**Python**

* Development and coding are easier.
* More libraries for raster data processing.
* Use Pyodide for interpreting the code.
* https://www.earthdatascience.org/courses/use-data-open-source-python/intro-raster-data-python/

**C/C++**

* Learning curve for coding.
* Faster performance.
* Use Emscripten for compiling.
* File loading is done by preloading or embedding (Emscripten provides a virtual file system that simulates the local file system).
  + <https://emscripten.org/docs/porting/files/file_systems_overview.html#file-system-overview>
* Sample program for raster data
  + https://www.machinet.net/tutorial-eng/write-cpp-program-advanced-geospatial-analysis-system

**Zarr Dataset and Programs**

* Loaded data from S3 [Multi-Scale Ultra High Resolution (MUR) Sea Surface Temperature (SST) - Registry of Open Data on AWS](https://registry.opendata.aws/mur/).
* Loading and slicing done in JS.
* Tried in python as the tutorial from S3 Zarr link mentioned but couldn’t be done in Pyodide due to package loading problem. Some packages were loaded others had issue. The issue might be because of wrong wheel package.

**Download Raster Data and converting to Zarr file**

* Create account in CDS for API key
* Install CDSAPI in conda
* Run the python script

**MinIO instead of AWS S3**

* Used minio for storage of the zarr dataset from above.
* Updated JS code accordingly.

**Implementation of DataLoader in Pyodide**

* Tested a simple loader program in python.
* Implemented a modified version of the above to work with pyodide.

**Data to frontend**

* Tested a simple data to plot with pyodide.
* Modified it for raster data plotting, still need modification since it is just a straight line now.

**Modular Approach**

* Split the program as loading packages takes time. Split – (Import Packages, Data Loading, Processing, Plotting)
* The problem with plotting is solved by taking T2M data.

**Memory Usage**

WebAssembly (Wasm) does not create a separate virtual machine instance; instead, it runs within the JavaScript engine of the browser. As such, when inspecting memory usage or performance in the browser's developer tools, WebAssembly is considered part of the JavaScript environment.

Here's how WebAssembly's memory is typically represented in the browser's developer tools:

1. **Memory Tab**:
   * **JavaScript VM**: The memory used by WebAssembly is often reflected in the same area as JavaScript. This is because WebAssembly operates as a lower-level, more efficient representation of code that runs alongside JavaScript.
   * **Heap Snapshots**: If you take a heap snapshot, you might see memory usage attributed to "ArrayBuffer" or "TypedArray" objects. These are often used by WebAssembly for managing memory, as WebAssembly modules typically use a linear memory model that JavaScript exposes through these constructs.
2. **Performance Tab**:
   * **Call Tree and Flame Graph**: When profiling performance, you can sometimes see WebAssembly functions intermingled with JavaScript functions. They may be labeled with the .wasm extension or as "native" code, depending on the browser and the specific developer tools being used.
3. **Sources Tab**:
   * **WebAssembly Module**: The sources panel might list .wasm files, and you can inspect the compiled WebAssembly code here. However, this doesn't provide direct insight into memory usage but shows the source of the code.

To summarize, WebAssembly does not have its separate virtual machine; it runs within the JavaScript VM provided by the browser. Consequently, its memory and performance are generally tracked and reported alongside JavaScript in the developer tools.

**Caching Issues**

Pyodide runs in a WebAssembly (Wasm) virtual machine environment within the JavaScript runtime in your browser. This means that every time you load or refresh a page, a new instance of the Pyodide runtime is created, and it does not retain any state from previous instances.

**Key Points to Understand:**

1. **Isolation of Pyodide Instances:**
   * Each time you load a page, Pyodide starts fresh. It loads a new Python runtime instance, so any packages you installed or any variables you defined in a previous session are not preserved.
2. **Package Installation on Each Load:**
   * Since the Python environment is reset with each page load, you have to reload or reinstall all necessary packages every time the page is refreshed. This is why you see the need to reload packages like xarray, numpy, etc., each time.
3. **No Persistent Caching:**
   * Due to this isolated nature, even if you attempt to cache packages using sessionStorage or other browser-based storage, it won't prevent the need for the Pyodide environment to load and initialize these packages in each session. The packages need to be loaded into the new Python VM on every page load.

**Implications:**

* **Performance:** Package loading and initialization can be slow, especially for large packages, because this process has to occur each time the page is loaded.
* **No Persistence:** You can't persist the Python environment across page reloads or between different pages unless you serialize and deserialize some states manually, which is complex and still wouldn't avoid package reinstallation.

**Possible Optimization:**

* **Preload Packages:** One way to mitigate load times is to load Pyodide and required packages as early as possible, possibly in a hidden or background process, to make them ready when the user needs them.
* **Minimal Package Loading:** Only load the packages that are absolutely necessary for the specific operations you're performing to reduce initialization time.

**Conclusion:**

Given Pyodide's design, every page load will reinitialize the Python environment. Understanding this behavior is crucial when developing web applications that rely on Pyodide, as it directly impacts performance and how you structure your code.

**Next Steps to be done**

**Questions**

* Why object store is used from the given 3 types.