HEP Computing
Part II
Python
Marcella Bona

Lectures 3-4

© Bob Dowling
Launching Python interactively

Unix prompt

Unix command

Bold face means you type it.

Python prompt

Introductory blurb

Python 2.6 ...

[ GCC 4.3.2 ...

Type "help", ...

>>> $ python
Using Python interactively

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

>>> 'Hello, world!
'Hello, world!'
```
Using Python interactively

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

Python prints a 3

>>> 5  # Give Python a literal 5
5

Python evaluates and displays a 5

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

Python evaluates and displays a 5
Quitting Python interactively

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

Read / edit the script  Run the script
Launching Python scripts

Edit a file with one single line:
$ emacs -nw hello.py
print('Hello, world!')

$ python hello.py
Hello, world!

$
Launching Python scripts

```
print(3)
5
```

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

No “5”!
Types of values

Numbers

Whole numbers

Decimal numbers

Text

“Boolean”

True

False
Integers

Addition behaves as you might expect it to.

Spaces around the “+” are ignored.

Subtraction also behaves as you might expect it to.
Integers

>>> 3 * 5
15

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

>>> 5 / 3
1

Division rounds down.

>>> -5 / 3
-2

Strictly down.

>>> 4**2
16

Raising to powers uses “4**2” instead of “4^2”.
Integers

>>> 65536 * 65536
4294967296L

Long integer

>>> 4294967296 * 4294967296
18446744073709551616L

>>> 18446744073709551616 * 18446744073709551616
340282366920938463463374607431768211456L

No limit to size of Python's integers!
Floating point numbers

1
1 \(\frac{1}{4}\)
1 \(\frac{1}{2}\)

But

1 \(\frac{1}{3}\)
Floating point numbers are... ...printed in decimal ...stored in binary

17 significant figures
\[
4294967296.0 \times 4294967296 = 18,446,744,073,709,551,616
\]

Difference

384
Floating point limits

$1.2345678901234567 \times 10^N$

17 significant figures

$-325 < N < 308$

Positive values:
$4.94065645841e-324 < x < 8.98846567431e+307$
Summary on Numbers

Floating Point numbers

1.25 → 1.25
1.25×10^5 → 1.25e5

Limited accuracy
(but typically good enough)

Limited range of sizes

Mathematical operations

a+b   a-b   a×b   a÷b   a^b
a+b   a-b   a*b   a/b   a**b
Strings

quotes

>>> 'Hello, world!'

The value of the text object

Quotes: Hey, this is text!

How Python represents the text object.
Why do we need quotes?

3 ➔ It’s a number

print ➔ Is it a command?

Is it a string?

'print' ➔ It’s a string

double quotes work the same

print ➔ It’s a command
Mixed quotes

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

>>> print "He said 'Hello' to her."
He said 'Hello' to her.
Joining strings together

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

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

Repeated text

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

>>> 3 * 'Bang!'
'Bang! Bang! Bang!'
```
Line breaks

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

Special characters

- `\a` → ♬ bell
- `\n` → ¶ new line
- `\t` → →| tab
- `\'` → ' single quote
- `\"` → " double quote
- `\\` → \ backslash

m.bona@qmul.ac.uk
Comparisons

A comparison operation

>>> 5 > 4
True

A comparison result

Only two values possible

n.b. double equals

>>> 5.0 < 4.0
False

>>> 5 == 4
False
Comparing strings

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

>>> 'bad' < 'bud'
True

>>> 'Cat' < 'cat'
True

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

Alphabetic order…

ABCDEFGHIJKLMNOPQRSTUVWXYZ…
abcdefghijklmnopqrstuvwxyz
Combining booleans

```plaintext

>>> 1 < 2 and
False

>>> 1 < 2 or
True

>>> 5 > 6
False

Not both True

Either True
```
Negating booleans

>>> 1 > 2
False

<table>
<thead>
<tr>
<th>True</th>
<th>False</th>
</tr>
</thead>
<tbody>
<tr>
<td>False</td>
<td>True</td>
</tr>
</tbody>
</table>

>>> not 1 > 2
True

>>> 1 == 2
False

>>> 1 != 2
True

>>> not 1 == 2
True

m.bona@qmul.ac.uk
Precedence

**  %  /  *  -  +

==  !=  >=  >  <=  <

not  and  or

Arithmetic

Comparison

Logical

Last
How Python stores values

Lump of computer memory

42

Identification of the value’s type

identification of the specific value

int

42
How Python stores values

- **int**: 42
- **float**: $4.2 \times 10^1$
- **str**: 'Forty two'
- **bool**: True

Example values and their corresponding data types.
Variables

Attaching a name to a value.

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

An expression

The expression’s value

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

Attaching the name `answer` to the value 42.

```python
>>> answer
```

The name given

42

The attached value returned
if ... then ... else ...

Run a test

Do something if it succeeds.

Do something else if it fails.

Colon...Indentation

if test:
    something
else:
    something else
```python
if middle**2 > 2.0:
    upper = middle

else:
    lower = middle
```
Example script:

```python
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)
```

middle1.py
Example script: before

```python
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...

```python
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: then...

```python
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...

```python
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: after

```python
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

```python
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

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

Run a test

Do something if it succeeds.

Finish if it fails.

Go back to the test.
while

Keep going while...

...then stop.

doubler1.py

```
number = 1
limit = 1000

while number < limit:
    print(number)
    number = number * 2

print('Finished!')
```

while condition:
    action
    action
afterwards
Example script: before

```python
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...

```python
doubler1.py

number = 1
limit = 1000

while number < limit:
    print(number)
    number = number * 2

print('Finished!')
```

keyword: “while”
condition
colon
doubler1.py
Example script: loop body

```python
number = 1
limit = 1000

while number < limit:
    print(number)
    number = number * 2

print('Finished!')
```

`doubler1.py`
Example script: after

```python
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

```python
number = 1
limit = 1000

while number < limit:
    print(number)
    number = number * 2

print('Finished!')
```

```bash
> python doubler1.py
1
2
4
8
16
32
64
128
256
512
Finished!
```
while... if ... then ... else ...

lower = 1.0
upper = 2.0

while upper - lower > 1.0e-15 :
    middle = (upper+lower)/2.0
    if middle**2 > 2.0:
        print('Moving upper')
        upper = middle
    else:
        print('Moving lower')
        lower = middle

print(middle)
Running the script

```python
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 upper
Moving lower
...
Moving upper
Moving upper
Moving lower
Moving lower
1.41421356237
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)
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)
Arbitrary nesting

Not just two levels deep
As deep as you want
Any combination

number = 20

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

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

Lines starting with “#” are ignored
Partial lines too.

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

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

lower = 1.0 # Initial bounds.
upper = 2.0

while upper - lower < 1.0e-15 :
    ...

Comments

Reading someone else’s code. – Writing code for someone else.

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


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

[0.0, 1.5707963267948966, 3.1415926535897931]
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”
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

Here’s a list

[1, 2, 3]

Yes, that’s a list

Attaching a name to a variable.

Using the name

Empty list

[1, 2, 3]

[1, 2, 3]

[ ]

[ ]
Anatomy of a list

Square brackets at end

Individual element

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

Elements separated by commas
Order of elements

No “reordering”

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

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

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

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

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

“+” used to join lists.

“3” appears twice
How long is the list?

```python
>>> len([10, 20, 30])
3
```
How long is a string?

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

Recall:
Quotes say “this is a string”.
They are not part of the string.
How long is a *number*?

```python
>>> 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”. 
Picking elements from a list

```python
>>> letters = ['a', 'b', 'c', 'd']
```
The first element in a list

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

Count from zero

"Index"

```python
letters = ['a', 'b', 'c', 'd']
letters[0]  # Count from zero
```

```
<table>
<thead>
<tr>
<th>str</th>
<th>a</th>
</tr>
</thead>
<tbody>
<tr>
<td>str</td>
<td>b</td>
</tr>
<tr>
<td>str</td>
<td>c</td>
</tr>
<tr>
<td>str</td>
<td>d</td>
</tr>
</tbody>
</table>
```
“Element number 2”

numbers[$N$]  Indexing into a list

>>> letters[2]

'c'  The third element
Going off the end

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

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

<table>
<thead>
<tr>
<th>letters</th>
<th>list</th>
</tr>
</thead>
<tbody>
<tr>
<td>str</td>
<td>a</td>
</tr>
<tr>
<td>str</td>
<td>b</td>
</tr>
<tr>
<td>str</td>
<td>c</td>
</tr>
<tr>
<td>str</td>
<td>d</td>
</tr>
</tbody>
</table>

Maximum index is 3!
“Element number -1 !”

```python
>>> letters[-1]
'd'
The final element
```
Negative indices

```python
>>> letters[-3]
'b'
```
Going off the end

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

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

-4  -3  -2  -1  0  1  2  3

```
letters[-4]  str a  letters[0]
letters[-3]  str b  letters[1]
letters[-2]  str c  letters[2]
letters[-1]  str d  letters[3]
```
Assigning list elements

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

The name attached to the list as a whole

```python
>>> letters[2] = 'X'
```

The name attached to one element of the list

```python
>>> letters
['a', 'b', 'X', 'd']
```

Assign a new value

The new value
Doing something with a list

1. Given a list
2. Start with the first element of the list
3. Do something with that element
4. Are there any elements left?
5. If yes, move on to the next element; if no, finish.
The “for loop”

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

- **Keyword**: for, in
- **List**: ['a', 'b', 'c']
- **Indentation**
- **Using the loop variable**
- **Repeated block**
The “for loop”

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

```
Have a letter: a
Have a letter: b
Have a letter: c
Finished!
```

$ python for1.py
“Slices” of a list

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

>>> abc[1:5]  # Slice index
['b', 'c', 'd', 'e']  # A new list

>>> abc[1:5]  # Slice index
['b', 'c', 'd', 'e']  # A new list

>>> abc[1:5]  # Slice index
['b', 'c', 'd', 'e']  # A new list

>>> abc[1]  # Element 1
'a'

>>> abc[4]  # Element 4
'f'

>>> abc[5]  # Element 5
'g'

>>> abc[1]  # Element 1
'a'

>>> abc[4]  # Element 4
'f'

>>> abc[5]  # Element 5
'g'
```

A new list

“Slice”

“from” index

“to” index

Element 5

not in slice
Slice feature

```
abc[1:3] + abc[3:5]
['b', 'c'] + ['d', 'e']
['b', 'c', 'd', 'e']
```

abc[1:5]
Open-ended slices

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

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

Open ended at the end

abc[3]

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

Open ended at the start

abc[4]

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

Open ended at both ends
```
Modifying lists — recap

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

>>> abc[2] = 'X'

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

New value

Changed
Modifying vs. replacing?

```python
>>> xyz = ['x', 'y']
>>> xyz[0] = 'A'
>>> xyz = ['A', 'B']
>>> xyz[1] = 'B'
>>> xyz
['A', 'B']
```

Modifying the list

Replacing the list
What's the difference? — 1

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

**Old value unused and cleaned up.**

New value assigned
What's the difference? — 2

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

Old value unused and cleaned up.

New value assigned.
What's the difference?

Modification: same list, different contents

Replacement: different list

Does it matter?
Two names for the same list

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

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

>>> xyz
['x', 'y']
One last trick with slices

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

Length 6
New length
Appending to a list

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

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

>>> abc.append('z')

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

Add one element to the end of the list.
List “methods”

A list

A dot

A built-in function

Round brackets

Argument(s) to the function

Built-in functions: “methods”
Methods

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

Privileged access to object

“Object-oriented programming”
The append() method

```python
>>> 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!

```python
>>> 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

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

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

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

All in one go rather unnecessary
Avoiding extend()

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

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

>>> abc
['x', 'y', 'z', 'A', 'B', 'C']
```
Changing the list “in place”

```python
>>> abc.append('w')
No value returned

>>> abc
['x', 'y', 'z', 'w']
List itself is changed

>>> abc.extend(['A', 'B'])
No value returned

>>> abc
['x', 'y', 'z', 'w', 'A', 'B']
List itself is changed
```
Another list method: sort()

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

```python
>>> abc.sort()
```

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

Any type of sortable element

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

>>> abc = [3.142, 1.0, 2.718]
>>> abc.sort()
>>> abc
[1.0, 2.718, 3.142]
Another list method: `insert()`

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

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

Insert just before element number 2

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

```
"old 2"
```
Summary on lists

List methods:  

- `list.append(item)`  
- `list.extend([item_1, item_2, item_3])`  
- `list.sort()`  
- `list.insert(index, item)`

Change the list itself  
Don't return any result
Creating new lists

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

Simple copying

```python
>>> copy
[0,1,2,3,4,5,6,7,8,9]
```
Creating new lists

```python
>>> 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]
```

Boring!

Changing the value
Lists of numbers

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

range(0, 10)
Creating new lists

```python
>>> numbers = range(0, 10)
>>> squares = []
>>> for number in numbers:
...     squares.append(number**2)
...```

```python
>>> squares
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```
Lists of words

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

Python command

file name string

Python file object

refers to the file with name 'data.txt'

initial position at start of data
Reading from a file

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

the Python file object
a dot
a “method”

>>> data.readline()

'line one
'

first line of the file
complete with “\n”

>>> data.readline()

'line two
'

same command again
second line of file
```
```python
>>> data = open('data.txt')
```

```
data
line one
line two
line three
line four
```

**position:** start of file
```python
>>> data = open('data.txt')
>>> data.readline()
'line one
'
```
>>> data = open('data.txt')

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

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

after end of second line at start of third line
```python
>>> 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.close()
```
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()
p
print(count)
Write a script `counting.py` from scratch to do this:

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

15 minutes
Converting the type of input

Problem:

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

```
['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

```python
>>> float('1.0
')
1.0

>>> str(1.0)
'1.0'

>>> float(1)
1.0

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

- **String → Float**: `float('1.0
')` results in `1.0`.
- **Float → String**: `str(1.0)` results in `'1.0'`.
- **Int → Float**: `float(1)` results in `1.0`.
- **Float → Int**: `int(-1.5)` results in `-1`.

- *No newline*
- *No newline*

- **Rounding to zero**
Type conversions to lists

```python
>>> list('hello')  # String → List
['h', 'e', 'l', 'l', 'o']
```

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

>>> list(data)  # File → List
['line one
', 'line two
', 'line three
', 'line four
']
```
Example script

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

output = open('output.txt')  # Default

output = open('output.txt','r')  # Equivalent

output = open('output.txt','w')  # Open for writing

output = open('output.txt','w')  # Open for writing
```python
>>> 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**
- **alpha\n**
>>> output = open('output.txt','w')
>>> output.write('alpha\n')
>>> output.write('bet')
```python
>>> output = open('output.txt','w')
>>> output.write('alpha
')
>>> output.write('bet')
>>> output.write('a
')
```

Remainder of the line

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

Method to write a list of lumps

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

```python
>>> output.close()
```
Example

```python
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”

```python
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.
')

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

**Setup**

**Ugly!**

**Shutdown**

`filter1.py`
Exercise

Change `counting.py` to do this:

Read `file.txt` and write `file.out`. For each line write to the output:
- line number
- number of words on the line
- number of characters in the line separated by TABs.
At the end output a summary line
- number of lines
- total number of words
- total number of characters separated by TABs too.

⏰ 15 minutes
Defining our function

```python
def my_function(limit):
    
    # Indentation
```

- **def**
- **function name**
- **input**
- **colon**
- **indentation**
Defining our function

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

Names are used only in the function

Function definition
Defining our function

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

```python
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?

- **Clarity**
  - Clearly separated components are easier to read.

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

- **Reliability**
  - Isolation of variables leads to fewer accidental clashes of variable names.
Example 1

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

\[ (a_i) \rightarrow \sum |a_i|^2 \]

```python
def norm2(values):
    sum = 0.0
    for value in values:
        sum = sum + value**2
    return sum
```
Example 1

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

```
50.0
```

```
$ python norm2.py
```

```
50.0
```

```
[3.0, 4.0, 5.0]
```

```
169.0
```

```
[12.0, 5.0]
```
Example 2

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

\[(a_i) \rightarrow \min(a_i)\]

def minimum(a_list):
    a_min = a_list[0]
    for a in a_list:
        if a < a_min:
            a_min = a
    return a_min
Example 2

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

```
1.0
```

```bash
$ python minimum.py
```

```
3.0
```

```
[4.0, 3.0, 5.0]
```

```
5
```

```
[12, 5]
```
Example 3

Write a function to “dot product” two vectors.

\[(a_i,b_j) \rightarrow \sum a_k b_k\]

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

    return sum
Example 3

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

```
11.0
```

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

```
115
```
Example 3 — version 2

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

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

    return sum
Example 4

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

e.g. \([1, -2, 0, 5, -5, 3, 3, 6]\) \[\rightarrow 1, 5, 3, 3, 6]\)

```
def positive(a_list):
    answer = []
    for a in a_list:
        if a > 0:
            answer.append(a)
    return answer
```
How to return more than one value?

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

```python
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)
```
Receiving two values

\[ \text{values} = [1, 2, 3, 4, 5, 6, 7, 8, 9] \]

\[
(\text{minval}, \text{maxval}) = \text{min\_max(values)}
\]

print \text{minval}
print \text{maxval}
Pairs, triplets, ...

singles
doubles
triples
quadruples
quintuples
...

“tuples”
### Lists vs. Tuples

<table>
<thead>
<tr>
<th>Lists</th>
<th>Tuples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Concept of “next entry”</td>
<td>All items at once</td>
</tr>
<tr>
<td>Same types</td>
<td>Different types</td>
</tr>
<tr>
<td>Mutable</td>
<td>Immutable</td>
</tr>
</tbody>
</table>
Tuple examples

Pair of measurements of a tree

\((\text{height}, \text{width})\)  \((7.2, 0.5)\)

\((\text{width}, \text{height})\)  \((0.5, 7.2)\)

Details about a person

\((\text{name, age, height})\)  \((\text{'Bob', 45, 1.91})\)

\((\text{age, height, name})\)  \((\text{45, 1.91, 'Bob'})\)
Exercise

Copy the `min_max()` function. Extend it to return a triplet: 
`(minimum, mean, maximum)`
Tuples and string substitution

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

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

%s Substitute a string.
Simple integer substitution

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

%d Substitute an integer.
Tuple substitution

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

'My name is Bob and I am 46 years old.'
Lists of tuples

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

for (person, age) in data:
    print '%s %d' % (person, age)
Problem: ugly output

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Bob</td>
<td>46</td>
</tr>
<tr>
<td>Joe</td>
<td>9</td>
</tr>
<tr>
<td>Methuselah</td>
<td>969</td>
</tr>
</tbody>
</table>

Columns should align

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Bob</td>
<td>46</td>
</tr>
<tr>
<td>Joe</td>
<td>9</td>
</tr>
<tr>
<td>Methuselah</td>
<td>969</td>
</tr>
</tbody>
</table>

Columns of numbers should be right aligned
Solution: formatting

'\%s'  % 'Bob'  \rightarrow  'Bob'

'\%5s'  % 'Bob'  \rightarrow  'Bob'      \textbf{Five characters}

'\%-5s'  % 'Bob'  \rightarrow  'Bob'      \textbf{Left aligned}

'\%-5s'  % 'Bob'  \rightarrow  'Bob'      \textbf{Right aligned}

'\%5s'  % 'Charles'  \rightarrow  'Charles'
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)
Floats

'\%f' % 3.141592653589 \rightarrow '3.141593'

'\%.4f' % 3.141592653589 \rightarrow '3.1416'

'\%.4f' % 3.1 \rightarrow '3.1000'
Exercise

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

<table>
<thead>
<tr>
<th>Name</th>
<th>Age</th>
<th>Height</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alfred</td>
<td>46</td>
<td>1.90</td>
</tr>
<tr>
<td>Bess</td>
<td>24</td>
<td>1.75</td>
</tr>
<tr>
<td>Craig</td>
<td>9</td>
<td>1.50</td>
</tr>
<tr>
<td>Diana</td>
<td>100</td>
<td>1.66</td>
</tr>
</tbody>
</table>

↑       ↑     ↑
1       9     15

5 minutes
Reusing our functions

Want to use the same function in many scripts

def min_max(a_list):
    ...
    return (a_min,a_max)

vals = [1, 2, 3, 4, 5]
(x, y) = min_max(vals)
print(x, y)

five.py
vals = [1, 2, 3, 4, 5]
(x, y) = min_max(vals)
print(x, y)

def min_max(a_list):
    ...
    return (a_min, a_max)

Move the definition of the function to a separate file.
import utils
vals = [1, 2, 3, 4, 5]
(x, y) = min_max(vals)
print(x, y)

def min_max(a_list):
    ...
    return (a_min, a_max)

Identify the file with the functions in it.
import utils

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

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

print(x, y)

---

def min_max(a_list):
    ...
    return (a_min, a_max)

---

def min_max(a_list):
    ...
    return (a_min, a_max)

---

Indicate that the function comes from that import.
A library of our functions

“Module”

Container → Functions → Objects → Parameters
## System modules

<table>
<thead>
<tr>
<th>Module</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>os</td>
<td>operating system access</td>
</tr>
<tr>
<td>subprocess</td>
<td>support for child processes</td>
</tr>
<tr>
<td>sys</td>
<td>general system functions</td>
</tr>
<tr>
<td>math</td>
<td>standard mathematical functions</td>
</tr>
<tr>
<td>numpy</td>
<td>numerical arrays and more</td>
</tr>
<tr>
<td>scipy</td>
<td>maths, science, engineering</td>
</tr>
<tr>
<td>csv</td>
<td>read/write comma separated values</td>
</tr>
<tr>
<td>re</td>
<td>regular expressions</td>
</tr>
</tbody>
</table>
Using a system module

```python
>>> import math

>>> math.sqrt(2.0)
1.4142135623730951
```

Keep track of the module with the function.
Don't do this

```python
>>> from math import sqrt

>>> sqrt(2.0)
1.4142135623730951
```

```python
>>> from math import *

>>> sqrt(2.0)
1.4142135623730951
```
Do do this

```python
>>> import math

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

1. Edit your utils.py file.

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

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

5 minutes
Interacting with the system

>>> import sys
Standard input and output

```python
>>> import sys

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

sys.stdout
Treat like an open(..., 'w') file
```
import sys
for line in sys.stdin:
    sys.stdout.write(line)

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)

Treat a file like a list  
Acts like a list of lines
Line-by-line copying — 3

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

An open file

The file's write() method

Standard output
import sys
for line in sys.stdin:
    sys.stdout.write(line)

$ python copy.py < in.txt > out.txt
Line-by-line actions

- Copying lines unchanged
  - Changing the lines
  - Gathering statistics
  - Only copying certain lines
Line-by-line rewriting

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

$ python process.py < in.txt > out.txt
Line-by-line filtering

```python
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
```

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: \( \text{len(line.split())} == 0 \) for blank lines.
The command line

We are putting parameters in our scripts.

We want to put them on the command line.

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

```
$ python script.py 1.25
```
Reading the command line

```python
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!
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

```python
import sys
number = float(sys.argv[1])
number = number + 1.0
print(number)
```
Worked example

Write a script to print points

\[(x, y) \quad y = x^r \quad x \in [0,1], \text{ uniformly spaced}\]

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 \((\text{float, integer})\) 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.

```python
import sys

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

    return (pow, num)
```
1b. Write a script that tests that function.

```python
import sys

def parse_args():
    ...
    (r, N) = parse_args()
    print 'Power: %f' % r
    print 'Points: %d' % N
```
1b. Write a script that tests that function.

```bash
$ python curve.py 0.5 5
Power:  0.500000
Points: 5
```
2a. Write a function that takes \((r, N)\) as \((\text{float, integer})\) and does the work.

```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 '%f %f' % (x, y)
```
2b. Write a script that tests that function.

def power_curve(pow, num_points):
    ...

power_curve(0.5, 5)
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.

```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):
    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)
```
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.
Back to our own module

```python
>>> 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

```python
>>> 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)
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)
Annotated function

```python
>>> import utils
>>> help(utils.min_max)
Help on function min_max in module utils:

min_max(numbers)
This function takes a list of numbers and returns a pair of their minimum and maximum.
```
def min_max(numbers):
    minimum = numbers[0]
    maximum = numbers[0]
    for number in numbers:
        ...
Annotated module

```python
>>> import utils
>>> help(utils)
Help on module utils:
NAME
    utils
FILE
    /home/rjd4/utils.py
DESCRIPTION
    A personal utility module full of all the pythonic goodness I have ever written.
```
Exercise

Annotate your utils.py and the functions in it.

3 minutes
Simple data processing

input data

What format?

Python script

output data
Comma Separated Values

<table>
<thead>
<tr>
<th>input data</th>
</tr>
</thead>
<tbody>
<tr>
<td>A101, Joe, 45, 1.90, 100</td>
</tr>
<tr>
<td>G042, Fred, 34, 1.80, 92</td>
</tr>
<tr>
<td>H003, Bess, 56, 1.75, 80</td>
</tr>
<tr>
<td>...</td>
</tr>
</tbody>
</table>

| 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”

```python
>>> line = '1.0, 2.0, 3.0, 4.0

>>> line.split(',

[ '1.0', ' 2.0', ' 3.0', ' 4.0
```

More likely to have come from sys.stdin

Split on commas rather than spaces.

Note the leading and trailing white space.
Quick and dirty .csv — 2

>>> line = '1.0, 2.0, 3.0, 4.0\n'
>>> strings = line.split(',','
')
>>> numbers = []
>>> for string in strings:
...    numbers.append(float(string))
...    
>>> numbers
[1.0, 2.0, 3.0, 4.0]
Quick and dirty .csv — 3

Why “quick and dirty”?

Can't cope with common cases:

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

Dedicated module:  csv
Proper .csv

Dedicated module:  csv

```python
import csv
import sys

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

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

### Storing data in the program

<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
<th>age</th>
<th>height</th>
<th>weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>A101</td>
<td>Joe</td>
<td>45</td>
<td>1.90</td>
<td>100</td>
</tr>
<tr>
<td>G042</td>
<td>Fred</td>
<td>34</td>
<td>1.80</td>
<td>92</td>
</tr>
<tr>
<td>H003</td>
<td>Bess</td>
<td>56</td>
<td>1.75</td>
<td>80</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

? `id → (name, age, height, weight)` ?
### Simpler case

Storing data in the program

<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>A101</td>
<td>Joe</td>
</tr>
<tr>
<td>G042</td>
<td>Fred</td>
</tr>
<tr>
<td>H003</td>
<td>Bess</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

id → name
Not the same as a list...

<table>
<thead>
<tr>
<th>index</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Joe</td>
</tr>
<tr>
<td>1</td>
<td>Fred</td>
</tr>
<tr>
<td>2</td>
<td>Bess</td>
</tr>
</tbody>
</table>

...  

['Joe', 'Fred', 'Bess', ...]

names[1] = 'Fred'
...but similar: a “dictionary”

<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>A101</td>
<td>Joe</td>
</tr>
<tr>
<td>G042</td>
<td>Fred</td>
</tr>
<tr>
<td>H003</td>
<td>Bess</td>
</tr>
</tbody>
</table>

`names['G042'] = 'Fred'`

`{ 'A101':'Joe', 'G042':'Fred', 'H003':'Bess', ... }`
Dictionaries

“key” → “value”

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

Generalized look up

Python object (immutable) → Python object (arbitrary)

string → string
int → int
tuple → tuple
string → string
tuple → list
Building a dictionary — 1

data = { 'A101': 'Joe', 'G042': 'Fred', 'H003': 'Bess' }
Building a dictionary — 2

data = {}

Empty dictionary

Square brackets

Key

Value

data[ 'A101' ] = 'Joe'

data[ 'G042' ] = 'Fred'

data[ 'H003' ] = 'Bess'

A101  →  Joe
G042  →  Fred
H003  →  Bess
Example — 1

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

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

Order is not preserved!
```
Example — 2

```python
>>> 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[\textit{key}]  Looking up in a dictionary
Example — 3

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

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

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

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

data['G042'] = ('Fred', 34)

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

Write a script that:

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

10 minutes
Worked example — 1

Reading a file to populate a dictionary

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

elements.txt

File

dictionary_to_name

Dictionary
Worked example — 2

data = open('elements.txt')  

symbol_to_name = {}  

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

data.close()  

Now ready to use the dictionary
Worked example — 3

Reading a file to populate a dictionary

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()
symbol_to_name = {}

data = open('elements.txt')

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

data.close()
symbol_to_name = {}

data = open('elements.txt')

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

data.close()
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()
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()
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()
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)
Exercise

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

2. Write a script that does this:
   a. Loads the file `elements.txt` as a dictionary
      (This maps 'Li' → 'lithium' for example.)
   b. Reads each line of `inputs.txt`
      (This is a list of chemical symbols.)
   c. For each line, prints out the element name

   10 minutes
Keys in a dictionary?

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

“Treat it like a list”
“Treat it like a list”

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

- File $\rightarrow$ List of lines
- String $\rightarrow$ List of letters
- Dictionary $\rightarrow$ 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?

```python
>>> data = {'a':'alpha', 'b':'beta'}
>>> data['g']
```

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

KeyError: 'g'

Dictionary equivalent of “index out of range”
Convert to a list

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

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

```
list(dictionary) ➔ [keys]
```
Exercise

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

```
symbol_to_name = 'Li' → 'Lithium'

name_to_symbol = invert(symbol_to_name)

name_to_symbol = 'Lithium' → 'Li'
```

10 minutes
One last example

Word counting

Given a text, what words appear and how often?
Word counting algorithm

Run through file line-by-line
Run through line word-by-word
Clean up word
Is word in dictionary?
If not: add word as key with value 0
Increment the counter for that word
Output words alphabetically
Word counting in Python: 1

```python
# Set up
import sys

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

- Need sys for sys.argv
- Empty dictionary
- Filename on command line
for line in data:
    for word in line.split():
        clean_word = cleanup(word)

We need to write this function.
def cleanup(word_in):
    word_out = word_in.lower()
    return word_out
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

```python
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]))
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