# Designing an Open-SAS VS Code Extension for SAS Program Execution

**Goal:** To transform the **Open-SAS** project into a full-featured Visual Studio Code extension. This extension will allow users to open .osas files in VS Code, write SAS code (using SAS syntax), and run those programs directly within the editor. The execution will be powered by an underlying Python-based interpreter (the Open-SAS backend), so the user does not need a licensed SAS installation or to run Python scripts manually. The result should be a **feasible open-source alternative to SAS**, supporting core SAS data step functionality, key PROC procedures, the macro language, and integration with the VS Code UI (syntax highlighting, error feedback, etc.). Below we outline the detailed steps and components required to achieve this.

## 1. Project Overview and Architecture

* **Open-SAS Interpreter:** First, assess the current state of the Open-SAS project (the Python interpreter for SAS syntax). Identify which SAS features it already supports and which need to be added or improved. Key components likely include a parser for SAS code, execution engine (probably using Pandas/Polars for data), and implementations of some SAS procedures (PROCs).
* **VS Code Extension:** Plan the VS Code extension structure. The extension will consist of:
* A **language support component** for SAS/OSAS syntax (for highlighting, snippets, etc.).
* A **runtime execution component** that invokes the Open-SAS Python interpreter when the user runs a .osas file.
* Configuration for file type .osas to be recognized and associated with the SAS language mode.
* Possibly a **debug adapter** or at least an output viewer for results and logs.
* **User Workflow:** The user should be able to:
* Install the “Open-SAS” extension from the VS Code marketplace.
* Open or create a .osas file in VS Code – which will be treated as a SAS program. The editor should provide SAS syntax highlighting (and ideally some auto-completion and diagnostics similar to SAS language support).
* Run the program (via a command or “Run” button in VS Code). This triggers the extension to call the Open-SAS Python backend, execute the SAS code, and then display outputs (log messages, results, any output data sets created).
* View outputs: For example, see textual results of PROC analyses in a VS Code output panel, and have any output data sets saved to a workspace directory on disk (so the user can inspect them or use in subsequent code).
* **No Manual Python/CLI Steps:** The extension should abstract away all Python commands. Users won’t run Python scripts or CLI tools themselves; they just press run in VS Code. Under the hood, however, the extension will spawn a Python process to execute the code using the Open-SAS interpreter.

## 2. VS Code Extension Setup (Language Support and Commands)

**Language Identification:** Register a new language in the extension manifest for SAS/OSAS code. Associate .osas file extension with this language. This allows VS Code to recognize .osas files and apply syntax rules.

**Syntax Highlighting:** Provide a TextMate grammar or similar for SAS syntax. SAS syntax highlighting is well-understood (keywords like DATA, PROC, RUN, etc., comments \*...; and /\*...\*/, macro syntax with % and & tokens, etc.). We may leverage existing SAS VS Code grammars. In fact, SAS Institute provides an official VS Code extension with SAS language support (syntax highlighting, code completion, hover help, etc.[[1]](https://github.com/sassoftware#:~:text=This%20SAS%20Extension%20for%20Visual,folding%2C%20outline%2C%20SAS%20code%20sni%E2%80%A6)). We can use a similar grammar definition for our extension to color-code SAS syntax appropriately. This ensures .osas programs *look* like SAS in the editor (making it easier for SAS users to adopt).

**Editor Features:** Beyond highlighting, consider simple editor features: - **Snippets or Auto-completion:** We can include snippets for common SAS constructs (e.g. a snippet for PROC MEANS block or DATA step structure). - **Outline/Navigation:** Optionally, detect PROC and DATA blocks for a document outline. - **Hover Help:** Optionally, basic hover info for SAS keywords (could be added later or reuse documentation).

These make the extension feel polished, but core effort is highlighting and running code. Initially, focusing on highlighting and run/debug is priority.

**Run Command Implementation:** Create a VS Code command (e.g., “Open-SAS: Run Program”) accessible via the Command Palette and a toolbar button (the play icon) when a .osas file is open. This command will: - Save the current file (ensure latest code is on disk). - Invoke the Python backend to execute the file. This likely means spawning a child process running the Python interpreter with the Open-SAS execution module.

**Spawning the Python Process:** Use Node’s child\_process.spawn or exec to run Python. For example:

import { spawn } from 'child\_process';  
const pythonProcess = spawn('python', ['-u', '/path/to/osas\_runner.py', programPath]);

Here, osas\_runner.py would be a Python script (packaged with the extension or installed) that loads the Open-SAS interpreter and runs the given program file. We include the -u flag (unbuffered mode) so that output is flushed in real-time[[2]](https://stackoverflow.com/questions/68794540/vscode-extension-executes-local-python-bash-code-and-append-to-output-channel#:~:text=After%20some%20more%20digging%2C%20I,to%20unbuffer%20the%20python%20output). This way, as the SAS program produces log or output lines, we can stream them live to the VS Code UI rather than waiting until the process ends.

**Output Capture:** Create an **Output Channel** in VS Code (e.g., vscode.window.createOutputChannel("Open-SAS Output");). As the Python process runs: - Capture stdout data events and append to the output channel[[3]](https://stackoverflow.com/questions/68794540/vscode-extension-executes-local-python-bash-code-and-append-to-output-channel#:~:text=export%20function%20execute,). - Capture stderr similarly (perhaps to the same channel or a separate “Open-SAS Log” channel to distinguish errors). - For real-time feedback, each time data arrives from the process, immediately append it so the user sees the SAS log/results streaming as if in SAS console. Using unbuffered Python mode (as noted) is important to achieve this.

**Termination and Errors:** When the process exits, check exit code. If non-zero, indicate an error (the log likely already shows the error details from the SAS program or interpreter). We might also implement basic error location: if the interpreter can output the line number of a syntax error, the extension could parse that and use VS Code’s diagnostics API to highlight the offending line in the editor.

**No External Dependencies for User:** To ensure a seamless experience, the extension should manage the Python environment: - **Bundling vs. Relying on System Python:** Ideally, package the Open-SAS interpreter so that users do not have to manually install Python packages. There are a few approaches: - **Use system Python:** Require that the user has Python (of a specific minimum version, e.g., 3.9+) installed. On first run, the extension can check for Python and prompt if not found. Then it can automatically install the open-sas package (if published to PyPI) or include the Python code in the extension. - **Bundle a Python environment:** It’s possible to ship a minimal Python runtime with the extension or use Node to run the Python code via something like Pyodide (WebAssembly Python) if feasible. However, bundling CPython increases extension size and complexity. A middle ground is to create a dedicated virtual environment upon extension install. - **Isolated Virtual Env:** To avoid version conflicts with the user’s other Python packages, consider creating a venv under the extension directory and installing Open-SAS and dependencies there. The extension can then call that specific python interpreter. This ensures the Open-SAS backend uses the correct library versions and doesn’t clash with other Python uses on the system.

**Configuration:** Provide extension settings for things like: - Path to Python interpreter (if user wants to override). - Choice of backend engine (Pandas vs Polars, if we allow switching – more on that below). - Default data library directory (path for WORK or for libname mappings if desired). - Log level or verbosity settings.

## 3. Data Engine: Pandas vs. Polars Decision

The core interpreter will rely on a DataFrame library to handle dataset operations (similar to how SAS datasets work). We have two primary candidates: - **Pandas:** Widely used Python library for data manipulation (tables, grouping, joins, etc.). - **Polars:** A newer Rust-based DataFrame library with Python bindings, known for high performance on large data.

**Performance Consideration:** Polars is **significantly faster** and more memory efficient for many common operations – often 5-10× faster than Pandas, and in some cases up to 100× faster on large data[[4]](https://blog.jetbrains.com/pycharm/2024/07/polars-vs-pandas/#:~:text=In%20a%20word%3A%20performance,4%20times%20needed%20for%20Polars). Polars can also handle larger-than-memory datasets more gracefully and utilize multiple CPU cores by default[[5]](https://blog.jetbrains.com/pycharm/2024/07/polars-vs-pandas/#:~:text=One%20of%20the%20other%20advantages,about%20how%20Polars%20achieves%20this). This suggests Polars would be a strong choice for building a SAS-like system aimed at big data analytics.

**Feature and Ecosystem Consideration:** Pandas, however, is extremely feature-rich and has a large ecosystem: - Many statistical libraries (e.g., statsmodels, scikit-learn) expect Pandas DataFrames or NumPy arrays as input. If we implement procedures like GLM (regressions), connecting with these libraries might be simpler with Pandas. - Pandas has built-in capabilities that map to SAS functionality (e.g., merging datasets, grouping and summarizing for PROC MEANS, pivoting for PROC TRANSPOSE, etc.) and plenty of built-in date/time and categorical handling. - Polars API is similar but not identical; it may lack some of the more specialized statistical routines (though basic aggregation and joins are well supported).

**Decision:** For the initial implementation, using **Pandas** might be more straightforward given its maturity and compatibility. We can implement the SAS features on top of Pandas DataFrames (and possibly NumPy/SciPy for certain stats). Once correctness and functionality are achieved, we could consider optimizing using Polars later, or allow the engine to be swappable via a setting. (If performance is a key concern from the start, using Polars is viable – but the team must be ready to implement certain analytics routines manually or via other libraries, since Polars focuses on data manipulation rather than statistical modeling).

Regardless of choice, ensure the library is included in the extension’s Python environment (list it in requirements). If using Pandas, target a stable version (e.g., Pandas 2.x, which even uses Apache Arrow in the backend for performance). If using Polars, ensure we pick a version and test for compatibility with the rest of our code.

## 4. Implementing SAS **Data Step** Functionality

The **DATA step** is the heart of SAS programming. We need to implement the majority of data step features so that typical SAS data manipulation code runs correctly. Key aspects to cover:

* **Reading Data Sets:** In SAS, a DATA step usually reads from one or more sources (SET, MERGE, or reading raw data). For our interpreter:
* Implement SET statement to read from an existing dataset (which would be, in our system, likely a CSV or Parquet file on disk, or a DataFrame held in memory from a prior step).
* Implement MERGE for merging two or more datasets by key (respect SAS behavior: if a BY statement is used, perform a keyed merge, akin to SQL join; if no BY, perform simple interleaving merge).
* Optionally, support INFILE/INPUT for reading raw text data, though this is less crucial if users can load data via CSV through a simpler method. We might initially skip complex raw data parsing.
* **Column Operations:** The data step allows creating new columns, recoding values, dropping/keeping columns, etc. Using the DataFrame library, we can implement:
* Assignment statements (e.g., NewVar = expression;) to create or replace columns. Parse the expression using Python’s eval or a custom evaluator to apply it on the DataFrame (taking care to vectorize operations).
* Functions: Common SAS functions (SUM, MEAN, etc.) should be mapped to their Pandas equivalents (or implemented). Many SAS functions (date functions, math functions) have counterparts in Python’s libraries (e.g., use numpy.mean or Pandas series methods).
* Conditional logic: SAS uses IF/THEN/ELSE within data step to conditionally set values or output records. We will parse IF conditions and apply boolean filtering on the DataFrame to assign values accordingly. For example, if age < 18 then minor = 1; else minor = 0; can translate to something like:
* df['minor'] = 0  
  df.loc[df['age'] < 18, 'minor'] = 1
* This needs to be done carefully to mimic SAS (which processes row by row logically, but we can use vectorized operations).
* DO loops: SAS data step can have loop constructs (do i=1 to 10; ... end; or looping over arrays). Implementing loops that generate multiple output observations (e.g., output multiple times per iteration) is tricky in a vectorized world. We might need to simulate by expanding the DataFrame (for example, if a loop outputs multiple times, we may need to repeat records). This is an advanced feature; initially focus on loops used for iterative calculations within a single row (which could often be done with vectorized operations or apply functions).
* **BY-group processing:** If a DATA step has a BY statement (usually when data is sorted by certain keys), SAS generates automatic flags like FIRST. and LAST. for group boundaries. We should implement FIRST.<var> and LAST.<var> as boolean indicators when data is grouped by the BY variable. In Pandas, this can be achieved by using .groupby and identifying first and last indices of each group. We can add corresponding columns in the DataFrame to represent these.
* **Retention of values:** SAS’s retain statement keeps variable values across iterations. In a vectorized approach, this may correspond to shifting or cumulative operations. We might simulate retain by initial value setting and then fill forward logic.
* **Output and Return:** In SAS, each iteration of the data step writes an output observation by default. The OUTPUT statement can control this (write additional outputs or prevent output). We need to decide how to simulate the row-wise output behavior. A straightforward way in a batch data framework is to construct the resulting DataFrame entirely by the end of processing. We might not simulate record-by-record streaming output (not typically needed). But we must honor if output; is suppressed (like within conditional to skip output).
* **Drop/Keep/Rename:** SAS data step syntax includes DROP=, KEEP=, RENAME= data set options and statements. We should implement support to drop or keep specific columns in the output dataset easily (Pandas can drop columns or filter columns; implement these after processing, based on metadata).
* **Format and Informats:** Full SAS format support is complex (user-defined formats etc.), so initially we may skip or implement only basic numeric format rounding. The core functionality doesn’t strictly require formats for the data to be processed (they mostly affect presentation).

**Testing Data Step:** We should gather some sample SAS data step code (with IF/THEN, loops, BY-group logic, etc.) and ensure our interpreter reproduces the expected results. This will validate our data step implementation is “complete” enough for common tasks.

## 5. Implementing Key PROC Procedures (Analytics & Data Management)

A major part of making Open-SAS a viable alternative is supporting the most commonly used SAS procedures. We will prioritize **Base SAS procs and essential statistical procs**. Research shows that procedures like PROC FREQ and PROC MEANS are among the most frequently used in SAS analytics[[6]](https://support.sas.com/resources/papers/proceedings/proceedings/sugi30/263-30.pdf#:~:text=Procs%20FREQ%20and%20MEANS%20have,where%20they%20may%20be%20most), so those are high priority. The user specifically requested support for *“top PROCs”* such as PROC MEANS, PROC UNIVARIATE, PROC CONTENTS, PROC TIMESERIES, PROC GLM, etc. Below, we break down the implementation approach for each of these and other important ones:

* **PROC PRINT:** (Not explicitly mentioned, but fundamental) – Simply display the contents of a dataset (perhaps first N rows by default). Implement by pretty-printing the DataFrame. In VS Code, output can be text (formatted table) or potentially we could render as markdown table for nicer view. A simple text table is fine to start.
* **PROC CONTENTS:** This procedure displays dataset metadata – variable names, types, lengths, etc. We can implement a function that, given a DataFrame, prints out its schema. E.g., list columns with their data type (numeric or character), and perhaps number of observations. This is straightforward: df.info() in Pandas provides some info but we'll format it SAS-style. For example, SAS shows variables, type, length, and labels if any. Since our data is in Python, we might not enforce “length” for character (pandas doesn’t have fixed lengths), but we can display max length or simply note “string”. *Example:* *“proc contents data=MyData;”* would output a list of variables with attributes[[7]](http://webhome.auburn.edu/~zengpen/teaching/STAT-6110/pdf/01C-proc.pdf#:~:text=match%20at%20L80%20Commonly,Dataset)[[8]](http://webhome.auburn.edu/~zengpen/teaching/STAT-6110/pdf/01C-proc.pdf#:~:text=Commonly,tables%20for%20var1%20and%20var2).
* **PROC MEANS:** Calculates descriptive statistics for numeric variables – count, mean, std, min, max by default, optionally other percentiles. In Pandas, df.describe() provides count/mean/std/min/25%/50%/75%/max, which covers many of these. We will implement PROC MEANS by grouping (if a BY statement or CLASS variables are specified) and computing aggregates. For each analysis variable:
* Compute N, Mean, Std Dev, Min, Max by default.
* Support statements like VAR to choose which variables, and options like MAXDEC= for decimal places (we can format output accordingly).
* Support BY groups: Pandas groupby can handle that easily, computing stats per group.
* Output can be shown as tables in the output channel. If OUTPUT OUT=dataset option is used, we should capture the result stats in a DataFrame and save it as a dataset.
* **PROC FREQ:** (Not explicitly requested but essential) – Generates frequency tables for categorical variables and cross-tabulations. Implementation:
* For each variable (or combination if TABLES X\*Y) count distinct values.
* Pandas value\_counts() for one-way frequencies; for two-way, use pd.crosstab or groupby with size.
* Include percentages if requested (SAS by default shows frequency and percent). This can be computed from counts.
* For higher-dimensional tables (n-way), can nest groupby.
* If statistical tests (Chi-square, etc.) are desired for 2-way, we might use SciPy or statsmodels to compute them.
* Initially focus on basic frequency counts and percentages.
* **PROC UNIVARIATE:** Provides detailed summary for one variable at a time: all descriptive stats plus distribution measures (skewness, kurtosis), extreme observations, and normality tests. We can implement parts:
* Many stats (mean, median, mode, std, variance, range, quartiles, etc.) can be computed via Pandas/NumPy.
* Quantiles and percentiles (e.g., P1, P5, P95, etc.) can be obtained with numpy.percentile.
* If needed, use SciPy for tests like Shapiro-Wilk for normality, etc., or skip initially.
* We can output a structured report text similar to SAS’s output.
* Since PROC UNIVARIATE often prints a lot, we ensure formatting is clear (maybe break into sections: Moments, Basic Statistical Measures, Tests for Normality, etc., similar to SAS layout).
* **PROC SORT:** Sorts a dataset by one or more variables. In Pandas, simply use df.sort\_values(by=[...]). We should support OUT= to write the sorted result to a new dataset (or replace the original if no OUT given, similar to SAS default behavior). Also support DESCENDING keywords for sort order. Sorting is straightforward to implement.
* **PROC DATASETS:** Used to manage data libraries (list datasets, rename, delete, etc.). We may implement a limited subset:
* At least PROC DATASETS; CONTENTS DATA=lib.\_ALL\_; which lists all datasets in a library. We can map that to listing files in the library directory.
* DELETE or SAVE statements to delete datasets – corresponding to removing files.
* This is more of a nice-to-have for completeness of managing the workspace.
* **PROC SQL:** SAS has a whole SQL environment. This might be too large to implement fully at first. Users can often achieve similar results with data step or other procs. We might skip direct PROC SQL support in initial version to focus on data step and basic procs. (Alternatively, we could internally use Python’s sqlite3 or Pandas query to handle simple SELECTs if needed. But full SQL support would be another big project.)
* **Statistical PROCs (GLM, REG, etc.):**
* **PROC GLM**: General Linear Models (ANOVA, regression). Implementing this completely is complex, but we can leverage Python’s statsmodels or sklearn:
  + For linear regression: use statsmodels.OLS or LinearRegression from sklearn to fit models.
  + Extract outputs: parameter estimates, ANOVA table, R-squared, etc., and format similar to SAS output (Parameter Estimates table, Analysis of Variance table).
  + Support CLASS (categorical) variables by creating dummy variables (statsmodels can do this via formulas).
  + Likely focus on least squares linear regression and ANOVA in initial support. More advanced features (random effects, etc.) can be skipped.
* **PROC REG** (if needed separately) – similar to GLM for regression specifically.
* **PROC LOGISTIC** (if ever needed) – could map to logistic regression in statsmodels.
* We list GLM because user specifically mentioned it; ensure at least linear regression and ANOVA are covered.
* **PROC TIMESERIES / Time Series Analysis:** SAS/ETS has various procs (like PROC TIMESERIES, PROC ARIMA, etc.). For “timeseries” as given:
* Possibly the user wants the ability to do time-series data prep and basic forecasting.
* We can use **Pandas** or **statsmodels** for this:
  + Compute time-based summaries or transformations (e.g., moving averages, differencing) with Pandas built-in rolling or diff functions.
  + For forecasting, statsmodels’ ARIMA or exponential smoothing can provide similar outputs.
* If focusing on PROC TIMESERIES specifically: this SAS procedure can generate summary statistics over time intervals, detect seasonality, etc. We might implement a simpler subset: resampling data to a time frequency, computing stats per period, etc., using Pandas resample/rolling.
* Advanced time series (like ARIMA modeling results) might be more akin to PROC ARIMA or PROC FORECAST. It may be enough to demonstrate we can do basic time series operations. We should clarify with the user if possible which specific functionality they need. In absence, implement core abilities:
  + Read a time series dataset (with a datetime index).
  + Produce simple plots or stats (maybe outside scope for now).
  + Possibly output some forecast values (if using a library).
* Since graphical output (like actual time series plots) would be complex to show in text, we might skip plotting and focus on numeric results.
* **Additional Useful PROCs:**
* **PROC TRANSPOSE**: transposing a dataset (wide-to-long or vice versa). Can implement with Pandas df.transpose() or pd.melt/pivot depending on use.
* **PROC IMPORT/EXPORT**: to read/write CSV, Excel, etc. For user convenience, maybe provide a simplified way to import data. However, since this is an extension, the user could also use Python code inside if absolutely needed. Perhaps not needed if we allow reading CSV via a data step or a small utility.
* **PROC MACONTENT** (just kidding – macro is separate topic below).

For each implemented PROC, thoroughly test with known SAS examples: - e.g., compare a simple PROC MEANS on a dataset with the Pandas describe output to ensure we include the same statistics and that results match (within rounding error). - Validate PROC GLM by comparing with an equivalent statsmodels regression on a small dataset to ensure coefficients and p-values match SAS outputs (SAS outputs might use Type I/III SS for ANOVA, etc., but we can aim for Type III sums of squares and typical regression output).

All these procedure implementations will live in the Python backend (Open-SAS code). The VS Code extension will just call into them and display results. It’s wise to have the Open-SAS backend structured as a library (with functions or classes for each PROC and the data step), so the extension’s runner script can simply import and invoke them.

## 6. SAS Macro Language Support

Supporting the SAS macro facility is crucial for compatibility with many real-world SAS programs. The macro language allows for dynamic code generation and is processed *before* the actual data step/PROC execution. Key tasks for implementation:

* **Macro Variable Processing:** Implement the resolution of macro variables (&var references) and macro variable assignment:
* %LET statements to assign global macro variables (store them in a dictionary).
* References like &VAR in the code should be replaced by their value *before execution* of that step. This means our interpreter should first scan through the code for & references and substitute values from the macro var table.
* Keep in mind macro variables can be created during execution too (e.g., via CALL SYMPUT in a data step or PROC SQL INTO: – these might be advanced to handle later). At least handle %LET and %PUT (which just prints a message or macro var value to the log).
* **Macro Definitions:** Support %MACRO name(param1=..., ...) ... %MEND;:
* When encountering a %MACRO definition, the lines between it and %MEND should be stored (in an internal structure) as the macro’s definition (essentially like a function body).
* Macro parameters (positional or keyword) should be handled: replace &param with the passed values when the macro is invoked.
* Macro scope: We can keep it simple and treat all macros as global (SAS has local macro variables and macro recursion, which we might avoid initially).
* **Macro Invocation:** When %macroname is called (with optional arguments), the interpreter’s macro processor should:
* Take the stored macro definition, substitute in any parameters and global macro vars referenced, and generate the corresponding SAS code lines.
* Insert those lines into the program for execution. Essentially, the macro call is replaced by the macro’s body (with substitutions) in the code stream.
* **Conditional Logic in Macros:** Macros have their own %IF %THEN %ELSE (which are evaluated during macro execution, not to be confused with data step IF). Also %DO loops in macro. We should implement a basic parser for these within macro definitions:
* For example, %IF &VAR = value %THEN %PUT ...; should check the macro variable value at macro processing time and decide whether to generate the %PUT (or other code).
* %DO i=1 %TO 10; ... %END; inside a macro should replicate the contained text multiple times with &i substituted each time.
* Essentially, our macro processor acts like a simple scripting engine that can process these directives and manipulate the code text.
* **Nesting and Macro Quoting:** Advanced macro features (like masking special characters via %STR, etc.) are probably beyond initial scope. We can document that some complex macro behaviors might not be fully supported initially. Focus on the common use cases: generating repetitive code, using macros as functions (passing parameters), and simple conditional or iterative macro logic.

The macro facility is a **string-based code generator**[[9]](https://support.sas.com/publishing/pubcat/chaps/61885.pdf#:~:text=The%20macro%20facility%20is%20a,does%20not%20support%20the%20use). We may implement it by reading the entire program as text and performing a two-pass approach: 1. **Macro resolution pass:** Handle all macro definitions and expansions, producing a fully expanded SAS code (with no macro directives, just “open code”). 2. **Execution pass:** Then feed that expanded code into the data step/PROC interpreter for actual execution.

This separation helps, as we can reuse the macro expansion logic independently. It should mimic SAS: *“When you use a macro facility name in a SAS program ... the macro facility generates SAS statements and commands as needed. The rest of SAS receives those statements and uses them as if you entered them.”*[[10]](https://support.sas.com/publishing/pubcat/chaps/61885.pdf#:~:text=When%20you%20use%20a%20macro,communicate%20with%20the%20macro%20processor). We essentially do the same: our macro processor will inject generated statements into the code flow.

We will test macro processing with simple scenarios: - Define a macro that generates a data step, call it, and ensure the output dataset is created. - Use %LET and &var in code (e.g., %let cutoff=10; data a; set b; if x > &cutoff then output; run;) – check that &cutoff is replaced with 10 in the condition. - Recursion or macro referencing macro (maybe not initially). - Macro generating multiple procs or data steps to ensure overall integration.

## 7. Data Storage and LIBNAME Handling

In SAS, data sets are stored in libraries (which correspond to directories). The extension should define how datasets are saved and referenced: - **Default Library (Work):** SAS has a temporary WORK library for datasets that are not saved permanently. We can simulate this by using a default directory (maybe the project’s root or a temp folder) for any dataset not explicitly given a library. If the user doesn’t specify a libname, we treat it as WORK library which is ephemeral (could be cleared each session or kept until VS Code closes). For simplicity, using the workspace root as the default data directory is fine (or a subfolder like work/). - **LIBNAME Statement:** Implement LIBNAME name 'path'; so the user can map a name to an actual directory path on their file system. Then, when code references name.dataset, we know to look in that path for a file corresponding to dataset. - We will maintain a mapping of libnames to directories. The default LIBNAME WORK could map to a temp dir or workspace. - Also support LIBNAME without arguments to list current libraries, if needed (not critical).

* **Dataset Storage Format:** Decide on a file format for saved datasets. Options:
* **CSV:** Easy and human-readable, but doesn’t preserve types/metadata well (e.g., might guess types on read).
* **Parquet/Feather:** Columnar, efficient, preserves schema, good for large data.
* **Pickle (Python-specific):** Could preserve exactly, but not cross-language and can be brittle across versions.
* **SAS7BDAT:** Writing SAS’s proprietary format is possible via pyreadstat library, but not necessary unless we need true SAS compatibility. It adds complexity and dependency.

Using **Parquet** is a good modern choice (fast, compressed, widely supported). We can implement reading/writing via Pandas (df.to\_parquet() and pd.read\_parquet()). - When a data step or proc produces an output dataset, save it as <libdir>/<data\_name>.parquet (or maybe <name>.csv if small, but Parquet is better). - When a data step does set lib.dataset, we load from <libdir>/dataset.parquet. - Document this: so users know the data persists as Parquet files. (We might allow an option to use CSV for transparency or if Parquet not available, but Parquet is fine if we include pyarrow).

* **Naming Conventions:** SAS datasets names are case-insensitive and <=32 chars. We should handle name case consistently (maybe treat everything lowercase internally).
* **Cleaning up:** If using a temporary WORK library, clear it on session start (delete old files to avoid stale data interfering).

By implementing LIBNAME and persistent storage, users can build workflows where intermediate data sets are saved and reused, just like in SAS. For example:

libname proj "C:\project\data";  
data proj.cleaned;  
 set proj.rawdata;  
 /\* cleaning logic \*/  
run;

This would read rawdata.parquet from C:\project\data, process it, and write cleaned.parquet in the same location.

## 8. Packaging the Python Backend and Managing Dependencies

To ensure the extension works reliably, we must package or install the Open-SAS Python backend correctly: - **As a Python Package:** Ideally, the Open-SAS interpreter (data step + procs implementation) is structured as a Python package (with a setup.py or pyproject.toml). We can publish it to PyPI (e.g., as opensas library). Then the extension can simply pip install opensas in its environment. This makes updates easier (just update the version). If the code is not on PyPI, we could include it as part of the extension (e.g. ship the .py files in extension’s folder). - **Dependency Versions:** Pin specific versions of Pandas/Polars, statsmodels, etc., that are known to work. The extension’s internal environment should use those versions to avoid conflicts. We need to monitor Python compatibility (e.g., ensure compatibility with Python 3.11+ if that’s current, and update the package when needed). - **Python Version Handling:** Document which Python versions are supported (e.g., 3.9 to 3.12). Possibly include checks in the extension that warn if an older Python is selected. For seamless migration, ensure our code doesn’t use deprecated APIs that will break in upcoming versions. Regularly test with new Python releases. - **Isolation:** Because VS Code might have other Python extensions or user may run other Python code, keep our environment separate to avoid any contamination. This prevents issues where the user’s environment lacks a dependency or has a different version.

* **No CLI Required:** The user never directly invokes pip or python. The extension’s activation can handle environment setup:
* On install/activate, if needed, create venv and install dependencies.
* Provide a command to manually reinstall/upgrade the backend in case of issues (for support).
* If the extension can include pre-built wheels for dependencies (to avoid lengthy pip install), that’s a bonus, but not strictly necessary.
* **Testing Packaging:** After setting this up, test on a clean machine: install extension, and try running a simple .osas file. It should prompt for Python if not found, set up environment, and successfully execute the code.
* **Continuous Updates:** We should plan for releasing updates to both the extension and the Open-SAS Python package. If a new PROC is added or bug fixed, bump the version and have the extension update. Possibly integrate with VS Code extension’s auto-update mechanism for extensions and have it trigger pip update for the package if needed.

## 9. Editor Integration: Error Highlighting and Debugging Aids

To improve user experience, implement features to catch and display errors, mimicking SAS’s feedback: - **Syntax Errors:** If the user writes incorrect SAS syntax, our interpreter’s parser should throw an error with a message and ideally position (line number). We can intercept that and surface it in VS Code: - Use the vscode.Diagnostic API to underline the error in the editor and show the message. For example, if a user forgets a semicolon, the parser might report “ERROR: Statement not terminated with semicolon at line X.” We output that in the log and also mark line X with a red squiggly and that message. - This requires that our parser is implemented to catch errors gracefully rather than just crash. So building a robust parser (maybe with a library like ANTLR or a hand-written recursive descent) is important. We could even leverage an existing SAS grammar if available.

* **Runtime Errors/Warnings:** If during execution there are issues (e.g., divide by zero, missing data causing calculations to fail, etc.), log them to the output channel as SAS would (SAS prints notes/warnings to the log). We can tag them with prefixes “NOTE:”, “WARNING:”, “ERROR:” just like SAS. Perhaps color-code them in the output (maybe not trivial in output channel, but we could simply use all-caps to make them stand out).
* **Debugging Capabilities:** Full step-through debugging (setting breakpoints, examining variable values mid-step) is a **huge** feature that might be beyond initial scope. However, we can lay groundwork:
* Possibly implement a mode where the interpreter runs in a step-by-step manner and communicates back to the extension (e.g., a basic debug adapter protocol implementation). This would be advanced; initially we might skip.
* A simpler approach: provide a way for users to insert something like PUT statements in code to print variable values (SAS uses PUT for debugging prints). We can support %PUT for macro vars and PUT in data step to output to the log (we’ll just print those lines).
* If syntax is wrong or unsupported (not yet implemented feature), we should clearly message that so the user knows it’s an Open-SAS limitation, not just silent failure.

**Syntax Coloring for Macros:** Ensure the syntax highlighter covers macro keywords (%macro, %mend, etc.) and macro variables (&something) distinctly, to help users spot them. The official SAS VSCode extension likely covers that, and we can follow suit.

**Testing Editor Integration:** - Deliberately introduce a syntax error in a .osas file (e.g., omit a semicolon) and run – confirm that the error appears in the output and that the extension highlights it. - Try an unsupported PROC – extension should catch it and perhaps output “ERROR: PROC XXX is not implemented” rather than crashing. - If possible, also support a “dry run” or syntax-check-only mode (SAS has an OPTIONS OBS=0 or SYNTAXCHECK mode). We could have a command “Open-SAS: Check Syntax” that runs the parser without executing, to quickly give feedback on errors.

## 10. Verification with Real SAS Examples and Edge Cases

To reach "full maturity," we must rigorously test the extension and interpreter with real-world SAS code: - Collect sample SAS programs that cover a wide range of functionality: reading data, multiple data steps, macro-driven code, various procs. For example, sample code from SAS documentation for PROC MEANS, PROC GLM (like an ANOVA example), etc. - Ensure that running these in our extension produces results **consistent with SAS** (at least for the features we claim to support). Some minor differences in output format are acceptable, but statistics and data results should align. - Compare performance on large data with SAS if possible. If our approach is significantly slower, document that and consider optimizations (e.g., using Polars or improving algorithms).

Also test integration in VS Code on multiple platforms (Windows, Mac, Linux) since path and Python setup differences can affect the experience (especially around launching Python and library dependencies).

## 11. Publishing and Documentation

Once implementation and testing are done: - Prepare to **publish the extension** on the VS Code Marketplace. Ensure all licensing is correct (Open-SAS likely MIT or similar, and any third-party libs have acceptable licenses). The extension’s README should clearly state what is supported. - Provide usage documentation: how to use libname, which PROCs are available, any known limitations (for example, “Graphs are not supported” or “Some statistical tests in PROC UNIVARIATE are not yet implemented”), etc. - Include example .osas programs in the repo or as part of extension samples that users can refer to. - Possibly include a link to the Open-SAS GitHub for users to report issues or contribute.

Finally, inform the user community that this extension exists as an open-source SAS alternative. The ultimate goal is that a user can search in VS Code for “SAS” or “Open SAS”, install this extension, and effectively run SAS programs without an official SAS installation, achieving much of the same functionality.

By following all the steps above, **Cursor** (or any developer) should have a clear blueprint to implement the Open-SAS VS Code extension successfully, bringing the project to full maturity. This will deliver a tool where .osas files can be executed seamlessly inside VS Code with authentic SAS syntax and a robust Python-powered backend doing the heavy lifting.

**Sources:**

* SAS Institute’s official VS Code extension description – showing that syntax highlighting, completion, etc., are feasible[[1]](https://github.com/sassoftware#:~:text=This%20SAS%20Extension%20for%20Visual,folding%2C%20outline%2C%20SAS%20code%20sni%E2%80%A6).
* SAS procedures usage: PROC FREQ and PROC MEANS are among the most commonly used analytics procedures in SAS[[6]](https://support.sas.com/resources/papers/proceedings/proceedings/sugi30/263-30.pdf#:~:text=Procs%20FREQ%20and%20MEANS%20have,where%20they%20may%20be%20most), indicating their importance in our implementation.
* Polars vs Pandas performance discussion: Polars can be **10-100×** faster than pandas for common operations, due to its Rust implementation and efficient memory use[[4]](https://blog.jetbrains.com/pycharm/2024/07/polars-vs-pandas/#:~:text=In%20a%20word%3A%20performance,4%20times%20needed%20for%20Polars).
* SAS Macro Facility overview: *“The macro facility is a tool for extending and customizing SAS and for reducing the amount of text you must enter... The macro facility enables you to assign a name to... programming statements.”*[[9]](https://support.sas.com/publishing/pubcat/chaps/61885.pdf#:~:text=The%20macro%20facility%20is%20a,does%20not%20support%20the%20use) – guiding our implementation of a similar facility in Open-SAS.
* Example of using VS Code extension API to spawn a process and capture output in real-time (using unbuffered output)[[3]](https://stackoverflow.com/questions/68794540/vscode-extension-executes-local-python-bash-code-and-append-to-output-channel#:~:text=export%20function%20execute,)[[2]](https://stackoverflow.com/questions/68794540/vscode-extension-executes-local-python-bash-code-and-append-to-output-channel#:~:text=After%20some%20more%20digging%2C%20I,to%20unbuffer%20the%20python%20output), which we will employ to run the Open-SAS backend and stream the SAS log/output to the user.

[[1]](https://github.com/sassoftware#:~:text=This%20SAS%20Extension%20for%20Visual,folding%2C%20outline%2C%20SAS%20code%20sni%E2%80%A6) SAS Software · GitHub

<https://github.com/sassoftware>

[[2]](https://stackoverflow.com/questions/68794540/vscode-extension-executes-local-python-bash-code-and-append-to-output-channel#:~:text=After%20some%20more%20digging%2C%20I,to%20unbuffer%20the%20python%20output) [[3]](https://stackoverflow.com/questions/68794540/vscode-extension-executes-local-python-bash-code-and-append-to-output-channel#:~:text=export%20function%20execute,) VSCode Extension executes local python (bash) code and append to output channel - Stack Overflow

<https://stackoverflow.com/questions/68794540/vscode-extension-executes-local-python-bash-code-and-append-to-output-channel>

[[4]](https://blog.jetbrains.com/pycharm/2024/07/polars-vs-pandas/#:~:text=In%20a%20word%3A%20performance,4%20times%20needed%20for%20Polars) [[5]](https://blog.jetbrains.com/pycharm/2024/07/polars-vs-pandas/#:~:text=One%20of%20the%20other%20advantages,about%20how%20Polars%20achieves%20this) Polars vs. pandas: What’s the Difference? | The PyCharm Blog

<https://blog.jetbrains.com/pycharm/2024/07/polars-vs-pandas/>

[[6]](https://support.sas.com/resources/papers/proceedings/proceedings/sugi30/263-30.pdf#:~:text=Procs%20FREQ%20and%20MEANS%20have,where%20they%20may%20be%20most) 263-30: PROC FREQ and PROC MEANS: To Stat or Not to Stat

<https://support.sas.com/resources/papers/proceedings/proceedings/sugi30/263-30.pdf>

[[7]](http://webhome.auburn.edu/~zengpen/teaching/STAT-6110/pdf/01C-proc.pdf#:~:text=match%20at%20L80%20Commonly,Dataset) [[8]](http://webhome.auburn.edu/~zengpen/teaching/STAT-6110/pdf/01C-proc.pdf#:~:text=Commonly,tables%20for%20var1%20and%20var2) STAT 5110/6110: SAS Programming and Applications - 1-C. Commonly-Used SAS Procedures

<http://webhome.auburn.edu/~zengpen/teaching/STAT-6110/pdf/01C-proc.pdf>

[[9]](https://support.sas.com/publishing/pubcat/chaps/61885.pdf#:~:text=The%20macro%20facility%20is%20a,does%20not%20support%20the%20use) [[10]](https://support.sas.com/publishing/pubcat/chaps/61885.pdf#:~:text=When%20you%20use%20a%20macro,communicate%20with%20the%20macro%20processor) SAS 9.2 Macro Language: Reference

<https://support.sas.com/publishing/pubcat/chaps/61885.pdf>