# SNMPeek

Complete Program Breakdown

SNMPeek is a CLI tool that queries an AKiPS network monitoring server to look up network devices by CIDR subnet, hostname pattern, or MAC address. It displays results in rich terminal tables and optionally exports to CSV.

## Table of Contents

## 1. Imports & Global Setup (Lines 1-33)

### Purpose

Sets up the runtime environment: the shebang line points to the project's virtual environment Python, standard library and third-party modules are imported, and global constants are initialized.

### Step-by-Step

- Line 1 (Shebang): #!/home/tuna/.../. venv/bin/python tells the OS to run this script with the project's virtual environment Python, ensuring all installed packages are available.
- Lines 4-11 (Stdlib imports): argparse (CLI parsing), csv (CSV export), fnmatch (wildcard hostname matching), ipaddress (CIDR/IP validation), os (env vars), re (regex), sys (exit codes), datetime (timestamps).
- Lines 13-18 (Third-party imports): akips.AKIPS (AKiPS API client), dotenv.load_dotenv (.env file loader), rich.Console/Panel/Table/Text (terminal formatting).
- Line 20: console = Console() creates a single Rich console instance used throughout the program for styled output and status spinners.
- Lines 24-33 (MAC_PATTERN): A compiled regex that matches all common MAC address formats: colon-separated (AA:BB:CC:DD:EE:FF), dash-separated (AA-BB-CC-DD-EE-FF), dot-quad (AABB.CCDD.EEFF), dash-quad (AABB-CCDD-EEFF), and bare hex (AABBCCDDEEFF). Case-insensitive via character classes.

## 2. MAC Address Utilities (Lines 36-75)

### normalize_mac(mac_str) - Lines 36-43

Converts any MAC format to a canonical colon-separated lowercase form.

- Step 1: Strip all non-hex characters using re.sub(r"[^0-9a-fA-F]", "", mac_str) and lowercase the result.
- Step 2: If no hex characters remain, return the lowered original string.
- Step 3: Re-join hex chars in pairs with colons: "aabbccddeeff" becomes "aa:bb:cc:dd:ee:ff".

### is_partial_mac(s) - Lines 46-75

Detects partial MAC addresses (fewer than 6 octets) that still look MAC-like. Requires at least one delimiter character to avoid false positives on short hex strings like hostnames.

- Checks colon-separated partial: e.g. "aa:bb:cc" (2-5 octets).
- Checks dash-separated partial: e.g. "aa-bb-cc".
- Checks dot-quad partial: e.g. "aabb.ccdd" (HPE format).
- Checks dash-quad partial: e.g. "aabb-ccdd".
- Returns True if any pattern matches, False otherwise.

## 3. Argument Parsing (Lines 78-149)

### parse_args()

Defines and processes all CLI arguments using argparse.

### Positional Argument

- query (nargs="+"): One or more queries. Can be CIDR subnets (10.1.0.0/24), MAC addresses

(aa:bb:cc:dd:ee:ff), or hostname patterns (*switch*). Multiple queries can be mixed.

### Optional Field Flags

- -g / --groups: Include AKiPS group memberships in output.
- -d / --descr: Include sysDescr (OS/model info) in output.
- --location: Include sysLocation in output.
- -l / --lldp: Include LLDP neighbor discovery data in output.
- --snmp: Include SNMP reachability state in output.
- -a / --all-fields: Shortcut to enable ALL optional fields at once.

### Output Options

- -o / --output FILE: Write CSV to a specific filename (implies --csv).
- --csv: Enable CSV export with an auto-generated filename.

### Post-Processing Logic

- If -o is provided, args.csv is auto-set to True.
- If --all-fields is set, all individual field flags (groups, descr, location, lldp, snmp) are set to True.

## 4. Query Classification (Lines 152-177)

### classify_query(query_str)

Determines the type of each user-provided query string and returns a (type, normalized_value) tuple. This is the core routing logic.

- Step 1: Strip whitespace. Test against MAC_PATTERN for full MAC addresses.
- Step 2: Test with is_partial_mac() for partial MAC addresses.
- Step 3: If MAC matched, return ("mac", normalize_mac(stripped)).
- Step 4: Try ipaddress.ip_network(query_str, strict=False). If valid, return ("subnet", network). The strict=False flag allows host-bits to be set (e.g. 10.1.0.5/24 is treated as 10.1.0.0/24).
- Step 5: If not a MAC or subnet, treat as hostname pattern. If no wildcards (* or ?) are present, automatically wrap in wildcards: "switch" becomes "*switch*" for substring matching.
- Returns ("hostname", pattern).

## 5. AKiPS Connection (Lines 180-210)

### connect_akips()

Initializes the AKiPS API client from environment variables.

- Step 1: Call load_dotenv() to load variables from the .env file in the project root.
- Step 2: Read environment variables:

**AKIPS_SERVER:** Required. The hostname/IP of the AKiPS server.
**AKIPS_USERNAME:** Defaults to "api-ro" (read-only API user).
**AKIPS_PASSWORD:** Required. The API password.
**AKIPS_VERIFY_SSL:** Defaults to "true". Set to "false" to skip SSL verification.
**AKIPS_TIMEZONE:** Defaults to "America/Los_Angeles".

- Step 3: If AKIPS_SERVER or AKIPS_PASSWORD is missing, print an error message and exit with code 1.
- Step 4: Create and return an AKIPS client object initialized with all credentials.

## 6. Data Fetching (Lines 214-242)

### fetch_data(api, args)

Retrieves device data and optional enrichment data from AKiPS. Each API call displays a Rich spinner while running.

- Step 1 (Always): Fetch all devices via api.get_devices(). Returns a dict of {device_name: {attribute: value}}.
- Step 2 (Always): Fetch ICMP ping states via api.get_attributes(attribute="PING.icmpState"). Used to determine up/down status and uptime.
- Step 3 (If --groups): Fetch group memberships via api.get_group_membership().
- Step 4 (If --snmp): Fetch SNMP states via api.get_attributes(attribute="SNMP.snmpState").
- Step 5 (If --lldp): Fetch LLDP neighbor data using a regex attribute filter that matches both lldpRemSysName and lldpRemPortId.
- Returns tuple: (devices, ping_states, extras_dict).

## 7. Enum State Parsing (Lines 245-266)

### parse_enum_state(enum_data)

Parses AKiPS enum strings. SNMP enumerated types (like interface status) are stored by AKiPS as comma-delimited strings with 5 fields:

```
Format: number,value,created,modified,description
Example: 1,up,1700000000,1707000000,Interface is operational
```

- Step 1: Guard clause - return (None, None) if enum_data is empty.
- Step 2: Apply regex with 5 capture groups matching non-whitespace fields separated by commas (the description field can contain spaces).
- Step 3: Extract group 2 as the value (e.g. "up", "down").
- Step 4: Parse group 4 as a Unix epoch timestamp and convert to a datetime object. If parsing fails, set modified to None.
- Step 5: Return (value, modified_datetime).

Note: Line 253 contains a debug print(modified) statement that should be removed for production use.

## 8. Uptime Formatting (Lines 269-282)

### format_uptime(delta)

Converts a timedelta object into a human-readable uptime string.

- Step 1: Convert to total seconds. If negative, return "N/A".
- Step 2: Use divmod to break into days, hours, minutes.
- Step 3: Format contextually:

**Days present:** "5d 3h 22m"
**Hours only:** "3h 22m"
**Minutes only:** "22m"

## 9. Device Filtering & Merging (Lines 285-388)

### filter_and_merge(networks, hostname_patterns, devices, ping_states, extras, args)

The core data processing function. Iterates all devices, applies filters, extracts status info, and builds result rows. This is where raw AKiPS data becomes structured output.

#### Filtering Phase (Lines 293-315)

- For each device, parse its IPv4 address with ipaddress.ip_address().
- Check if the IP falls within ANY of the provided subnets (using Python's "ip in network" operator).
- If no subnet match, check if hostname or device_name matches ANY hostname pattern (case-insensitive fnmatch wildcard matching).
- If neither matches, skip the device entirely.

#### Status Extraction Phase (Lines 317-336)

- Look up the device in ping_states data.
- Parse the PING.icmpState enum value using parse_enum_state().
- If value is "up": status = "Active", last_seen = now, uptime = now - modified_time.
- If value is anything else: status = "Inactive", last_seen = modified_time.

- If no ping data found: status remains "Unknown".

### Row Building Phase (Lines 338-382)

- Build a base row dict with: hostname, ip, status, uptime, last_seen, _ip_obj (for sorting).
- Conditionally add optional fields based on CLI flags:

**--descr:** SNMPv2-MIB.sysDescr from device attributes

**--location:** SNMPv2-MIB.sysLocation from device attributes

**--groups:** Group list from extras["groups"] data

**--snmp:** SNMP state parsed from extras["snmp"] enum data

**--lldp:** LLDP neighbors assembled from lldpRemSysName + lldpRemPortId

### Sorting (Line 387)

- Sort all results by IP address (ascending). Devices without IPs sort to 0.0.0.0.

## 10. Rich Table Display (Lines 391-484)

### display_results(results, networks, hostname_patterns, args)

Renders the filtered results as a styled terminal table using the Rich library.

### Summary Panel (Lines 393-413)

- Count Active, Inactive, and Unknown devices.
- Build a styled Rich Text object showing: query parameters, total found, active (green), inactive (red), unknown (yellow).
- Display inside a Rich Panel with blue border, titled "AKIPS Host Query".

### Table Construction (Lines 418-484)

- Create a Rich Table with alternating row styles (plain / grey background) for readability.
- Default columns: Hostname (cyan), IP Address, Status (colored indicator), Uptime (green if active), Last Seen.
- Optional columns added dynamically based on CLI flags: Description, Location, Groups, SNMP, LLDP Neighbors.
- Status rendering: green circle + "Active", red circle + "Inactive", yellow circle + "Unknown".
- Uptime uses format_uptime() helper; shows "N/A" in dim style if unavailable.

## 11. CSV Export (Lines 487-527)

### write_csv(results, filename, args)

Writes the same data shown in the terminal table to a CSV file.

- Step 1: Open file with csv.writer. Build header row with default + optional field names.
- Step 2: For each result row, format uptime and last_seen as strings.
- Step 3: Append optional field values (description, location, groups as comma-joined, SNMP state, LLDP neighbors as comma-joined).
- Step 4: Write each row to the CSV.

## 12. Switch Port Mapper Functions (Lines 533-619)

### query_spm(api, mac, ip) - Lines 533-545

Queries the AKiPS Switch Port Mapper (SPM) API endpoint.

- Builds a params dict from optional mac/ip arguments.
- Calls api._get(section="api-spm") with a 30-second timeout.
- Returns raw text response, or None on error (prints a warning).

### parse_spm_response(text) - Lines 548-619

Parses the raw SPM API text response into a list of dicts. Handles variable formats since AKiPS SPM output is not strictly standardized.

- Step 1: Detect delimiter by checking for tab, semicolon, or defaulting to comma.
- Step 2: Detect if the first line is a header row by counting known keywords (mac, vendor, switch, interface, vlan, ip). If 2+ match, treat as headers.
- Step 3a (With headers): Map header names to canonical keys (e.g. "manufacturer" -> "vendor", "port" -> "interface"). Parse each subsequent line into a dict.

- Step 3b (Without headers): Assume positional field order: mac, vendor, switch, interface, vlan, ip.
- Returns a list of entry dicts.

## 13. MAC Data Fetching & Enrichment (Lines 622-686)

### fetch_mac_data(api, mac_addresses)

Orchestrates MAC lookups: queries SPM, then enriches each result with detailed port information from the switch.

#### SPM Query Phase

- For each MAC address, call query_spm() and parse the response with parse_spm_response().

#### Port Enrichment Phase

- For each SPM entry that has a switch + interface:
- Query the switch for port attributes using a regex filter matching: ifOperStatus, ifAlias, ifHighSpeed, ifAdminStatus.

**ifOperStatus:** Parsed as enum -> port_status (up/down) + port_last_change timestamp

**ifAdminStatus:** Parsed as enum -> admin_status (up/down, distinguishes admin-shutdown)

**ifAlias:** Port description label -> port_descr

**ifHighSpeed:** Port speed in Mbps -> port_speed

#### Event History Phase

- Query api.get_events() for the switch/interface over the last 7 days.
- Store up to 20 most recent events for display in the port history table.

#### Return Structure

- Returns a list of dicts, each containing: query_mac, entries (enriched list), raw (original SPM text).

## 14. MAC Results Display (Lines 694-798)

### format_speed(speed_raw) - Lines 694-703

- Converts Mbps integer to human-readable: 1000 -> "1G", 100 -> "100M".

### display_mac_results(mac_data_list) - Lines 706-798

Renders MAC lookup results with two tables per query.

#### Summary Panel

- Shows the queried MAC address and number of results found, in a magenta-bordered panel.

#### Main Results Table

- Columns: MAC Address, Vendor, IP Address, Switch, Port, VLAN, Port Status, Speed, Description.
- Port status color-coded: green (up), red (down), yellow (admin down), dim (unknown).

#### Event History Table

- Only shown if events were found. Green-bordered table with columns: Time, Switch, Port, Type, Details.
- Timestamps converted from Unix epoch to human-readable format.

## 15. MAC CSV Export (Lines 801-828)

### write_mac_csv(mac_data_list, filename)

Writes MAC lookup results to CSV with 12 columns.

- Headers: Query MAC, MAC Address, Vendor, IP Address, Switch, Port, VLAN, Port Status, Admin Status, Speed (Mbps), Port Description, Port Last Change.
- Iterates all entries across all MAC queries and writes one row per SPM entry.

## 16. Main Entry Point (Lines 831-920)

### main()

The orchestrator function that ties everything together. Handles the complete program lifecycle from argument parsing to output.

#### Phase 1: Argument Parsing & Classification (Lines 832-849)

- Call parse_args() to get CLI arguments.
- Initialize three empty lists: networks, hostname_patterns, mac_addresses.
- For each query string, call classify_query() and route the result to the appropriate list.

#### Phase 2: API Connection (Line 852)

- Call connect_akips() to initialize the API client. Exits on credential errors.

#### Phase 3: MAC Address Processing (Lines 856-873)

- If mac_addresses is non-empty, call fetch_mac_data() to query the SPM endpoint.
- Display results with display_mac_results().
- If --csv is enabled, generate filename (from MAC hex digits) and call write_mac_csv().
- On error: print error and exit only if there are no subnet/hostname queries to fall back to.

#### Phase 4: Subnet/Hostname Processing (Lines 876-920)

- If networks or hostname_patterns is non-empty, call fetch_data() to retrieve device/state data.
- Call filter_and_merge() to apply filters and build result rows.
- Call display_results() to render the Rich table.
- If --csv is enabled, generate filename (from subnet/pattern labels) and call write_csv().

#### Phase 5: Entry Guard (Lines 922-923)

- if __name__ == "__main__": main() ensures the script only runs when executed directly, not when imported.

## 17. Program Flow Diagram

High-level execution flow when a user runs SNMPeek:

```
User runs: ./SNMPeek 10.1.0.0/24 *switch* aa:bb:cc:dd:ee:ff -a --csv

  1. parse_args()
     +-- Parse CLI arguments
     +-- --all-fields -> enable all optional field flags
     +-- --csv -> enable CSV export

  2. classify_query() x3
     +-- "10.1.0.0/24"         -> ("subnet", IPv4Network)
     +-- "*switch*"            -> ("hostname", "*switch*")
     +-- "aa:bb:cc:dd:ee:ff"  -> ("mac", "aa:bb:cc:dd:ee:ff")

  3. connect_akips()
     +-- Load .env credentials
     +-- Return AKIPS API client

  4. MAC Path (if mac_addresses):
     +-- fetch_mac_data()
```

```
   |   +-- query_spm() -> parse_spm_response()
   |   +-- Enrich with port attrs (ifOperStatus, etc.)
   |   +-- Fetch 7-day event history
   +-- display_mac_results()
   +-- write_mac_csv() [if --csv]

5. Subnet/Hostname Path (if networks or hostname_patterns):
   +-- fetch_data()
   |   +-- get_devices()
   |   +-- get_attributes(PING.icmpState)
   |   +-- [optional] get_group_membership()
   |   +-- [optional] get_attributes(SNMP.snmpState)
   |   +-- [optional] get_attributes(LLDP-MIB.*)
   +-- filter_and_merge()
   |   +-- Match devices against subnets/patterns
   |   +-- Parse ping state -> status/uptime
   |   +-- Build enriched row dicts
   |   +-- Sort by IP address
   +-- display_results()
   +-- write_csv() [if --csv]
```