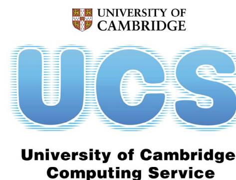
Python: Further Topics

Day One

Bruce Beckles
University of Cambridge Computing Service



1

Note that this course covers Python 2.4 to 2.7, which are the most common versions currently in use – it does **NOT** cover the recently released Python 3.0 (or 3.1) since that version of Python is so new. Python 3.0 is significantly different to previous versions of Python, and this course will be updated to cover it as it becomes more widely used.

The official UCS e-mail address for all scientific computing support queries, including any questions about this course, is:

`scientific-computing@ucs.cam.ac.uk`

Introduction

- Who:
  - Bruce Beckles, e-Science Specialist, UCS
- What:
  - Python: Further Topics course, *Day One*
  - Part of the ***Scientific Computing*** series of courses
- Contact (questions, etc):
  - scientific-computing@ucs.cam.ac.uk
- Health & Safety, etc:
  - Fire exits
- **Please switch off mobile phones!**

2

As this course is part of the Scientific Computing series of courses run by the Computing Service, all the examples that we discuss will be more relevant to scientific computing than to other programming tasks.

This does not mean that people who wish to learn about Python for other purposes will get nothing from this course, as the techniques and underlying knowledge taught are generally applicable. However, such individuals should be aware that this course was not designed with them in mind.

Note that there are various versions of Python in use, the most common of which are releases of Python 2.2, 2.3, 2.4, 2.5 and 2.6. (The material in this course is applicable to versions of Python in the 2.4 to 2.7 releases.)

On December 3rd, 2008, Python 3.0 was released. Python 3.0 is significantly different to previous versions of Python, is not covered by this course, and breaks backward compatibility with previous Python versions in a number of ways. As Python 3.0 and 3.1 become more widely used, this course will be updated to cover them.

Related/Follow-on courses

"Python: Operating System Access":

- Accessing the underlying operating system (OS)
- Standard input, standard output, environment variables, etc

"Python: Regular Expressions":

- Using *regular expressions* in Python

"Programming Concepts: Pattern Matching Using Regular Expressions":

- Understanding and constructing regular expressions

"Python: Checkpointing":

- More robust Python programs that can save their current state and restart from that saved state at a later date
- "Python: Further Topics" is a pre-requisite for the "Python: Checkpointing" course

"Introduction to Gnuplot":

- Using gnuplot to create graphical output from data

3

For details of the "Python: Operating System Access" course, see:

http://www.training.cam.ac.uk/ucs/course/ucs-pythonopsys

For details of the "Python: Regular Expressions" course, see:

http://www.training.cam.ac.uk/ucs/course/ucs-pythonregexp

For details of the "Programming Concepts: Pattern Matching Using Regular Expressions" course, see:

http://www.training.cam.ac.uk/ucs/course/ucs-regex

If you are unfamiliar with regular expressions, the following Wikipedia article gives an overview of them:

http://en.wikipedia.org/wiki/Regular_expression

…although that article does not express itself as simply as it might, so it may be most useful for the references it gives at the end. If you have met regular expressions before, but haven't yet used them in Python, then the "Python: Regular Expressions" course will teach you how to use them in Python. Alternatively, the Python "*Regular Expression HOWTO*" introductory tutorial also provides a good introduction to using regular expressions in Python:

http://docs.python.org/howto/regex

For details of the "Python: Checkpointing" course, see:

http://www.training.cam.ac.uk/ucs/course/ucs-pythonchckpt

For the notes of the "Introduction to Gnuplot" course, see:

http://www-uxsup.csx.cam.ac.uk/courses/Gnuplot/

If you are unfamiliar with gnuplot, you may wish to have a look at its home page:

http://www.gnuplot.info/

Pre-requisites

- Ability to use a text editor under Unix/Linux:
  - Try gedit if you aren't familiar with any other Unix/Linux text editors
- Basic familiarity with the Python language (as would be obtained from the "Python: Introduction for Absolute Beginners" or "Python: Introduction for Programmers" course):
  - Interactive and batch use of Python
  - Basic concepts: variables, flow of control, functions, Python's use of indentation
  - Simple data manipulation
  - Simple file I/O (reading and writing to files)
  - Structuring programs (using functions, modules, etc)

4

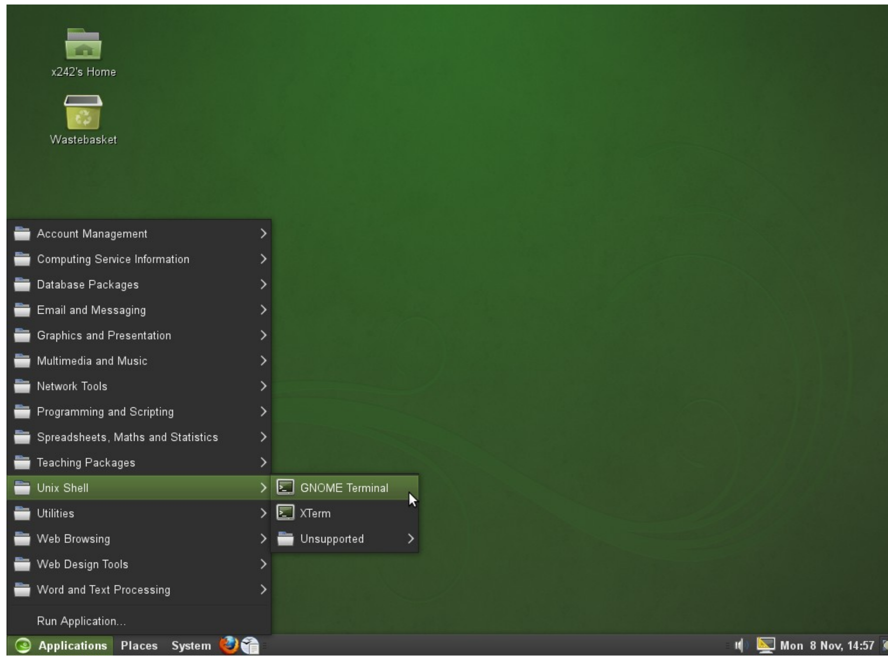
For details of the "Python: Introduction for Absolute Beginners" course, see:

http://www.training.cam.ac.uk/ucs/course/ucs-python

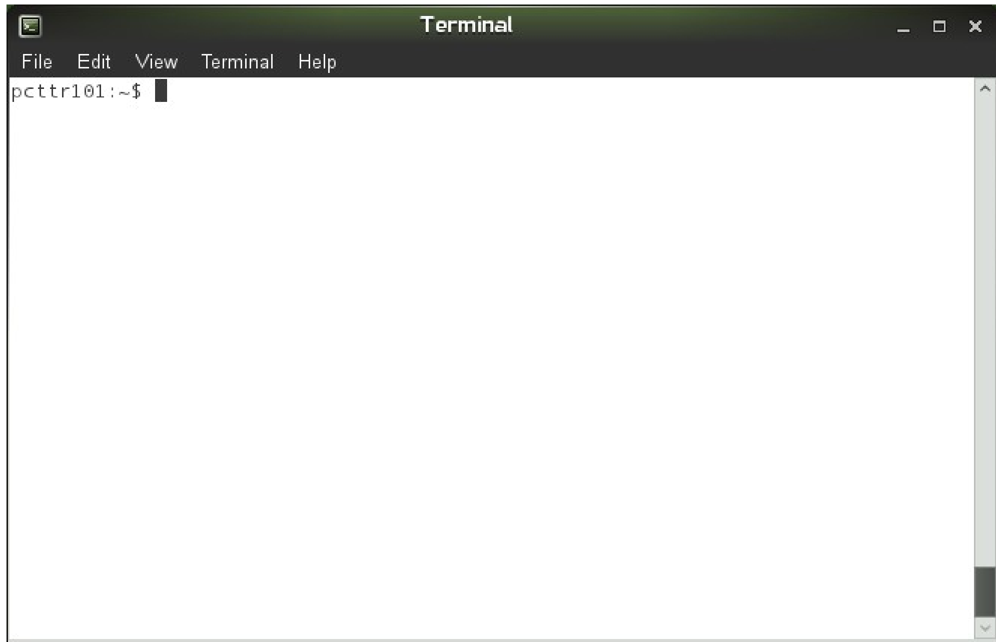
For details of the "Python: Introduction for Programmers" course, see:

http://www.training.cam.ac.uk/ucs/course/ucs-python4progs

Start a shell

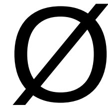
- 

Screenshot of newly started shell



6

None



None is a special value in Python, with its own data type (**NoneType**). It is Python's way of representing “nothing”. Its “truth value” is **False** (i.e. for the purpose of tests it is equivalent to **False**).

It is often used as “placeholder” value, or to mean that there is “no data”.

```
>>> None
>>>
```

```
>>> not None
True
```

```
>>> type(None)
<type 'NoneType'>
```

7

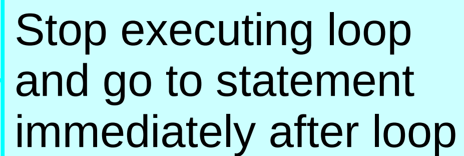
The value **None** is a Python special value (Python calls it a “null object”) designed for situations when you need a value, but that value should not represent anything. For instance, it is often used as a “placeholder” for variables that will be assigned a value later in the script. It has its own separate type (**NoneType**). For the purpose of tests, it is equivalent to **False** (i.e. its “truth value” is **False**).

Many Python functions use **None** to mean that there are no appropriate value(s) for whatever they were asked to do. We will see some examples of how it can be used later in this course.

It is also the value that is returned by a function that doesn't explicitly **return** anything else. So if your function does not explicitly **return** anything, then it actually returns **None**.

Controlling loops in Python: **break**

```
while x % 2 == 0 :
    print x, 'still even'
    x = x/2
    break

for prime in primes :
    print prime
    if prime > 5 :
        break
```



8

The **`break`** statement causes Python to stop executing whatever loop it might be executing and jump to the first statement immediately after that loop. If you have nested loops (i.e. loops within loops, e.g. a **`for`** loop within a **`while`** loop), the **`break`** statement will terminate the current innermost loop, transferring control to the next statement in the containing loop.

If your loop has an **`else`** block (**`while`** and **`for`** loops can have **`else`** blocks, although these are seldom used) then **`break`** will skip any statements in the **`else`** block, and jump to the first statement after the loop *and* its **`else`** block.

(If a **`while`** loop has an **`else`** block, the statements in this block will be executed when the test in the **`while`** loop evaluates to **`False`**, after which the rest of the script will be executed. If a **`for`** loop has an **`else`** block, the statements in the **`else`** block will be executed after the **`for`** loop completes, after which the rest of the script will be executed.)

Controlling loops in Python: **continue**

```
while x % 2 == 0 :
    if   x == 4   :
        continue
    x = x/2

for prime  in  primes :
    print prime
    continue
    print 'This line never executed'
```

Stop executing this iteration of the loop and go to the next iteration

9

The **`continue`** statement causes Python to stop executing the current iteration of whatever loop it might be executing and start on the next iteration of that loop. If you have nested loops (i.e. loops within loops, e.g. a **`for`** loop within a **`while`** loop), the **`continue`** statement will start the next iteration of the current innermost loop.

Function Arguments

```
>>> import utils
>>> utils.greet("Afternoon", "Bruce")
Good Afternoon, Bruce.
```

1st argument: "Afternoon"

2nd argument: "Bruce"

```
>>> utils.greet(person="Bruce", time="Afternoon")
Good Afternoon, Bruce.
```

named argument: **time**

named argument: **person**

10

We have already seen that when we use a function in Python we can give it some arguments (parameters). Up to now we have specified the function's arguments (its input) by *position*. We have called the function, specifying its arguments, and the **position** of each argument determines what the function does with that argument. Such arguments are called *positional arguments*, and the order in which they are given to the function is crucial. In the example above, the **`greet()`** function (a function I have specially written for this course and put in the **`utils`** module in your course home directory) prints out a greeting on the screen – the first argument is the time of day (morning/afternoon/evening/etc) and the second argument is the name of the person to whom the greeting is addressed.

Python also has another way of organising a function's arguments: rather than the position of an argument being used to determine what the function should do with it, the argument is given a *name* (Python calls the name a "*keyword*"), and that **name** determines what the function does with the argument. This allows the arguments to be given to the function in any order. Such arguments are called *named arguments* (or *keyword arguments*). In Python, the functions that the programmer writes automatically support both positional arguments and named (or keyword) arguments, and we can use either mechanism to specify the functions arguments when we use the function. The advantage of using named arguments is that we can specify them in any order we like, and the purpose of each argument is immediately clear (assuming the programmer chose sensible names for the arguments).

Note that some of the built-in functions in Python do not support named arguments, but the functions that you write will do so automatically.

Default Values and Optional Arguments

person is now an *optional* argument

Default value for **person**

```
def greet(time, person='user'):
    print "Good %s, %s." % (time, person)
```

utils.py

```
>>> import utils
>>> utils.greet('Afternoon')
Good Afternoon, user.
```

11

Open up the **utils.py** file in your course home directories with a text editor and edit the definition of the **greet()** function as shown above (note that to fit everything on the slide we haven't shown the function's doc string), i.e. add:

```
='user'
```

immediately after "person" on the first line (the line with the def keyword) of the greet() function's definition. Thus first line should now look like this:

```
def greet(time, person='user'):
```

The rest of the function is unchanged. Make sure you save the file after you've made this change.
(If you have been following along in the Python interpreter and still have a copy of the interpreter running after you've edited the **utils.py** file, quit the interpreter and restart it before trying the changed **greet()** function.)

This function now has what is called "a *default* value" for the **person** argument (parameter). If we import the **utils** module, and then call the **greet()** function without specifying a value for the **person** argument, then **person** will be set to this default value, i.e.

```
>>> utils.greet('Afternoon')
Good Afternoon, user.
```

is exactly the same as

```
>>> utils.greet('Afternoon', 'user')
Good Afternoon, user.
```

This means that **person** is now an *optional* argument: we no longer have to specify a value for it when we call the function; if we don't specify a value, Python will use the default value we've given in the function definition.

VERY IMPORTANT: Once you've defined a default value for one argument in the function definition, you need to define default values for ***all*** the *following* arguments taken by the function. So, in the **greet()** function, if we had given a default value for the **time** argument, we would **have** to also specify one for the **person** argument as well.

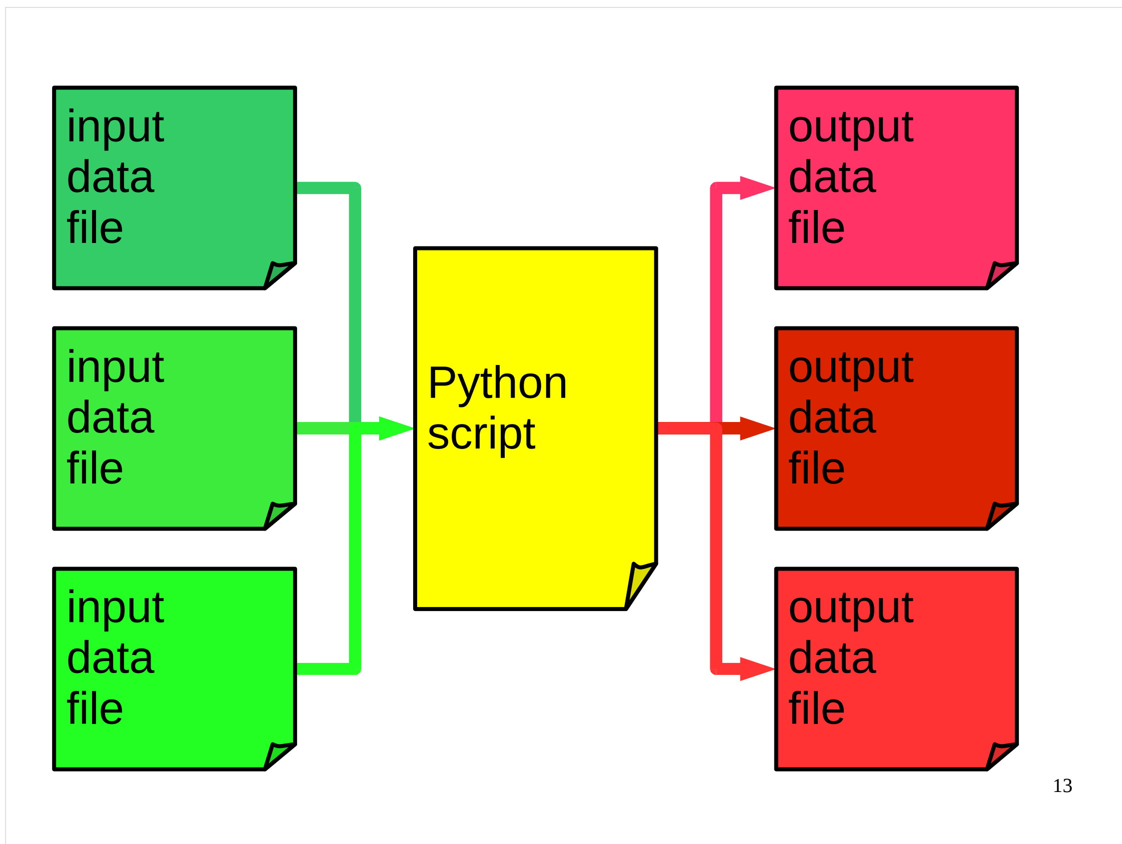
Accessing files

1. Direct access to files
2. Structured files: **csv** module

12

Today, we're going to examine file I/O (input and output) in Python. We'll start off with a quick recap of the basics (as was covered in the "Python: Introduction for Absolute Beginners" course, and in a more abbreviated fashion in the "Python: Introduction for Programmers" course) and then move on to more advanced topics.

First we will consider directly accessing files ourselves, and then move on to how we can access structured files with the help of the **csv** module.



We want to be able to directly access files from Python. In particular, we want to be able to cope with the situation where there is more than one input and/or output file.

Reading a file

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

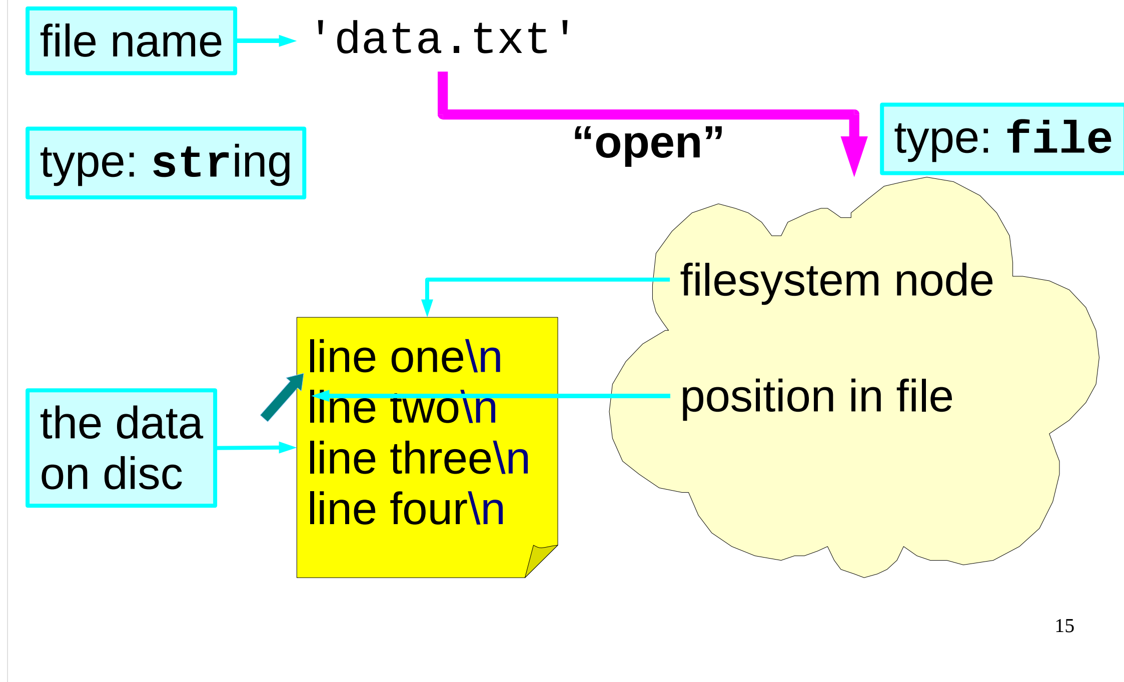
14

So, we will start by reminding ourselves how we read a file in Python. There are three phases if we start with a file name.

We “open” the file. This takes the name of the file, checks to see if it exists and gives us a “handle” − a special type of data type known as a **`file`** object − on the contents of the file itself.

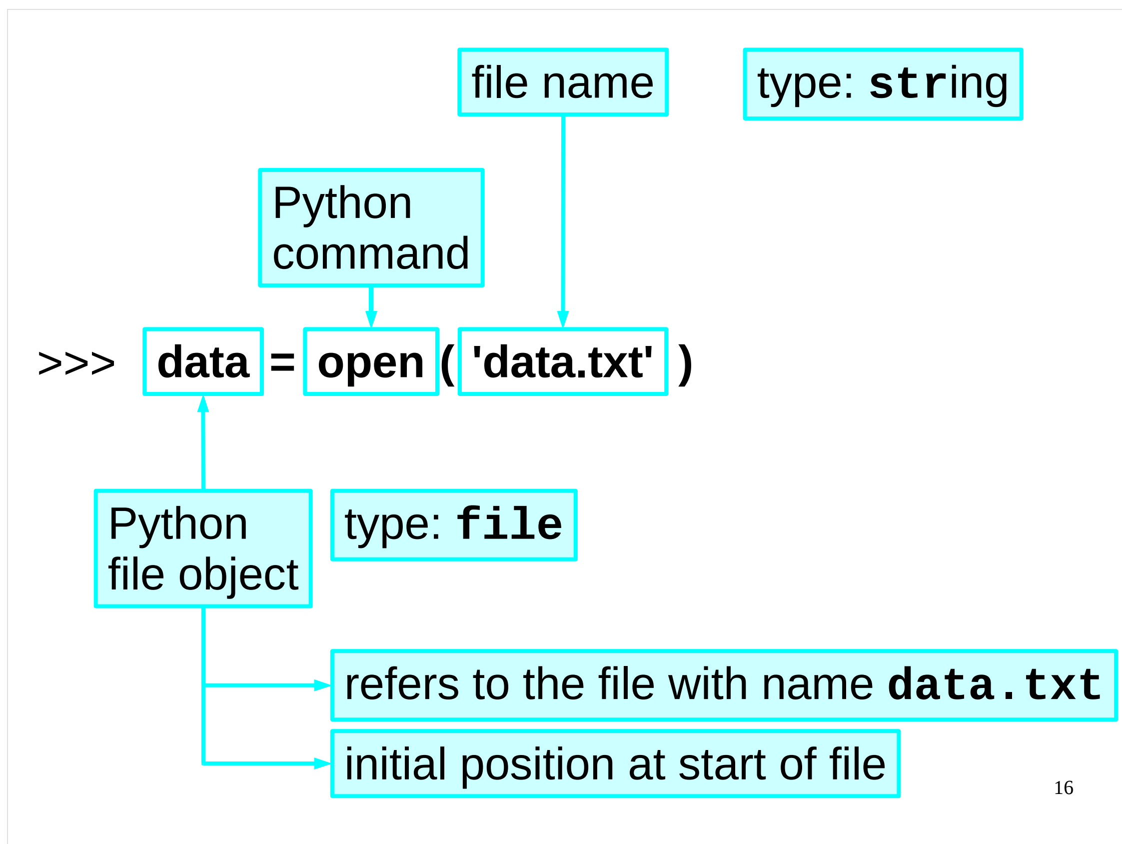
Then we use that **`file`** object to read the file’s contents.

Then we disconnect from the file by declaring that we are done with it now. This is called “closing” the file (in contrast to “opening” it in the first place).



Opening a file involves taking the file name and getting some Python object whose internals need not concern us – it's a Python type called "**`file`**" (logically enough). If there is no file of the name given, or if we don't have permission to get at this file, then the opening operation fails. If it succeeds we get a **`file`** object that has two properties of interest to us. It knows the file on disc that it refers to, obviously. But it also knows how far into the file we have read so far. This property, known as the "offset", obviously starts at the beginning of the file when it is first opened. As we start to read from the file the offset will change.

Think of opening a file, given its name, as being in a library. You are given a book's name, you fetch the book from the shelves (if it exists and you have permission), you open the book and place your finger under the first letter of the first word of the book, ready to read.



In Python, we open a file with the "**open**" command. (You may also meet some scripts that use the "**file**" command instead of the "**open**" command. The "**file**" command is used to create **file** objects and so can be used to do the same thing as the "**open**" command, but this is not really a very good idea. Stick to the "**open**" command for opening files.)

In your Python directories there is a file called "**data.txt**". If you enter Python interactively and give the command

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

then you should get a **file** object for this file inside the Python interpreter.

Note that we just gave the name of the file, we didn't say where it was. If we don't give a path to the file then Python will look in the current directory. If we want to open a file in some other directory then we need to give the path as well as the name of the file to the **open** command. For instance, if we wanted to open a file called "**data.txt**" in the **/tmp** directory, we would use the **open** command like this: **open('/tmp/data.txt')**.

If you want to know which directory is your current directory, you can use a function called **getcwd()** ("**get** **c**urrent **w**orking **d**irectory") that lives in the **os** module:

```
>>> import os
>>> os.getcwd()
'/home/x282'
```

(If you try this on the computer in front of you, you will find that it displays a different directory to the one shown in these notes.)

You can change your current directory by using the **chdir()** ("**ch**ange **dir**ectory") function, which also lives in the **os** module. You give the **chdir()** function a single argument: the name of the directory you want to change to, e.g. to change to the **/tmp** directory you would use **os.chdir('/tmp')**. However, ***don't try this now***, as if you change the current directory to something other than your course home directory then many of the examples in these notes will no longer work! (If you have foolishly ignored my warning and changed directory, and don't remember what your course home directory was called (and so can't change back to it), the easiest thing to do is to quit the Python interpreter and then restart it.)

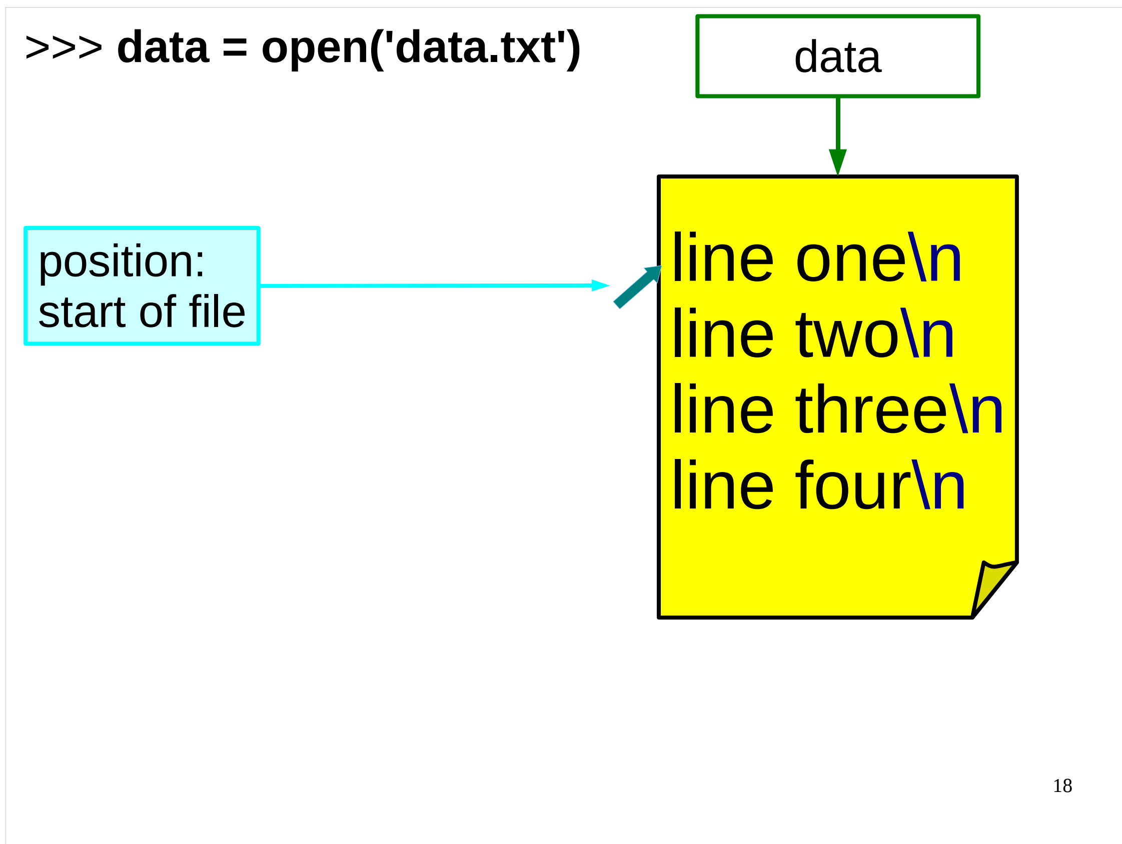
Reading a file

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

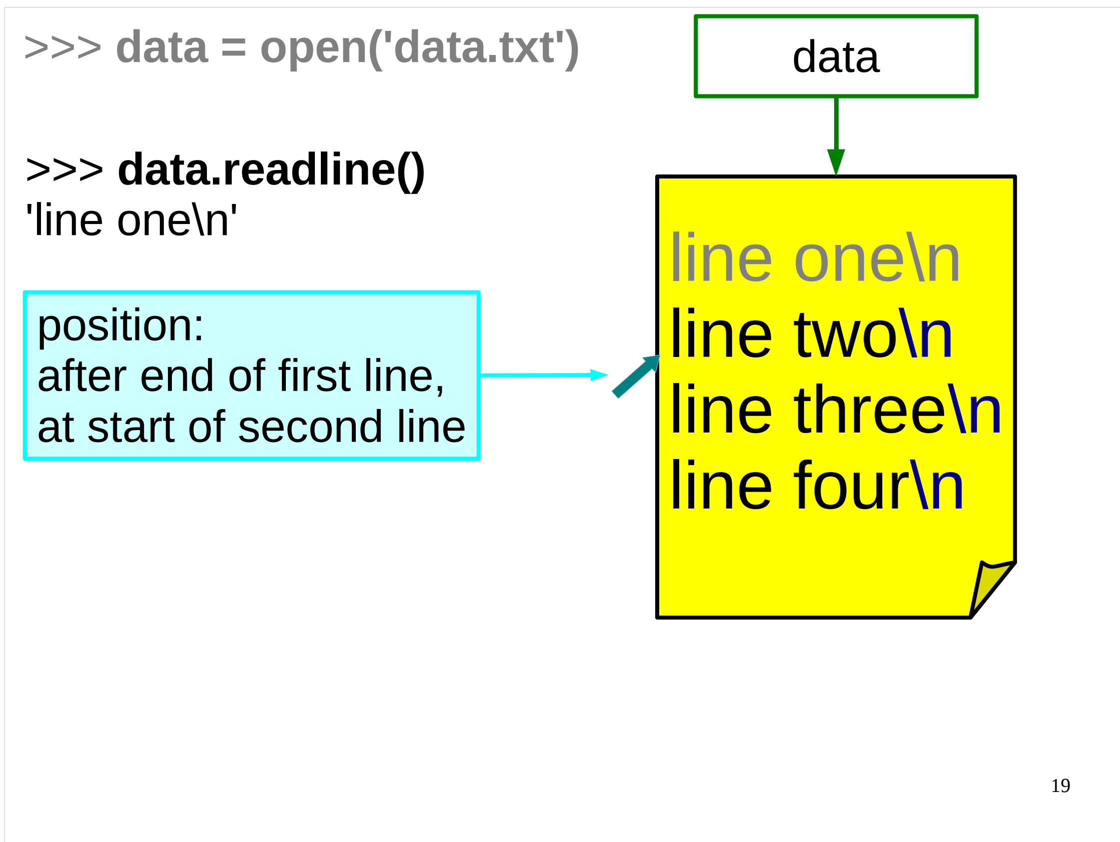

To read a file line by line (which is typical for text files), the **`file`** object provides a method to read a single line. (Recall that methods are the “built in” functions that objects can have.) The method is called “**`readline()`**” and the **`readline()`** method on the **`data`** object is run by asking for “**`data.readline()`**” with the object and method name separated by a dot.

There are two important things to notice about the string returned. The first is that it’s precisely that: one line, and the first line of the file at that. The second point is that it comes with the trailing “new line” character, shown by Python as “**`\n`**”.

Now observe what happens if we run exactly the same command again. (Remember that Python on PWF Linux has a history system. You can just press the up arrow once to get the previous command back again.) This time we get a different string back. It’s the second line.



Let’s take a closer look at what just happened to be sure we understand how reading from a file works. We started with the **`file`** object for the “**`data.txt`**” file having its offset point to the start of that file. That's what we always get immediately after an **`open()`**.



Then we ran **`data.readline()`**. This read one line from the file and returned it to us in a string. As it did so it moved the offset along to just beyond the last character read. It's now at the start of the second line.

In our book analogy, as you read the words you slide your finger along. It's now at the start of the second line ready for you to read that.

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

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

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

data

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

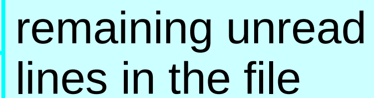
line one\n
line two\n
line three\n
line four\n

20

Next time we call **`readline()`** we get the line following on from the current position in the file.

(It is possible to put the position in the middle of a line using a method that I'm not going to mention yet. What **`readline()`** generates is everything from the current position to the end of the line, including the new line character.)

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

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

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



21

Just so you know, there's another method which is occasionally useful. The "**`readlines()`**" method gives all the lines from the current position to the end as a list of strings.

We won't use **`readlines()`** much as there is a better way to step through the lines of a file.

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

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

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

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

data

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

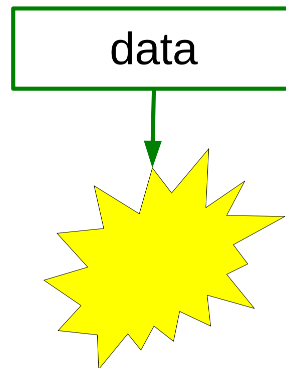
position:
at end of file

22

Once we have read to the end of the file the position marker points to the end of the file and no more reading is possible (without changing our position in the file, which we're not going to discuss yet).

Closing a file

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

>>> data.close()
```

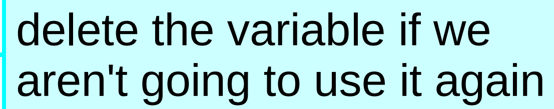


The method to close a file is, naturally, "**`close()`**".

It's only at this point that we declare to the underlying operating system (Linux in this case) that we are finished with the file. On operating systems that lock down files while someone is reading them, it is only at this point that someone else can access the file.

Closing files when we are done with them is important, and even more so when we come to examine writing to them.

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

>>> data.close()

>>> del data
```



24

We should practice good Python variable hygiene and delete the **`data`** variable if we aren't going to use it again immediately.

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

25

Some Python objects have the property that if you treat them like a list they act like a particular list. **`file`** objects act like the list of lines of the file, but be warned that as you run through the lines you are running the offset position forward, i.e. you won’t get the same results twice (in fact, most likely you won’t get anything the second time) unless you move the offset back to where it was. (We’ll see how to do this later.)

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`

26

So let’s look at a sensible example. In the “Python: Introduction for Absolute Beginners” course we created a function – that we put in a module called “**`utils`**” – that took a file name as its argument, read in the lines, expecting each line to consist of two simple words, and returned a Python dictionary where the first word on each line is a key of the dictionary and the second word is the corresponding value.

In your course home directory you will find this **`utils`** module (the module is in a file called **`utils.py`**), containing the function that does this. The function is called “**`file2dict()`**” and we’ll examine it now to see what it does. Then we’ll try improving it.

(For the pedants reading, the author knows that the file we’re using this function on really contains the atomic elements rather than chemicals, and so it would be better to call it “**`elements.txt`**” rather than “**`chemicals.txt`**”. However, the word “element” is often used to refer to the individual items in lists or dictionaries (as in “an element of a list” or “an element of a dictionary”), and I’d rather avoid any confusion.)

The 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.

27

This is what the function does in words…

Simple, really.

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

utils.py

28

…and this is what it does in Python.

We know how to create a function – chiefly, we need to decide on its name and the name(s) of its argument(s). This function is called "**`file2dict()`**" and its single argument is "**`filename`**". Remember that the body of the function must be indented.

This function takes the approach to creating a dictionary of starting with an empty one and adding entries as it proceeds. So first it creates an empty dictionary.

It needs access to the content of the file so it **`open`**s the file to read it.

Next it needs to step through the lines of the file. It uses the standard Python trick of treating something like a list and having it behave like a list.

Then it needs to split the line into two words. It does this using the **`split()`** method (with which you should be familiar). Note that **`split()`** discards all white space, including the new line character as well as the spaces between the words. Observe that it assigns the list of words generated to a list of two variable names. If any lines in the file don't consist of exactly two words, this line (and the function) will fail.

Having got two words it adds them into the dictionary.

Once it finishes with the file it **`close`**s it.

Now that it is done it hands back the dictionary, and the function ends.

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

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

mkdict.py

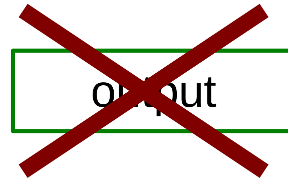
29

So we can use it to create a dictionary from a text file structured in a particular way.

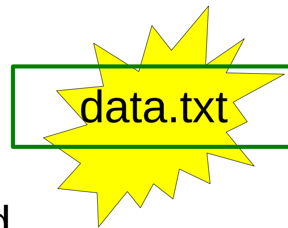
(Again, for the pedants reading, the author knows that the text file really contains the atomic elements rather than chemicals, and so it would be better to call the dictionary we create "**`chemicals`**" rather than "**`elements`**". However, as I said, I want to avoid confusion with the use of the word "elements" in "elements of lists" or "elements of dictionaries" to refer to the individual items in lists or dictionaries.)

What if something goes wrong?

```
>>> data = open('output')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IOError: [Errno 2] No such file or directory: 'output'
```




```
>>> data = open('data.txt')
>>> data.readlines()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IOError: [Errno 13] Permission denied
```

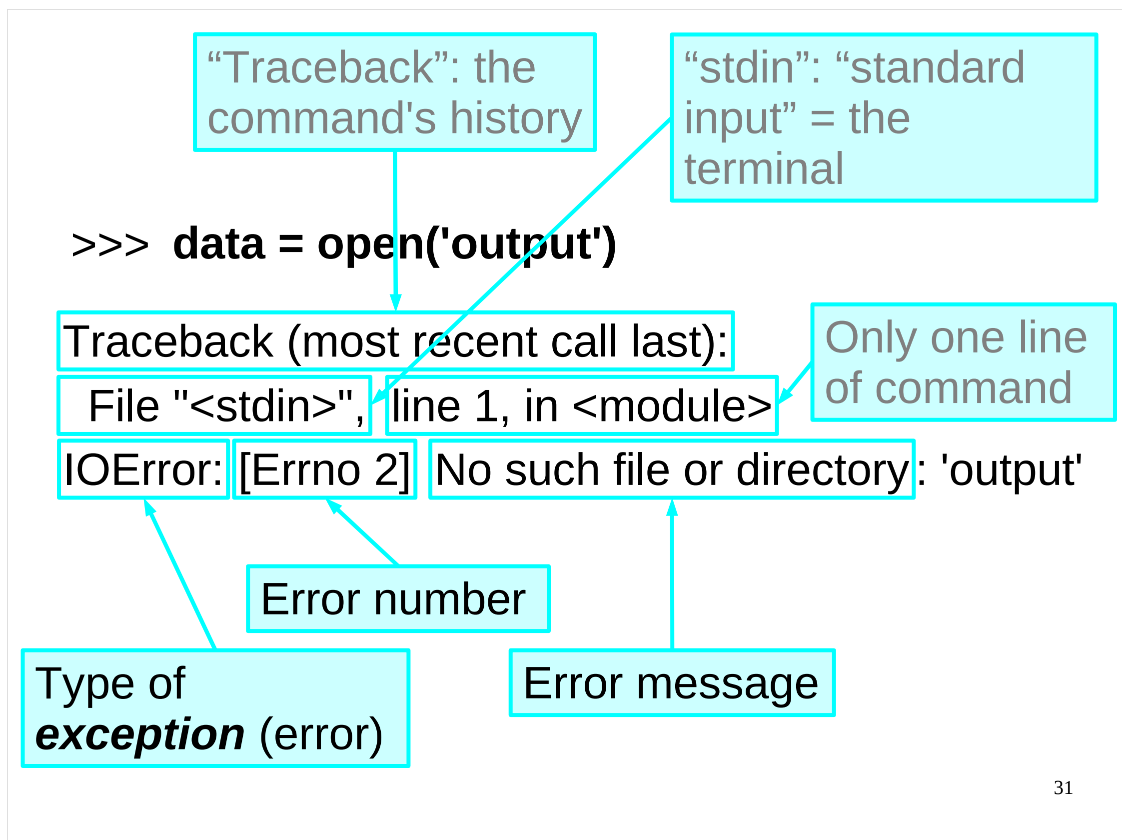


30

What happens if we try to open a file that doesn't exist? Or what if something happens to the file whilst we're reading it? – perhaps it is on a network filesystem and there's a network problem, or maybe there's a bad sector on the disk where the file is stored.

Dealing with files is one of the places where we are most likely to encounter *unexpected* errors that are beyond our control or impossible to predict in advance (or avoid). So how can we handle such unexpected errors?

(Note that the errors shown above are *examples* of what *could* happen while you are performing, or attempting to perform, file I/O operations in Python. You won't necessarily get those results if you try the Python commands above now in class. If you do open the file **`data.txt`** make sure you **`close()`** it before continuing.)



Errors in Python are called *exceptions*. When something goes wrong, Python will "*raise an exception*", i.e. generate an error. It is up to your script to *handle* (deal with) that error. If your script does not have a mechanism for handling the exception, then Python's default exception handler is used, which normally halts your script and prints out some error messages similar to those above.

The first line of the error message above declares that what we are seeing is a "traceback" with the "most recent call last". A traceback is the command's history: how we got to be here making this mistake, if you like. The error itself will come at the end. It will be preceded with how we got to be there. (If our script has jumped through several hoops to get there we will see a list of the hoops.) In the above example the error occurs as soon as we try to open the file, so the traceback is pretty trivial.

The next two lines are the traceback. In more complex examples there would be more than two but they would still have the general structure of a "file" line followed by "what happened" line.

The file line says that the error occurred in a file called "<stdin>". What this actually means is that the error occurred on Python being fed to the interpreter from "standard input". Standard input means our terminal. We typed the erroneous line and so the error came from us.

Each line at the ">>>" prompt is processed before the prompt comes back. Each line counts as "line 1".

If the error had come from a script we would have got the file name instead of "<stdin>" and the line number in the script.

The "<module>" refers to what function (and module) we were in. This command wasn't in a function (or a module), so we get "<module>" as the function name (indicating we weren't in a function, or a module for that matter).

The third line gives information about the error itself. First comes a description of what type of error (*exception*) has occurred. This is followed by a more detailed error message – some errors (such as this one) also have an error number. If the error had come from a script the line of Python that generated this error would also be reproduced here.

Exception handling

try:
Python commands
except:
Exception handler

Python exception handling:

0. ***try*** some commands
1. if there's an error…
2. …execute the ***except*** block…
3. …but if there's no error, don't execute the ***except*** block.

(Similar to **if**…**else** statements)

32

Exception handling in Python is done using the **try**…**except** construct. Essentially, whenever I think that some commands may fail, I create an exception handler for the errors I expect using the **try**…**except** construct. If I don't specify a specific exception (error) to handle, then my exception handler is used for *any* errors that occur while executing those commands. If my exception handler only handles certain exceptions and my script produces an exception that my handler wasn't written to deal with, Python's default exception handler will handle that error for me.

Using the **try**…**except** construct will become clearer as we do some examples.

You can get a list of all the built-in exceptions that Python knows about in the Python documentation here:

http://docs.python.org/library/exceptions.html

Note that those are the exceptions that are part of Python itself – however, it is also possible to define new exceptions (these are known as "user-defined exceptions"), and many Python modules do this. If a module defines a new exception, it should document this in its documentation, and, if it is a well written module, in its doc string.

```
def file2dict(filename):
    import sys
    dict={}
    try:
        data = open(filename)
        for line in data:
            [ key, value ] = line.split()
            dict[key] = value
        data.close()
    except IOError:
        print "Problem with file %s" % filename
        print "Aborting!"
        data.close()
        sys.exit(1)
    return dict
```

utils.py

33

Modify the **`file2dict()`** function as shown above. ***Note that the commands in the `try` and `except` blocks have to be indented.***

As already mentioned, when dealing with files, it is very important that you **`close()`** any files you were using when you are finished with them. This is especially true if something has gone wrong. So, if we run into problems while reading the file, we print an error message and close the file (or try to). Since we don't really know what it would be safe to do at this point, we just make whatever script called us exit with an error. Note that although this may seem a draconian response, it is no more than what Python would do anyway if we didn't handle this error.

Apart from the **`try`**…**`except`** construct, you should be familiar with all the Python in the function above with the exception of the **`exit()`** function.

The **`exit()`** function lives in the the **`sys`** module – which is why we have to put "**`import sys`**" at the start of the **`file2dict()`** function – and causes your script to stop what it is doing and return to the operating system (or whatever program called it). If you give the **`exit()`** function an integer as input, then that integer will be the *exit status* of the program. If you don't supply an integer, then the **`exit()`** function behaves as though it had been called with the integer 0 (i.e. the exit status will be 0). By convention, the exit status of a program or script should be 0 if it completed successfully, and non-zero if it didn't. If we get to the **`except`** block then there's been a problem, so we should exit with a non-zero exit status.

If you are unfamiliar with the exit status concept (also called *exit code*, *return code*, *return status*, *error code*, *error status*, *errorlevel* or *error level*), the following Wikipedia article gives a bit more detail:

http://en.wikipedia.org/wiki/Exit_status

Apart from the **`exit()`** function and the **`try`**…**`except`** construct, if there is any Python above which you don't understand please put up your hand now and ask the course giver to explain.

Don't forget to save the file after you've finished it or your changes won't take effect.

```
>>> import utils
>>> mydict = utils.file2dict('output')
Problem with file output
Aborting!
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "utils.py", line 110, in file2dict
    data.close()
UnboundLocalError: local variable 'data' referenced before assignment
>>>
```

34

Well, that didn't work quite as expected.

Observe that the error is in our **`except`** block, and so Python's default exception handler takes over to handle it (well, an **`except`** block can hardly be expected to handle an error within itself).

The problem here is that, since the file doesn't exist, the variable **`data`** never gets assigned a value, and so it doesn't exist when we reference it to **`close()`** the file in our **`except`** block. The easiest way of fixing this problem is to assign **`data`** a value before we enter the **`try`** block. What value should we use? Well, it doesn't really matter so long as it is not another **`file`** object. We'll use the value **`None`** that we met earlier, as it is a Python special value designed for such purposes. Many Python functions use it to mean that there are no appropriate values for whatever they were asked to do.

So let's fix this error and try again.

```
def file2dict(filename):
    import sys
    dict={}
    data = None
    try:
        data = open(filename)
        for line in data:
            [ key, value ] = line.split()
            dict[key] = value
        data.close()
    except IOError:
        print "Problem with file %s" % filename
        print "Aborting!"
        if type(data) == file:
            data.close()
        sys.exit(1)
    return dict
```

utils.py

35

Modify the **`file2dict()`** function as shown above. ***Remember to indent the "`data.close()`" line for the `if` statement.***

You should be able to see why the above modifications will fix the error we encountered before. If you are confused, please ask the course giver to explain.

Don't forget to save the file after you've finished it or your changes won't take effect.

```
>>> import utils
>>> mydict = utils.file2dict('output')
Problem with file output
Aborting!
$
```

36

Note that the **`exit()`** function not only causes our function to stop running, it throws us out of the Python interpreter as well.

Now, our error message is a little vague (“Problem with file”) and it would be nice if we could be a bit more specific. Recall that Python’s default exception handler not only tells you the type of error but gives you some detail in the form of an “error message”. How can we get access to this?

```
def file2dict(filename):
    import sys
    dict={}
    data = None
    try:
        data = open(filename)
        for line in data:
            [ key, value ] = line.split()
            dict[key] = value
        data.close()
    except IOError, error:
        (errno, errdetails) = error
        print "Problem with file %s: %s" % (filename, errdetails)
        print "Aborting!"
        if type(data) == file:
            data.close()
        sys.exit(1)
    return dict
```

utils.py

37

Modify the **`file2dict()`** function as shown above.

Don't forget to save the file after you've finished it or your changes won't take effect.

As well as specifying what type of exception we want to handle, we can also get the error information produced by Python put into a variable of our choice. We do this by specifying the name of our chosen variable, separated by a comma, after the type of exception we want to handle. So a more complete syntax for the **`try`**…**`except`** construct is:

```
try:
    commands
except ExceptionType, errorvariable:
    exception handler commands
```

where ***ExceptionType*** is the type of exception we want to handle (**`IOError`** is the type of exception raised if there is a problem with file I/O) and ***errorvariable*** is the optional variable in which we want information about the error to be placed.

The information about the error may actually contain several pieces of information: the error number, the error message, etc. If so, we would normally want to use these separately, so we would "unpack" this variable using a tuple of variables to hold its constituent parts (recall that we can do exactly this sort of "unpacking" for lists or tuples to get the values in the list or tuple into individual variables). The way the error information is stored for errors that result in the **`IOError`** exception being raised is that the first value is the error number and the second is the actual error message.

The Python documentation gives some details about the error information available with the different sorts of exception:

http://docs.python.org/library/exceptions.html

```
>>> import utils
>>> mydict = utils.file2dict('output')
Problem with file output: No such file or directory
Aborting!
$
```

38

Now we have a much better exception handler. It tells us quite specifically what the problem is, and then **exit**s our program.

We now have a reasonably well-written Python function that reads a dictionary from a file, exiting with an informative error message if something goes wrong with the file I/O. Can we make improve this function's error handling even further?

```
>>> line = "Too many values"
>>> [ key, value ] = line.split()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: too many values to unpack
>>> line = "notenough!"
>>> [ key, value ] = line.split()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: need more than 1 value to unpack
>>>
```

39

What happens if the data in the file isn't in the format we were expecting?

Using the Python interpreter, we can simulate this error quite easily.

The variable **`line`** gets set to the next line of data in the file we are reading. We then use **`split()`** on this variable, which returns a list of *words* in the line we've just read, i.e. a list of collections of characters separated by *whitespace* (spaces, tabs, etc). We then "unpack" this list into two variables, **`key`** and **`value`**. If the list does not contain **exactly** two items (i.e. if line does not contain exactly two "words"), then Python will complain.

We can simulate such errors by setting **`line`** to a string that contains more than two words, and one that contains fewer than two words, and then using **`split()`** and trying to unpack the resulting list, exactly as we do in our function. If we do this, we see that the exception that Python raises (in both error cases) is a **`ValueError`** exception. (Note that the information about the error that we get with a **`ValueError`** exception is just the error message, there is *no* error number as there was with **`IOError`** exceptions.)

So, perhaps we could modify our function to handle this type of exception?

The plan

0. Find the part of the function where the **`ValueError`** may occur.

1. Place it in a **`try`**…**`except`** construct:

   1a. **`try`** block contains statement(s) that may fail.

   1b. **`except`** ***`ExceptionType`*** block says what to do on failure:

   i). print a message saying what has gone wrong.

   ii). Set the dictionary to **`None`**.

   iii). Stop processing the file.

40

This is the approach we will take.

We first identify the part of the function where the exception we intend to handle (the **`ValueError`** exception) occurs.

Then we surround this part of the function with a **`try`**…**`except`** construct.

We have to decide what we want to do on failure. The sensible thing is to tell the user that there is something wrong with the file. We'll use a **`print`** statement for that.

We have two options at this point: we can either halt the Python program using the **`exit()`** function, as we have done before, or we can carry on.

Arguably, a file that is in the wrong format is not so serious an error that we should automatically force the program to quit. However, we don't want to return a garbled or incomplete dictionary either. The sensible thing would be to return something that couldn't possibly be a valid dictionary (the special **`None`** value, for instance). The program or user who called our function can then look at what they got back and, if it is not a dictionary, then *they* can decide whether to quit or do something else (for example, they could try reading from a different file).

We then need to stop processing the file (well, we could continue, but there would be no point reading any more of the file as at this point we've decided to return **`None`** anyway). In this function we read and process data from the file in a **`for`** loop, so we need a way of telling Python to quit the **`for`** loop. (Recall that the Python statement that forces a **`for`** (or **`while`**) loop to terminate is the **`break`** statement.)

Exercise

Add exception handling to the **`file2dict()`** function for the **`ValueError`** exception.

```
def file2dict(filename):

        try:
            statement(s) from original function
        except ...
            print ...
            ...
            break
```

utils.py

41

Now I want you to modify the **`file2dict()`** function in the **`utils.py`** file in your course home directory to add exception handling for the **`ValueError`** exception.

In case you get stuck, here are a couple of hints/pointers:

- Remember that the statement(s) from the original function that you are putting into the **`try`** block need to be indented!
- Remember to specify the type of exception your **`except`** block should handle (if you don’t it will try to handle *all* exceptions).
- If you’ve positioned your exception handler properly within the function then you don’t need to worry about closing the file, since your exception handling for this exception should occur before the file would normally be closed. Thus you don’t need to worry about closing the file in your exception handler as this will happen anyway.
- The last line of your **`except`** block will be a **`break`** statement.

If (and only if!) you really can't manage it take a look at the page after next.

After you’ve done this exercise take a short break (i.e. **stop** using the computer) and then we’ll continue.

This page intentionally left blank

Deze bladzijde werd met opzet blanco gelaten.

このページは計画的にブランクを残ている

Ta strona jest celowo pusta.

Esta página ha sido expresamente dejada en blanco.

Эта страница нарочно оставлена пустой.

Denne side med vilje efterladt tom.

Paĝon intence vaka.

اين صفحه خالي است

An leathanach seo fágtha folamh in aon turas.

42

This page intentionally left blank: nothing to see here. If you're stuck for an answer to the exercise, have a look at the next page.

```
def file2dict(filename):

    Beginning of function unchanged

        for line in data:

            try:
                [ key, value ] = line.split()
            except ValueError:
                print "File %s is not in the correct format." % filename
                dict = None
                break

            dict[key] = value

        data.close()

    Rest of function unchanged
```

utils.py

43

You should have modified the **`file2dict()`** function in a similar way to the modifications shown above.

(Don't forget to save the file after you've finished it or your changes won't take effect.)

There are two points worth mentioning regarding the above exception handler. Firstly, note that we not only tell the user what the problem is, but we tell them what file has caused the problem – this will help them track down the problem. Secondly, note that if our exception handler for the **`ValueError`** exception is called we will exit the **`for`** loop and the file will then be closed, thus we don't need to worry about closing it in the exception handler.

You can now try this function on two files in your course home directory that aren't in the correct format: `bad-format1.txt` and `bad-format2.txt`.

```
>>> import utils
>>> dict1 = utils.file2dict('bad-format1.txt')
File bad-format1.txt is not in the correct format.
>>> print dict1
None
>>> dict2 = utils.file2dict('bad-format2.txt')
File bad-format2.txt is not in the correct format.
>>> print dict2
None
```

If you had problems with the exercise, or if you don't understand the Python above, please let the course giver know.

Handling multiple exceptions

try:
Python commands
except *Exception1*:
Exception handler1
except *Exception2*:
Exception handler2
...
except:
Handler for all other exceptions

0. ***try*** some commands
1. if there's an error…
2. …examine the ***except*** blocks…
3. …if the error is ***Exception1*** use that ***except*** block…
4. …if it's ***Exception2*** use that ***except*** block…
5. …and so on…
6. …if it's not any of the listed exceptions, use the final ***except:*** block if it exists.

If you need to handle multiple exceptions in the same block of code, you simply list each of the exception handlers in separate `except` blocks one after the other with each `except` block indicating what type of exception it is supposed to handle in its `except` statement, You can have as many of these `except` blocks as you want.

After you've finished defining exception handlers for specific exceptions, you can then have a final `except` block which will handle any other exceptions not previously handled – this `except` block will be written just as "**`except:`**" since it is supposed to handle multiple types of exception.

For example, consider the code snippet below:

```
try:
    my_function(data)
except IOError:
    print "There was an I/O error."
except ValueError:
    print "There was something wrong with a value."
except:
    print "There was some other error!"
```

This code snippet will call the **`my_function()`** function with **`data`** as its argument. If an **`IOError`** exception occurs, it will print "`There was an I/O error.`". If a **`ValueError`** exception occurs, it will print "`There was something wrong with a value.`". If any other type of exception occurs it will print "`There was some other error!`".

Exception handling: exc_info()

import sys

exc_info() returns a *tuple* of three items of information about the current exception: (Exception*Type*, Exception*Details*, *Traceback*)

(err_type, err_value) = **sys.exc_info() [:2]**

It is **dangerous** to access the traceback so **don't**: use a slice of the *first two items* in the tuple

Variable for *type* of exception e.g. **`IOError`**

Variable for exception *details* e.g. (2, 'No such file or directory')

45

The **`exc_info()`** function (which lives in the **`sys`** module) returns **exc**eption **info**rmation. It returns a tuple of three items relating to the current exception. (If no exception has been raised then it returns the tuple `(None, None, None)`.)

The first item is the *type* of the exception, e.g. **`IOError`**, **`ValueError`**, etc.

The second item contains the exception *details*, e.g. for an **`IOError`** exception it might contain the tuple `(2, 'No such file or directory')`. These details are the information about the error that we've accessed earlier using a **`try`**…**`except`** construct for a specific type of exception.

The third item contains the *traceback*, which is the listing of the lines of our script that have gotten us to this error. We've seen Python display the traceback before when we've made errors, but we have not tried to access it. In fact, it can be quite dangerous to interfere with the traceback, and, in particular, **assigning it to a local variable in a function will cause a circular reference**. Therefore it is best not even to access this item. Consequently you normally call the **`exc_info()`** function like this:

```
sys.exc_info()[:2]
```

The "**`[:2]`**" tells Python to take a slice of the first two items of the tuple returned by the **`exc_info()`** function, i.e. (the type of the exception, the exception details), and so it ignores the traceback.

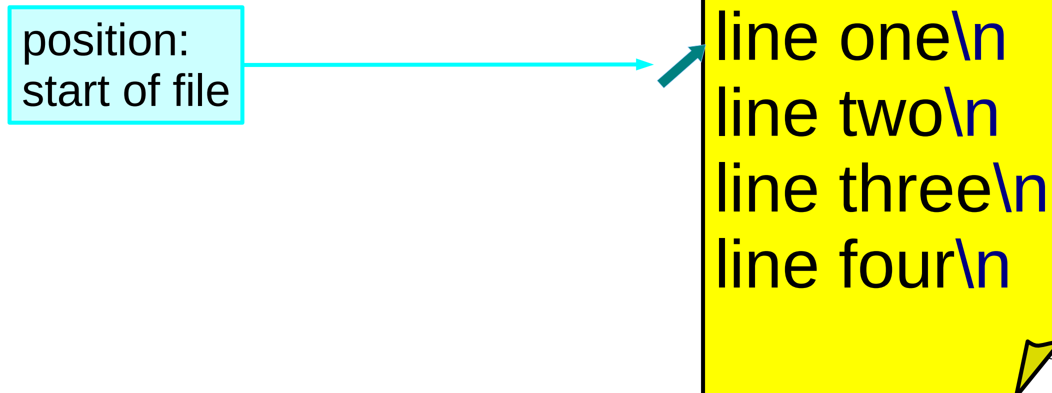
Given that we already know how to get the exception details when handling a specific type of exception, why might we want to use the **`exc_info()`** function?

Recall that we can have an exception handler that handles *all* types of exception. Inside such an exception handler, we might want to know what type of exception we were handling, as well as the details of the exception. In such circumstances the best way to get this information is with the **`exc_info()`** function.

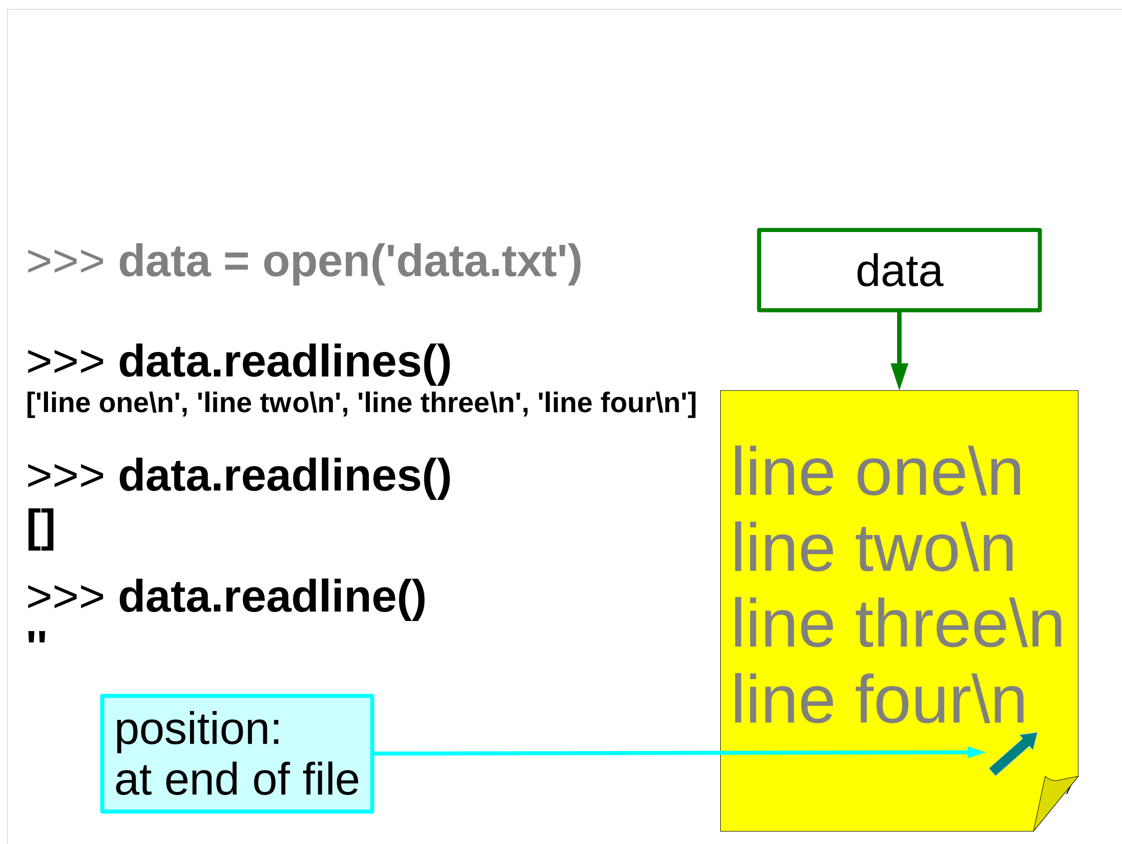
Moving around a file

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

data



As we know, when we **`open()`** a file, we start at the beginning of the file: the offset of our file object is set to the start of the file (unsurprisingly, as we will see in a moment, this position is offset ***`0`*** of the file).



We can move to the end of the file by reading all the data in it using the **`readlines()`** method. Once we are at the end, if we try to use the **`readlines()`** or **`readline()`** methods we don't get any more data, we just get an empty list or an empty string (respectively).

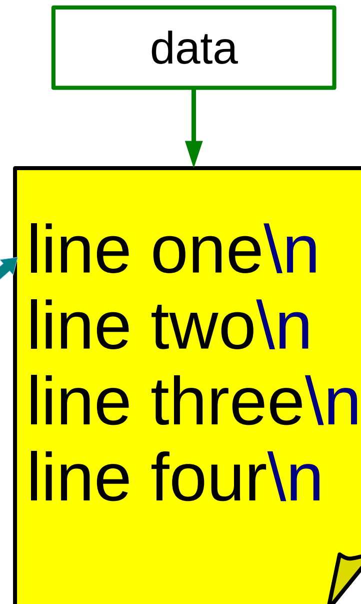
Suppose we want to read from somewhere else in the file, how can we do that? We could always close the file and then open it again, but that seems a bit silly – is there a better way?

Moving to the start of a file

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

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

>>> data.seek(0)
```

offset in file

position:
start of file



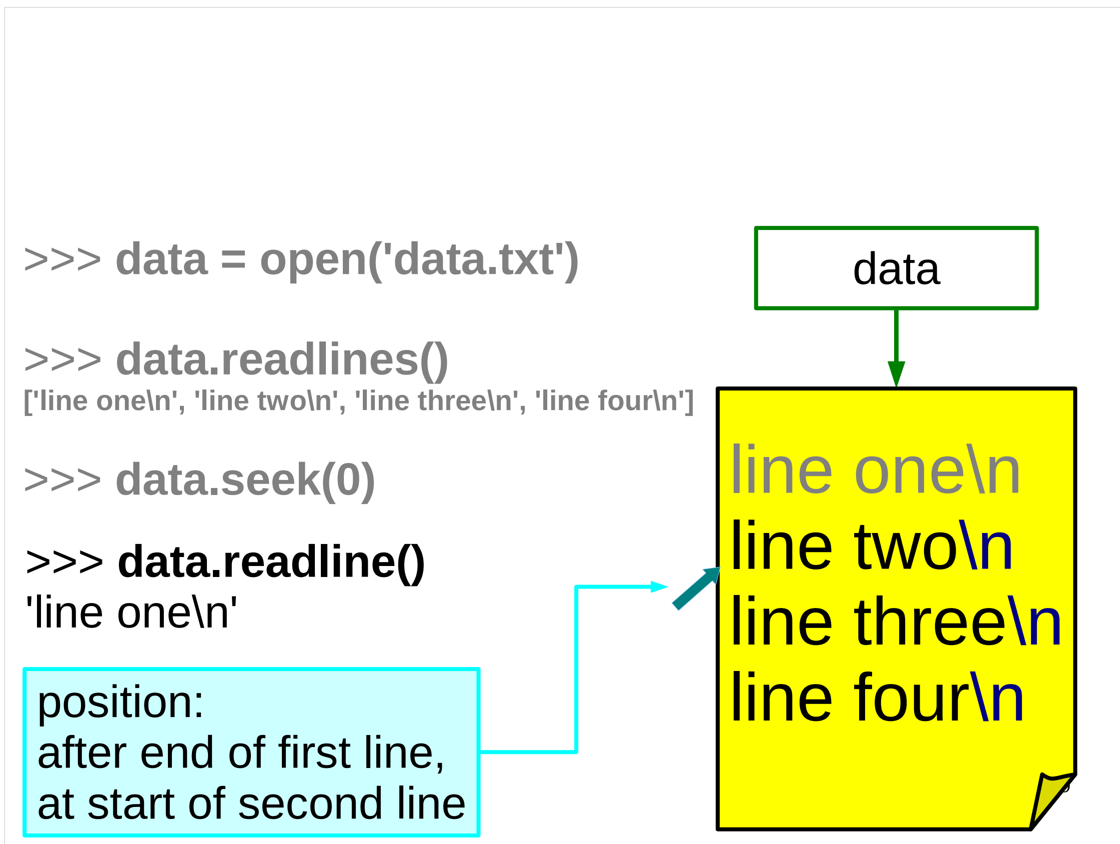
The better way is to use the **`seek()`** method. The **`seek()`** method is a method that moves the offset of the file object to the specified location in the file without reading (or writing) any data. You need to be very careful with this method, as it is very easy to accidentally move yourself to a random position in a file, which can then play havoc with subsequent read or write operations.

As we see here, the start of the file has an offset of 0, and so, to move to the start of a file we would call the **`seek()`** method with an argument of **`0`**, as shown above.

You can find out what the current offset of the **`file`** object is using the **`tell()`** method, as we will see shortly.

A very important point to note is that not all offsets you can give the `seek()` method are valid! This is one reason why the **`seek()`** method is rarely used except to move to the beginning or an end of a file (and even that is relatively uncommon). (In case you are curious, what offsets are valid depends on the type of file (whether it is text or binary) and the operating system that Python is running on. In this course we are only going to deal with text files (although we'll mention binary files briefly a little later), and we're not going to use the **`seek()`** method except to show you how to move to the start and end of a file.)

There are also certain sorts of **`file`** object that do not support the **`seek()`** method – however, if you are reading from a normal file that is stored on a file system somewhere, the **`seek()`** method should work.



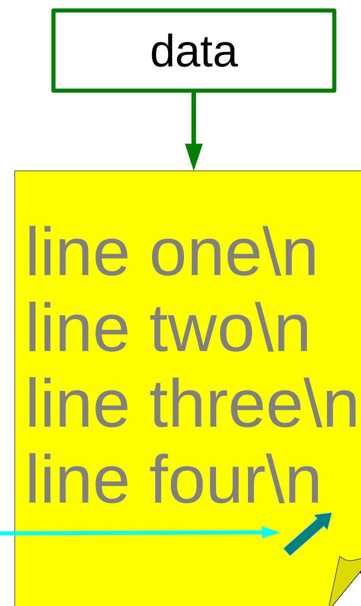
We can verify that we are indeed at the start of the file by trying to read a line from it. As we see, we indeed at the start of the file, and have now moved to the start of the second line of the file.

Moving to the end of a file

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

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

>>> data.seek(0, 2 )
```

specifies that offset is ***relative to the end of the file***

position:
at end of file

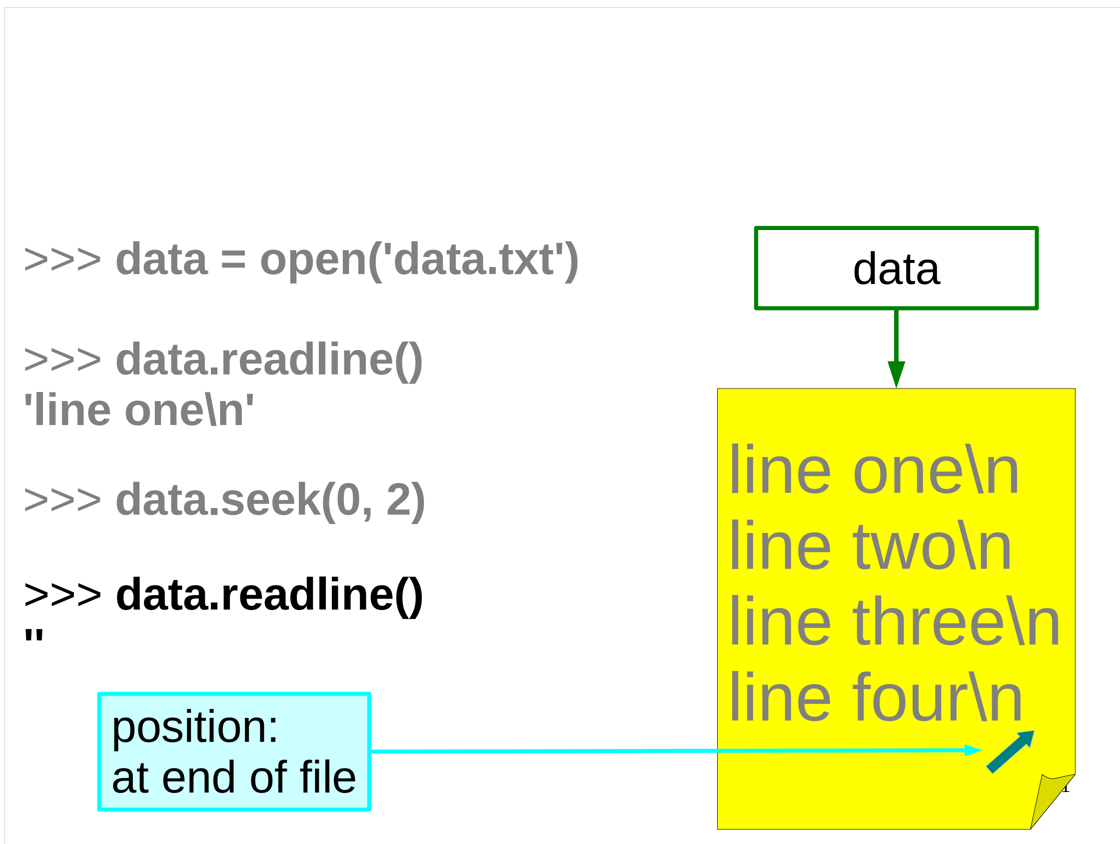


We can move to the end of the file using the **`seek()`** method as well. If we know exactly what value corresponded to the offset of the end of the file, we could instruct the **`seek()`** method to move to that offset, but that depends on us knowing exactly what the offset of the end of the file is.

So the **`seek()`** method can be instructed to move to an offset ***relative*** to the end of the file (or relative to the current position in the file). The syntax is as follows:

`seek(offset)` *or* **`seek(offset, 0)`**	move to the specified **`offset`** (*absolute* position, i.e. relative to the start of the file)
`seek(offset, 1)`	move to the specified **`offset`** **relative to the current position in the file**
`seek(offset, 2)`	move to the specified **`offset`** **relative to the end of the file**

So to move to the end of the file, we call the **`seek()`** method with the arguments **`0`**, **`2`**. **`0`** for the offset, and **`2`** to signify that this offset is relative to the end of the file.



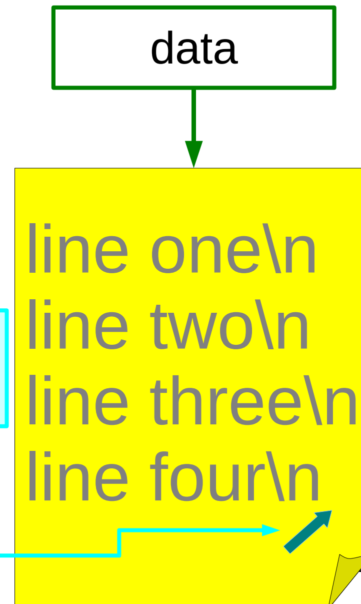
We can verify that we are at the end of the file by trying to use the **`readline()`** method (or the **`readlines()`** method, if you prefer). As we are at the end of the file, if we try to use the **`readline()`** or **`readlines()`** methods we don't get any more data, we just get an empty string or an empty list (respectively).

Finding your position in a file

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

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

>>> data.seek(0, 2)

>>> data.tell()
39L

>>> data.close()
```

current offset as a long integer

position: at end of file

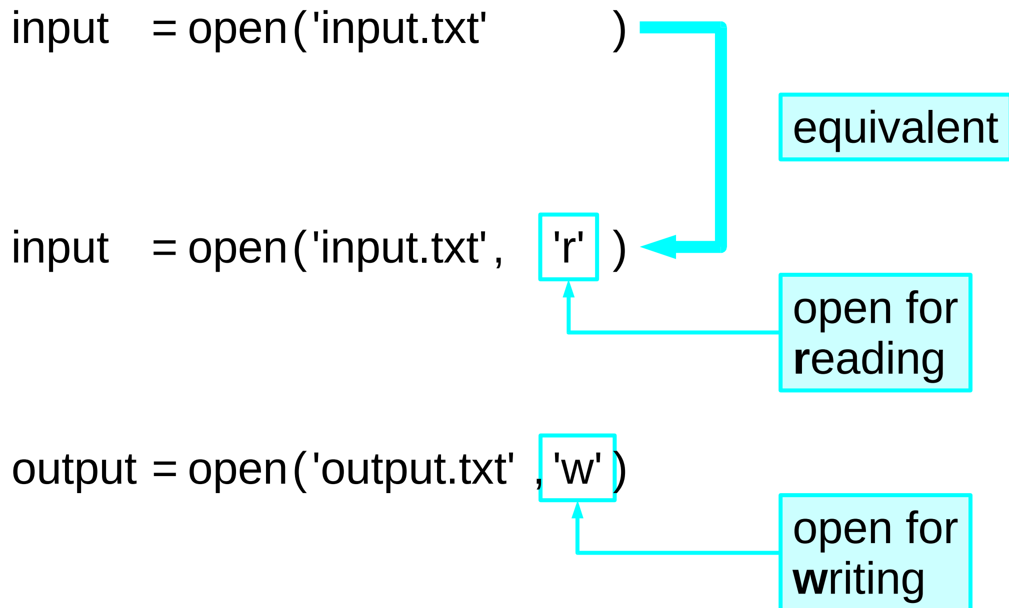


We can determine the value of the offset of a **`file`** object (i.e. our position within the file) using the **`tell()`** method. The offset is an integer, and because files can be very large, it is a *long integer* (in Python long integers can be of arbitrary size, limited only by the amount of memory the computer possesses), hence the ‘**`L`**’ that is printed after the ‘**`39`**’ above.

(Note that the **`tell()`** method only returns *valid* offsets, ***except*** when you have opened a Unix text file as a text file on the Windows platform, when it is **unreliable** – the solution to this is to open Unix text files as binary files on the Windows platform (we’ll see how to open a file in binary mode later). However, apart from in this pathological case, any offset returned by the **`tell()`** method can be used as the offset argument for the **`seek()`** method.)

We’ve finished playing around with this file now, so make sure you **`close()`** it. If you’re being good you get rid of the **`data`** variable as well:

```
>>> del data
```

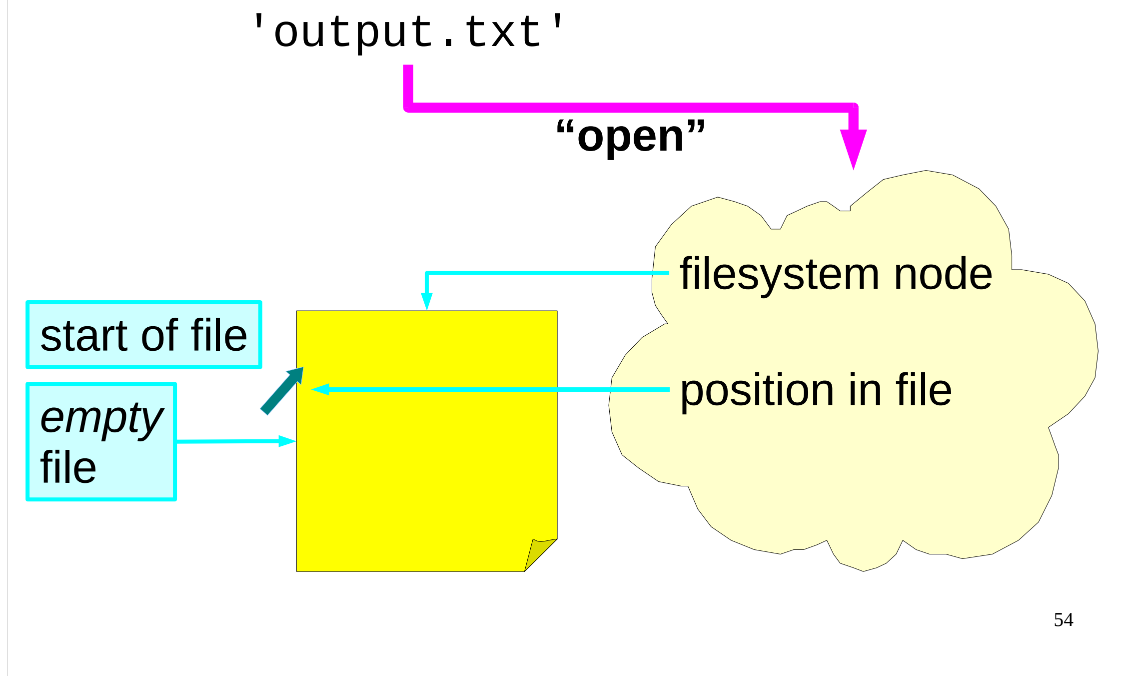


To date we have been only reading from files. What happens if we want to write to them?

The **`open()`** function we have been using actually takes more than one argument. The second argument specifies the *mode* in which we want to open the file. Amongst other things, the mode specified whether we want to read or write the file. If the mode is not specified, then the default is to open the file for reading only.

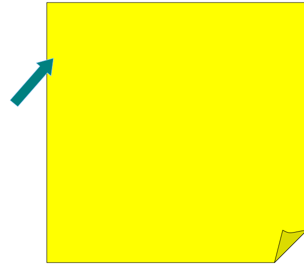
The explicit value you need to open a file for **r**eading is the single letter string **`'r'`**. That's the default value that the system uses. The value we need to use to open a file for **w**riting is **`'w'`**.

There are other modes apart from the above two simple ones, some of which we'll meet later.



As ever, a newly opened `file` has its position pointer (“offset”) pointing to the start of the file. This time, however, the file is empty. **If the file previously had any content then it would get completely replaced.**

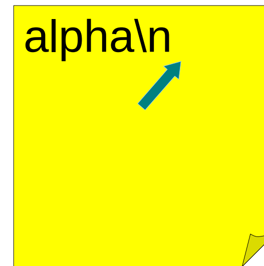
```
>>> output = open('output.txt','w')
```

55

Apart from the explicit second argument, the **`open()`** function is used exactly as we did before.

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

56

Now that we've opened our file ready to be written to we had better write something to it. There is no “**`writeline()`**” equivalent to **`readline()`**. What there is is a method “**`write()`**” which might be thought of as “writelump()”. It will write into the file whatever string it is given whether or not that happens to be a line.

When we are writing text files it tends to be used to write a line at a time, but this is not a requirement.

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

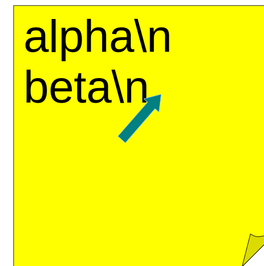
lump of data to be written

57

For example, we could write three characters in one **`write()`**…

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

remainder of the line



58

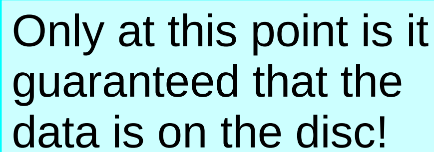
…and the last two characters (the new line counts as one character) in a second **`write()`**.

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


There is a writing equivalent of **`readlines()`** too: "**`writelines()`**". Again,the items in the list to be written do not need to be whole lines.

```
>>> 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.



60

Closing the file is particularly important with files opened for writing. As an optimisation, the operating system does not write data directly to disc because lots of small writes are very inefficient and this slows down the whole process. When a file is closed, however, any pending data is "flushed" to the file on disc. This makes it particularly important that files opened for writing are closed again once finished with.

It is only when a file is closed that the writes to it are committed to the file system.

Exercise

In **`utils.py`**, write a function called **`dict2file()`** that takes a dictionary and an ***optional*** filename as its arguments and writes the contents of the dictionary to the file:

1. Open a file for writing.
2. For each key in the dictionary:
   - 2a. Write the key to the file.
   - 2b. Write a tab (**`'\t'`**) to the file.
   - 2c. Write the value corresponding to the key, followed by a newline (**`'\n'`**), to the file.
3. Close the file.

61

So now let's try creating a function that writes to a file. You should create your function in the `utils.py` file in your course home directory. We have covered all the Python I/O you need to write this function (*hint:* you can use either the **`write()`** method or the **`writelines()`** method (or both if you're particularly creative)). And you should already know the basic Python you'll need (*hint:* recall that if you treat a dictionary like a list (say in a **`for`** loop) it behaves like a list of its *keys*).

Make sure you put in exception handling for any `IOError` exceptions that may be raised when writing to the file. Your function should also use a *default* filename (i.e. a filename to use if the user does not specify one when they call the function) of `dictionary.txt`.

Once you've finished writing this function (make sure you saved the file) you can test it out on the dictionary of atomic symbols and element names in the file `chemicals.txt` in your course home directory:

```
>>> import utils
>>> chemicals = utils.file2dict('chemicals.txt')
>>> utils.dict2file(chemicals)
```

Now exit the Python interpreter, and have a look at the file `dictionary.txt`:

```
$ more dictionary.txt
Ru      ruthenium
Re      rhenium
Ra      radium
Rb      rubidium
```

(For space reasons, only an excerpt of the `dictionary.txt` file is shown above.)

If – and only if – you really get stuck, have a peek at the answer on the page after next.

When you've finished make sure and take at least a 5 minute break (preferably at least a 10 minute break) – and that means a ***break*** from the computer, not checking your e-mail.

This page intentionally left blank

Deze bladzijde werd met opzet blanco gelaten.

このページは計画的にブランクを残ている

Ta strona jest celowo pusta.

Esta página ha sido expresamente dejada en blanco.

Эта страница нарочно оставлена пустой.

Denne side med vilje efterladt tom.

Paĝon intence vaka.

اين صفحه خالي است

An leathanach seo fágtha folamh in aon turas.

62

This page intentionally left blank: nothing to see here. If you’re stuck for an answer to the exercise, have a look at the next page.

```
def dict2file(dict, filename='dictionary.txt'):
    import sys
    data = None
    try:
        data = open(filename, 'w')
        for key in dict:
            output = "%s\t%s\n" % (key, dict[key])
            data.write(output)
        data.close()
    except IOError, error:
        (errno, errdetails) = error
        print "Problem with file %s: %s" % (filename, errdetails)
        print "Aborting!"
        if type(data) == file:
            data.close()
        sys.exit(1)
    return
```

utils.py

63

Your **`dict2file()`** function should look similar to the one above. If it doesn't, or if you had problems with the exercise, please let the course giver know.

Also, if there is anything in the above function that you don't understand please ask the course giver.

Note that in my version of the function above I only use the **`write()`** method once as I process each key, value pair from the dictionary. This makes my code fairly compact, but not as readable as it might be. It is perfectly acceptable to use the **`write()`** method multiple times, e.g.

```
data.write(key)
data.write('\t')
data.write(dict[key])
data.write('\n')
```

Note that one problem with the above sequence of Python code is that there is no guarantee that either the key or the value is a string and the **`write()`** method *only* accepts a string as an argument. This is why it is better to use the string formatting operator (**`%`**) as I do here:

```
output = "%s\t%s\n" % (key, dict[key])
```

and then use the **`write()`** method like this:

```
data.write(output)
```

The use of the string formatting operator guarantees that **`output`** will be a string (unless, of course, there is something wrong with **`key`** or **`dict[key]`**, say if **`dict`** is not actually a dictionary, in which case the function will fail anyway). An alternative would be to use **`str()`** function to convert the key and value to strings before giving them to the **`write()`** method.

Checking whether a file exists

```
>>> import os.path
>>> os.path.exists('chemicals.txt')
True
>>> os.path.exists('rubbish.txt')
False
>>>
```

64

Those of you who have ever accidentally overwritten a file may have spotted the glaring flaw with our otherwise well-behaved **`dict2file()`** function: it happily overwrites (or attempts to overwrite) whatever file it is given as a file name (or `dictionary.txt` if it is not given a file name).

Clearly, this is dangerous and we should fix this glaring bug. How do we do this? We need to check whether the file already exists before we try to write to it.

The **`exists()`** function – which lives in the **`os.path`** module – returns **`True`** if the specified file exists, **`False`** if it doesn't (or if the specified file is a broken *symbolic link*).

(A symbolic link (also known as a *symlink* or a *soft link*) is similar to a shortcut in the Microsoft Windows operating system (if you are familiar with those) – essentially, a symbolic link points to another file elsewhere on the system. When you try and access the contents of a symbolic link, you actually get the contents of the file to which that symbolic link points. If the file to which the symbolic link points does not exist, then the symbolic link is said to be *broken*. For a more detailed explanation of symbolic links see the following Wikipedia article:

http://en.wikipedia.org/wiki/Symbolic_link

)

```
def dict2file(dict, filename='dictionary.txt'):
    import sys, os.path

    if os.path.exists(filename):
        print "File %s exists." % filename
        print "Not overwriting it."
        return

    data = None

    Rest of function unchanged
```

utils.py

65

Modify the **`dict2file()`** function as shown above.

Remember to save the file after you've made your modifications.

Now the first thing our function does is to see whether the file name it has been given is of a file that already exists. If it does, it prints out a message to that effect, says that it is not overwriting the file, and simply **`return`**s without doing anything.

If we wanted to, we could make the function cause our program to **`exit()`**, but that seems an overly draconian response – it makes more sense to just let the user know that we haven't overwritten the file and let the program continue.

Of course it would be even better if the *program* (and not just the user) could tell whether the function had succeeded or not. To do that we need to make our function do something that a program calling it could easily check to see whether the function had succeeded or not.

```
def dict2file(dict, filename='dictionary.txt'):
    import sys, os.path

    if os.path.exists(filename):
        print "File %s exists." % filename
        print "Not overwriting it."
        return False

    data = None

    Rest of function unchanged except last line:

    return True
```

utils.py

66

Modify the **dict2file()** function as shown above.

Remember to save the file after you've made your modifications.

Now, if the passed file name is of a file that already exists, our function returns the Boolean value **False**, as well as printing out a message for the user. If the function succeeds, it returns the Boolean value **True**. A well-written program can now test whether the function succeeded or not and behave accordingly.

We can now try out our greatly improved function:

```
>>> import utils
>>> chemicals = utils.file2dict('chemicals.txt')
>>> writeout = utils.dict2file(chemicals,'dict1.txt')
>>> print writeout
True
>>> writeout = utils.dict2file(chemicals,'dict1.txt')
File dict1.txt exists.
Not overwriting it.
>>> print writeout
False
```

Renaming a file

```
>>> import os.path
>>> os.path.exists('data1.txt')
True

>>> import os
>>> os.rename('data1.txt', 'data2.txt')
```

rename() renames files.
It lives in the **os** module.

Under Unix/Linux if the new name is a file that already exists, then that file is **deleted**, i.e. **rename()** behaves like the Unix **mv** command.

```
>>> import os.path
>>> os.path.exists('data1.txt')
False
```

67

So what do you do if there already exists a file with the name you want to use? You could use a different name for your file, or you could rename the existing file to avoid overwriting it.

You can rename a file (or directory) using the **rename()** function that lives in the **os** module.

IMPORTANT: If the new name of the file or directory you want to rename is a file that already exists, then under Unix/Linux that file will be ***deleted*** and then the renaming will be done. If the new name is the name of an existing directory, then an **OSError** will be raised. Under Windows, if the new name is an existing file *or* directory then an **OSError** will be raised.

Under Unix/Linux **rename()** basically behaves like Unix's **mv** command (which means that on Unix/Linux you can use **rename()** to **m**o**v**e files around, not just rename them in the same directory). Note though, that on some versions of Unix **rename()** will fail if the new "name" is on a different file system to the original file, i.e. if, instead of just renaming the file, you use the **rename()** function to *move* the file to a ***different*** file system.

Appending output to a file

```
$ cat output.txt
alpha
beta
gamma
delta

>>> output = open('output.txt', 'a')
>>> output.write('epsilon\n')
>>> output.close()
>>> del output
>>>

$ cat output.txt
alpha
beta
gamma
delta
epsilon
```

open for **a**ppending

68

So we know how to read from files, and how to write to a file. However, if we write to a file that already exists then we will *overwrite* it. Now suppose we don't want to overwrite the file, but just add some more data to the end of it – how can we do this?

Another value for the mode of the **`open()`** function that we can use is **`'a'`**. This means that we should open the file for writing, but we want to ***a**ppend* our writes to it, i.e. we don't want to destroy the data already in the file, and instead we want to add whatever we write to the end of the file.

Note that on some systems (most commonly some (but not all) versions of Unix) if you open a file in append mode then ***all*** writes to the file are ***always** appended* to the end of the file, regardless of the position you may have moved to in the file using **`seek()`**. This means that if you use **`seek()`** on a file opened in append mode, you should not rely on it working in the way you might hope.

(Note that in the above slide the **`cat`** command is given at the Unix/Linux prompt. We then issue some Python commands in the Python interpreter. Finally, we quit the interpreter and issue the **`cat`** command (at the Unix/Linux prompt) again to see what effect our Python commands have had.)

Checking whether a file is open

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

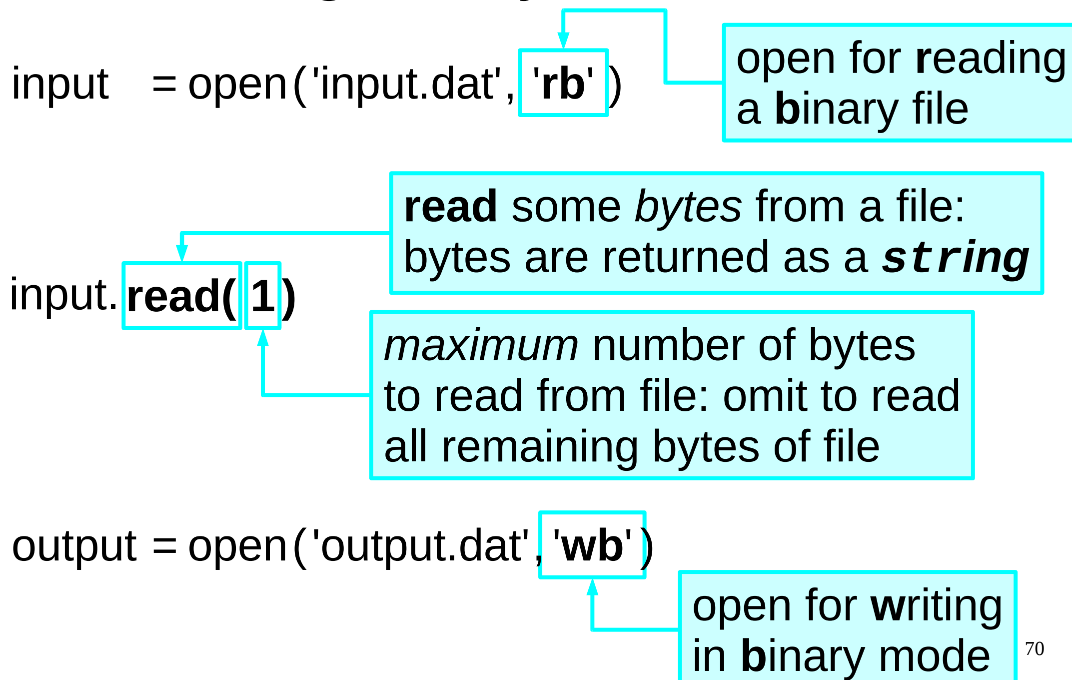


```
>>> data.closed
False

>>> data.close()
>>> data.closed
True
```

69

Some of you may be wondering how we can tell whether a file is open or closed.

`file` objects have an attribute called **`closed`**, which is set to **`True`** if the file is closed and set to **`False`** if the file is open. (Attributes are "built in" variables that objects can have.) If we need to tell whether a file is open or closed, we can do so by examining the **`closed`** attribute of the corresponding **`file`** object to see whether it is **`True`** or **`False`**.



To date we have only been accessing text files. What about binary files?

The first thing to understand is that many operating systems (such as Unix/Linux) **do not** treat text and binary files differently: so on these platforms we can carry on much as before. Some platforms (such as Windows) do treat text and binary files differently however, and on those platforms it is **very important** to know what sort of file you are dealing with.

Recall that the **`open()`** function for opening a file takes more than one argument, and the second argument specifies the mode in which we want to open the file. The mode tells Python whether to treat the file as a text file or a binary file. If we don't explicitly say it is a binary file, then the default is to treat the file as a text file.

The explicit value you need to use to open a file in binary mode is the single letter string **`'b'`** which you add to the end of the file mode. So for **r**eading a **b**inary file, use **`'rb'`** and for **w**riting in **b**inary mode, use **`'wb'`** (and to **a**ppend to a **b**inary file you would use **`'ab'`**).

If you are working with binary files, then the concept of lines probably no longer applies, so we need another method to read data from such files. That method is **`read()`**. When you use the **`read()`** method you can specify the *maximum* number of bytes you wish to read – **`read()`** will return that number of bytes or fewer if it comes to the end of the file. If you don't specify the maximum number of bytes then **`read()`** will read all the bytes in the file from the current offset in the file to the end of the file. Just as with **`readline()`** or **`readlines()`**, using **`read()`** advances the position in the file (the offset). And also as with **`readline()`** and **`readlines()`**, **`read()`** returns the bytes it reads as a ***string***. If your binary file does not contain strings, it is up to you to convert the data in the string returned by **`read()`** to the correct format.

We already have a method (**`write()`**) that allows us to write arbitrary length strings, rather than whole lines, to a file, so we don't need a new method for writing to a binary file.

Reading files from other OSes

```
>>> data = open('Win-text.txt')
>>> data.readline()
'line one\r\n'
>>> data.close()

>>> data = open('Win-text.txt', 'rU')
>>> data.readline()
'line one\n'
>>> data.close()
```

open as a text file in ***U**niversal newline* mode (for **r**eading)

71

Those of you who frequently work on both Unix/Linux and Windows platforms, or old Macintosh (pre-MacOS X) and Windows platforms, etc will probably have come across the annoying fact that text files are handled differently on all these platforms.

On Unix/Linux systems, each line of text in a text file is terminated with a single line feed character (‘**`\n`**’).

On Windows systems, each line of text in a text file is terminated with a single carriage return character (‘**`\r`**’) followed by a single line feed character (‘**`\n`**’), i.e. the *two* character combination ‘**`\r\n`**’.

On old Macintosh systems (up to MacOS 9), each line of text in a text file is terminated with a single carriage return character (‘**`\r`**’).

As long as you are always working on the same platform, or always working with binary files, this is irrelevant and you are unlikely to care. However, when working with text files on different platforms, it can be a problem, since some applications will not recognise a file as text if it doesn’t have the correct *end-of-line* (EOL) character combination for that platform, or else they may get confused.

Python has a special mode – *universal newline support* – for handling such files. Instead of just opening such files as we have been doing up to now, we use the special mode **`'rU'`** (or just **`'U'`**) as shown above. In this mode, Python treats any of the EOL combinations as a single newline character, which it represents as ‘**`\n`**’. (Note that the copy of Python you are running needs to have been compiled with universal newline support enabled – this is the default, so normally you shouldn’t need to worry about whether or not universal newline support is enabled on your particular copy of Python.) This mode is only for *reading* text files – when Python writes to text files it is uses the EOL character combination for the platform on which it is running whenever you specify the ‘**`\n`**’ character.

The file `Win-text.txt` in your course home directory is a Windows text file version of the file `data.txt`. As you can see, if we open it normally it doesn’t look right: there’s an extra ‘**`\r`**’ character hanging around near the end of each line. If we open it using Python’s universal newline support mode then it behaves “normally”.

If you want to know more about line endings for text files on different platforms, see the following Wikipedia article:

http://en.wikipedia.org/wiki/Newline

Accessing files

1. Direct access to files
2. Structured files: **csv** module

72

We've already met the limitations of the **split()** function for handling input, and we've already come across two structured files that we can't read because they aren't in the simple "whitespace" delimited format we're used to. How can we handle such files?

Obviously, we could write our own functions for making sense of (*parsing*) every single file format we came across, but there is an easier way, at least for a large class of text files. We use one of the standard Python modules (introduced in Python 2.3): the **csv** module.

This module allows us to read and write so-called CSV (***c***omma ***s***eparated ***v***alue) files. These are files where each line has the same structure, consisting of a number of values (called *fields*) separated by some specified character (or sequence of characters), typically a comma (**,**). The character (or characters) that separates the fields is called a *delimiter*.

This module also defines its own special sort of exception that is used when something goes wrong with the functions and methods it provides. This exception is "**Error**", but as it is defined in the **csv** module, you would normally refer to it by prefixing it with "**csv.**", e.g.

```
try:
    …
except csv.Error:
    print "Can't read CSV file!"
```

CSV files

Fields containing data

1.0, 2.5, 3.0, -6.8, 23.4
5.6, 2.8, 9.3, -4.6, 9.8
-3.9, 25.0, 1.23, 5.6, 7.8
2.9, 5.2, 6.7, 2.4, 5.6

Fields are separated by ***delimiters***.
A comma (**,**) is often used as a delimiter.

Sometimes spaces may follow delimiter between fields, or may be used as "padding" to make fields a particular size (*width*).

73

Although CSV stands for "*c*omma *s*eparated *v*alues", as far as Python is concerned CSV files are any type of text file which have a particular structure.

For Python, the CSV file will consist of a number of lines, each of which will have a number of items of data (called *fields*), separated from each other by a particular character known as a *delimiter*. (A comma (,) is often used as a delimiter, hence the name "comma separated values".)

There may also be spaces after (or before) the delimiter to separate the fields, or the fields may be "padded" out with spaces to make them all have the same number of characters in them. (The number of characters in a field is called the field's *width*.)

Some programs that work with CSV files will only accept a comma as a delimiter, or else will only accept a comma, a space or a tab as delimiters. Python, however, will accept any single character as a delimiter.

Also, some programs that work with CSV files require each line of the file to have the same number of fields (although some of the fields may be empty). Python doesn't care how many fields there are on each line, provided that the same delimiter is used throughout the file.

Quoting in CSV files

```
"Fred Smith", red, 56.9
"Joe Bloggs", blue, 27.8
"Jill East", brown, 28.9
```

Data with spaces may be *quoted* (surrounded by quotation marks).

```
"Smith, Fred", red, 56.9
"Bloggs, Joe", blue, 27.8
"East, Jill", brown, 28.9
```

Data containing special characters, e.g. the delimiter, is also quoted.

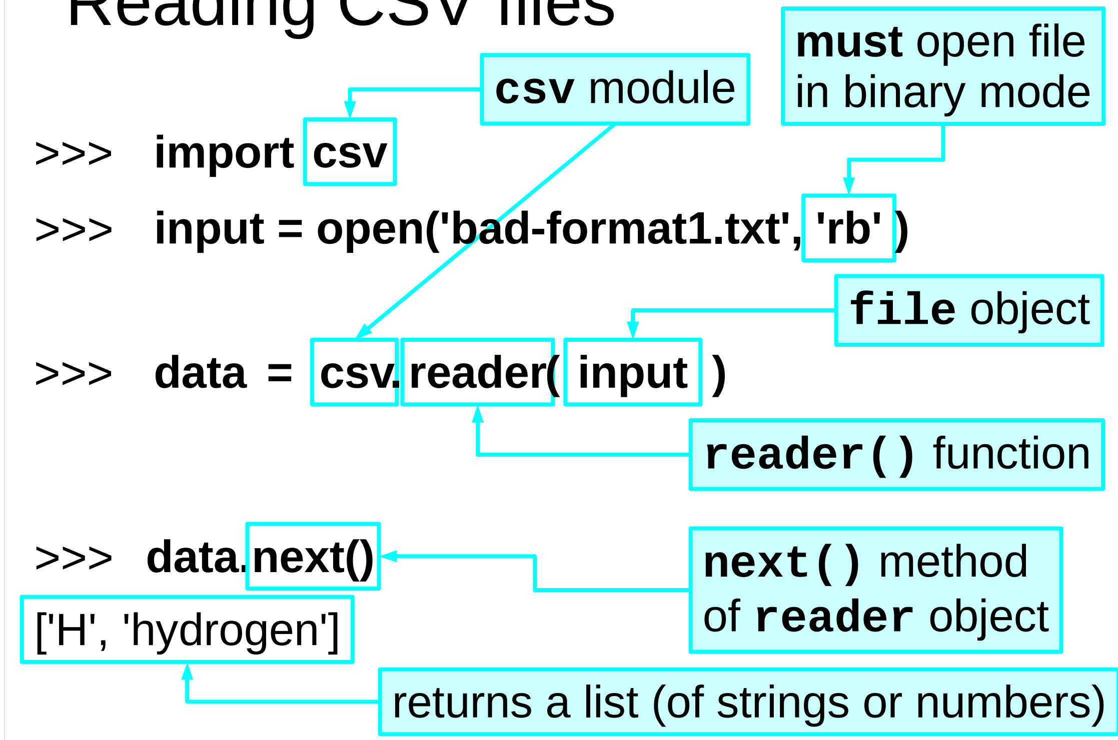
```
"Fred Smith", "red", "56.9"
"Joe Bloggs", "blue", "27.8"
"Jill East", "brown", "28.9"
```

We can even quote *all* data (or all *text* data) if we wish.

In CSV files, data with spaces in it is often *quoted* (surrounded by quotation marks) to make clear that it is one single item of data and should be treated as such. In Python this is not necessary unless you are using a space as your delimiter, but you will often find that programs that produce CSV files automatically quote data with spaces in it.

If your data contains special characters, such as the delimiter or a new line ('**\n**') character, then you will need to quote that data or Python will get confused when it reads the CSV file.

If you want, you can quote all the data in the file, or all the text data. Python doesn't mind if you quote data even when it is not strictly necessary.



Make sure you close the file (**`data.close()`**) after you've finished reading from it!

You read a CSV file using the **`reader()`** function in the **`csv`** module. This function requires a **`file`** object as input. **The `file` object must be opened for *reading* in *binary* mode.**

The **`reader()`** function returns a special sort of object, called a **`reader`** object. The **`reader`** object has a **`next()`** method which reads the next line of the CSV file and returns it as a list of strings or numbers. The first field on the line is the first item in the list, the second field is the second item in the list, and so on.

It is ***your*** responsibility to delete the **`reader`** object, close the CSV file and delete its **`file`** object when you are finished with the file.

The **`reader()`** function can take a number of optional formatting parameters that specify what sort of CSV file it is reading. See the **`csv`** module's documentation for a list of these parameters and their default values:

http://docs.python.org/library/csv.html#csv-fmt-params

You may also find the examples given in the module's documentation useful:

http://docs.python.org/library/csv.html#csv-examples

Treat a **reader** object like a list…

…it behaves like a list of the *lines of the CSV file*, where each line is itself a list (of fields of the CSV file).

```
import csv
import utils

input = open('bad-format1.txt','rb')
data = csv.reader(input)

chemicals = {}

for line in data:
	[ key, value ] = line
	chemicals[key] = value
del key, value, line, data
input.close()
del input

utils.print_dict(chemicals)
```

csv1.py

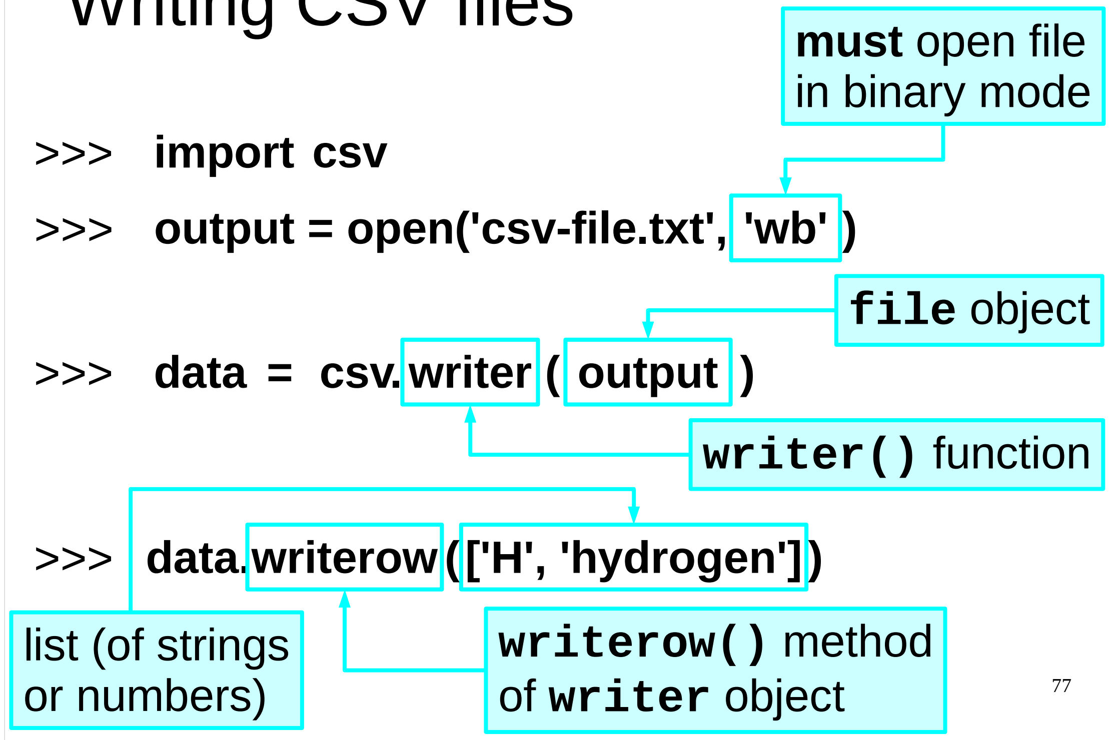
76

…so here is a typical example of using the **csv** module to read a CSV file. This script is in the file `csv1.py` in your course home directory.

Recall that in Python very often if you treat an object like a list, it will behave like a list of something, e.g. a dictionary will behave like a list of its keys. If we treat a **reader** object like a list, it behaves like a list of the lines of the CSV file, where each of those lines is itself a list (of strings or numbers). (Of course, just as when we treat a **file** object like a list we move our position in the file forward until we get to the end of the file, the same thing happens when we treat a **reader** object like a list. This means that we can only really treat a **reader** object like a list once, unless we then use the **seek()** method of the underlying **file** object to reset our position in the file.)

The **print_dict()** function is not a standard Python function. It is a function that prints out the keys and values of a dictionary that we wrote in the "Python: Introduction for Absolute Beginners" and the "Python: Introduction for Programmers" courses. For this course the function has been put in the **utils** module in your course home directories.

Finally, note that the bits of Python that read the CSV file and set up the dictionary would normally be hived off as a separate function in the script – that way we wouldn't have to worry about deleting the loop variable and all the other temporary variables. I haven't bothered with structuring this script that way as its purpose here is just to demonstrate the simple use of **reader** objects. Similarly, I haven't put in any exception handling. If this script was actually meant to be used for any serious task, it would be better structured and would have some exception handling.



Make **sure** you close the file (**`data.close()`**) after you've finished writing to it!

It is only when a file is closed that the writes to it are committed to the file system.

You write a CSV file using the **`writer()`** function in the **`csv`** module. This function requires a **`file`** object as input. **The `file` object must be opened for *writing* in *binary* mode.**

The **`writer()`** function returns a special sort of object, called a **`writer`** object. The **`writer`** object has a **`writerow()`** method which writes takes a list of *strings* or *numbers* and writes them out as a complete line of the CSV file, formatted appropriately. The first item in the list will be the first field of the line that is written to the CSV file, the second item will be the second field of the line, and so on. Python will handle quoting any data that needs to be quoted, you do not have to do it yourself.

Note that it is ***your*** responsibility to delete the **`writer`** object, close the CSV file and delete its **`file`** object when you are finished with the file. It is ***only*** when you ***close*** the **`file`** object that the data you've written to the CSV file will be committed to the file system.

The **`writer()`** function can take a number of optional formatting parameters that specify what sort of CSV file it is reading (these are the same as the optional formatting parameters for the **`reader()`** function). See the **`csv`** module's documentation for a list of these parameters and their default values:

http://docs.python.org/library/csv.html#csv-fmt-params

You may also find the examples given in the module's documentation useful:

http://docs.python.org/library/csv.html#csv-examples

```
import csv

symbol_to_properties = {…}

output = open('chem_props.txt','wb')

data = csv.writer(output)

for symbol in symbol_to_properties:
    (name, anum, boil) = symbol_to_properties[symbol]
    data.writerow([symbol, name, anum, boil])
del symbol, name, anum, boil
del data

output.close()
del output
```

csv2.py

78

Above is an example of using the **`csv`** module to write to a CSV file. This example is in the file `csv2.py` in your course home directories. Again, for simplicity I have not included any exception handling in this script, but if it was intended for serious use then it would include some exception handling.

We can try out this script if we wish:

```
$ python csv2.py
$ more chem_props.txt
Ru,ruthenium,44,4423.0
Re,rhenium,75,5900.0
Ra,radium,88,2010.0
Rb,rubidium,37,961.0
Rn,radon,86,211.3
Rh,rhodium,45,3968.0
```

(For space reasons, only an excerpt of the `chem_props.txt` file is shown above.)

You should understand all the Python in the script above. If there is anything you do not understand, please ask the course giver now.

Formatting options for CSV files

1.0,2.5,3.0,-6.8,23.4
5.6, 2.8, 9.3, -4.6, 9.8
-3.9, 25.0, 1.23, 5.6, 7.8
2.9, 5.2, 6.7, 2.4, 5.6

delimiter = ','
delimiter is a string that specifies the character being used as the delimiter.
Default value: **','**

skipinitialspace = True
will ignore any whitespace immediately after the delimiter. Default value: **False**

79

We'll look at some of the most common of these optional formatting parameters now.

Note that these parameters are *optional*, so you don't have to specify them: if you don't specify a parameter then Python will use that parameter's default value. When you call the **reader()** or **writer()** functions you can specify as many, or as few, of these optional parameters as you wish. If you specify any of these optional parameters, they must come *after* the **file** object you give to the **reader()** or **writer()** function, separated by a comma (**,**) as is usual for multiple arguments for a function in Python..

The **delimiter** parameter specifies the delimiter that separates the fields in the CSV file. It is a one-character string and defaults to the comma (**','**).

The **skipinitialspace** parameter is a Boolean that controls whether any spaces immediately following a delimiter should be ignored or considered part of the data in the field. When set to **True** any whitespace immediately following the delimiter is ignored, when set to **False** any whitespace is considered part of the data in the field. Its default value is **False**.

These optional parameters are specified as named (or keyword) arguments that you give when you call the **reader()** or **writer()** functions, e.g. if **data** is a **file** object for a CSV file that has been opened for writing, and the CSV file uses the tab character ('**\t**') as its delimiter, you would call the **writer()** function like this:

```
writer(data, delimiter='\t')
```

Similarly, if you had a **file** object for a CSV file that was opened for reading in **data**, and that CSV file used a space as its delimiter, and you wanted to ignore any extra spaces between fields, you would call the **reader()** function like this:

```
reader(data, delimiter=' ', skipinitialspace=True)
```

Formatting options for CSV files

"Fred Smith", red, 56.9
"Joe Bloggs", blue, 27.8
"Jill East", brown, 28.9

`quotechar = '"'`
`quotechar` is a string that specifies the character to be used for quoting.
Default value: **`'"'`**

The **`quoting`** parameter controls when things should be quoted.
The default is to only quote fields that contain special characters (the delimiter, etc).

80

The **`quotechar`** parameter specifies the character to be used for quoting. It is a one-character string and defaults to the double quote character (**`'"'`**).

It is another optional parameter that you can specify when you call the **`reader()`** or **`writer()`** function. For example, if **`data`** is a **`file`** object for a CSV file that has been opened for writing, and you want to use the single quote (**`'`**) character for quoting, you would call the **`writer()`** function like this:

```
writer(data, quotechar="'")
```

When things are quoted is controlled by the optional **`quoting`** parameter. The default value for this tells Python only to quote a field when the data in that field contains a special character, such as the delimiter. We'll look at the values the **`quoting`** parameter can take next.

The slide at the top of the page is shown here as text:

Controlling quoting in CSV files

"Smith, Fred", red, 56.9
"Bloggs, Joe", blue, 27.8
"East, Jill", brown, 28.9

default behaviour

`quoting = csv.QUOTE_MINIMAL`

"Fred Smith", "red", 56.9
"Joe Bloggs", "blue", 27.8
"Jill East", "brown", 28.9

`quoting = csv.QUOTE_NONNUMERIC`

"Fred Smith", "red", "56.9"
"Joe Bloggs", "blue", "27.8"
"Jill East", "brown", "28.9"

`quoting = csv.QUOTE_ALL`

81

The **`quoting`** parameter can take one of four values. These values are special constants that are defined in the **`csv`** module. This means that to use one of these special values we have to prefix it by "**`csv.`**" so that Python knows where to find it. The four values (and their meanings) are:

`QUOTE_ALL` Makes **`writer`** objects quote all fields.

`QUOTE_MINIMAL` Makes **`writer`** objects only quote those fields which contain special characters in the data, such as the delimiter.

`QUOTE_NONNUMERIC` Makes **`writer`** objects quote all non-numeric fields (i.e. fields whose data cannot be represented as a number). Makes **`reader`** objects convert all non-quoted fields to **`float`**s.

`QUOTE_NONE` Makes **`writer`** objects never quote fields. If the field contains a special character, then the **`writer`** object will try to *escape* it. For further details on this behaviour see the **`QUOTE_NONE`** entry in the **`csv`** module's documentation:

http://docs.python.org/library/csv.html#csv-contents

`QUOTE_NONE` makes **`reader`** objects do no special processing of quote characters, i.e. any quote characters will be treated as part of the data in the field.

For example, if **`data`** is a **`file`** object for a CSV file that has been opened for writing, and you want all text fields (i.e. all non-numeric fields) to be quoted, you would call the **`writer()`** function like this:

```
writer(data, quoting=csv.QUOTE_NONNUMERIC)
```

Any questions?

82

If there are any questions about what I have said today I'll (try to) answer them now. There will be another opportunity to ask questions at the start of the next day.