# From Notebook to Script

Note: we've supplied a simple Notebook which opens an image file, plots it, and then blurs and plots it again. This is called **ImageBlur.ipynb** You can use this notebook if you'd like, or you can use any notebook of your choice. The details below assume you're using our Notebook, but you can easily adapt this to any other code. You'll need access to a system running jupyter and for our notebook, you'll need numpy, numba and PIL (pillow).

We've provided some code etc - these are coloured **blue and bold** within this document to make them easier to spot. Directory names within our materials are *blue and italic*

## - **Step 1** - running the conversion

- You can export a notebook while its running (File->Download As->Python (.py)), but that requires firing up your notebook server, and downloads to the client machine, not where the server is running
- Using the command line tool is just as easy and has a lot of advantages

**Info:**

It's always a good idea to run the Notebook before doing the conversion process, just to make sure it works. Run it from the top (all cells in order) to ensure there are no missing definitions or such. You might want to restart the kernel to be absolutely sure

**Try:**

Run the notebook!

Run the convert process (`jupyter nbconvert --to script <notebookname>`)

If you have problems check:

that the notebook file exists and the name is correct, and you're running the command in the directory the notebook is in

that jupyter and nbconvert exist - try `jupyter nbconvert --version` and you should get something like 5.6.2

check the nbconvert issue tracker (https://github.com/jupyter/nbconvert/issues) in case other people have your issue

if it still isn't working, try running the conversion on a very simple notebook first, to see if the problem is the notebook, or the converter. Running a simple case makes problems much easier to diagnose

- **Step 2** - checking the result

- Just because the conversion worked, it doesn't always mean you have a working script! You have to check and test!

**Info:**

Since you always want to run the notebook before converting, you can use that to grab a simple test-case to verify your script

**Try:**

Run the script. Does it work? Does the result match the notebook result?

If not, can you find why and fix it?

Once the script is generally working, we can go on to the next step

- **Step 3** - fixing hidden problems

- Your script now works, but that doesn't aways mean it's useful! Some "notebooky" things can be left behind
- Other things can need adding to make the script more useful

**Info:**

The Notebook we provided has options and control that normally a user would edit in-place. For a script, this is less useful - you don't want to be editing things in all sorts of places. It's clumsy and it's easy to lose track of what you ran

Also, the notebook we provide doesn't produce any image files. You might have spotted this issue in Step-2, and if you got them to show, be aware that in a script, we'd usually want the images preserved as files too

**Try:**

Take the script produced and make sure that user-inputs can be changed without unexpected effects. For example, in our Notebook, the image filename just appears - should this be a file? a filename? the entire path to the file? A user can't tell

It helps to turn inputs into named variables, and add comments. You want a user (or you) to easily find the places to change, and know what the values mean

The next step will show how we can take this input when the code runs

For the ImageBlur script version we'll want to modify it to save the plots rather than show them. We should also add axis labels, a title, and a descriptive filename. Try and do this automatically within the script - don't rely on the user clicking buttons

## **Step 4** - nice things in scripts

- There's some things you can add or tweak to make your scripts nicer
- Generally you should ONLY do this once they're working

**Info:**

For example, we don't generally want to edit a script to change parameters or input data - we'd rather supply this as we go

Also, we might have functions defined in our script that we want to be able to import for use elsewhere - this is messy if we also have "free code" as it will run when we import, which we probably don't want

**Try:**

Check out the small script "**UserInputs.py**" which demonstrates three ways of taking input from a user. Modify your scriptified notebook to use one, or multiple, of these

Mixtures of these are common, although we warn against prompting with 'input" if your script might need to run unattended

Have a look at the script "**RunOrImport.py**". This shows how you can have a script which you can run OR import and have both versions work "as people would expect". Try running it (`./**RunOrImport.py**`) and importing it (start Python, `import **RunOrImport**` Modify your script to work in both cases.

Often modules when run as a script will run tests, or an example case to demonstrate the use of the module. Command-line parameters are a great way to swap between these

In the video, we mentioned the "Shebang" line which tells the computer what program to run our script with. Try putting this into your script, and verify that it can now be run EITHER as "python3 <scriptname>" or "<./scriptname> where the angle-brackets mean "fill in the name here". You can find an example of the line in the scripts we provided here

- **Addendum** - writing better code

  - There's a lot of ways to make your code more robust, whether it stays as a notebook or turns into a script
  - A lot of these can be "retrofitted", but some need you to write a bit differently
  - Sometimes these things don't make sense and make your code more complicated for no good reason. ***Don't use them just because you think you "should"!***

  **Info:**

  The Notebook **ImageBlur-RobustComputing.ipynb** shows these features. For this notebook, they are a bit "overdone", but in more complicated codes they are almost essential

  **Things to think about:**

  - Write good comments!
    - Explain WHY things are done, since the code is already telling you WHAT and HOW it is done

    - Mark things that might be surprising or unexpected and smooth this over
- Name variables well!
    - Use short but descriptive names
    - Don't use names that are hard to read, or too similar (no nl and n1 please!)
- Write functions, and write them well
    - Each should have a descriptive name and do one task
    - Think about what they take (arguments) and return. Make sure they always return something, and it is always the same kind of thing (not sometimes a number and sometimes an error string)
    - If you do something many times, it definitely belongs in a function. If you do it only once, it might still be a good idea
    - Write docstrings! It's a super easy way to get documentation right into your code
- Put things where they belong
    - Global variables aren't terrible, but they are only good for some things
    - Define variables inside the function that uses them
    - If you often need to talk about some data-items as a single entity, consider writing a class to hold them
- Check user inputs (even if the user is you!)
    - Use try/except blocks or asserts to make sure values are usable
    - Where possible, tell the user what is needed or expected
    - Don't start a long, time-consuming calculation before checking you have the data you need to finish it
- Handle errors where possible
    - If you can fix something, consider doing so
    - For instance, you wanted a real (decimal) input but got an integer - does it matter or can you simply fix the type?
- If you can't handle them, catch the errors and write helpful messages

    - Don't just `except / pass`
    - Don't swallow the exception message - catch it and repeat it
    - Catch expected sorts of Exception explicitly (such as a KeyError or an IOError) because then you can give a really good message
    - You can make your own Exceptions if you ever need to, and raise and catch these
- Write tests of your functions where possible
    - Check they give sensible results in known cases
    - Check they give errors in known-bad cases
    - Try and cover as many input cases as possible
    - Give as clear and detailed an error report as you can - imagine you're the one having a problem with somebody else's code - what information might you need to find and fix it
- Notebooks and scripts are complementary
    - Notebooks are best used with "rich" outputs - inline plots, tables etc
    - Scripts are great for automatic tasks like generating figures for publications
    - You can mix these!
        - You can import functions from other scripts you have written just by putting them in your working directory and running `import <name>` in a script or notebook
        - In this way you can collect useful, general functions for tasks you do a lot, while also using notebooks as a sort of lab-book where you try things out
        - Remember to document your functions! You can't re-use them if you don't know what they do