Quickstart

You should have two files with this resource:

- **SnapFiles.py**
- **SnapFiles.xml**

Using a Python 3 interpreter run the first file.

Start Snap! from http://snap.berkeley.edu/run.

Import the second file or drag and drop it into the Snap! main window.

Click on the 'green flag' icon to run a series of tests: if these finish normally (without the blocks being outlined in red) then Snap! is communicating with the Python program successfully. If you look in your Documents folder you should now see a directory/folder called **SnapFiles**, within that one called **SFS** and within that a number of files: these can all be deleted.

You can now delete the **FileTesting** sprite and just use the blocks in your own projects. A future resource contains all the file handling blocks with a simple SQL system built on top of it that may give you some ideas.

Now read on!

Snap! file system

Since Snap! (http://snap.berkeley.edu) is written in JavaScript it cannot directly access the world outside the browser except in very limited ways, and so the Snap! developers use an intermediary between Snap! and devices, e.g.,

- Orbotix Sphero
- Lego NXT
- Nintendo Wiimote
- Finch and Hummingbird robots
- Parallax S2
- LEAP Motion
- Speech synthesis
- Arduino
- Fischertechnik ROBOTICS TXT
- Snap! for Raspberry Pi (this runs on a Raspberry Pi and allows access to the GPIO pins)

The way these all work is that a program (usually written in Python) runs and acts like a web server, on a port that isn't going to conflict with other servers.

Using the **`[http:// []]`** block in the **Sensing** category your blocks connect to the web server and send details, in a standard URI format, either requesting information to be returned as the result of the block or requesting actions to be carried out.

Since the web server does not have the restrictions that a JavaScript program running in a browser does it can communicate with external devices, or (as in this case) write to and read from files.

The web server does not have to be running on the same computer as the Snap! programmer is using, so that, for example, a RaspberryPi can be used as a Snap! File Server for an entire classroom (see **Network Use** below).

Handling files

In what follows we'll refer to the web server as **SFS** (for **S**nap! **F**ile **S**erver).

The Python program provided (**`SnapFiles.py`**) implements **SFS** and allows Snap! blocks to

- create files
- write to them
- read from them
- copy, rename and delete them

SFS (in its current form) will only work with files in the **SnapFiles** directory/folder in the **Documents** folder of the current user, i.e., if user **john** is logged on then (but see **Network Use** below).

C:\Users\john\Documents\SnapFiles	on Windows machines (Windows 7 on)
/home/john/Documents/SnapFiles	on Raspberry Pi
/Users/john/Documents/SnapFiles	on Mac OS X

The **`[http:// []]`** block is a low-level way to communicate with **SFS** and higher level blocks are provided in **SnapFiles.xml**. These are[i]:

File manipulation

```
[close all files]
[close file called []]
[copy file called [] to []]
[delete file called []]
[rename file called [] to []]
[truncate file called []]
```

Reading data from files

```
(contents of file called [])
(lines of file called [])
(next () characters from file called [] as [])
(next line of file called [])
```

Writing data to files

```
[add line [] to file called []]
[write [] to file called []]
```

Information about files

```
<at end of file called []>
<file called [] exists>
<position in file called []>
```

Positioning for reading or writing within a file

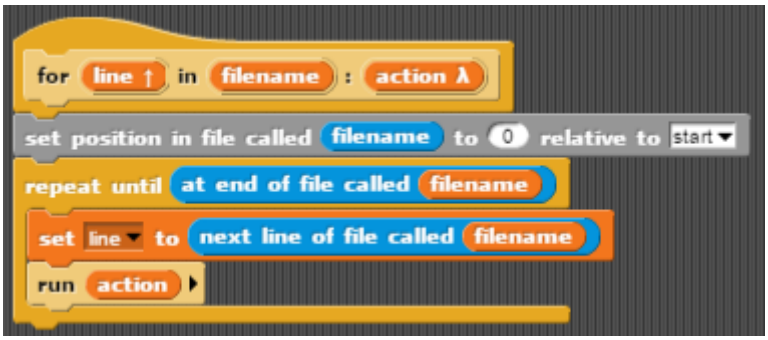
```
[set position in file called [] to () relative to []]
```

Information about the file server program

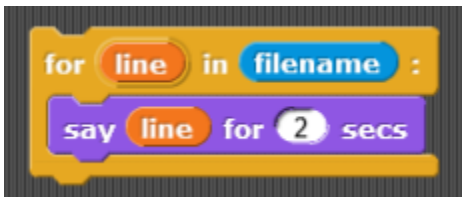
(SFS)
(SFS [])
<SFS is running>

Top level blocks

Even higher level blocks can be created, often using the functional programming facilities of Snap!, e.g.,



defines a block that implements a Python-like `[for [] in [] : []]` block, which iterates over all the lines in a file and performs an action on it, e.g.,



Categories

The blocks are split over two categories, **Sensing** and **Variables**. Anything that is a reporter block and returns information to the user's programs is in the **Sensing** category and is coloured blue. All the rest are in the **Variables** category and are coloured grey.

Error handling

Where it makes sense for it to do so **SFS** returns a status, either the string **OK** or a string with **ERROR:** followed by a short message describing the problem. Blocks that detect this call an error block which displays a message, outlines the block in error in red, and then stops all scripts.

If this simple behaviour is not what you want or need you can edit the behaviour in the block (I suggest importing the *Catch errors in a script* library that's provided as part of the standard Snap! system).

Working with files

When working with files using **SFS** you don't have to worry about opening a file, assigning the open file to a variable, and then using this handle/file ID/file variable. You always refer to a file by using its name[ii] (if this name includes a path, e.g., **N:\Scratch Projects\infections.log** it is ignored). If you are referring to the files more than once therefore it's best to create variables for the filenames then use these variables in file blocks that need a file name. If you need to change the name of a file then you only have one place to change it[iii]. **SFS** doesn't care about extensions, every file is opened in (Python) mode **rb+**.

The blocks

File manipulation

`[close all files]`

Close all the files that are in the internal file cache and remove them from it. It's always a good idea to close all files before starting and when finishing

`[close file called [name]]`

Close the file **[name]** and remove the reference to it from the internal file cache. Any more accesses to it will reopen it and position at the start of the file. It's always a good idea to close a file after you've finished with it since otherwise the behaviour may not be what is expected, particularly if more than one process/task is accessing the file[iv]

`[copy file called [fromName] to [toName]]`

Copy the file **[fromName]** (it will be closed before being copied) to the file **[toName]**: this block can cause an error

`[delete file called [name]]`

Delete the file **[name]** (it will be closed before being deleted): this block can cause an error

`[rename file called [oldName] to [newName]]`

Rename the file (it will be closed before being renamed): this block can cause an error

Reading data from files

`(contents of file called [name])`

The entire contents of the file **[name]** as one long string of characters, separated by line feed characters (ASCII LF), whatever the operating system your Snap! project is running on and whatever operating system SFS is running on; in this way the effect of a file operation is completely platform independent[v]

`(lines of file called [name])`

A list which contains all the lines in the file **without** the line terminator

(next <integer> characters from file called [name] as <type>)

The next **<integer>** characters from the file **[name]** starting at the current position and converted into the specified **<type>**, which can be **number**, **string**, **boolean** (for Boolean if the first character of those read is **t**, **T**, **y**, or **Y** then **true** is returned, otherwise **false**)

(next line of file called [name])

The next line in the file **[name]**, consisting of the characters from the current position to just before the line terminator. The way files are handled means that the current position for reading is remembered from block to block

Writing data to files

[add line <something> to file called [name]]

The file **[name]** is created (if it doesn't exist), then opened (if it isn't already open), the position for reading/writing is set to the end of the file (i.e., beyond the last character), and the string representation of **<something>** added and terminated by the appropriate line ending (ASCII CR LF (character 13, character 10) for Windows, ASCII LF (character 10) for Mac/Linux)[vi].

This block allows for text file processing, although behind the scenes everything is done using Python binary files

[write <something> to file called []]

<something> is converted to a string (if it isn't already a string) and output to the file **[name]** at the current position **without** a line terminator. This lets you use **SFS** to write binary format or fixed width field files.

A word of warning: be careful mixing **[write ...]** and **(next <integer> ...)** blocks with **(contents of ...)** and **(next line ...)** blocks (binary format write and text format read)

Information about files

<at end of file called [name]>

This predicate is **true** if the position for reading/writing is at the end of the file **[name]**, where no more characters can be read but writing will not overwrite existing contents

<file called [name] exists>

This predicate is true if the file **[name]** exists in the **SnapFiles** directory

<position in file called [name]>

Returns the current position for reading/writing the file. After a **[set position in file called [name] to (0) relative to [start]]** this will be 0, after a **[set position in file called [name] to (0) relative to [end]]** this will be one more than the number of bytes in the file, and by using **[set position in file called [name] to (0) relative to [current]]** you can move the position relative to the current position, both forward or backward (use negative numbers for moving backward)

Positioning for reading or writing within a file

`[set position in file called [name] to (<integer>) relative to []]`

Set the position for reading/writing the file to **`<integer>`**. As indicated this is particularly useful for random access files (as long as each record in the file is the same length) since records can be accessed in any order. The **`relative to []`** dropdown lets you select **`start`** to move relative to the start of the file, **`current`** to move relative to the current position in the file, or **end** to move relative to the end of the file

Information about SFS itself

(SFS)

Returns the URL for the server **without the protocol http://** (by default this is just **`localhost:7083`**, i.e., **SFS** will be running on the same computer as Snap!, and listening on port **7083** (**70** is the UTF-8 encoding for **F** and **83** the UTF-8 encoding for **S**). See **Network Use** below for situations where you might want to change this

(SFS [])

Returns information about **SFS**: currently the dropdown lets you select **`sfs_version`** (at the time of writing this documentation this was **1.4**) or **`python_version`** to return the version number of the **SFS** program or the version number of the Python interpreter running **SFS** (currently there is only a Python 3 version)

<SFS is running>

Returns true if **SFS** is running, false otherwise; an **`(SFS [])`** request is made and if this returns an empty string (after a delay of a few seconds) then **SFS** is NOT running and false is returned, whereas if any non-empty string response is returned, the server is assumed to be running, and true is returned

Network use

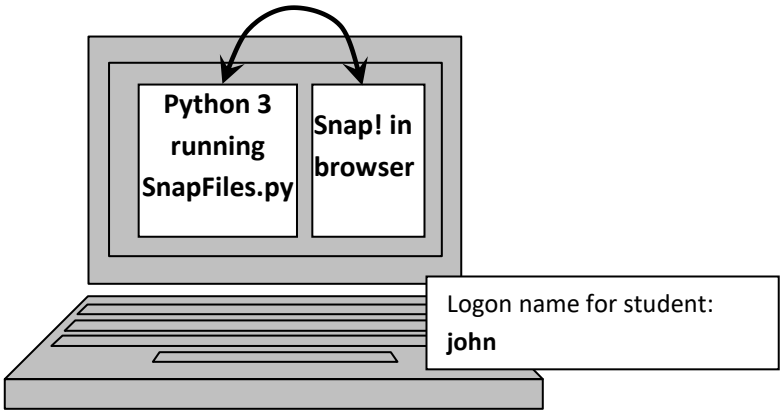
For one reason or another you may not be able to run **SFS** on the same machine that your students are using Snap! and so you can run **SFS** on a remote machine and share it with an entire class (I've used a Raspberry Pi as mentioned above).

If you want to do this then there are two changes to make in **SnapFiles.xml**:

- edit the definition of the function **(SFS)** in the Sensing category so that instead of returning the default **`localhost:7083`** it returns the name or IP address of the machine on which you'll be running **SFS**, e.g., **`raspberrypi:7083`**: this is a change that will need to be done on every project which communicates with the server
- get each student to edit the definition of the function **(username)** so that instead of returning the default **`SFS`** it returns their username (the important thing is that name is unique, so something like their network logon name is ideal). This will be used to generate a directory/folder for their Snap! files within the **SnapFiles** folder in the **Documents** folder of the user running **SFS** on the remote machine.

The diagrams below may make clearer the distinction between the two situations.

SFS running on the same machine as student using Snap! (client and server computer the same)



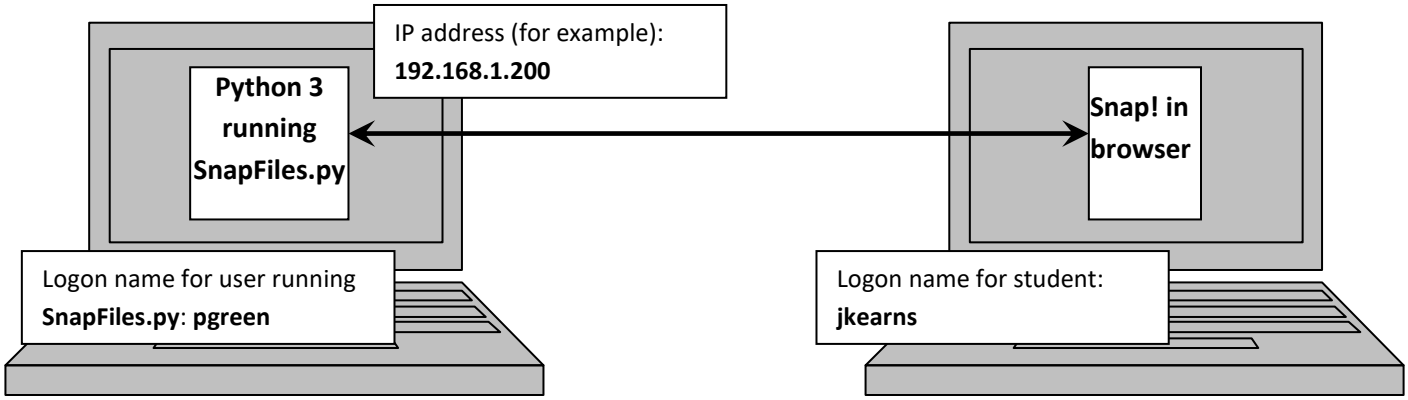
(SFS) function in **Sensing**

(username) function in **Sensing**

Snap! files are stored in the **SFS** subdirectory/folder of the **SnapFiles** subdirectory/folder of the **Documents** of the current user, i.e.,

C:\Users\john\Documents\SnapFiles	on Windows machines (Windows 7 on)
/home/john/Documents/SnapFiles	on Raspberry Pi
/Users/john/Documents/SnapFiles	on Mac OS X

SFS running on a different machine to the student's (server on left, client on right)



(SFS) function in **Sensing**

(username) function in **Sensing**

In this situation Snap! files are stored in the **jkearns** subdirectory/folder of the **SnapFiles** subdirectory/folder of the **Documents** folder of the user **pgreen** running **SnapFiles.py**, i.e.,

C:\Users\john\Documents\SnapFiles	on Windows machines (Windows 7 on)
/home/john/Documents/SnapFiles	on Raspberry Pi
/Users/john/Documents/SnapFiles	on Mac OS X

WARNING: This is not ideal since if two students use the same definition for **(username)**, and the same file names, the contents of the files may get corrupted. If this turns out to be a problem a future version of the SnapFiles.py software could use the IP address of the client machine to distinguish between the two, but this will mean that accessing files in a lesson that were created in an earlier one might not be possible.

Further details and hints

Startup options

There are two startup options for **SnapFiles.py**:

- `-p` followed by an integer which gives a different port address (rather than the default 7083) for the server to listen on, e.g., `python3 SnapFiles.py -p8000` will listen on port 8000
- `-t` will turn on tracing information, printing startup information and details of every request that it handles to the screen

Using SFS remotely

If you are using something like a Raspberry Pi to run **SFS** then you should consider setting it up so that **SnapFiles.py** is run automatically: you can then just plug the Pi in to power and network connectors and turn on.

With the 'Jessie' distribution of Raspbian (from https://www.raspberrypi.org/downloads/raspbian/) edit the file **`/etc/rc.local`** using your favourite text editor, and add the line

```
/usr/bin/python3 /home/pi/Documents/SFS/SnapFiles.py &
```

before the line

```
exit 0
```

at the end[vii]:

Check that this works by rebooting the Raspberry Pi and, after you have loaded **SnapFiles.xml** and changed the **(SFS)** function in the **Sensing** category to reflect the Raspberry Pi's IP address, that the reporter **(SFS is running)** in the **Sensing** category returns true.

Useful Snap! techniques

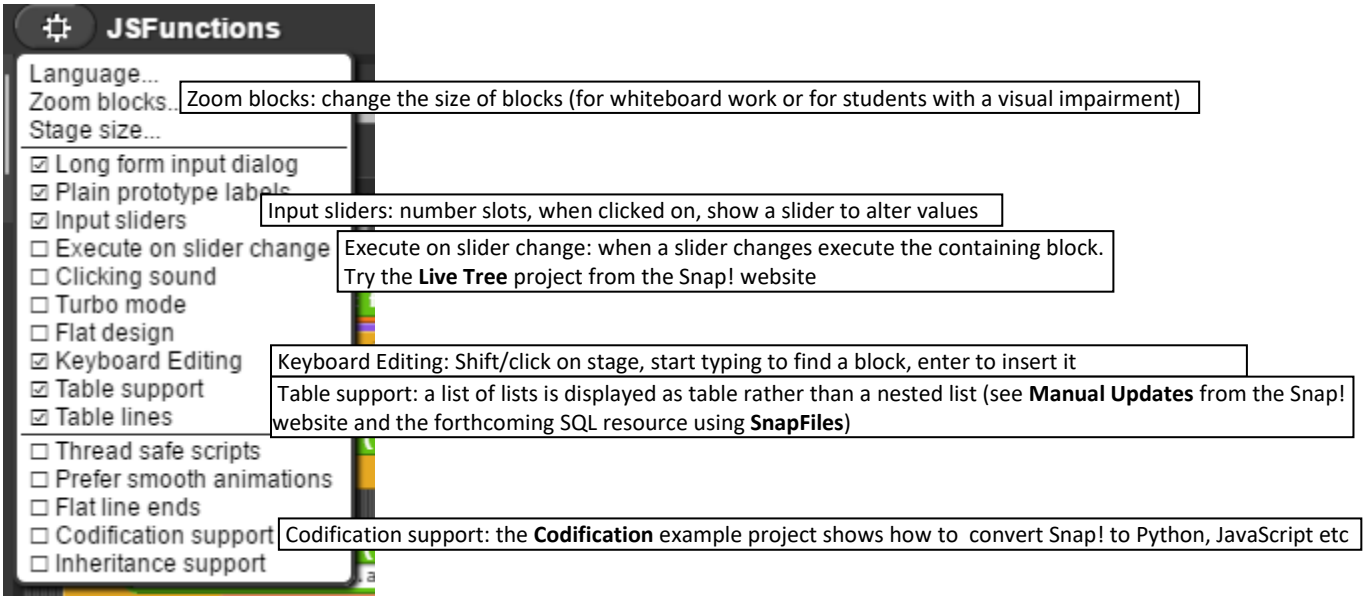
If you right click on the palette of any block category you can hide the primitive blocks, i.e., the blocks that are predefined in Snap! such as **[move () steps]**. By doing this and providing palettes with

just user-defined blocks you can avoid other blocks distracting your students, allowing them to concentrate on the task in hand.

There are a number of useful libraries available in the standard Snap! distribution that can be imported into an existing project (pick **Libraries...** from the 'file' menu).

Similarly there are a number of useful examples available from **Open...**: click the **Examples** button to see the range. Snap! has a facility to call JavaScript functions and the **JSFunctions** examples include one to call your browser's speech synthesis system (so you can make a new **[say []]** block which actually speaks) and one to draw text in any size on the stage. Other blocks show how JavaScript can be used to speed up slow Snap! blocks.

Explore the **Settings** menu (the 'cog' button on the menu bar) for useful (and sometimes experimental) features[viii]:



Keyboard editing: key assignments

If you use keyboard editing this list will help you when you get started. I find it's often easier to **shift + click**, type a few letters of a block's name, press **down** if needed to select the correct block and then **enter** to insert it rather than remember which category a block's in, switch to it, scroll down for to locate it and drag it to the scripting area.

activate:
- **shift + click** on a scripting pane's background
- **shift + click** on any block
- **shift + enter** in the IDE's edit mode

stop editing:
- **left-click** on scripting pane's background
- **esc**

navigate among scripts:
 - **tab**: next script
 - **backtab** (**shift + tab**): last script

start editing a new script:
 - **shift + enter**

navigate among commands within a script:
 - **down arrow**: next command
 - **up arrow**: last command

navigate among all elements within a script:
 - **right arrow**: next element (block or input)
 - **left arrow**: last element

move the currently edited script (stack of blocks):
 - **shift + arrow** keys (left, right, up, down)

editing scripts:
 - **backspace**:
   * delete currently focused reporter
   * delete command above current insertion mark (blinking)
   * collapse currently focused variadic input by one element

 - **enter**:
   * edit currently focused input slot
   * expand currently focused variadic input by one element

 - **space**:
   * activate currently focused input slot's pull-down menu, if any
   * show a menu of reachable variables for the focused input or reporter

 - **any other key**:
   start searching for insertable matching blocks

 - in menus triggered by this feature:
   * navigate with **up** / **down** arrow keys
   * trigger selection with **enter**
   * cancel menu with **esc**

 - in the search bar triggered by this feature:
   * keep typing / deleting to narrow and update matches
   * navigate among shown matches with **up** / **down** arrow keys
   * insert selected match at the focus' position with **enter**
   * cancel searching and inserting with **esc**

running the currently edited script:
   * **shift + ctrl + enter** simulates clicking the edited script with the mouse

[i] Blocks are represented in text using the following notation

[and **]** surround an instruction block that can be linked to others, top and bottom, e.g., **[delete file called []]**

[and **]** inside a block represent a text or any type parameter slot, e.g., the **[]** in the previous bullet point is a slot that can take a filename (or indeed anything, e.g., a number, which is converted to text before being used)

(and **)** surround a reporter block that returns a value that can be used in other reporters, e.g., operators, or instruction blocks

(and **)** inside a block represent a number parameter slot

< and **>** surround a predicate, i.e., a reporter block that returns either true or false

< and **>** inside a block represent the type of thing that can be placed there, e.g., **<integer>** can be replaced with an integer literal (25, 67, etc), or an expression that generates an integer, e.g., **((25) + (45))**

[ii] Behind the scenes **SFS** uses a dictionary, keyed by the full file path, of the file objects that have been opened by the program. The first time I saw this way of handling files was with the **awk** program

[iii] In the **FileTesting** sprite **variables** (local to the script) are used but for the file blocks themselves **functions** returning values are used (defined in the **Sensing** category)

[iv] Wrap any accesses to a file that may conflict with another process/task inside a `[warp []]` block, `save` the position in the file at the start of the block and restore it at the end although even this isn't foolproof

[v] Windows uses the ASCII CR LF control sequence to terminate files, Mac OS X/Linus the LF character only. Storing the correct sequence when the data is written using the `[add line <something> to file called []]` means that such files can be edited correctly using a standard text editor, e.g., Notepad or Nano.

[vi] This makes sharing text files between Windows and Mac OS X/Linux a bit of a problem, although the **[contents ...]** block helps by converting the Windows sequence to the Mac OS X/Linux one.

[vii] Because path variables and the environment are not set up when this is executed you cannot use `python3` on its own or ~ (representing the current user's HOME directory) in this command. The & character means that the command will run in the background

[viii] Some of these features are covered in my CAS TV video at https://www.youtube.com/watch?v=7tjNnF4fAgI