Development Rules & Standards

Code Style & Formatting

Python Version

- **Minimum**: Python 3.12+
- Use modern Python syntax (f-strings, type hints, etc.)

Naming Conventions

```python
# Functions and variables: snake_case
def search_contact_by_email(email: str) -> Dict:
    pass

contact_list = []
email_validation_error = None

# Constants: UPPER_SNAKE_CASE
HUBSPOT_API_KEY = "..."
BATCH_SIZE = 100
EXCEL_FILE_PATH = "emails.xlsx"

# Classes: PascalCase (if adding classes)
class HubSpotClient:
    pass

# Private functions: _leading_underscore
def _parse_email(raw_email: str) -> str:
    """Internal helper function"""
    pass
```

Imports Organization

```python
# 1. Standard library imports
import csv
import logging
import os
import re
import time
from typing import Dict, List, Optional

# 2. Third-party imports
import dotenv
import pandas as pd
import requests
```

```
# 3. Local imports
# (add when creating internal modules)
```

Type Hints

Use type hints for all functions (no exceptions):

```python
# Good
def search_contact_by_email(email: str) -> Optional[Dict]:
    """Search HubSpot for a contact by email."""
    pass

def add_contacts_to_segment(
    segment_id: str,
    contact_ids: List[str],
    batch_size: int = 100
) -> bool:
    """Add contacts to a segment in batches."""
    pass

# Avoid
def search_contact_by_email(email):
    pass
```

Logging Standards

Setup

Every script must configure logging identically:

```python
import logging
import os

LOGS_DIR = "logs"
os.makedirs(LOGS_DIR, exist_ok=True)

logger = logging.getLogger(__name__)
logger.setLevel(logging.INFO)
log_file = os.path.join(LOGS_DIR, f"{os.path.splitext(__file__)[0]}.log")
file_handler = logging.FileHandler(log_file, encoding="utf-8")
file_handler.setLevel(logging.INFO)
formatter = logging.Formatter(
    "%(asctime)s - %(name)s - %(levelname)s - %(message)s",
    datefmt="%Y-%m-%d %H:%M:%S"
)
```

```
file_handler.setFormatter(formatter)
logger.addHandler(file_handler)
```

Log Levels

- **INFO**: Major operations, results, summaries
  - "Starting HubSpot segment builder..."
  - "Successfully added 50 contacts to segment"
- **WARNING**: Recoverable issues, validation warnings
  - "Email validation failed for 3 addresses"
  - "Rate limit approaching - sleeping 30 seconds"
- **ERROR**: Operations that failed, exceptions
  - "Failed to create segment: 403 Forbidden"
  - "HubSpot API error: contact not found"

Log Message Format

Include enough context to debug issues:

```
# Good - includes what, count, and status
logger.info(f"Processing 150 contacts from emails.xlsx - found 120 in HubSpot")
logger.warning(f"Email validation failed for john@invalid: missing domain")
logger.error(f"API error (500): {response.text} - retry attempt 2/3")

# Poor - not enough context
logger.info("Done processing")  # What was processed?
logger.error("Error")  # What error? In which function?
```

Never Log

- ❌ API keys or tokens
- ❌ Full contact records with PII (phone, address)
- ❌ Raw HTTP response bodies (unless in ERROR with sanitization)
- ☑ Email addresses only (already semi-public)
- ☑ Contact IDs
- ☑ Sanitized error messages

```
# Good - safe to log
logger.info(f"Searching for contact with email: {email}")

# Bad - don't log the full API key
logger.debug(f"Authorization header: {HUBSPOT_API_KEY}")

# Good - log the error without exposing sensitive data
logger.error(f"HubSpot API returned 403: access denied for operation:
{operation_name}")
```

Error Handling

Try-Except Pattern

```
try:
    response = requests.post(url, headers=headers, json=payload)
    response.raise_for_status()  # Raise on 4xx/5xx
except requests.exceptions.Timeout:
    logger.error(f"HubSpot API timeout after {timeout_seconds}s")
    raise
except requests.exceptions.HTTPError as e:
    logger.error(f"HubSpot API error ({e.response.status_code}):
{e.response.reason}")
    if e.response.status_code == 429:
        logger.warning("Rate limited - caller should implement retry")
    raise
except Exception as e:
    logger.error(f"Unexpected error: {type(e).__name__}: {str(e)}")
    raise
```

API Rate Limiting

Always handle 429 responses:

```
def api_call_with_retry(
    url: str,
    headers: Dict,
    json_payload: Dict,
    max_retries: int = 3
) -> requests.Response:
    for attempt in range(max_retries):
        response = requests.post(url, headers=headers, json=json_payload)

        if response.status_code == 429:
            retry_after = int(response.headers.get("Retry-After", 60))
            logger.warning(f"Rate limited. Retry-After: {retry_after}s (attempt
{attempt + 1}/{max_retries})")
            time.sleep(retry_after)
            continue

        return response

    logger.error(f"Max retries ({max_retries}) exceeded for {url}")
    raise Exception(f"Failed after {max_retries} attempts")
```

Data Validation

Email Validation

```python
import re

def validate_email(email: str) -> bool:
    """Validate email format"""
    if not email or not isinstance(email, str):
        return False

    pattern = r'^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$'
    return bool(re.match(pattern, email.strip()))

def normalize_email(email: Optional[str]) -> Optional[str]:
    """Clean and normalize email - lowercase and strip whitespace"""
    if not email:
        return None
    return email.strip().lower()

# Usage
for email in email_list:
    normalized = normalize_email(email)
    if validate_email(normalized):
        # Process
        pass
    else:
        logger.warning(f"Invalid email: {email}")
        invalid_emails.append(email)
```

Data Integrity

- Remove duplicates before processing
- Filter null/empty values
- Log counts before and after validation
- Never process data without validation

HubSpot API Patterns

Pattern 1: Search with Batch Processing

```python
def search_contacts_and_process(
    emails: List[str],
    batch_size: int = 100
) -> Dict[str, any]:
    """Search for contacts and process results"""
    found_count = 0
    not_found = []

    for i in range(0, len(emails), batch_size):
        batch = emails[i:i + batch_size]
        logger.info(f"Processing batch {i // batch_size + 1} ({len(batch)}
emails)")

        for email in batch:
            try:
                contact = search_hubspot_contact(email)
                if contact:
                    found_count += 1
                else:
                    not_found.append(email)
            except Exception as e:
                logger.error(f"Error searching for {email}: {str(e)}")
                not_found.append(email)

    logger.info(f"Results: {found_count} found, {len(not_found)} not found")
    return {"found": found_count, "not_found": not_found}
```

Pattern 2: Batch Operations (Add to Segment)

```
def add_contacts_to_segment_in_batches(
    segment_id: str,
    contact_ids: List[str],
    batch_size: int = 100
) -> int:
    """Add contacts to segment in batches (max 100 per request)"""
    total_added = 0

    for i in range(0, len(contact_ids), batch_size):
        batch = contact_ids[i:i + batch_size]

        try:
            logger.info(f"Adding batch {i // batch_size + 1} ({len(batch)}
contacts)")
            response = requests.post(
                f"
{HUBSPOT_BASE_URL}/crm/v3/objects/contacts/segments/{segment_id}/add-members",
                headers={"Authorization": f"Bearer {HUBSPOT_API_KEY}"},
                json={"vids": batch}
            )
            response.raise_for_status()
            total_added += len(batch)
        except requests.exceptions.HTTPError as e:
            logger.error(f"Failed to add batch: {e.response.status_code} -
{e.response.reason}")
            raise

    logger.info(f"Successfully added {total_added} contacts to segment")
    return total_added
```

File Operations

Reading Files

```python
import pandas as pd

# Excel
try:
    df = pd.read_excel("emails.xlsx", sheet_name=0)
    logger.info(f"Loaded {len(df)} rows from Excel file")
except FileNotFoundError:
    logger.error("Excel file 'emails.xlsx' not found")
    raise

# CSV
try:
    df = pd.read_csv("client_export.csv")
    logger.info(f"Loaded {len(df)} rows from CSV file")
except FileNotFoundError:
    logger.error("CSV file 'client_export.csv' not found")
    raise
```

Writing Files

```python
# Write CSV
output_file = "output.csv"
try:
    df.to_csv(output_file, index=False, encoding="utf-8")
    logger.info(f"Wrote {len(df)} rows to {output_file}")
except Exception as e:
    logger.error(f"Failed to write {output_file}: {str(e)}")
    raise

# Write text file (for reports, not found emails, etc.)
try:
    with open("not_found_emails.txt", "w", encoding="utf-8") as f:
        f.write("\n".join(not_found_emails))
    logger.info(f"Wrote {len(not_found_emails)} not-found emails to report
file")
except Exception as e:
    logger.error(f"Failed to write report: {str(e)}")
    raise
```

Testing

Test File Location

All tests go in `test/` directory with naming: `test_*.py`

Test Structure

```python
import unittest
from pathlib import Path
import sys

# Add parent directory to path for imports
sys.path.insert(0, str(Path(__file__).parent.parent))

class TestEmailValidation(unittest.TestCase):
    def test_valid_email(self):
        """Test that valid email passes validation"""
        from makelist import validate_email
        self.assertTrue(validate_email("john@example.com"))

    def test_invalid_email(self):
        """Test that invalid email fails validation"""
        from makelist import validate_email
        self.assertFalse(validate_email("invalid-email"))

    def test_normalize_email(self):
        """Test that email is normalized correctly"""
        from makelist import normalize_email
        self.assertEqual(
            normalize_email("  JOHN@EXAMPLE.COM  "),
            "john@example.com"
        )

if __name__ == "__main__":
    unittest.main()
```

Running Tests

```
# Run all tests
python -m pytest test/

# Run specific test file
python -m pytest test/test_makelist.py

# Run with verbose output
python -m pytest test/ -v
```

Commits & Version Control

Commit Message Format

```
[TYPE] Brief description (imperative mood)

Detailed explanation of what and why (if needed).

Related files: makelist.py, synclist.py
Related issue: #123 (if applicable)
```

Types: `feat`, `fix`, `refactor`, `docs`, `test`, `perf`

Examples:

```
[feat] Add dry-run mode to synclist.py
[fix] Handle HubSpot rate limiting with exponential backoff
[refactor] Extract HubSpot API logic to separate module
[docs] Update README with environment setup instructions
[test] Add email validation test cases
```

Files to Never Commit

Add to `.gitignore` (should already exist):

```
.env
*.log
logs/
__pycache__/
*.pyc
.venv/
.DS_Store
```

Documentation

Function Docstrings

```python
def search_hubspot_contact(email: str) -> Optional[Dict]:
    """
    Search HubSpot for a contact by email address.

    Args:
        email: Email address to search for (will be normalized)

    Returns:
        Dictionary with contact data if found, None otherwise.
        Structure: {"id": "...", "properties": {...}}
    Raises:
        requests.exceptions.HTTPError: On API errors (4xx, 5xx)
        ValueError: If email validation fails

    Example:
        contact = search_hubspot_contact("john@example.com")
        if contact:
            print(f"Found: {contact['properties']['email']}")
    """
    pass
```

Inline Comments

```
# Use comments to explain WHY, not WHAT
# Bad comment (what it does is obvious)
email = email.strip().lower()  # Strip and lowercase the email

# Good comment (why it's needed)
email = email.strip().lower()  # Normalize for HubSpot API consistency
```

Performance Considerations

Avoid in Production Code

- ❌ `print()` statements (use logger instead)
- ❌ Global state modifications
- ❌ Nested loops without optimization
- ❌ Unbounded loops without exit conditions
- ❌ Loading entire files into memory unnecessarily

Optimize For

- ☑ Batch API calls (HubSpot batch endpoints)
- ☑ Minimize API requests
- ☑ Stream large files
- ☑ Cache repeated values
- ☑ Progress tracking for long operations

Security

Credentials Management

- Always store API keys in `.env` (never hardcode)
- Never log credentials
- Validate environment variables exist before use
- Rotate API keys regularly

Data Handling

- Validate all external input (user files, API responses)
- Sanitize error messages before logging
- Don't expose internal system details in error messages
- Filter PII before external reporting

When in doubt, follow Python PEP 8 and the existing code patterns in this project.