Writing a Programming Language Handbook
A Brief Introduction to Jupyter for Language Acquisition

Daniel Szelogowski
Independent Programming Literacy

Jupyter & Computational Essays

Jupyter is a tool frequently used by computer and data scientists, engineers, and physicists to create "computational essays" (a type of essay that includes text, computational tools, interactive diagrams, and small programs to express an idea or small application, see https://uio-ccse.github.io/computational-essay-showroom/ for examples). These essays, or notebooks, allow the use of live code integration, along with computational output, LaTeX equations, explanatory text/documentation, multimedia resources, etc., to create and share complex computational documents.

Jupyter Lab & Jupyter Notebooks

Jupyter can be installed in a number of ways, but perhaps the most common is to use the Python package manager "pip" with the command "**pip install jupyterlab**" entered in the Command Line (sometimes Terminal, CMD, etc.) – see https://jupyter.org/install for additional options, and https://www.python.org/downloads/ to install the latest version of Python. This can then be launched with the command "**jupyter-lab**" which starts a live web server that can be accessed in the browser through the URL http://localhost:8888/lab, which provides a robust interface for numerous languages. To close the server, simply press "**Control + C**" in the Terminal or press "**Quit**" on the web page.

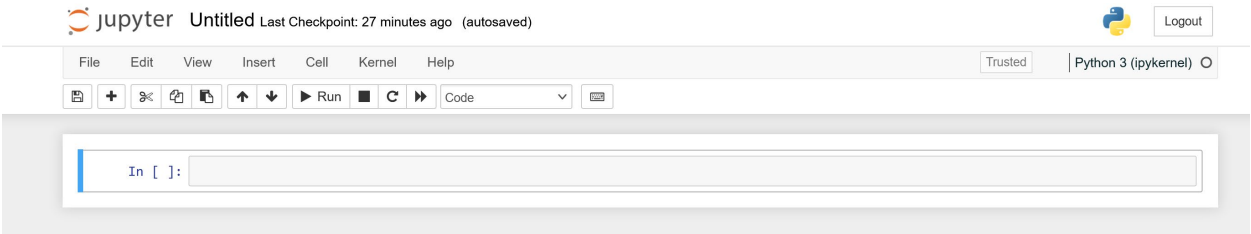
For a simpler (more focused) interface, use Jupyter Notebooks instead, which can be installed using "**pip install notebook**" and launched with "**jupyter notebook**". This again opens web server on the same port as Jupyter Lab, which can be accessed by the URL http://localhost:8888/notebooks/.

The variance between these two is that of comparing a simple text editor to a rich-text editing suite (e.g., Notepad to Microsoft Word) – Jupyter Lab contains a more thorough interface as well as an in-built tutorial system, whereas Jupyter Notebook is much like that of a hand-written notebook. For the rest of this guide, we will use Jupyter Notebook for its simplicity and ease of use, though Jupyter Lab is recommended for more complex projects.

Getting Started with Jupyter Notebooks

Upon launching Notebook, you will be greeted by a File Tree where you can navigate to the desired location where you want your notebook to be stored.

Press the "**New**" button in the upper right corner to create a new Notebook. You will be prompted to choose either a specific programming language (Kernel) or create a new text file, folder, or launch a new terminal. If the programming language of your choice is not listed, you need to install the kernel for it, which can be found on https://github.com/jupyter/jupyter/wiki/Jupyter-kernels (ideally look for a kernel beginning with "**I**" for Interactive, such as IElixir, IGo, IScala, etc.). Otherwise, choose the desired kernel and a new notebook blank notebook file will be created at the desired location.



A more thorough guide to Jupyter can be found on https://jupyter4edu.github.io/jupyter-edu-book/jupyter.html if desired, as this guide will briefly discuss installing a new kernel and getting started writing a simple formatted computational essay for the purpose of programming language acquisition.

Installing a Language Kernel

For this guide, we will use C#; navigate the list to the "**ICSharp**" kernel – the link will take you to a GitHub page containing the source code for the kernel. Scroll down to find the "Readme" which contains the installation instructions. **Note that these installation instructions will differ greatly between operating systems** – some kernels may only support installation through Docker on Windows especially. Close Jupyter first, then carefully follow the instructions for installation specific to your operating system. An example for install ICSharp on Windows can be seen below (from https://github.com/zabirauf/icsharp/wiki/Installation).

Installation from sources on Windows

Requirements

- Git
- Jupyter
- Visual Studio

Compile ICsharp

Get the sources from Github (including submodules):

```
git clone --recursive https://github.com/zabirauf/icsharp
```

Navigate to the engine folder, and open ScriptCS.sln in VS.

Change to Release from debug, Clean and rebuild solution (right click solution in solution explorer and rebuild twice works too).

Close VS and go back to root and open iCSharp.sln, switch the configuration to Release. Rebuild the solution.

Add ICsharp to the kernel list.

Open the file `kernel-spec/kernel.json`. Modify the `<INSTALL_PATH>` in the following line to match the PATH were you compiled ICsharp:

```
"argv": ["<INSTALL_PATH>/icsharp/Kernel/bin/Release/iCSharp.Kernel.exe", "{connection_file}"],
```

Inside the `icsharp` directory containing the code, run:

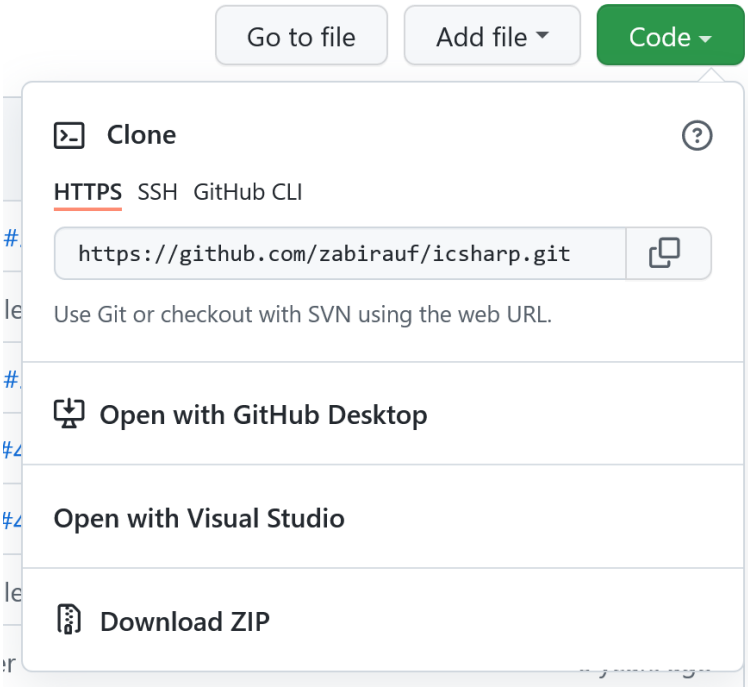
```
jupyter kernelspec install kernel-spec --name=csharp
```

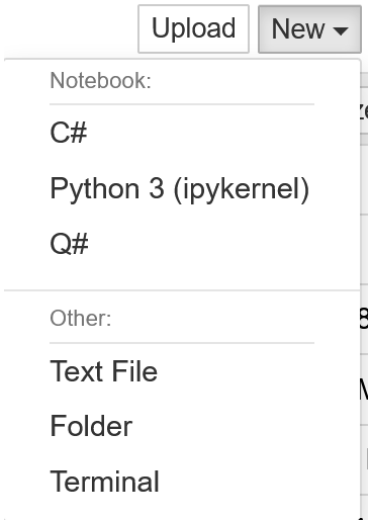
This command copies the `kernel-spec` directory in your `ProgramData\Jupyter\kernels\` where Jupyter looks for kernels, and renames it as `csharp`. Check that `csharp` is listed in:

```
jupyter kernelspec list
```

Now just launch `jupyter notebook` and `C#` will be available in the Kernel list once you opened a notebook.

If you do not have **git** installed, you can navigate to the main page of the kernel on GitHub and press the green "**Code**" button, which will allow you to download the source code as a Zip file, then extract the folder to a location such as your desktop. You can also open it directly through Visual Studio. After following the installation instructions, you should see the new language added to your list of available kernels in Jupyter when you make a new notebook.





Writing Notebooks

You can rename a handbook at the top of the page to change it from "Untitled" to something like "**C# Handbook**" (or whatever language you chose). By default, each notebook starts with a single empty **code** cell. You can change this classification to text for writing notes by clicking the dropdown menu with the word "**Code**" (between the keyboard icon and ⏩ arrows) and switching it to "**Markdown**" which allows for rich-text formatting.

To write a first-level header, use "**# Some Header Name**" – i.e., the header name of your choice, preceded by the # symbol. You can also use HTML or LaTeX code for formatting as well (see https://towardsdatascience.com/write-markdown-latex-in-the-jupyter-notebook-10985edb91fd), but we will use Markdown for its simplicity for now. LaTeX is highly recommended for mathematical equations, however.

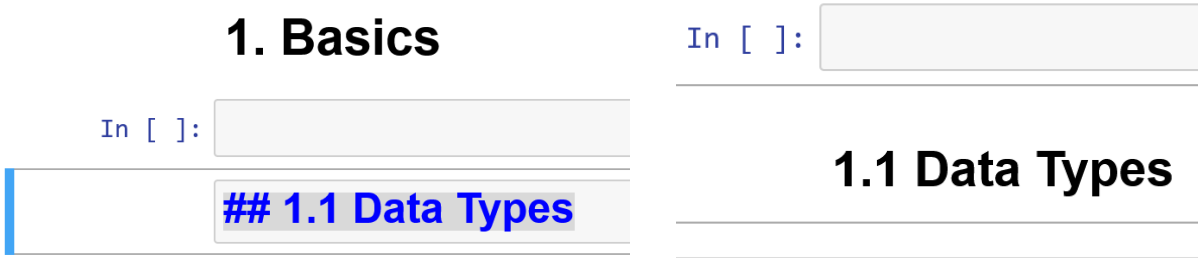
```
# 1. Basics
```

To see the formatted heading, press the "▶ **Run**" button which executes the code or markdown language in the cell. This will also create a new cell below. **You can also use the hotkey "Control + Enter" to execute a cell without creating a new cell below.**

1. Basics

In [ ]:

Likewise, to create a second, third, fourth-level heading (subheading/sub-subheading, …), etc., simply add additional # symbols before the header title. Remember to change the cell type to Markdown – all cells are Code by default.



To easily add new cells, click off the textbox of the cell (such as on the "In [ ]:" text preceding it) and hit **A** on the keyboard to add a new cell above, or **B** to add a new cell below, likewise. You can also hit **D** twice (type **DD**) to delete a cell. You can also add a table using Markdown (see https://www.markdownguide.org/extended-syntax/).

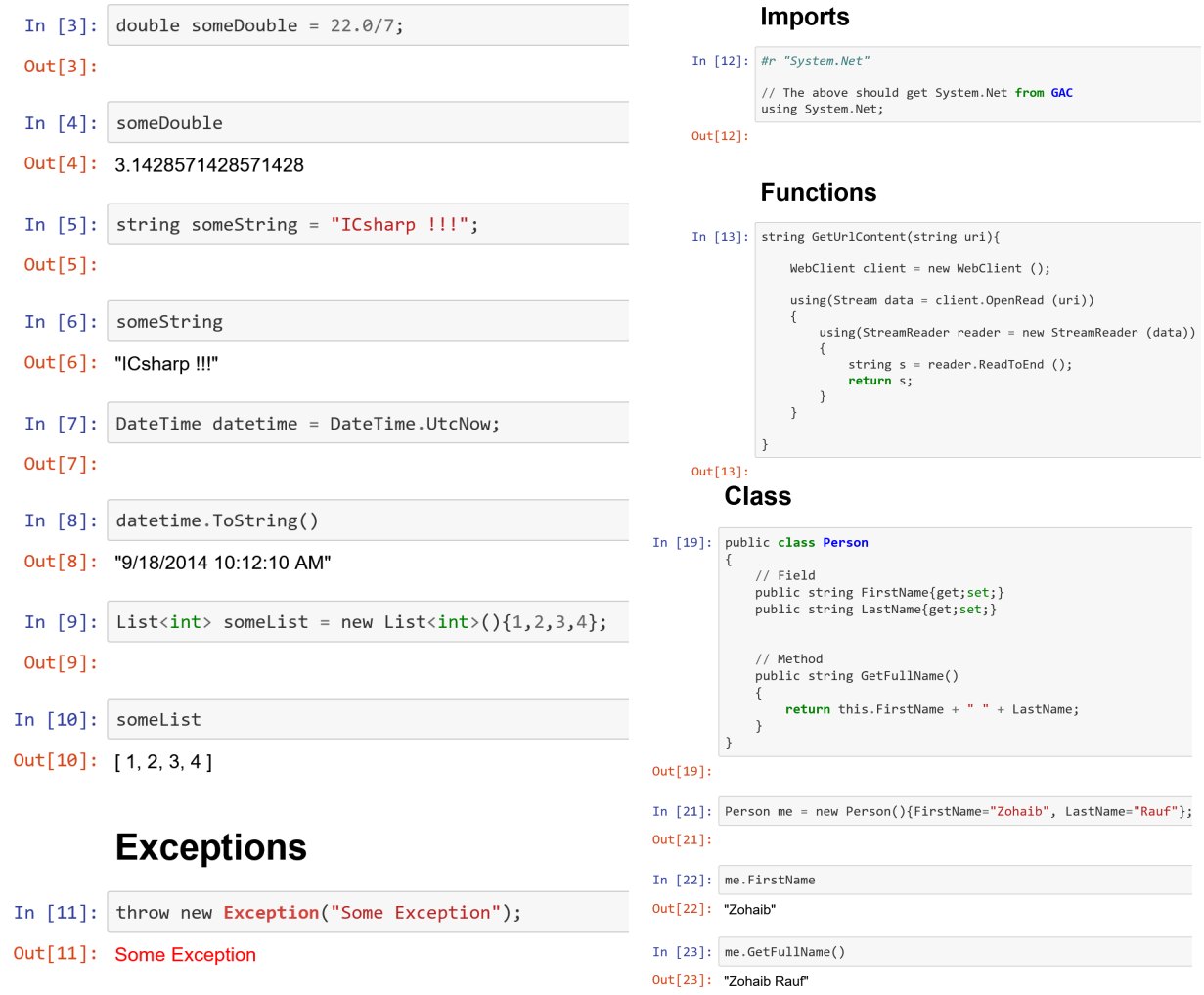
```
Common data types:

| Data Type | Size    | Description |
| :---      | :----:  | :---        |
| int       | 4 bytes | Stores whole numbers from -2,147,483,648 to 2,147,483,647 |
| long      | 8 bytes | Stores whole numbers from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807 |
| float     | 4 bytes | Stores fractional numberswith 6 to 7 decimal digits of precision |
| double    | 8 bytes | Stores fractional numbers with 15 decimal digits of precision |
| char      | 2 bytes | Stores a single character/letter, surrounded by single quotes |
| string    | 2 bytes/char | Stores a sequence of characters, surrounded by double quotes |
| bool      | 1 bit | Stores true or false values |
```

Common data types:

Data Type	Size	Description
int	4 bytes	Stores whole numbers from -2,147,483,648 to 2,147,483,647
long	8 bytes	Stores whole numbers from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
float	4 bytes	Stores fractional numberswith 6 to 7 decimal digits of precision
double	8 bytes	Stores fractional numbers with 15 decimal digits of precision
char	2 bytes	Stores a single character/letter, surrounded by single quotes
string	2 bytes/char	Stores a sequence of characters, surrounded by double quotes
bool	1 bit	Stores true or false values

From there, simply start writing your code as desired, and switch between using Markdown and Code cells to document your programming language (or program!) as desired. You can write your code either directly inside the cell or copy and paste it in as desired for formatting. Also note that **code does not have to be written/structured as a typical program, but as an interactive script – you can print variables and display values simply by typing their name.** This is very similar to how Python Interactive (or other interactive interpreters) would be used.

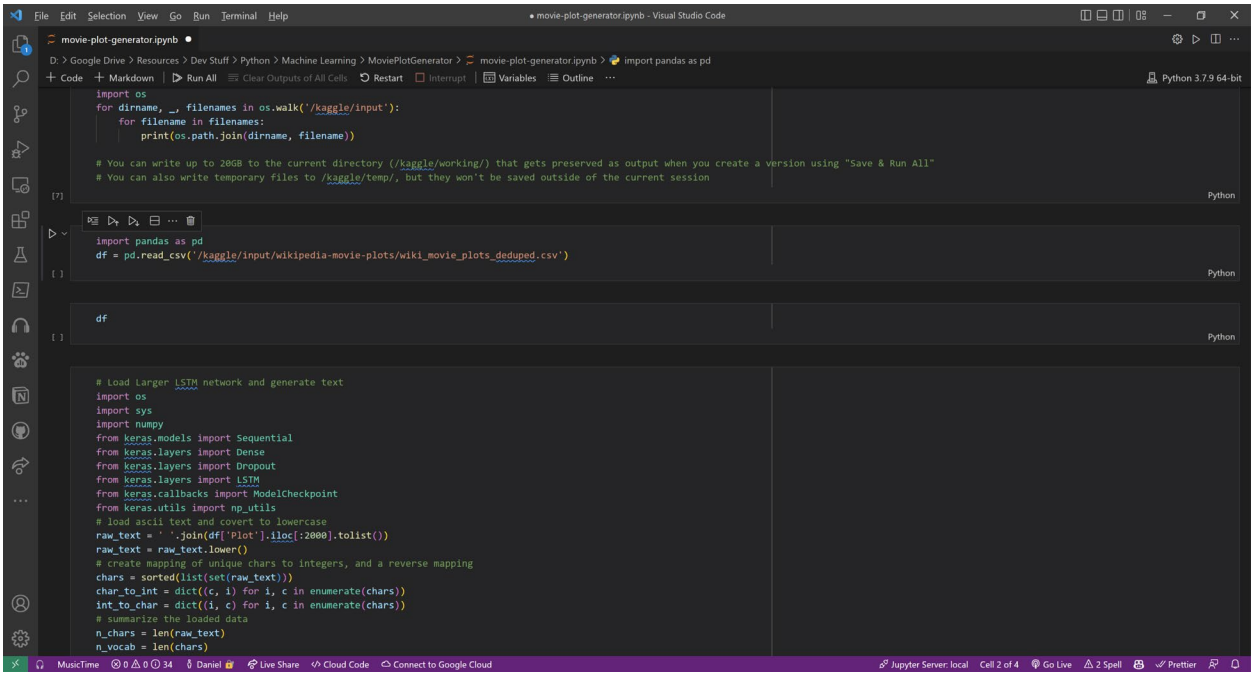


(Source: https://tinyurl.com/2fbycnv2)

See "Go Handbook.pdf" (which can be found at http://danielszelogowski.com/resources/Go%20Handbook.pdf) for a full example. You may use this for reference, but you must write your own handbook from scratch (including code, comments, and documentation) if you choose to use Golang for an assignment or it will be considered plagiarism.

Writing Notebooks in VS Code

One of the best solutions for efficiently writing a Jupyter notebook is using the popular text editor Visual Studio Code (which can be downloaded from https://code.visualstudio.com/Download), which has great support for notebooks (see the Jupyter extension: https://marketplace.visualstudio.com/items?itemName=ms-toolsai.jupyter).



This is essentially the same as using the Jupyter notebook app itself, but allows for the syntax highlighting, code-completion, and additional programming support provided by VS Code (and a nice dark theme). This allows enables the usage of remote Jupyter servers with additional kernels such as for C++ (i.e., with xeus-cling, found here: https://github.com/jupyter-xeus/xeus-cling). Additional resources and guides for using Jupyter in VS Code (locally and remotely) can be found at https://code.visualstudio.com/docs/datascience/jupyter-notebooks.

The Binder Project and NBViewer

Binder is a project and web tool (found at https://mybinder.org/) that allows a GitHub repository containing a Jupyter notebook (or set of notebooks) to be "bound" into a live, shareable page (an example of which can be found at https://mybinder.org/v2/gh/binder-examples/julia_python/master). Another similar project which is also very popular is nbviewer (found at https://nbviewer.org/) which renders the notebook (or set of notebooks) as a static HTML web page and generates a shareable static link for the notebook.

Language Acquisition – A Universal Curricula

When learning a programming language, there are hundreds of possible factors to consider – hence, it is important to narrow down the most important language aspects based on the type of language (dynamic/duck/static typed, explicit/implicitly typed, functional, procedural, object-oriented, imperative, reactive, etc.), its applications, and the goals of using the language. However, there are numerous factors that are universal to nearly all programming languages – these can be used to learn the basics of the language and scaffolded to learn more advanced aspects. While a variety of permutations is possible, the following is one of the most applicable orders of learning syntax and is highly recommended before tackling more difficult problems:

1. Common/primitive data types (int, float, char, …) OR mutable/immutable type declarations (let, var, …), acceptable variable names/naming rules
2. Compiling the language by hand (in a terminal)/running the interpreter
3. Arithmetic operators (+, -, *, /, %, **, …), type casting (float), literals
4. Assignment operators (+=, ++, --, /=, …), equals(), math functions
5. Comments and block comments (//, /*, #, (*, """, …)
6. Console I/O: printing variables with and without type formatting (print vs. printf vs. println, …), user input (scanf, input, readline/readkey, …)
7. Conditional statement syntax (if, else, elif, unless, …)
8. Conditional operators (<, >, != or <>, ==, …), language scope
9. Logic operators (&& or AND, || or OR, ! or NOT, …)
10. Nested/compound conditionals (switch), the ternary operator (?:)
11. Loops (for, foreach, while, do-while, …), break, continue
12. Functions/methods, parameters, recursion, anonymous functions, passing by reference vs. by value (ref, out, &, *, this, …)
13. Arrays and matrices (multidimensional/jagged array), enums, tuples
14. Major keywords (static, const, import/using, namespace, …)
15. String functions (comparing strings, substring, insert, …)
16. IF APPLICABLE (if included in the language):
    a. Pointers, function pointers, void pointer, etc., sizeof/typeof
    b. Classes (structs/unions) and object declaration, destructors, namespaces/packages, class accessors (public/private/protected)
    c. Memory management and dynamic memory allocation
    d. Inheritance, function/argument/operator overloading/overriding, abstraction (virtual, interface, abstract, friend), polymorphism
    e. Preprocessor directives, scope, headers, and operators
    f. Conditional compilation directives, symbols, macros, typedef
    g. Generic typing/templates (classes and functions), default(T)

17. Importing libraries/namespaces from files and/or DLLs
18. Error handling (try-catch, raise/throw, creating Exceptions)
19. Working with files, binary file I/O (write, read, append, create, …)
20. Random generation (int and float)
21. Working with time (calculating date/times, measuring runtime)
22. Importing libraries using package managers (pip, NuGet, conda, Git, …)
23. Bitwise and bitwise assignment operators (&=, !, ^, |, &, <<, &&=, …)
24. Common data structures (vectors, list, dictionary, set, ArrayList, HashMap/table, stack, etc.), hashing, and iterators (if applicable)
25. ADVANCED, IF APPLICABLE/DESIRED (any order):
    a. Indexing, slicing, and pattern matching
    b. Getter and setter shorthand
    c. Prototyping, subroutines
    d. Serialization
    e. Variadic functions
    f. Null operators (null coalescing, null forgiving, optional chaining)
    g. Ranges, map/filter, reduce, closures
    h. Fields, records, tables, databases, and other functional types
    i. Static members and static classes
    j. GUI development (if possible/useful, including video games)
    k. Abstract data structures
    l. Sorting algorithms
    m. Parallel processing, multithreading, asynchronous functions, defer, mutex, semaphores, routines/channels, yield
    n. Networking methods (TCP/UDP messages, web servers, hosting a web page, etc.)
    o. Encryption and security methods

This curriculum is very similar to many academic classes, textbooks, online learning platforms, tutorial series, among others. By learning these specified language features in the given order, it is much easier to maximize both the learning efficiency (in terms of both speed and degree of comprehension) and the ability to learn new language concepts in the given programming language and others due to the scaffolded approach. The learning path also covers most modern languages, including the skills necessary for everyday application, as the listed concepts are present (either by the language design, standard library, or extensions) in nearly every programming language – not including some esoteric and/or code golfing languages, however, as these are limited by design. As well, this guide can be applied to the development of a language when teaching compiler design/grammars, as these should almost always be possible in a language intended for everyday use and/or universal application.