Contest Log Analytics - Programmer's Guide

Version: 0.129.0-Beta Date: 2026-01-11

---

1. Introduction & Architecture

The Contest Log Analytics project is built on three core architectural principles:

- **Data-Driven:** The behavior of the analysis engine is primarily controlled by data, not code. Contest rules, multiplier definitions, and parsing logic are defined in simple `.json` files.
- **Extensible:** The application uses a "plugin" architecture. New reports and contest-specific logic modules are dynamically discovered at runtime.
- **Convention over Configuration:** Files and classes must be named and placed in specific, predictable locations to be discovered.

---

2. The Data Abstraction Layer (DAL)

The `contest_tools.data_aggregators` package is the sole authority for data summarization. All Aggregators must return standard Python primitives (Dictionaries, Lists, Ints), not Pandas DataFrames.

Primary Aggregators

- `CategoricalAggregator`: Handles set operations (Unique/Common QSOs) and categorical grouping.
- `ComparativeEngine`: Implements Set Theory logic to calculate Universe, Common, Differential, and Missed counts.
- `MatrixAggregator`: Generates 2D grids (Band x Time) for heatmaps.
- `MultiplierStatsAggregator`: Handles "Missed Multiplier" analysis and summarization.
- `TimeSeriesAggregator`: Generates the standard TimeSeries Data Schema (v1.4.0).
- `WaeStatsAggregator`: Specialized logic for WAE contests (QTCs and weighted multipliers).

---

3. Web Architecture (Phase 3)

The project includes a stateless, containerized web dashboard built on Django.

The Manifest & WORM Strategy

To ensure responsiveness, the application employs a **"Write Once, Read Many" (WORM)** strategy.

1. **Generation Phase:** During analysis, "Artifact Reports" (e.g., `json_multiplier_breakdown.py`) execute and serialize aggregation trees into JSON files.
2. **Hydration Phase:** When a user loads a dashboard view, the view hydrates its context directly from these pre-computed JSON artifacts via the `ManifestManager`.
3. **Result:** No re-parsing of Cabrillo logs occurs on page load.

---

4. Shared Utilities & Styles

- `MPLStyleManager`: Centralized Matplotlib styles for static plots.
- `PlotlyStyleManager`: Centralized Plotly styles for interactive visualizations.
  - `get_point_color_map()`: Standard color mapping for QSO points.
  - `get_qso_mode_colors()`: Standard scheme for Run/S&P modes.
- `CtyManager`: Manages the lifecycle and caching of the `cty.dat` country file.

---

5. How to Add a New Contest

To add a new contest (e.g., "My Contest"), you generally do not need to write Python code. You only need to define the rules in JSON.

Step 1: Create the Definition File

Create a JSON file in `contest_tools/contest_definitions/` with a filename that matches your contest name:

- Convert the contest name to lowercase
- Replace hyphens (`-`) and spaces with underscores (`_`)
- Add `.json` extension

Example: For contest name `MY-CONTEST`, create `my_contest.json`

Important: The `contest_name` field in the JSON must **exactly match** the `CONTEST:` tag in the Cabrillo file header (case-sensitive).

Step 2: Define Basic Metadata

Every contest definition requires these core fields:

```
{
  "contest_name": "MY-CONTEST",
  "dupe_check_scope": "per_band",
  "score_formula": "points_times_mults",
  "valid_bands": ["160M", "80M", "40M", "20M", "15M", "10M"]
}
```

Field Descriptions:

- `contest_name`: Must exactly match the `CONTEST:` header in Cabrillo files
- `dupe_check_scope`: Either `"per_band"` (dupes checked per band) or `"all_bands"` (dupes checked across all bands)
- `score_formula`: Scoring calculation method:
  - `"points_times_mults"`: Standard formula (QSO points × multipliers)
  - `"qsos_times_mults"`: QSO count × multipliers (e.g., NAQP)
  - `"total_points"`: Sum of points only (e.g., ARRL Field Day)
  - `"custom"`: Requires a `scoring_module` (e.g., WAE contests)
- `valid_bands`: Array of allowed band names (standard: `160M`, `80M`, `40M`, `20M`, `15M`, `10M`, `6M`, `2M`, etc.)

Step 3: Define Exchange Parsing Rules

Most contests require `exchange_parsing_rules` to parse QSO exchanges from Cabrillo format. This defines regex patterns to extract exchange components.

```
"exchange_parsing_rules": {
  "MY-CONTEST": {
    "regex": "(\\d{3})\\s+([A-Z]{2})\\s+([A-Z0-9/]+)\\s+(\\d{3})\\s+([A-Z]{2})",
    "groups": [
      "SentRST",
      "SentLocation",
      "Call",
      "RST",
      "RcvdLocation"
    ]
  }
}
```

Notes:

- The key (e.g., `"MY-CONTEST"`) must match the contest name from the Cabrillo header
- The `regex` field uses standard regex syntax with capturing groups
- The `groups` array maps each capturing group to a column name
- For contests with different exchange formats (e.g., W/VE vs DX), define multiple keys (see `arrl_dx_cw.json` for examples)

Step 4: Define Multiplier Rules

Multiplier rules define how multipliers are calculated and counted. **This must be an array, not an object.**

```
"multiplier_rules": [
  {
    "name": "Sections",
    "value_column": "Mult1",
    "name_column": "Mult1Name",
    "totaling_method": "once_per_log"
  },
  {
    "name": "Zones",
    "source_column": "Zone",
    "value_column": "Mult2",
    "totaling_method": "sum_by_band"
  }
]
```

Common Field Descriptions:

- `name`: Display name for the multiplier type
- `value_column`: DataFrame column storing multiplier values (typically `Mult1`, `Mult2`, etc.)
- `name_column`: DataFrame column storing multiplier names (typically `Mult1Name`, `Mult2Name`, etc.)
- `source_column`: Source column to extract multiplier from (e.g., `Zone` from QSO data)
- `source`: Pre-defined multiplier source (e.g., `"wae_dxcc"` for WAE DXCC list)
- `totaling_method`: How multipliers are counted:
  - `"once_per_log"`: Count each multiplier once across entire log
  - `"sum_by_band"`: Sum multipliers across all bands (same multiplier on different bands counts separately)
  - `"once_per_mode"`: Count once per mode (for multi-mode contests)
- `applies_to`: Optional filter (e.g., `"W/VE"` or `"DX"` for location-specific multipliers)

Step 5: Define Default QSO Columns (Optional)

Specify which columns appear in reports:

```
"default_qso_columns": [
  "ContestName",
  "MyCall",
  "Frequency",
  "Mode",
```

```
  "Datetime",
  "SentRST",
  "SentLocation",
  "Call",
  "RST",
  "RcvdLocation",
  "Band",
  "Date",
  "Hour",
  "Dupe",
  "DXCCName",
  "DXCCPfx",
  "Run",
  "QSOPoints",
  "Mult1",
  "Mult1Name"
]
```

If omitted, the system uses defaults from `_common_cabrillo_fields.json`.

Step 6: Additional Optional Fields

Many contests require additional configuration:

Scoring & Calculation:

- `scoring_module`: Python module name for custom scoring (e.g., `"wae_scoring"`)
- `time_series_calculator`: Custom time series calculation module (e.g., `"naqp_calculator"`)

Custom Parsers & Resolvers:

- `custom_parser_module`: Custom Cabrillo parser module (e.g., `"arrl_dx_parser"`)
- `custom_multiplier_resolver`: Custom multiplier resolution logic (e.g., `"arrl_ss_multiplier_resolver"`)
- `custom_location_resolver`: Custom location determination (e.g., `"wae_location_resolver"`)
- `custom_dupe_checker`: Custom duplicate QSO checking logic (e.g., `"laqp_dupe_checker"`)
- `custom_adif_exporter`: Custom ADIF export logic (e.g., `"cq_ww_adif"`)

Contest Period & Operating Rules:

- `contest_period`: Defines contest start/end times:

```
"contest_period": {
  "start_day": "Saturday",
  "start_time": "00:00:00 UTC",
```

```
    "end_day": "Sunday",
    "end_time": "23:59:59 UTC"
  }
```

- `operating_time_rules`: Operating time restrictions:

```
"operating_time_rules": {
  "min_off_time_minutes": 30,
  "single_op_max_hours": 24,
  "multi_op_max_hours": 24
}
```

Report Control:

- `excluded_reports`: Array of report names to skip (e.g., `["text_wae_score_report"]`)
- `included_reports`: Array of report names to include (if specified, only these run)

Multiplier Behavior:

- `mults_from_zero_point_qsos`: Boolean - whether zero-point QSOs can yield multipliers (default: `true`)
- `mutually_exclusive_mults`: Array of arrays defining mutually exclusive multiplier groups:

```
"mutually_exclusive_mults": [
  ["Mult_STPROV", "Mult_NADXCC"]
]
```

- `multiplier_report_scope`: Scope for multiplier reports (`"per_band"` or `"all_bands"`)

Other:

- `enable_adif_export`: Boolean - enable ADIF export functionality
- `points_header_label`: Custom label for points column in reports (e.g., `"QSO+QTC Pts"` for WAE)
- `is_naqp_ruleset`: Boolean - special flag for NAQP contest rules

Inheritance Mechanisms

The system supports two inheritance mechanisms:

1. Explicit Inheritance (`inherits_from`):

```
{
  "inherits_from": "cq_ww_cw",
  "contest_name": "MY-CONTEST",
  "score_formula": "points_times_mults"
}
```

The child definition inherits all fields from the parent, then overrides with its own values.

2. Implicit Generic/Specific Pattern: If a specific contest file isn't found (e.g., `arrl_ss_cw.json`), the system automatically tries the generic version (e.g., `arrl_ss.json`) by removing the last hyphen-separated segment.

3. Common Fields Base: All contest definitions automatically inherit from `_common_cabrillo_fields.json`, which provides:

- Standard Cabrillo header field mappings
- Default QSO column definitions
- Common field structures

Complete Example

Here's a complete example for a simple contest:

```
{
  "_filename": "contest_tools/contest_definitions/my_contest.json",
  "contest_name": "MY-CONTEST",
  "scoring_module": "my_contest_scoring",
  "valid_bands": ["80M", "40M", "20M", "15M", "10M"],
  "dupe_check_scope": "per_band",
  "score_formula": "points_times_mults",
  "enable_adif_export": true,
  "excluded_reports": [
    "text_wae_score_report",
    "text_wae_comparative_score_report"
  ],
  "exchange_parsing_rules": {
    "MY-CONTEST": {
      "regex": "(\\d{3})\\s+([A-Z]{2})\\s+([A-Z0-9/]+)\\s+(\\d{3})\\s+([A-Z]{2})",
      "groups": [
        "SentRST",
        "SentLocation",
        "Call",
        "RST",
        "RcvdLocation"
      ]
    }
  },
  "multiplier_rules": [
    {
      "name": "Sections",
      "value_column": "Mult1",
      "name_column": "Mult1Name",
      "totaling_method": "once_per_log"
    }
  ],
```

```
  "default_qso_columns": [
    "ContestName",
    "MyCall",
    "Frequency",
    "Mode",
    "Datetime",
    "SentRST",
    "SentLocation",
    "Call",
    "RST",
    "RcvdLocation",
    "Band",
    "Date",
    "Hour",
    "Dupe",
    "DXCCName",
    "DXCCPfx",
    "Run",
    "QSOPoints",
    "Mult1",
    "Mult1Name"
  ]
}
```

Reference Examples

For real-world examples, examine existing contest definitions in `contest_tools/contest_definitions/`:

- **Simple contests:** `cq_ww_cw.json`, `cq_wpx_cw.json`
- **Complex multipliers:** `arrl_dx_cw.json`, `naqp.json`
- **Custom parsers:** `arrl_ss.json`, `wae_cw.json`
- **Inheritance:** Check how `arrl_ss.json` is used by mode-specific variants
- **Special rules:** `wae_cw.json` (QTCs, custom scoring), `arrl_fd.json` (unique bands)

When Python Code is Needed

While most contests can be defined purely in JSON, you may need custom Python modules for:

1. **Custom Scoring:** Create a module in `contest_tools/contest_specific_annotations/` (e.g., `wae_scoring.py`)
2. **Custom Parsing:** Create a module in `contest_tools/contest_specific_annotations/` (e.g., `arrl_dx_parser.py`)
3. **Custom Multiplier Resolution:** Create a module in `contest_tools/contest_specific_annotations` (e.g., `arrl_ss_multiplier_resolver.py`)

4. **Custom Location Resolution:** Create a module in `contest_tools/contest_specific_annotations/` (e.g., `wae_location_resolver.py`)
5. **Custom Dupe Checking:** Create a module in `contest_tools/contest_specific_annotations/` (e.g., `laqp_dupe_checker.py`)

When Custom Dupe Checking is Needed:

- Rover stations that can be worked from multiple locations (e.g., state QSO parties with county/parish rovers)
- Contests where dupe checking must consider exchange data (e.g., location codes in exchange)
- Asymmetric dupe rules (different rules for in-state vs out-of-state stations)
- Any contest where standard `(Call, Band, Mode)` dupe checking is insufficient

Function Signature:

```
def check_dupes(df: pd.DataFrame, my_location_type: Optional[str],
                root_input_dir: str, contest_def: ContestDefinition) -> pd.DataFrame:
    """
    Performs custom duplicate QSO checking.
    Returns a new DataFrame with the 'Dupe' column updated.

    Note: Standard dupe checking runs first during apply_annotations().
    Custom dupe checker runs after multiplier resolution (so location data
    is available) but before scoring. It can override the standard dupe flags.
    """
    # Implementation
    return df.copy()  # Always return new DataFrame (immutability)
```

Execution Order:

1. Standard dupe checking (during `apply_annotations()`)
2. Multiplier resolution (populates location/exchange data)
3. **Custom dupe checking** (if specified, can override standard dupes)
4. Scoring (uses final `Dupe` column)

Reference existing modules in these directories for implementation patterns.

Data File Requirements

Some contests require external data files (`.dat` files) to be placed in the `data/` directory within your input directory. These files contain multiplier lists, section definitions, and other contest-specific data.

Required for All Contests:

- `cty.dat`: Country/prefix database (required for all contests)
- `band_allocations.dat`: Frequency validation data (required for all contests)

Contest-Specific Data Files:

Contest	Required Data File	Purpose
ARRL 10 Meter	`arrl_10_mults.dat`	Multiplier definitions
ARRL DX (CW/SSB)	`ARRLDXmults.dat`	Multiplier definitions
ARRL Sweepstakes	`SweepstakesSections.dat`	Section definitions
ARRL Field Day	`SweepstakesSections.dat`	Section definitions
CQ 160-Meter	`CQ160mults.dat`	Multiplier definitions
NAQP	`NAQPmults.dat`	Multiplier definitions
IARU HF	`iaru_officials.dat`	Official station list

How to Determine if Your Contest Needs Data Files:

1. **Check existing similar contests:** If your contest is similar to an existing one (e.g., uses ARRL sections), check what data files that contest uses.
2. **Check custom multiplier resolvers:** If your contest uses `custom_multiplier_resolver`, examine that Python module to see which data files it loads.
3. **Simple contests may not need data files:** Contests that use standard multipliers (e.g., CQ Zones, DXCC countries) typically don't require additional data files, as these are derived from the `cty.dat` file.

Where to Place Data Files:

- Data files must be in: `{CONTEST_INPUT_DIR}/data/`
- The system automatically looks in this location
- Ensure files are named exactly as expected (case-sensitive on some systems)

Exchange Parsing: Standard vs Custom

Understanding when to use standard parsing versus custom parsers is crucial for correctly adding a contest.

Standard Parsing (Most Contests):

The standard Cabrillo parser uses `exchange_parsing_rules` from your JSON definition to parse QSO exchanges. This works for most contests with straightforward exchange formats.

When Standard Parsing Works:

- Exchange format is consistent across all QSOs
- Exchange can be parsed with a single regex pattern
- No special handling needed for different operator locations or modes

Example: CQ WW, CQ WPX use standard parsing with simple regex patterns.

Custom Parsers (`custom_parser_module`):

Some contests need custom parsing logic when:

- Exchange format varies by operator location (e.g., W/VE vs DX in ARRL DX)
- Exchange format varies by mode (e.g., CW vs SSB in some contests)
- Special handling is needed (e.g., QTC lines in WAE contests)
- Pre-processing is required before parsing

Important: Even when using a custom parser, you still need `exchange_parsing_rules` in your JSON definition. The custom parser uses these rules to parse the exchange portion of each QSO line.

How the System Chooses:

1. If `custom_parser_module` is specified in JSON → Custom parser is used
2. Otherwise → Standard parser uses `exchange_parsing_rules` from JSON

Examples:

- `cq_ww_cw.json`: No `custom_parser_module` → Uses standard parser
- `arrl_dx_cw.json`: Has `custom_parser_module: "arrl_dx_parser"` → Uses custom parser (because W/VE and DX have different exchange formats)

Multiplier Resolution Flow

Understanding how multipliers are resolved helps you configure `multiplier_rules` correctly.

The Resolution Process:

1. **QSO Data Collection:** After parsing, each QSO has basic data (callsign, band, mode, exchange components, etc.)
2. **Location Determination:** The system determines the operator's location type (W/VE, DX, etc.) using:
   - `cty.dat` lookup of the operator's callsign
   - `custom_location_resolver` if specified
3. **Multiplier Resolution:** For each QSO, multipliers are resolved using one of these methods:

   Method A: Direct Column Mapping (`source_column`)

   ```
   {
     "name": "Zones",
     "source_column": "Zone",
     "value_column": "Mult2"
   }
   ```

   - Uses a column already present in the QSO data (e.g., `Zone` from exchange parsing)

- No additional lookup needed

Method B: Pre-defined Source (`source`)

```
{
  "name": "Countries",
  "source": "wae_dxcc",
  "value_column": "Mult1"
}
```

- Uses a built-in multiplier source (e.g., WAE DXCC list)
- System handles the lookup automatically

Method C: Custom Resolver (`custom_multiplier_resolver`)

```
{
  "custom_multiplier_resolver": "arrl_ss_multiplier_resolver"
}
```

- Python module performs custom logic
- Typically used when multipliers come from data files or need complex logic
- The resolver reads `multiplier_rules` from JSON to know which columns to populate

4. **Multiplier Counting:** After resolution, multipliers are counted according to `totaling_method`:
   - `once_per_log`: Each unique multiplier counts once
   - `sum_by_band`: Same multiplier on different bands counts separately
   - `once_per_mode`: Count once per mode (for multi-mode contests)

Common Patterns:

- **Simple contests (CQ WW, CQ WPX):** Use `source_column` pointing to `Zone` or rely on DXCC from `cty.dat`
- **Section-based contests (ARRL SS, ARRL FD):** Use `custom_multiplier_resolver` with `SweepstakesSections.dat`
- **Complex contests (NAQP, ARRL DX):** Use `custom_multiplier_resolver` with contest-specific data files

Field Dependencies

Some fields require or work together with other fields. Understanding these relationships prevents configuration errors.

Required Combinations:

If You Use	You Must Also Provide
`score_formula: "custom"`	`scoring_module` (Python module name)
`custom_parser_module`	`exchange_parsing_rules` (parser still uses these)

If You Use	You Must Also Provide
`custom_multiplier_resolver`	`multiplier_rules` (resolver reads these)
`time_series_calculator`	Usually `scoring_module` (for custom scoring)
`applies_to` in multiplier_rules	Location-specific logic (W/VE vs DX)

Optional but Recommended:

- If using `multiplier_rules` with `source_column`, ensure that column exists in your exchange parsing (e.g., if `source_column: "Zone"`, parse `Zone` in `exchange_parsing_rules`)
- If using `custom_multiplier_resolver`, check which data files it requires and ensure they exist
- If using `excluded_reports`, verify the report names match actual report modules

Validation Tips:

- The system will raise errors if required combinations are missing
- Missing `scoring_module` when `score_formula: "custom"` will result in zero points
- Missing `exchange_parsing_rules` when using standard parser will fail to parse QSOs

Testing Your Contest Definition

After creating your contest definition, test it thoroughly before using it in production.

Step 1: Validate JSON Syntax

```
# Use a JSON validator or Python to check syntax
python -m json.tool contest_tools/contest_definitions/my_contest.json
```

Step 2: Test with a Sample Log

1. **Prepare a test log:** Use a small Cabrillo log file from the contest you're adding
2. **Run a simple analysis:**
   `python main_cli.py --report summary path/to/test.log`
3. **Check for errors:** Look for:
   - `FileNotFoundError`: Contest definition not found (check filename and `contest_name`)
   - `ValueError`: Parsing errors, missing fields, or configuration issues
   - Warnings about missing multipliers or zero points

Step 3: Verify Parsing

Check that QSOs are parsed correctly:

- Open the generated CSV or check the summary report
- Verify exchange components are in correct columns
- Check that multipliers are being identified (if applicable)

Step 4: Verify Scoring

- Check that QSO points are calculated correctly
- Verify multiplier counts match expected values
- Compare against known good scores if available

Step 5: Test Reports

Generate different report types to ensure they work:

```
python main_cli.py --report all path/to/test.log
```

Common Test Scenarios:

- **Single QSO test:** Create a minimal log with one QSO to verify basic parsing
- **Multi-band test:** Ensure multipliers work across bands correctly
- **Edge cases:** Test with unusual callsigns, portable indicators, etc.

Common Pitfalls and Solutions

Learn from common mistakes to avoid frustration.

Pitfall 1: Contest Name Mismatch

Problem: `FileNotFoundError: No definition file found for 'MY-CONTEST'`

Causes:

- Filename doesn't match (e.g., created `my_contest.json` but contest name is `MY-CONTEST-CW`)
- `contest_name` in JSON doesn't match `CONTEST:` header in Cabrillo file

Solution:

- Ensure filename follows naming rules: `contest_name.lower().replace('-', '_').replace(' ', '_') + '.json'`
- Ensure `contest_name` in JSON exactly matches the `CONTEST:` header (case-sensitive)

Pitfall 2: Exchange Parsing Fails

Problem: QSOs parsed but exchange fields are empty or incorrect

Causes:

- Regex pattern doesn't match actual exchange format
- Number of capturing groups doesn't match `groups` array length
- Whitespace handling (tabs vs spaces)

Solution:

- Examine actual QSO lines from a Cabrillo file
- Test regex pattern separately (use online regex tester)
- Use `\s+` for flexible whitespace matching
- Ensure `groups` array has same number of elements as capturing groups in regex

Pitfall 3: Multipliers Not Counting

Problem: Multipliers identified but count is zero or incorrect

Causes:

- `totaling_method` incorrect for contest rules
- `custom_multiplier_resolver` not working correctly
- Missing data files
- `value_column` and `name_column` not matching what resolver populates

Solution:

- Check contest rules for how multipliers should be counted
- Verify data files exist if using custom resolver
- Check resolver code to see which columns it populates
- Ensure `multiplier_rules` column names match resolver output

Pitfall 4: Zero Points for All QSOs

Problem: All QSOs show 0 points

Causes:

- `scoring_module` missing when `score_formula: "custom"`
- Scoring module has errors
- `scoring_module` not found (import error)

Solution:

- If using `score_formula: "custom"`, ensure `scoring_module` is specified
- Check Python module exists and is importable
- Review error logs for import or execution errors
- For standard formulas, ensure `score_formula` is set correctly

Pitfall 5: Inheritance Not Working

Problem: Inherited fields not appearing or overrides not taking effect

Causes:

- Parent file not found
- Field name typo in `inherits_from`
- Deep merge not working as expected

Solution:

- Verify parent file exists and is readable
- Check `inherits_from` value matches parent filename (without `.json`)

- Remember: child values override parent values (deep merge)

Pitfall 6: Custom Parser Not Used

Problem: Custom parser specified but standard parser runs instead

Causes:

- Module not found (import error)
- Module doesn't have required function signature
- Module path incorrect

Solution:

- Ensure module is in `contest_tools/contest_specific_annotations/`
- Verify module has `parse_log()` function with correct signature
- Check module name matches `custom_parser_module` value exactly

Troubleshooting

When things go wrong, use this systematic approach to diagnose and fix issues.

Error: "No definition file found for 'CONTEST-NAME'"

1. Check filename matches naming convention
2. Verify file is in `contest_tools/contest_definitions/`
3. Check `contest_name` in JSON matches Cabrillo header exactly
4. Try implicit inheritance (remove last hyphen segment from contest name)

Error: "Parsing rule 'CONTEST-NAME' not found"

1. Verify `exchange_parsing_rules` has key matching contest name
2. Check if custom parser needs different key format (e.g., `CONTEST-NAME-W/VE`)
3. Ensure regex pattern is valid JSON (escape backslashes: `\\d` not `\d`)

Error: "Could not load scoring module"

1. Verify `scoring_module` name is correct
2. Check module exists in `contest_tools/scoring/` or `contest_tools/contest_specific_annotations/`
3. Ensure module has `calculate_points()` function
4. Check for Python syntax errors in module

Warning: "Could not determine own location"

1. Verify `cty.dat` file exists and is readable
2. Check callsign in Cabrillo header is valid
3. If using `custom_location_resolver`, verify it's working correctly

Multipliers Not Appearing:

1. Check if `custom_multiplier_resolver` is specified and working
2. Verify required data files exist
3. Check `multiplier_rules` column names match what resolver populates

4. Ensure QSOs have necessary data (callsign, location, etc.) for multiplier resolution

Reports Not Generating:

1. Check if contest is in `excluded_reports` list
2. Verify report modules exist
3. Check for errors in report generation logs
4. Ensure required data is present (QSOs, multipliers, etc.)

Getting Help:

- Review existing contest definitions for similar contests
- Check error messages carefully—they often point to the exact issue
- Test with minimal configuration first, then add complexity
- Use the reference examples listed earlier in this guide

JSON Quick Reference

Key	Type	Required	Description
`contest_name`	string	Yes	Matches the `CONTEST:` Cabrillo header exactly
`dupe_check_scope`	string	Yes	`"per_band"` or `"all_bands"`
`score_formula`	string	Yes	`"points_times_mults"`, `"qsos_times_mults"`, `"tota`
`valid_bands`	array	Yes	Array of band names (e.g., `["80M", "40M", "20M"]`)
`exchange_parsing_rules`	object	Usually	Regex patterns for parsing QSO exchanges
`multiplier_rules`	array	Usually	Array of multiplier definition objects
`default_qso_columns`	array	No	Columns to include in reports
`scoring_module`	string	No	Python module for custom scoring
`custom_parser_module`	string	No	Python module for custom parsing
`custom_multiplier_resolver`	string	No	Python module for custom multiplier logic
`custom_location_resolver`	string	No	Python module for custom location logic
`custom_dupe_checker`	string	No	Python module for custom duplicate checking
`custom_adif_exporter`	string	No	Python module for custom ADIF export
`time_series_calculator`	string	No	Python module for custom time series
`inherits_from`	string	No	Parent contest definition to inherit from
`excluded_reports`	array	No	Report names to skip
`included_reports`	array	No	Report names to include (if specified, only these)
`contest_period`	object	No	Contest start/end day and time
`operating_time_rules`	object	No	Operating time restrictions
`mults_from_zero_point_qsos`	boolean	No	Whether zero-point QSOs yield multipliers (default: t
`mutually_exclusive_mults`	array	No	Groups of mutually exclusive multipliers
`multiplier_report_scope`	string	No	`"per_band"` or `"all_bands"`
`enable_adif_export`	boolean	No	Enable ADIF export functionality
`points_header_label`	string	No	Custom label for points column
`is_naqp_ruleset`	boolean	No	Special flag for NAQP rules