PTP-P1-V00907185-Evan-Strasdin

March 3, 2024

Term Portfolio Project

Evan Strasdin
University of Victoria
SENG-265, Winter 2023-24

Table of Contents

Introduction: Jupyter Magics, Creating a Table of Contents

About notebooks, magics

I've included the section on magics and markdown tables of contents since, if you're like me, you love the idea of a single file which can be navigated like a wiki, can support both markdown and usable code in just about any language and can be exported directly to html for publishing online.

Magics are special commands that provide additional functionality within Jupyter notebooks. If you can't figure out how to do something specific with your notebook, there's a good chance you can do it with a magic.

I have a good amount of experience using Jupyter notebooks but little experience using magics (or python scripts for that matter), so I've decided to see if the all-knowing chatGPT could help me

out. My goal for now is to add support for C and Python cells in the same notebook, and to create a working table of contents to make this file navigable.

Creating a custom magic

1. Create a Python module for your magic command This chatGPT code defines the behavior of the custom magic:

[6]:
```
# Module contents for C language support

from IPython.core.magic import Magics, magics_class, cell_magic
from IPython import get_ipython
import subprocess

@magics_class
class CMagic(Magics):
    @cell_magic
    def c(self, line, cell):
        # Save the C code to a temporary file
        with open('temp.c', 'w') as f:
            f.write(cell)

        # Compile the C code
        compile_command = f'gcc -o temp_executable temp.c'
        compile_result = subprocess.run(compile_command, shell=True,␣
 ↪stdout=subprocess.PIPE, stderr=subprocess.PIPE, text=True)

        if compile_result.returncode == 0:
            # If compilation is successful, execute the compiled code
            execute_command = './temp_executable'
            execute_result = subprocess.run(execute_command, shell=True,␣
 ↪stdout=subprocess.PIPE, stderr=subprocess.PIPE, text=True)
            print(execute_result.stdout)
            print(execute_result.stderr)
        else:
            # If compilation fails, print the error message
            print(compile_result.stderr)

# Register the magic command
ip = get_ipython()
ip.register_magics(CMagic)
```

In order to make this code usable (a module) it must be in a .py file. To save some time we can run the cell below, which creates this file for us:

[7]:
```
# Define the code to be included in either module as multiline strings,
# copy-pasted from above
```

```
module_code_c = """
from IPython.core.magic import Magics, magics_class, cell_magic
from IPython import get_ipython
import subprocess

@magics_class
class CMagic(Magics):
    @cell_magic
    def c(self, line, cell):
        # Save the C code to a temporary file
        with open('temp.c', 'w') as f:
            f.write(cell)

        # Compile the C code
        compile_command = f'gcc -o temp_executable temp.c'
        compile_result = subprocess.run(compile_command, shell=True,␣
 ↪stdout=subprocess.PIPE, stderr=subprocess.PIPE, text=True)

        if compile_result.returncode == 0:
            # If compilation is successful, execute the compiled code
            execute_command = './temp_executable'
            execute_result = subprocess.run(execute_command, shell=True,␣
 ↪stdout=subprocess.PIPE, stderr=subprocess.PIPE, text=True)
            print(execute_result.stdout)
            print(execute_result.stderr)
        else:
            # If compilation fails, print the error message
            print(compile_result.stderr)

# Register the magic command
ip = get_ipython()
ip.register_magics(CMagic)
"""

# Write the strings to file
with open('c_magic.py', 'w') as f2:
    f2.write(module_code_c)
```

Our python module is complete!

2. Load and use the magic command

[8]:
```
# Commands to initialize %%c
%load_ext autoreload
%autoreload 2
%run c_magic.py
```

[9]:
```
%%c
#include <stdio.h>

int main() {
    printf("Hello, C programming in Jupyter!");
    return 0;
}
```

```
Hello, C programming in Jupyter!
```

Back to top

Creating a table of contents

I used chatGPT to get a basic idea of how a table of contents works in markdown (and thus Jupyter). Here's the output:

```
 Table of Contents

- [Link to Section 1](#section-1)
- [Link to Section 2](#section-2)

---

<a id="section-1"></a>
## Section 1

This is the content of Section 1.

---

<a id="section-2"></a>
## Section 2

This is the content of Section 2.
```

Note that the `<a id="section-1"></a>` HTML anchors aren't always necessary, since many markdown editors (and editing tools, such as markdown all in one) write anchors automatically based on headings.

Another fun fact about anchors is that we can use them for internal linking, for example the markdown text `[Some text](#section-1)` would create a link to the header `## Section 1` in the above document, regardless of where it's located in the document.

I'm sure you've seen it already, but here's the finished product

Back to top

Bash

The most useful command line tools: `man` and `--help`

While `man` (short for manual) and `--help` can't teach you commands, they can be extremely helpful for learning how to use them. For example, let's say we were wondering about the functionality of the `mkdir` command:

[10]:
```
%%bash
man mkdir
```

```
MKDIR(1)                          User Commands                          MKDIR(1)

NAME
       mkdir - make directories

SYNOPSIS
       mkdir [OPTION]… DIRECTORY…

DESCRIPTION
       Create the DIRECTORY(ies), if they do not already exist.

       Mandatory  arguments  to  long  options are mandatory for short options
       too.

       -m, --mode=MODE
              set file mode (as in chmod), not a=rwx - umask

       -p, --parents
              no error if existing, make parent directories  as  needed,  with
              their file modes unaffected by any -m option.

       -v, --verbose
              print a message for each created directory

       -Z     set  SELinux  security  context of each created directory to the
              default type

       --context[=CTX]
              like -Z, or if CTX is specified then set the  SELinux  or  SMACK
              security context to CTX

       --help display this help and exit

       --version
              output version information and exit

AUTHOR
```

```
       Written by David MacKenzie.

REPORTING BUGS
       GNU coreutils online help: <https://www.gnu.org/software/coreutils/>
       Report any translation bugs to <https://translationproject.org/team/>

COPYRIGHT
       Copyright © 2023 Free Software Foundation, Inc. License GPLv3+: GNU
       GPL version 3 or later <https://gnu.org/licenses/gpl.html>.
       This is free software: you are free to change and redistribute it.
       There is NO WARRANTY, to the extent permitted by law.

SEE ALSO
       mkdir(2)

       Full documentation <https://www.gnu.org/software/coreutils/mkdir>
       or available locally via: info '(coreutils) mkdir invocation'

GNU coreutils 9.4               January 2024                     MKDIR(1)
```

[11]:
```
%%bash
mkdir --help
```

```
Usage: mkdir [OPTION]… DIRECTORY…
Create the DIRECTORY(ies), if they do not already exist.

Mandatory arguments to long options are mandatory for short options too.
  -m, --mode=MODE   set file mode (as in chmod), not a=rwx - umask
  -p, --parents     no error if existing, make parent directories as needed,
                    with their file modes unaffected by any -m option.
  -v, --verbose     print a message for each created directory
  -Z                   set SELinux security context of each created directory
                         to the default type
      --context[=CTX]  like -Z, or if CTX is specified then set the SELinux
                         or SMACK security context to CTX
      --help        display this help and exit
      --version     output version information and exit

GNU coreutils online help: <https://www.gnu.org/software/coreutils/>
Full documentation <https://www.gnu.org/software/coreutils/mkdir>
or available locally via: info '(coreutils) mkdir invocation'
```

Note that the output differs slightly - this is because `man` reads directly from the bash manual, while help is a built-in command. This means that all commands should have a manual, and some commands will not have a common `--help` flag available .

Back to top

Shell scripting

In the same way that magics can make notebooks more useful, shell scripting allows you to do pretty much anything you can imagine from your command line (as long a you can write the code). In the following example we'll create a simple shell script which allows us to connect to a server using a one-word command.

1. Creating the file First we need to figure out where we want to keep our file. According to some guy online, if we want our script to only be accessible by a single user we should store them in `~/bin` or `~/.local/bin`, otherwise we store them in `/usr/local/bin`. Navigate appropriately using the `cd` command:

```
$ cd ~/bin
```

2. Writing the script To connect to a server in linux, we use the `ssh` command. Here's an example:

```
$ ssh username@remote_host
```

Note that this code is generic, and the username/host (etc) will differ depending on the server you're trying to connect to. Let's put this command into a new .sh file using `nano`, making sure to include the shebang `#!/bin/bash` which marks the file as executable:

```
$ nano server
```

The file contents should look like this:

```
#!/bin/bash

ssh username@remote_host
```

3. Making it executable We use the `chmod` command as follows (the 'a' allows all users to access the script, the +x makes it executable):

```
$ chmod a+x server
```

Now when we type `./path/to/file/server` in bash, our file will execute, connecting us to the desired server. Not only that, now we know how to make short commands that do anything we want. Sweet!

4. (Optional, extremely useful) Making it available globally In order for the command to be able to be executed from any directory, we have to export the location of the scripts to our PATH, if it's not already. We use the following command to check:

```
$ echo $PATH
```

This should output a series of directories delimited by ":". If you don't see "/home/usr/bin", we need to add this to our path permanently by making sure it's in our path each time we open a terminal. Otherwise skip this step. We `echo` the following line into our `.bashrc`, the initialization file for bash:

```
$ echo "export PATH=$PATH:/home/usr/bin" >> ~/.bashrc
```

As a last step, we reload `.bashrc` to update the contents using the following command:

```
$ source ~/.bashrc
```

Now when you run the command `server`, you should be automatically connected to the server of your choice.

Aliases

The above example was overkill for the sake of instruction. Generally we should leave scripting to running actual programs and use aliases to shorten commands. The folowing one-liner is a concise way to shorten a command:

```
$ echo "alias myalias='command_to_run'" >> ~/.bashrc
```

We can also use bash functions to get the command to take arguments, but in my opinion this should be left to scripts to avoid having an overcrowded ~/.bashrc file.

Back to top

C

C and bash

We use gcc (gnu C compiler) to compile and render executabe C files in bash. Here's a basic example:

```
$ gcc helloworld.c -std=c99 -o helloworld
```

This compiles the c file 'helloworld.c' using C99 (a version of C from 1999), outputting ('-o') an executable named 'helloworld', which we run using the command `./helloworld`.

Pointers

The distinguishing feature of C is it's direct access to hardware and data stored in memory. The first occurrence of this I ran into was the use of pointers, which are simple but strange without context. Here's some examples which are fairly close to those found in the course's lecture notes:

[29]:
```
%%c

#include<stdio.h>
#include<stdlib.h>

int main(void){
    int n = 10; //integer w value '10'
    printf("n = %d\n", n);
    int* p = NULL; //int pointer, points to NULL
    printf("p = %p\n", p);

    p = &n; // now p contains address of n
    printf("p = address of n = %p\n", p);\

    // Dereferencing p (*p) allows us to indirectly
```

```
    // access the data stored at whatever address is held
    // by p:
    printf("p dereferenced = %d\n", *p);

    int** p2 = &p; // Points to p's address
    // Dereferencing pp (**p) allows us to indirectly
    // access the data stored at whatever address is held
    // by p:
    printf("p2 dereferenced = %d\n", **p2);

    *p = 8; //changes value at address of n
    printf("n = %d\n", n);

    **p2 = 6; //changes value at address of n
    printf("n = %d\n", n);

}
```

```
n = 10
p = (nil)
p = address of n = 0x7ffca04c5aa4
p dereferenced = 10
p2 dereferenced = 10
n = 8
n = 6
```

Note that pointer types should match the type contained at the address they point to, and that we cam't access certain areas of memory (for example, we cannot dereference a `NULL` pointer). Pointer issues will generally result in runtime errors called segmentation faults.

A note on call-by-reference and call-by-value These are mostly self-explanatory methods for passing values to functions (in any programming language, not limited to C). chatGPT supplied the following definitions and examples:

1. Call-by-Value:

- In call-by-value, a copy of the argument's value is passed to the function.
- Changes made to the parameter within the function do not affect the original value outside the function.
- Primitive data types (int, float, char, etc.) are typically passed by value.

[30]:
```
%%c

#include <stdio.h>

void increment(int x) {
    x++; // Increment x (local copy)
```

```
    printf("Inside function: x = %d\n", x);
}

int main() {
    int num = 5;
    increment(num); // Pass num by value
    printf("Outside function: num = %d\n", num); // num remains unchanged
    return 0;
}
```

```
Inside function: x = 6
Outside function: num = 5
```

2. Call-by-Reference:

- In call-by-reference, a reference (or address) to the original argument is passed to the function.
- Changes made to the parameter within the function affect the original value outside the function.
- Pointers are often used to implement call-by-reference.

[31]:
```
%%c

#include <stdio.h>

void increment(int *x) {
    (*x)++; // Increment the value pointed to by x
    printf("Inside function: *x = %d\n", *x);
}

int main() {
    int num = 5;
    increment(&num); // Pass the address of num (call-by-reference)
    printf("Outside function: num = %d\n", num); // num is incremented
    return 0;
}
```

```
Inside function: *x = 6
Outside function: num = 6
```

This should help bring some context to how and why pointers are essential in C: Without them we would have no way to pass variables between functions with persistent results.

Back to top

C libraries

This section will be updated with functions as they are used - the libraries are not limited to the listed functions. For more extensive examples I would recommend checking out chatGPT.

Input and output: `<stdio.h>`

- Standard Input/Output:
  - `printf()`: Prints formatted output to the standard output stream (usually the console).
  - `scanf()`: Reads formatted input from the standard input stream (usually the keyboard).
- File Input/Output:
  - `fopen()`, `fclose()`: Opens and closes files for reading and writing.
  - `fread()`, `fwrite()`: Reads from and writes to files.
  - `fprintf()`, `fscanf()`: Similar to `printf()` and `scanf()`, but work with file streams.
  - `fputs`, `fgets`: Reads and writes from files to strings

String manipulation: `<string.h>`

- String Copying:
  - `strcpy()`: Copies a string.
  - `strncpy()`: Copies a specified number of characters from one string to another.
- String Concatenation:
  - `strcat()`: Concatenates two strings.
  - `strncat()`: Concatenates a specified number of characters from one string to another.
- String Comparison:
  - `strcmp()`: Compares two strings.
  - `strncmp()`: Compares a specified number of characters from two strings.
- String Length:
  - `strlen()`: Returns the length of a string.
- String Searching:
  - `strchr()`: Finds the first occurrence of a character in a string.
  - `strrchr()`: Finds the last occurrence of a character in a string.
  - `strstr()`: Finds the first occurrence of a substring in a string.
- String Tokenization:
  - `strtok()`: Splits a string into tokens based on a delimiter.
- String Modification:
  - `strtok()` (for tokenization and modification).
  - `strdup()`: Duplicates a string.
  - `strchr()`, `strrchr()`, `strstr()` (for modification).

Back to top

Answers to Selected Questions

1. **Required**. Describe your continued learning experience in SENG 265. Make it a weekly habit to document your learning experience.
   - The entire document serves as a 'weekly learning experience' notebook - The weeks during which specific materials were covered are included in the table of contentes.

2. What is the core functionality of Jupyter Notebooks?
   - About notebooks, magics
3. Summarize simple Jupyter Notebook markdown including how to insert links and images?
   - Headings: `$`, `$$`, …
   - Links/images: `[Name](link)/![Name](/path/to/image.jpg)`
4. How can you typeset mathematical formulas including Greek letters using LaTeX Markdown in Jupyter Notebooks? LaTeX is used extensively used for documents with mathematical formulas.
   - Inline: `$x \sim \text{Binomial}(n, p)$` → $x \sim \text{Binomial}(n, p)$
   - Display: `$$ |x| = \begin{cases} -x, \quad x < 0 \\ x, \quad x \geq 0 \end{cases} $$` →
$$|x| = \begin{cases} -x, & x < 0 \\ x, & x \geq 0 \end{cases}$$
5. How prevalent is Unix or Linux in software development today?
   - In short, Unix/Linux is like the C of operating systems: We wouldn't have the more popular options without it as it's their foundation. It's used extensively for servers and non-pc devices, albeit has quite a few users (myself included) who prefer it's functionality over less transparent OS's such as MacOS and Microsoft.
6. What are the most popular programming languages and why?

- Python comes first since it is by far the easiest language to write. Others include Java, JS/CSS and HTML, C and C++.
- C is popular because it's about the closest you can get to writing assembly without having to write assembly - if you want a fast program that does exactly what you tell it to, C and C++ are the way to go. They also have extensive libraries which (once you know what they are) make writing code fairly easy
- Java is popular because of it's portability, but as of late I don't see ths being a standout feature of java over any other programming language. It is also my least favorite to write.

7. Describe the fundamental differences between C and Python.

- The main difference is that C is a low-level language (pretty much a step up from assembly) and Python is pretty much the highest-level language.

8. How challenging were learning C and completing Assignment 1 for you?

- It was initially very challenging but once I figured out pointers and got into a 'groove' with chatGPT I found it easier to use than Java, granted I was not using the stack or dynamic memory.

9. **Required**: What are your personal insights, aha moments, and epiphanies you experienced in the first part of the SENG 265 course?

- The most significant insight I've had so far is how important it is to know how to *really* know bash. After I had written a couple simple scripts to automate lab server tasks I had a moment of awe at how powerful it could be - you can automate absolutely anything that a computer could do as long as you can figure out how to do it.
- This is less of an epiphany, but still significant: After learning a bit of C for the first assignment, other languages are starting to make more sense, especially when it comes to pointers/pass-by-value/ref.

10. **Required**: How did you experience chatGPT as a learning tool?

- Introduction: Jupyter Magics
- A note on call-by-reference and call-by-value
- I've found chatGPT is extremely helpful for small, explicitly defined tasks. For example:
  - I had it write me a minimal .config file for conky, a system manager
  - I frequently have it write examples of code using specific functions to see how they're used in context
  - It usually gives a better response than google in far less time

Back to top

Citations and References

1. Custom commands for linux terminal, GeeksforGeeks (2018) https://www.geeksforgeeks.org/custom-commands-linux-terminal/
2. Where/how should I store scripts? Unix & Linux Stack Exchange (n.d.) https://unix.stackexchange.com/questions/604818/where-how-should-i-store-scripts
3. –help vs man command, Unix & Linux Stack Exchange (n.d.)https://unix.stackexchange.com/questions/86571/command-help-vs-man-command