

# Research Software Engineering Meetups: Operating Systems, Subprocesses, and Command Lines

2023/11/10

# Overview

- Schedule
- Short intro
- Each subtopic with interspersed practicals

# Schedule (provisional)

Date	Location	Rough Topic	Status
16th October	34-E-3180	Introduction & tooling	Done
23rd October	34-E-3180	Sharing, publication, and collaboration	Done
30th October	34-E-3180	Software Design: command-lines, UIs, and notebooks	Done
6th November	34-E-3180	Software Implementation: writing algorithms, assertions, tests	Done
13th November	34-E-3180	Operating Systems, Subprocesses, and Command Lines	In development
20th November	34-E-3180	Performance	Proposed

**source:** <https://github.com/adamkewley/rse-meetups/>

# Introduction

- You've got the tools (**1. tooling**)
- You've got a platform (**2. sharing, collaboration**)
- You have an idea of what you want to build (**3. design**)
- And you're in the process of making it (**4. implementation**)
- ... But you're not quite sure how these things *actually* run on a typical computer
- ... So you're not quite sure how to join up various scripts/programs (**integration**)

# Operating Systems

(intro)

Name some?

Which ones are good?

What do they do?

Everything	Advertise bing	Install updates at the worst possible time
Run machine code	Sandbox stuff (e.g. users, processes, filesystems)	Schedule task execution (concurrency)
Communicate with hardware (e.g. GPUs, WiFi)	Control Access to Hardware	Provide abstractions for all of the other tiles in this grid

# Operating Systems

(the bits you need to know)

- OSeS take an awfully long time to study – usually an entire undergraduate module
- But for practical purposes, this will just be an overview of:
  - Processes
  - Command-Line Arguments
  - Standard Input/Output/Error
  - Terminals
- All major OSeS provide those three things. Being aware of them can sometimes make debugging/running/integrating software easier

# Processes (overview)

## A task that the OS is managing (simplifying)

- The OS can only run machine code. It doesn't (usually) care whether it's running ``python.exe`` or ``firefox.exe``
- Processes may spawn sub-processes (recursive). Task managers (e.g. ``top`` in Linux) can show you the current process hierarchy
- They're the fundamental unit that almost all code runs inside of (ignoring hardware drivers, RTOS, etc.)



# Processes (practical)

- Find the native executable for your web browser of choice on your computer e.g.:
  - C:\Program Files\Mozilla Firefox\firefox.exe
  - C:\Program Files (x86)\Google\Chrome\Application\chrome.exe
- Google “python subprocess module”
- Use ``subprocess.run([command, arguments])`` to launch a browser subprocess
- Try providing ``https://google.com`` as an argument to the process

# Command-Line Arguments (overview)

**An array of strings that are passed to a process when it is started**

- Command-line arguments are one of the most common ways to change the behavior of a program at runtime without having to change the source code
- Examples: ``input_file.csv``, ``--help``, ``--algorithm=mcmc``

# Command-Line Arguments (practical)

- Write a new script that prints ``sys.argv`` (import `sys` first – google `sys.argv` for more info)
- Modify your script from the first practical to run ``python.exe`` with your second script as the first argument (e.g. ``subprocess.run(["path/to/python.exe", "path/to/your/script.py"])`)
- Ensure it runs
- Provide more arguments to your first script's argument list
- (this is one way of passing information between two processes)

# Command-Line Arguments (practical - bonus steps)

- Add a command-line parser to your sys.argv-printing script
- Run your script via the command-line, as well as your `subprocess` practical

```
1 import argparse
2
3 parser = argparse.ArgumentParser("myscript", description="it does awesome science stuff")
4 parser.add_argument("data_files", nargs="+")
5 parser.add_argument("-a", "--algorithm")
6
7 args = parser.parse_args()
8 print(args)
9
10 if args.algorithm != "mcmc":
11     raise RuntimeError("Wrong argument etc.")
12
```

# Standard Input/Output/Error

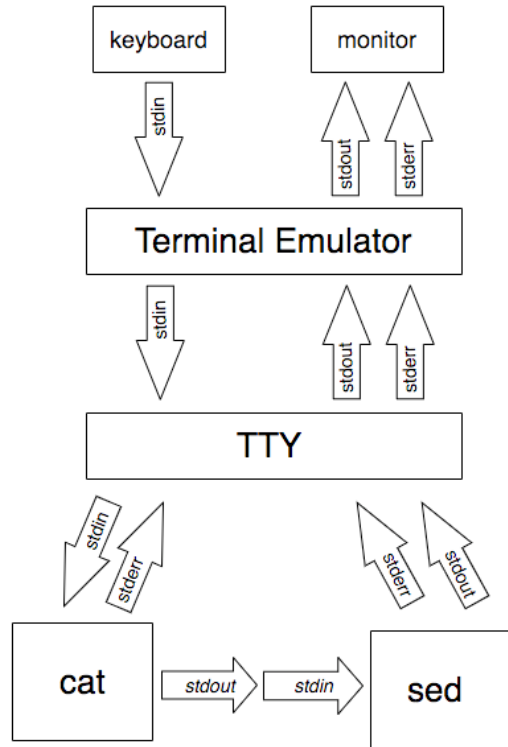
## Standard data streams for inputting/outputting data to/from a process

- E.g. in ``python`` you can read from the standard input with the ``input`` command:
  - ``name = input("What is your name?")``
- And you can write to the standard output with the ``print`` command:
  - ``print(f"Hello, {name}!")``
- And you can write to the standard error with the ``print`` command:
  - ``print("Error, no name entered!", file=sys.stderr)``

---

You can also read/write entire files etc. via these streams, and you can join the output stream of one process to the input stream of another.

# Standard Input/Output/Error



# Terminal [emulator] (overview)

## A program to launch/join/control processes

- Most of you probably write python in an IDE and run it from there
- And if you're unfamiliar with terminals, it's normal that they initially look very scary
- But terminals are a simple, standardized, way to run almost *any* non-GUI application – regardless of the language it was programmed in
- And because they're standardized across multiple languages, they're also a very good way to automate/integrate applications
- They provide a convenient syntax for launching processes:
  - **application ARG1 ARG2**
- And for joining them together:
  - **application ARG1 ARG2 | python script\_that\_reads\_input.py 2> errors.txt 1>output.txt**

# Terminal (practical)

- (for this, we'll use the ``git`` terminal, or your preferred bash terminal)  
<https://learnxinyminutes.com/docs/bash/>
- Use a terminal to run your script from the “Command-line arguments” practical, see if it prints **Command-Line Arguments**, etc.
- Then augment your script read ``sys.stdin`` in a for loop (i.e. ``for line in sys.stdin:``), exiting when given a unique word, such as “chicken”, or ``print``ing something otherwise (e.g. ``line.toupper()``)
- Run your script in a terminal (e.g. ``python yourscrip.py``) and make sure it works by typing stuff in the terminal (in a terminal, you're the input ;))
- Write another python script that prints stuff, possibly including “chicken”
- Use the terminal to pipe the output of your second script into the input of your first: ``python outputs_stuff.py | python yourscrip.py``
- Now switch ``python outputs_stuff.py`` for non-python commands, such as ``echo``, ``cat``, etc.



# Environment Variables (overview)

## Process-wide Key → Value storage

(e.g. PATH → ~/.local/bin:/usr/local/bin:/usr/bin)

- System-level environment variables are inherited by all processes on the system
- User-level environment variables are inherited by all user-initiated processes
- Process-level environment variables, are inherited by a given process, and all of its sub-processes

---

- The above three are merged together, top-to-bottom
- Overwrites are allowed (necessary, even)

# Environment Variables (practical)

- In your ``subprocess.run`` script (the first one)
  - Add a section that prints ``os.environ``
  - Run it and have a look at what environment variables your particular computer provides
- In the script your ``subprocess.run`` runs
  - Add a section that prints ``os.environ``
  - (it should print the same environment variables)
- Now set an environment variable in your top-level `subprocess.run` script with ``os.environ["MY_VAR"] = "some-var"``
- What does the sub-script now print?

# PATH (+ PYTHONPATH)

(the important environment variable)

## **A list of directories that processes can search through when looking for an executable**


- You can use ``subprocess.run`` to run an executable directly:
  - e.g. `subprocess.run(["C:\Path\To\Exe.exe"])`
- Or you can be looser, and ask it to run (e.g.) “python”:
  - e.g. `subprocess.run(["python"], shell=True)` # windows: `shell=True` required
- When you do the latter, the implementation needs a strategy to find ``python.exe`` on your computer
- The default strategy is to search through each directory in PATH until ``python.exe`` is found
- You'll find that many larger platforms (e.g. MATLAB, compilers, desktop software) will sometimes ask you to set PATH, or similar environment variables. The reason they do that is because they are inherited (so end up being propagated to all subprocesses inside MATLAB) and because it means that MATLAB can be looser about where external software is installed on your computer

# Summary

Lots of information (sorry about that)

- Operating systems run **native executables** (.exes, on Windows) as **processes**
- Processes can run other processes as **subprocesses**, forming a **process hierarchy**
- Processes almost always support
  - **Command-line arguments** – one-off list of strings given to one process
  - **Standard input/out/error** – IO streams that can be joined together
  - **Environment variables** – key-value store that is inherited by all subprocesses
- A **terminal** provides a standard way to launch a process:
  - **ENV\_VAR=hello PATH=extra:\${PATH} application ARG1 ARG2 > stdout.txt**

# Next Time (up for debate)

 **Note:** This list is just to give you an idea (it's very very provisional).

Topic	Description
Tools	IDEs, text editors, REPL, command line, basic git usage, LLMs (ChatGPT/GH Copilot)
Sharing, Publication, and Collaboration	Shared drives, GitHub, GitLab, writing a README, managing dependencies, releasing, publishing to package managers like <code>pip</code> , etc.
Software Design	Library vs. command-line vs. UI. User-/developer-experience. What is an application? The basics of how applications work.
Software Implementation	Iterative development methods. The REPL. Incremental application design. Assertions. Tests. How to implement things done more quickly - and with fewer surprises.
Libraries	Command-line parsers, configuration parsers, plotters, renderers
Advanced/Specialized topics (later)	Languages with type systems. Native application development (C/C++/Fortran /Rust). Multithreading. GP-GPU. Julia, hardware engineering, etc.