values is shown: a blue box around '42', a red box around '1.95', and a green box around ''Bob''. Lines connect the blue box of the format string to the blue box of the value '42', the red box of the format string to the red box of the value '1.95', and the green box of the format string to the green box of the value ''Bob''. Below this, the resulting string is shown with the placeholders replaced by their corresponding values: a blue box around '42', a red box around '1.950000', and a green box around 'Bob'.

String substitution

```
>>> '%d years old, %f metres tall, called %s .' %
```

```
( 42, 1.95, 'Bob' )
```





```
>>> '%d years old, %.2f metres tall, called %s .' %  
( 42, 1.95, 'Bob' )  
'42 years old, 1.95 metres tall, called Bob .'
```

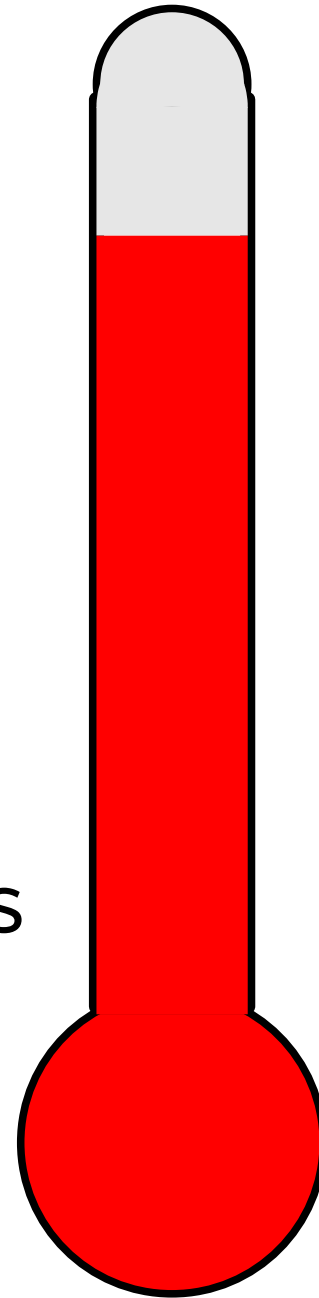
The diagram illustrates the mapping of format specifiers to values in a Python string formatting operation. The format string is `'%d years old, %.2f metres tall, called %s .'`. The values to be formatted are `42`, `1.95`, and `'Bob'`. The resulting output string is `'42 years old, 1.95 metres tall, called Bob .'`. The connections are as follows:

- `%d` (blue box) maps to `42` (blue box).
- `%.2f` (red box) maps to `1.95` (red box).
- `%s` (green box) maps to `'Bob'` (green box).

More complex formatting possible

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

- Modules
- Tuples:
 - Function arguments
 - Function return values
 - Gathering related values
 - String substitution



Exercise

Write a function `min_max()` in `utils.py`.

List of floating point numbers

List

→

(min, max)

Tuple

[1.0, 2.0, 3.0, -1.0, -5.0, 7.0]

→

(-5.0, 7.0)

[1.0, 2.0, 3.0]

→

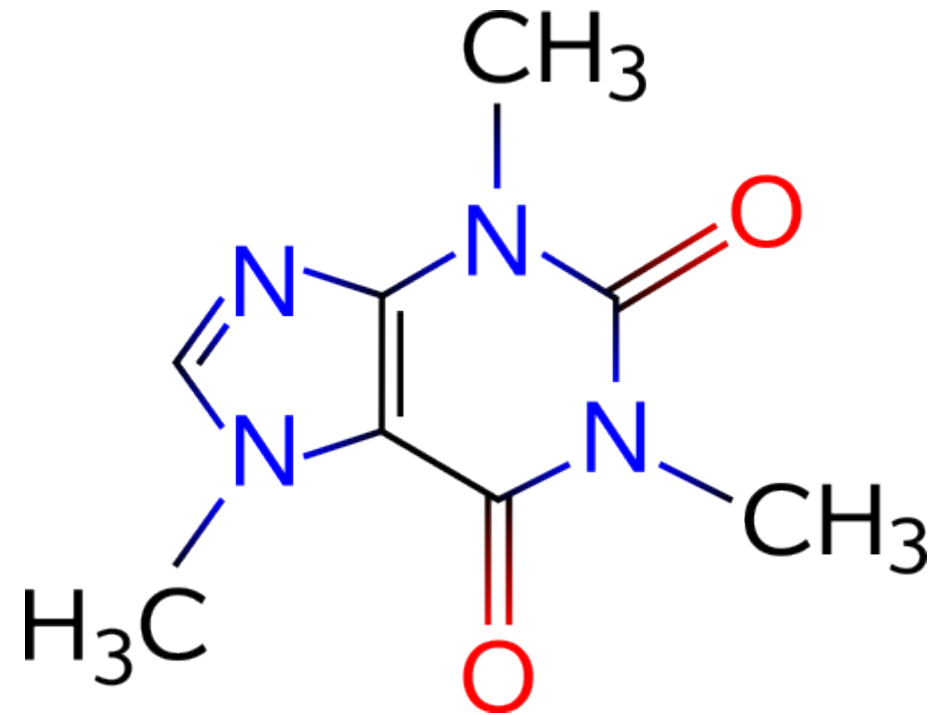
(1.0, 3.0)

[1.0, -2.0, 3.0]

→

(-2.0, 3.0)

10 minutes



`min_max()`

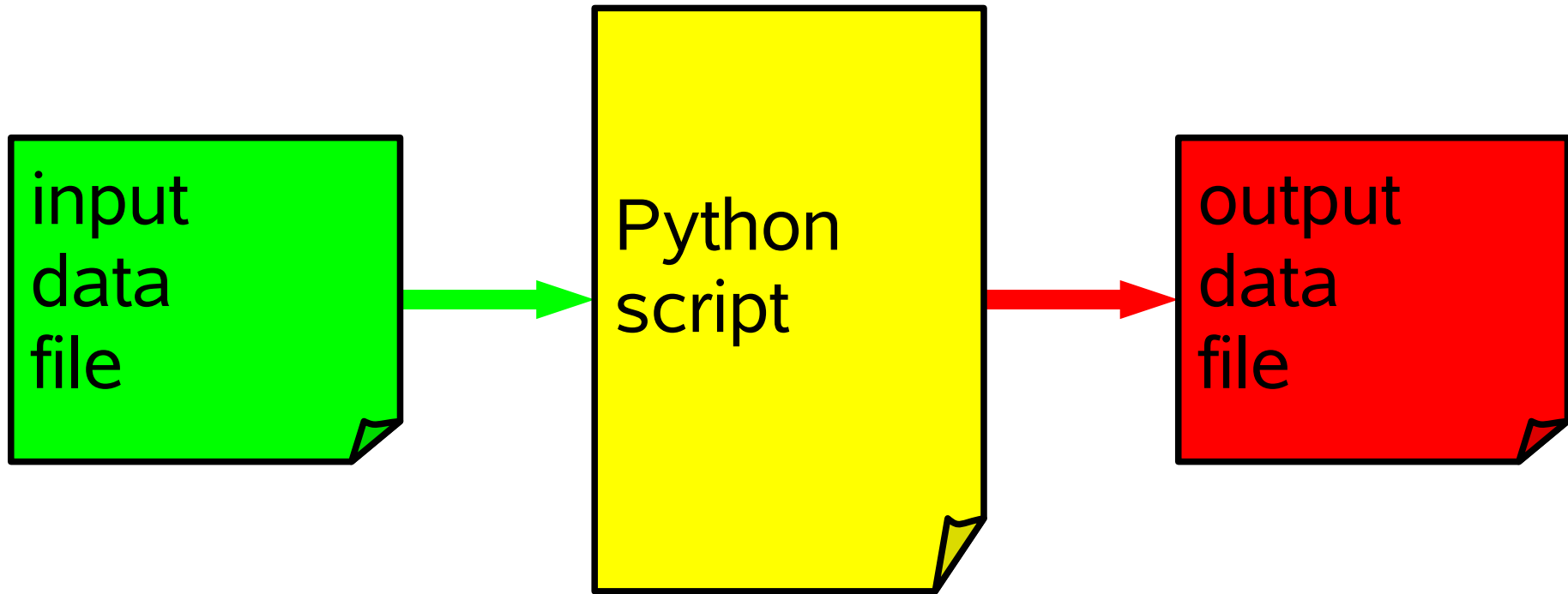
```
def min_max(numbers):  
  
    min = numbers[0]  
    max = numbers[0]  
  
    for number in numbers:  
        if number < min:  
            min = number  
        if number > max:  
            max = number  
  
    return (min, max)
```

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

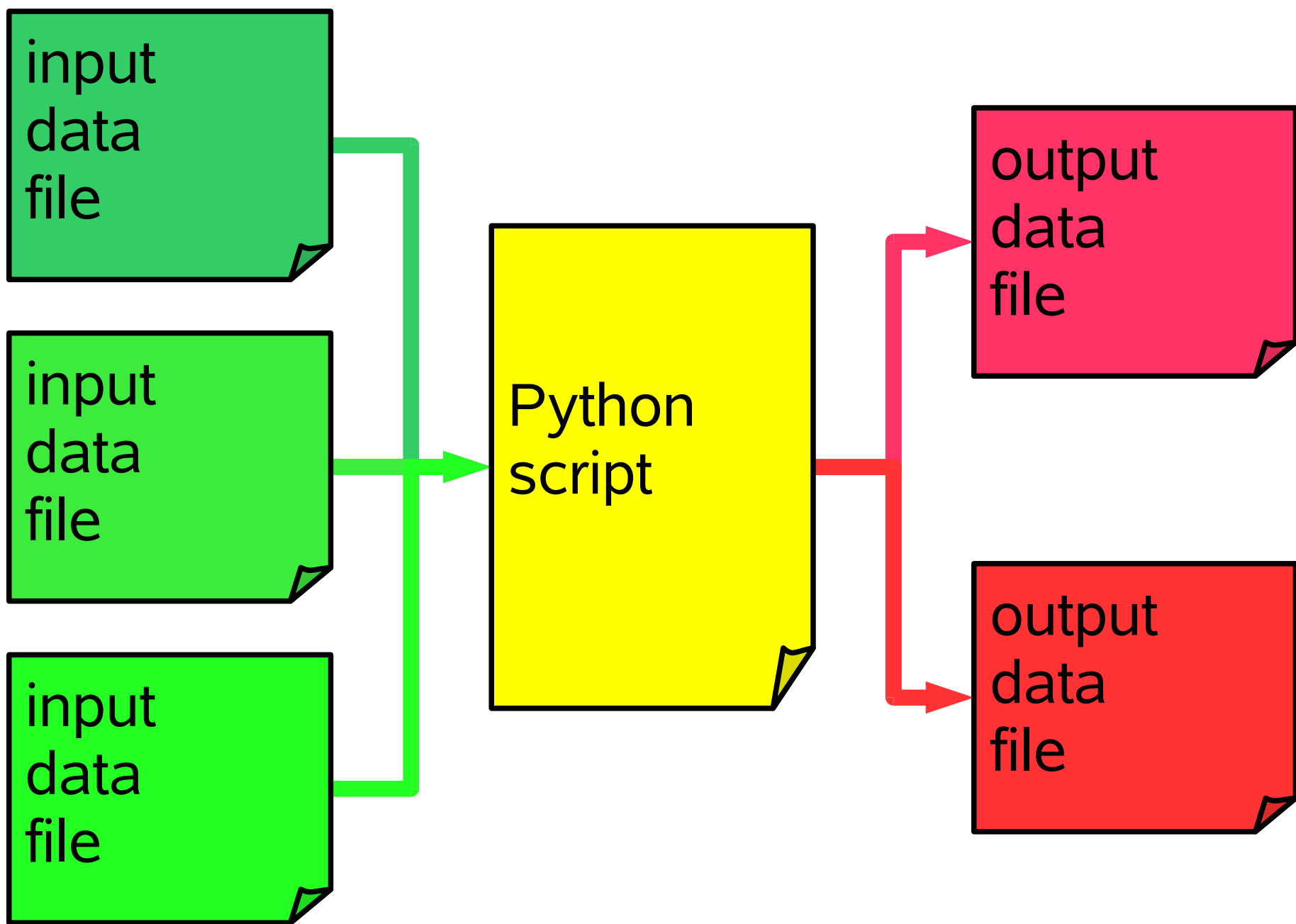
`utils.py`

Accessing the system

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



> command < input > output



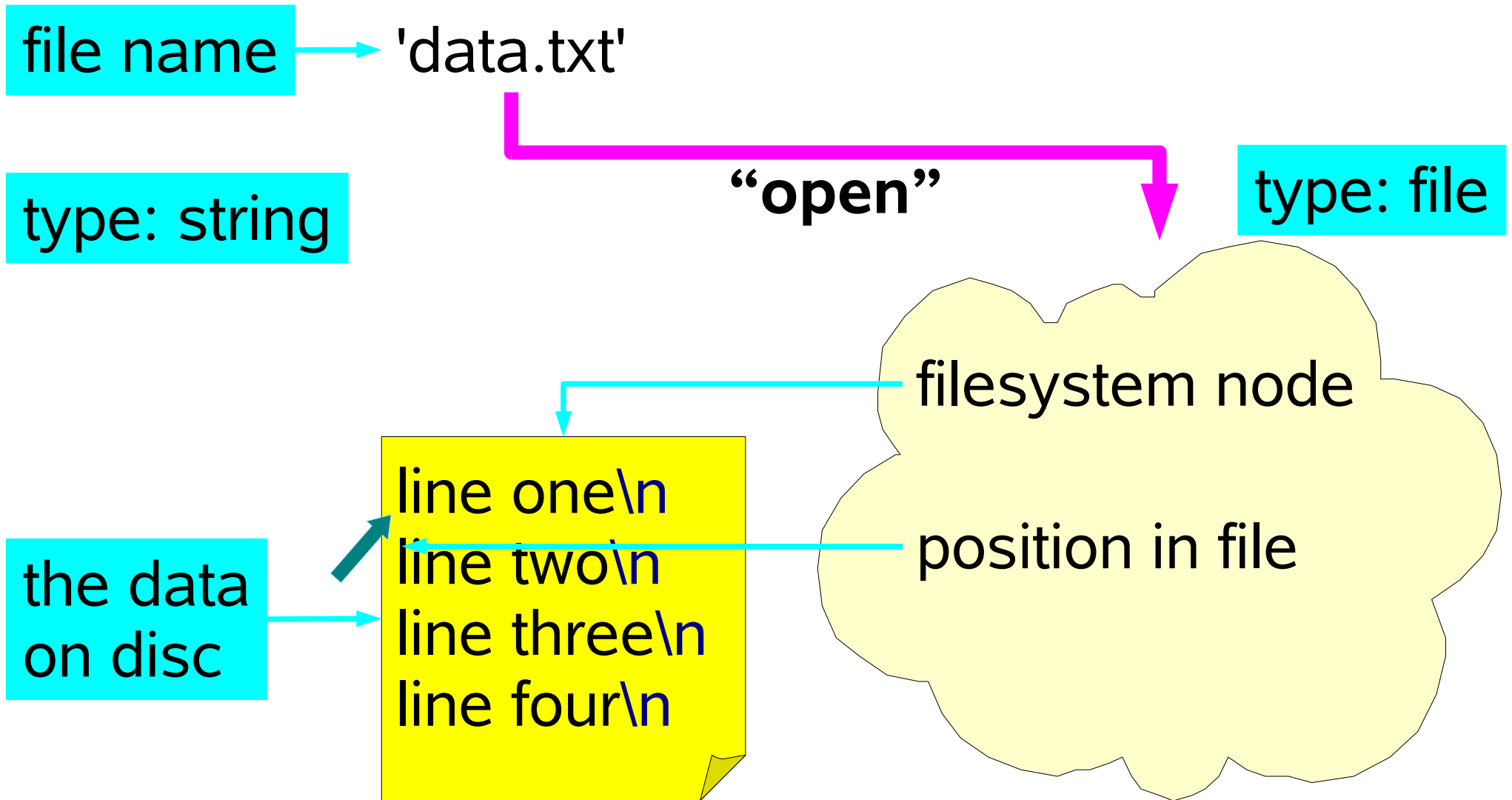
Reading a file

1. Opening a file
2. Reading from the file
3. Closing the file

Reading a file

1. Opening a file

Opening a file



file name

type: string

Python
command

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

Python
file object

type: file

refers to the file with name data.txt

initial position at start of file

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

```
>>> type(data)
```

```
<type 'file'>
```

Reading a file

1. Opening a file

2. Reading from the file

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

the Python file object
a dot
a “method”

```
>>> data.readline()
```

```
'line one\n'
```

first line of the file
complete with “\n”

```
>>> data.readline()
```

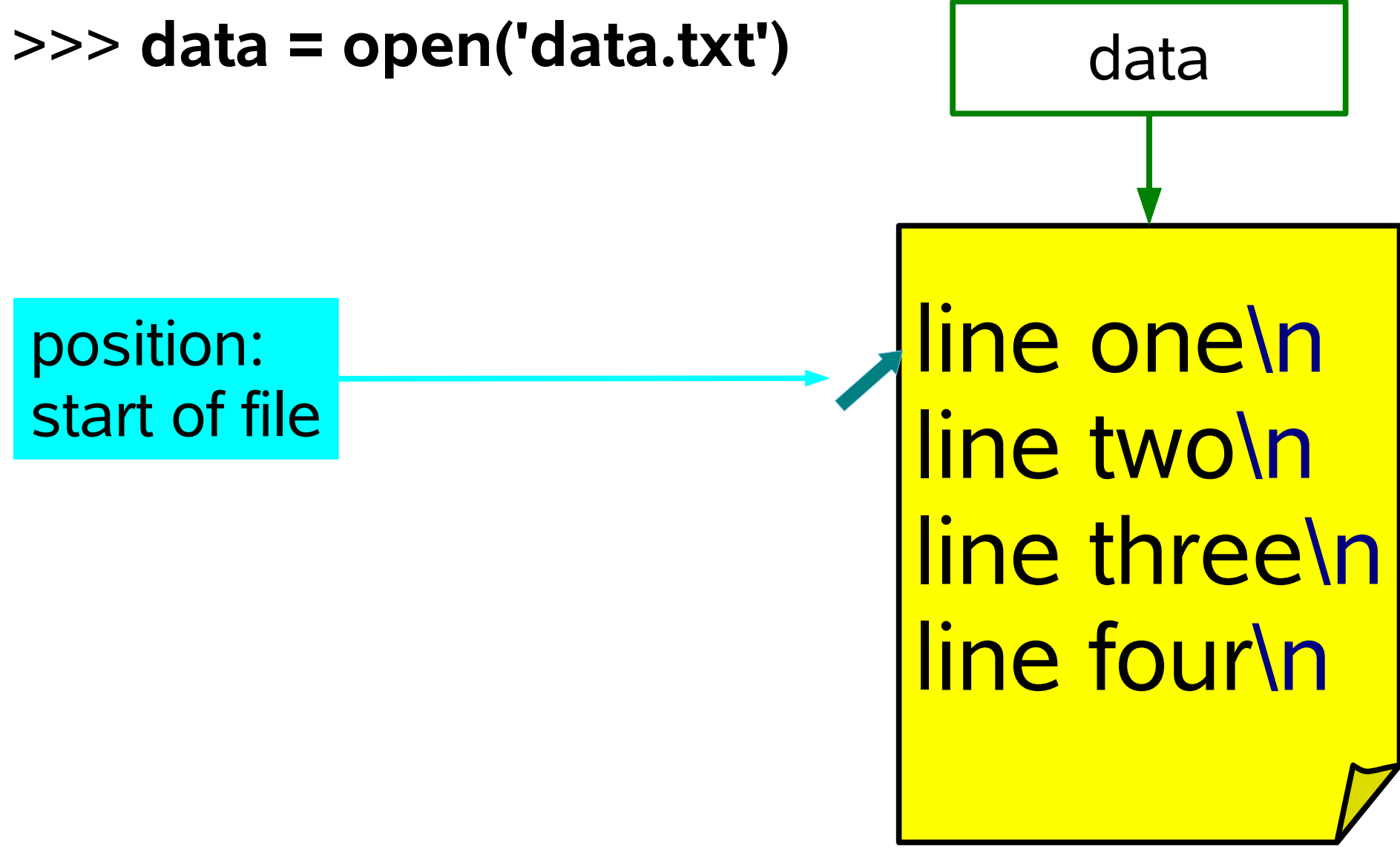
```
'line two\n'
```

same command again
second line of file

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

data

position:
start of file



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

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

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

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

data

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

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

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

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

position:
after end of read data,
at start of unread data

data

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

```
>>> data.readline()
```

```
'line one\n'
```


```
>>> data.readline()
```

```
'line two\n'
```

```
>>> data.readlines()
```

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

remaining unread
lines in the file



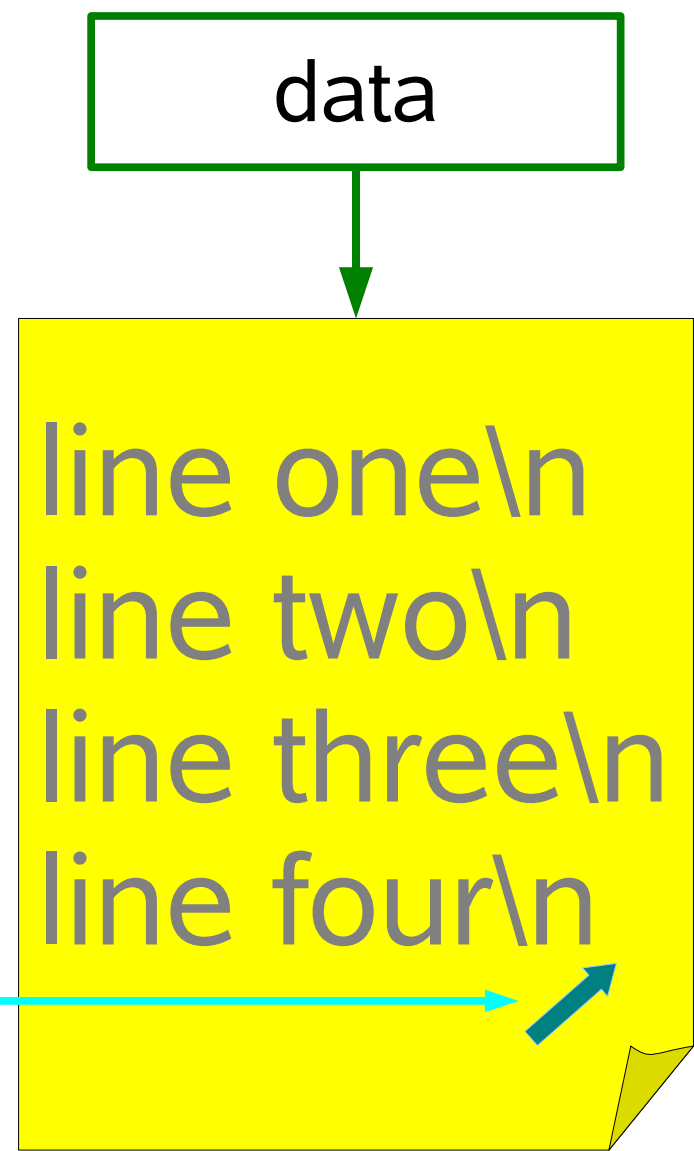
```
>>> data = open('data.txt')
```

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

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

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

position:
at end of file



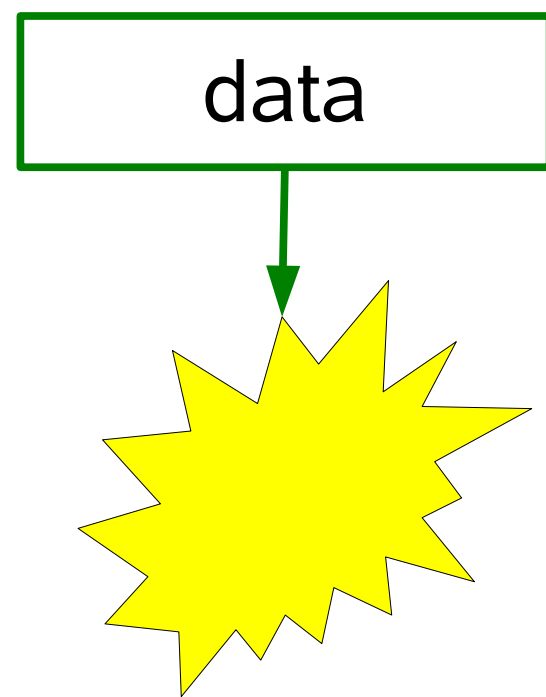
Reading a file

1. Opening a file
2. Reading from the file
3. Closing the file

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

```
>>> data.close()
```

disconnect




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

```
>>> data.close()
```

```
>>> del data
```

delete the variable if we aren't going to use it again



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

Putting it all together in a function

1. Take a file name as the function argument.
2. Read a file of “key/value” lines.
3. Create the equivalent dictionary.
4. Return the dictionary.

```
H hydrogen  
He helium  
Li lithium  
Be beryllium  
B boron
```

```
chemicals.txt
```

The plan

0. Create a function.
1. Create an empty dictionary.
2. Open the file.
3. For each line in the file:
 - 3a. Split the line into two strings (key & value).
 - 3b. Add the key and value to the dictionary.
4. Close the file.
5. Return the dictionary.



```
def file2dict(filename):
```

utils.py

0. Create a function.

```
def file2dict(filename):
```

```
    dict = {}
```

utils.py

1. Create an empty dictionary.

```
def file2dict(filename):  
    dict = {}  
    data = open(filename)
```

utils.py

2. Open the file.

```
def file2dict(filename):  
    dict = {}  
    data = open(filename)  
    for line in data:
```

utils.py

3. For each line in the file:

```
def file2dict(filename):  
    dict = {}  
    data = open(filename)  
    for line in data:
```

```
        [ key, value ] = line.split()
```

utils.py

3a. Split the line into two strings (key & value).

```
def file2dict(filename):  
    dict = {}  
    data = open(filename)  
    for line in data:  
        [ key, value ] = line.split()  
        dict[key] = value
```

utils.py

3b. Add the key and value to the dictionary.

```
def file2dict(filename):  
    dict = {}  
    data = open(filename)  
    for line in data:  
        [ key, value ] = line.split()  
        dict[key] = value  
    data.close()
```

utils.py

4. Close the file.

```
def file2dict(filename):  
    dict = {}  
    data = open(filename)  
    for line in data:  
        [ key, value ] = line.split()  
        dict[key] = value  
    data.close()  
    return dict
```

utils.py

5. Return the dictionary.

And that's it!

```
def file2dict(filename):  
    dict = {}  
    data = open(filename)  
    for line in data:  
        [ key, value ] = line.split()  
        dict[key] = value  
    data.close()  
    return dict
```

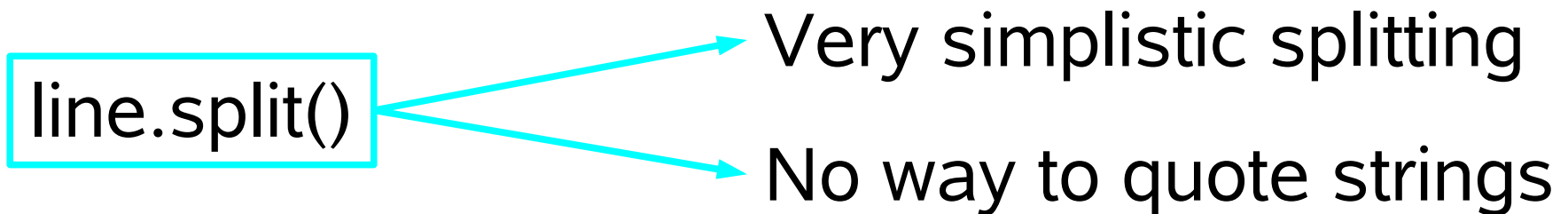
utils.py

```
#!/usr/bin/python  
import utils
```

```
chemicals = utils.file2dict('chemicals.txt')  
utils.print_dict(chemicals)
```

mkdict.py

Very primitive input



Comma separated values? **csv** module

Regular expressions? **re** module

“Python: Further Topics” course

“Python: Regular Expressions” course

Input gets *strings*

`input.readline()` → *string* per line

`input.readlines()` → list of *strings*

```
1.0  
2.0  
3.0  
-1.0  
-5.0  
7.0
```



```
[ '1.0', '2.0', '3.0',  
  '-1.0', '-5.0', '7.0' ]
```

list of
strings

≠

```
[ 1.0, 2.0, 3.0,  
  -1.0, -5.0, 7.0 ]
```

list of
floats

numbers.py

Converting from one type to another

In and out of strings

```
>>> float('0.25')  
0.25
```



```
>>> str(0.25)  
'0.25'
```

```
>>> int('123')  
123
```



```
>>> str(123)  
'123'
```

Converting from one type to another

Between numeric types

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

```
12
```

loss of
precision

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

```
12.0
```

Converting from one type to another

If you can treat it like a list ...

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

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

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

Example

Read in numbers from a file, one per line.
Print out each number plus one.

“read in numbers”



read in strings

and

convert strings
to numbers

```
#!/usr/bin/python  
  
data = open('nums.txt')  
  
for line in data:  
    number = float(line)  
    print number + 1.0  
  
data.close()
```

read_add.py 90

Exercise

Write a Python script that:

1. Reads from `numbers.py`.
2. Converts each line read into a float.
3. Appends each float to a list.
4. Applies `utils.min_max()` to that list.
5. Prints the minimum and maximum values.

This page intentionally left ~~blank~~

Silly



Answer

```
#!/usr/bin/python

import utils

data = open('numbers.py')

numbers = []

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

data.close()
del data

print utils.min_max(numbers)
```

What about output?

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

equivalent

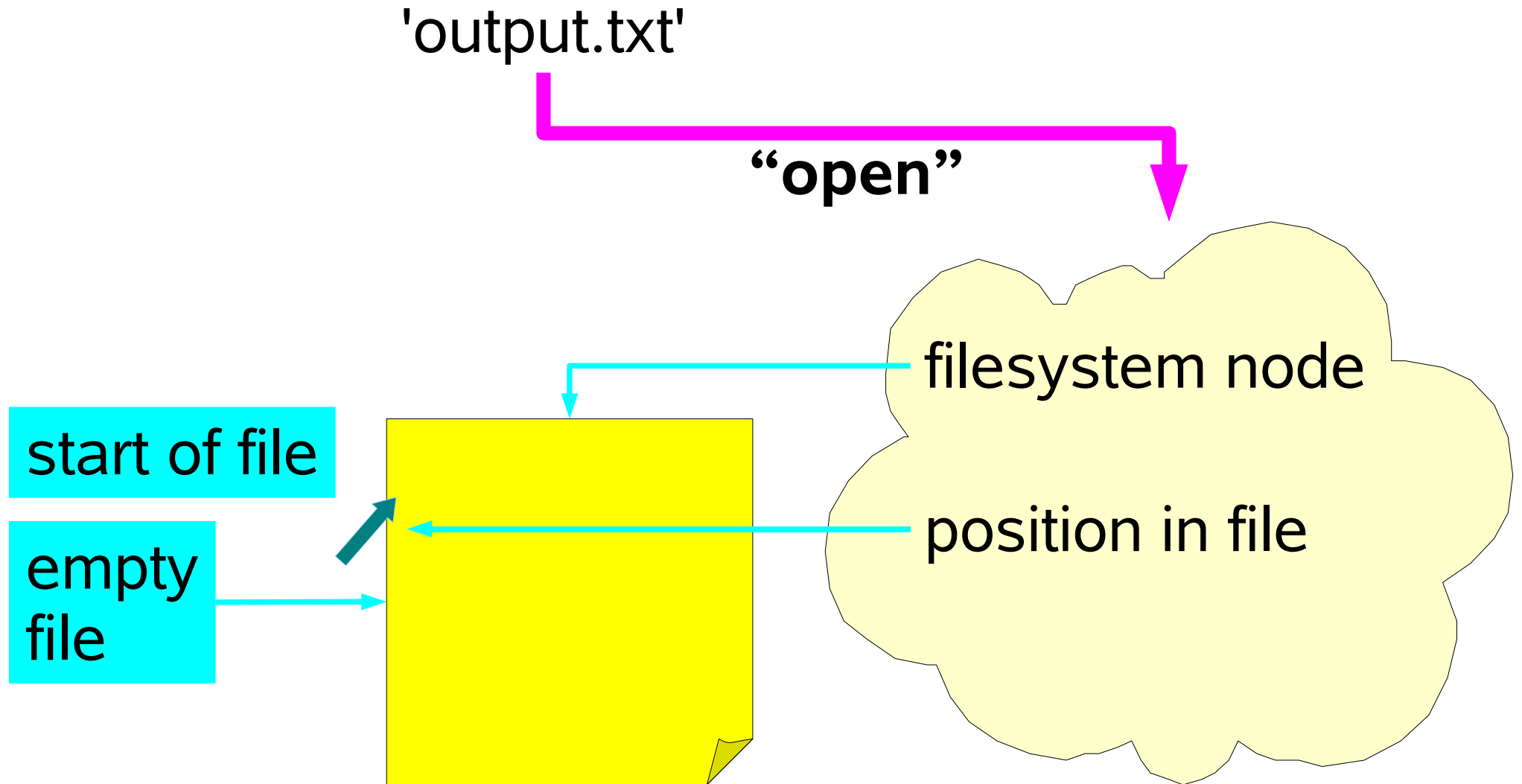
```
input = open('input.txt', 'r' )
```

open for reading

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

open for writing

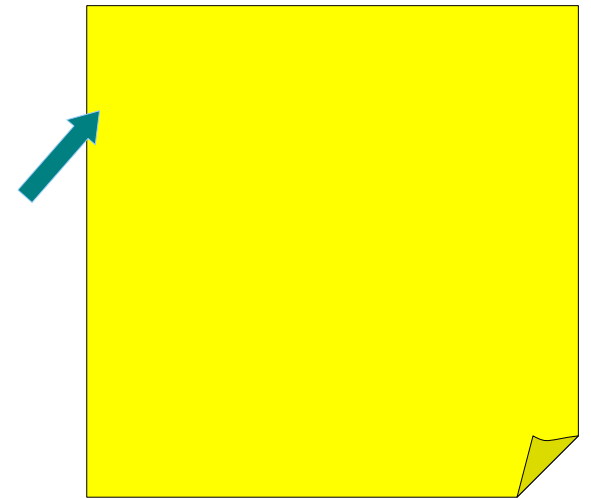
Opening a file for writing



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

file name

open for
writing



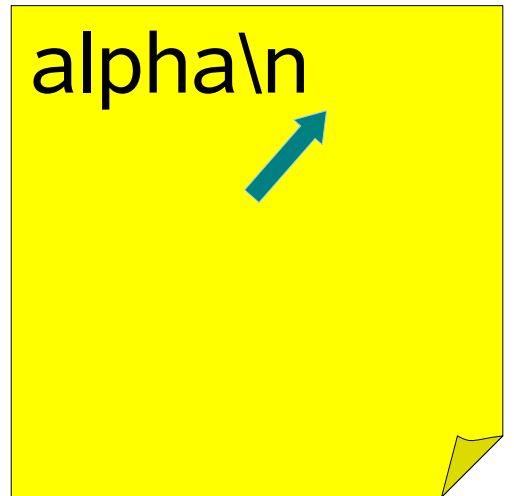
```
>>> output = open('output.txt', 'w')
```

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

method to write
a lump of data

lump of data
to be written

lump: not
necessarily
a whole line



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

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

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

lump of data
to be written



alpha\n
bet

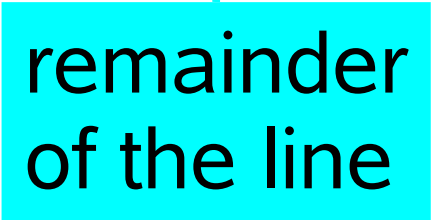
```
>>> output = open('output.txt', 'w')
```

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

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

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

remainder
of the line



alpha\n
beta\n



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

method to write
a list of lumps


the list of lumps
(typically lines)

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

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

Python is done
with this file.

Only at this point is it
guaranteed that the
data is on the disc!



Accessing the *sys* module

1. Files

2. Standard input and output

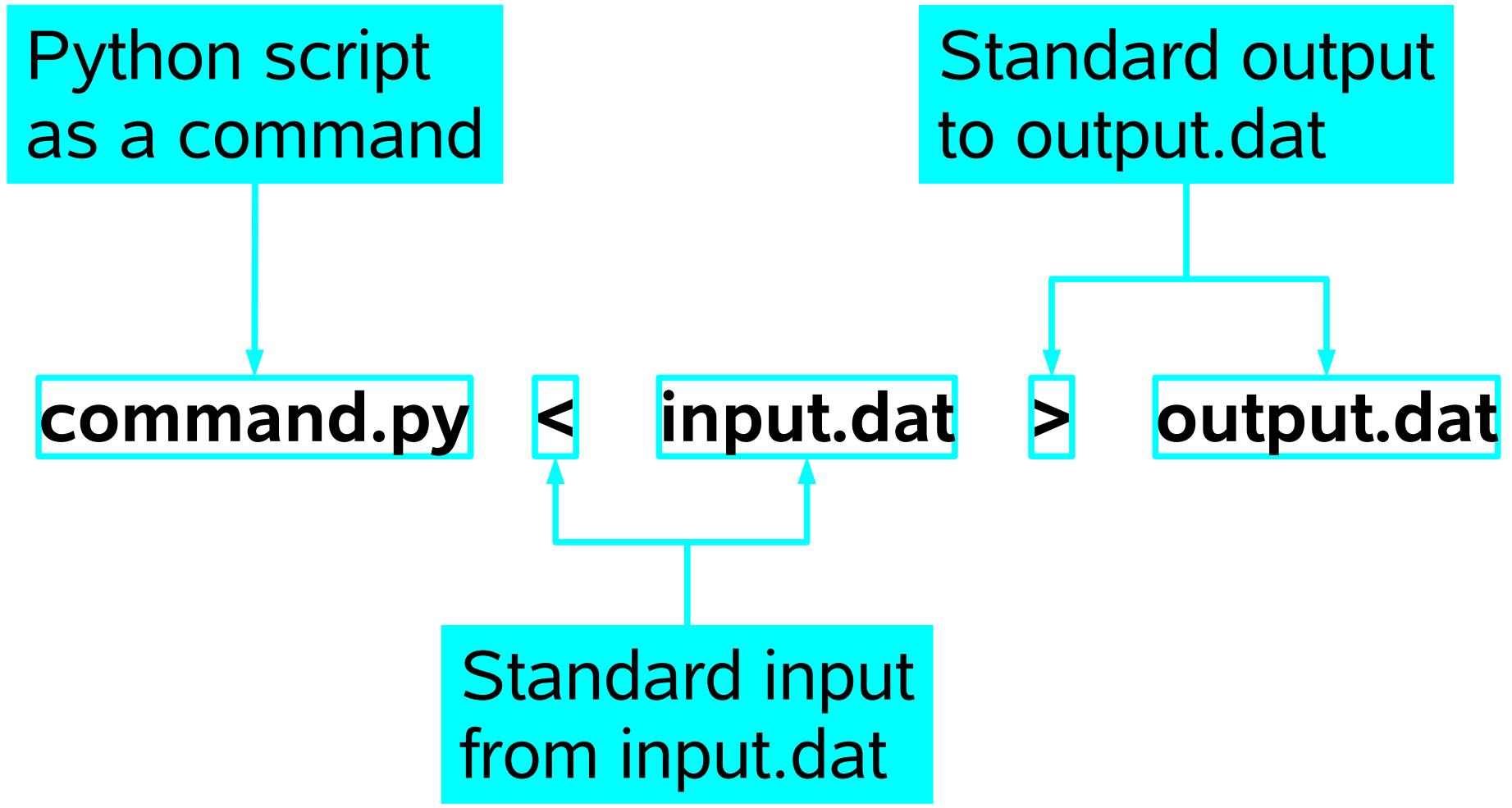
3. The command line

The “`sys`”
module

The “sys” module

```
>>> import sys
```

- Access to general “system” things
 - Standard input and output
 - The command line
 - Information about the Python environment



Inside the Python script

`import sys`

import the module

`sys.stdin`

treat exactly like an
`open(..., 'r')` file

“standard
input”

`sys.stdout`

treat exactly like an
`open(..., 'w')` file

“standard
output”

So, what does this script do?

Read lines in from standard input

Write them out again to standard output

It copies files, line by line

```
#!/usr/bin/python
```

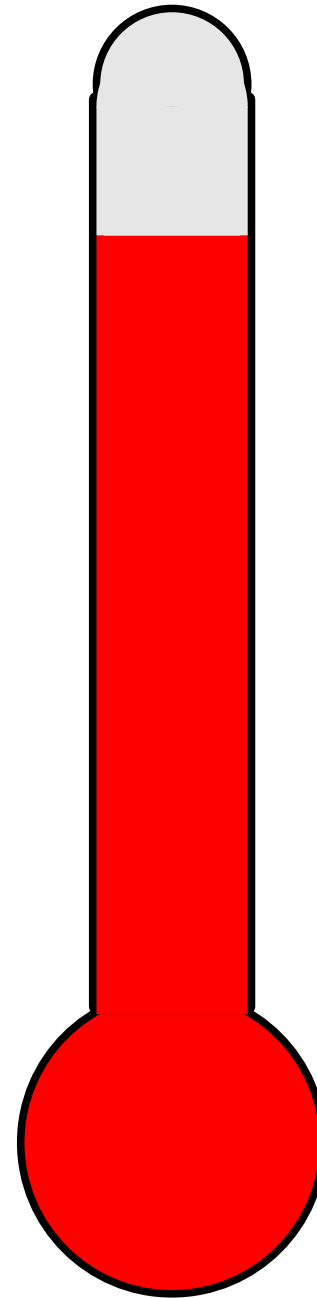
```
import sys
```

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

```
    sys.stdout.write(line)
```

```
stdin-stdout.py
```

- Access to files
- Opening
- Reading/writing
- Closing
- `sys` module
- standard input
- standard output



5 minutes



Abused substance	serving (ml)	caffeine (mg)
coffee	250	135
espresso	60	100
coca cola	355	34

Command line arguments

> python

args.py

0.25

Python script

Single command line argument



import the `sys` module

```
#!/usr/bin/python
```

```
import sys
```

```
print sys.argv
```

`sys.argv` —
the command
line arguments

argument values

```
args.py
```

```
> python args.py 0.25
```

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

sys.argv[0]

sys.argv[1]

The file name
of the script itself

The command
line argument

as a string!

```
#!/usr/bin/python
import sys
```

```
print repr(sys.argv[1])
print type(sys.argv[1])
```

`sys.argv[1]` —
the 1st command
line argument

`args.py`

```
> python args.py 0.25
```

```
'0.25'
```

```
<type 'str'>
```

1st command line argument

as a string

Recall:
float()
converts
strings
to floats

```
#!/usr/bin/python  
import sys  
print float(sys.argv[1])  
print type(float(sys.argv[1]))
```

args.py

> **python args.py 0.25**

0.25

<type 'float'>

1st command line argument

as a float

Now we can use the command line arguments

Write a script to print points

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

Two command line arguments:

r (float) power

N (integer) number of points

General approach

- 1a. Write a function that takes (r, N) as (float, integer) and does the work.
- 1b. Write a script that tests that function.
- 2a. Write a function that parses the command line for a float and an integer.
- 2b. Write a script that tests that function.
3. Combine the two functions.

- 1a. Write a function that takes (r, N) as (float, integer) and does the work.

```
#!/usr/bin/python
```

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

```
    for index in range(0, num_points):
```

```
        x = float(index)/float(num_points-1)
```

```
        y = x**pow
```

```
        print x, y
```

curve.py

1b. Write a script that tests that function.

```
#!/usr/bin/python  
  
...function...  
print x, y
```

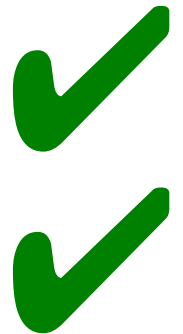
```
power_curve(0.5, 5)
```

```
curve.py
```

```
> python curve.py  
0.0 0.0  
0.25 0.5  
0.5 0.707106781187  
0.75 0.866025403784  
1.0 1.0
```

Five points

Each correct



2a. Write a function that parses the command line for a float and an integer.

```
#!/usr/bin/python  
import sys
```

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

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

curve.py

2b. Write a script that tests that function.

```
#!/usr/bin/python

...functions...

# power_curve(0.5, 5)
(r, N) = parse_args()
print 'Power: ', r, type(r)
print 'Points: ', N, type(N)
```

curve.py

```
> python curve.py 0.5 5
Power: 0.5 <type 'float'>
Points: 5 <type 'int'>
```

Right values
Right types



3. Combine the two functions.

```
#!/usr/bin/python
```

```
...functions...
```

```
...tests (commented out)...
```

```
(pow, num) = parse_args()  
power_curve(pow, num)
```

```
curve.py
```

“Doc” strings

A string immediately before the function body

A long string so typically done with
""" ... """

```
def file2dict(filename):
```

```
    """file2dict takes a file name as its argument. This file must contain pairs of words. It returns the equivalent Python dictionary."""
```

```
    dict = {}  
    data = open(filename)  
    data_lines = data.readlines()  
    for line in data_lines:  
        [ key, value ] = line.split()  
        dict[key] = value  
    data.close()  
    return dict
```

utils.py¹²⁴

```
>>> import utils
```

Name of module

Name of function

__doc__

```
>>> print utils.file2dict.__doc__
```

file2dict takes a file name as its argument. This file must contain pairs of words. It returns the equivalent Python dictionary.

The string

Functions can carry their own documentation

```
def file2dict(filename):
```

```
    """file2dict takes a file name  
    as its argument. This file  
    must contain pairs of words.  
    It returns the equivalent  
    Python dictionary."""
```

```
    dict = {}  
    data = open(filename)
```

```
    ...
```

```
utils.py
```

We can document a function.

What about the module?

The module's doc string.

```
"""A collection of my personal utility functions."""
```

```
def file2dict(filename):
```

```
    """file2dict takes a file name as its argument. This file must contain pairs of words. It returns the equivalent Python dictionary."""
```

```
    dict = {}
```

```
    data = open(filename)
```

```
    ...
```

```
utils.py
```

```
>>> import utils
```

Name of module

```
>>> print utils.__doc__
```

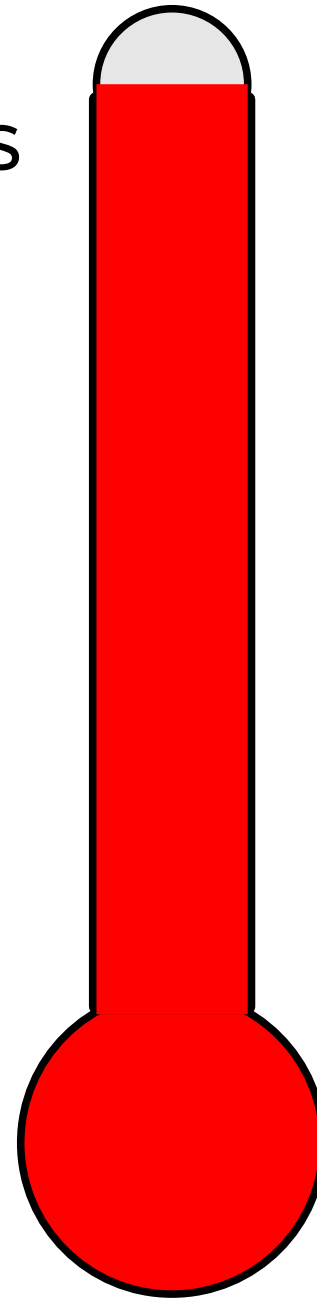
__doc__

A collection of my personal utility functions.

The string

Modules can carry their own documentation too

- Command line arguments
- Type conversions
- Script as set of functions
- Doc strings

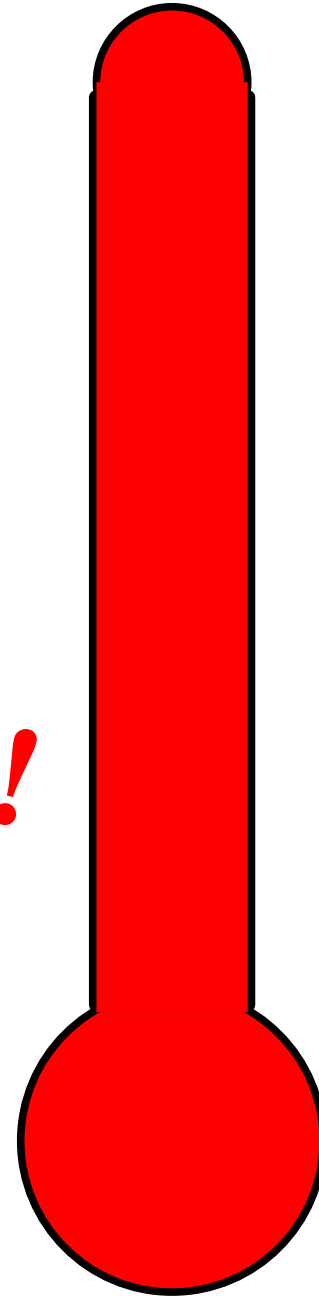


And that's it!

Congratulations!

Follow on course:

“Python: Further Topics”



References

Dive Into Python

Mark Pilgrim

Apress

ISBN: 1-59059-356-1

<http://diveintopython.org/>

Best book on Python I've found. (It was written for Python 2.3, though. Luckily, Python 2.4, 2.5 and 2.6 are very similar to Python 2.3.)

Python Programming: An Introduction to Computer Science

John M. Zelle

Franklin, Beedle & Associates, Inc.

ISBN: 1-887902-99-6

<http://www.fbeedle.com/99-6.html>

<http://mcsp.wartburg.edu/zelle/python/>

Superb introduction to computer programming, using Python as a first language (which also makes it a good introduction to Python).

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

Final Exercise / Homework

You have been given a collection of files that contain floating point numbers, one number on each line of each file. Write a Python script that accepts the name of one or more files on the command line, and processes each file as follows:

- Prints the name of the file and its minimum and maximum values.

After processing all the files, your script should then print out the smallest minimum value and the name of the file(s) containing it, as well as the the largest maximum value and the name of the file(s) containing that maximum value.