Ansible Filter Plugin - Line-by-Line Code Flow Explanation

Phase 1: File Header

```python
#!/usr/bin/python
```

Flow:

1. Shebang line - tells the operating system this is a Python script
2. Allows the file to be executed directly if permissions are set
3. Points to the Python interpreter location

Phase 2: Class Definition

```python
class FilterModule(object):
```

Flow:

1. Define a class named `FilterModule` (this EXACT name is REQUIRED by Ansible)
2. Inherit from `object` (base Python class)
3. Ansible specifically looks for a class with this name when loading filter plugins
4. If the class name is different, Ansible will not recognize it as a filter plugin

```python
"""Simple filters for Linux system tasks"""
```

Flow:

1. Documentation string (docstring) describing the plugin's purpose
2. Appears when users inspect the plugin
3. Good practice for documentation but not required for functionality

---

Phase 3: The filters() Method - Registration Point

```python
def filters(self):
```

Flow:

1. Define a method named `filters` (this EXACT name is REQUIRED)
2. This is the REGISTRATION METHOD - it tells Ansible which filters are available
3. Ansible calls this method when loading the plugin
4. Must return a dictionary mapping filter names to their implementation functions

```python
return {
    'to_megabytes': self.to_megabytes,
    'service_status': self.service_status
}
```

Flow:

1. Return a dictionary (key-value pairs)
2. KEY: The filter name that will be used in playbooks (e.g., `{{ var | to_megabytes }}`)
3. VALUE: Reference to the actual method that implements the filter logic
4. Structure: `'filter_name_in_playbook': self.method_that_does_work`
5. Each entry creates one usable filter
6. You can have as many filters as you want in this dictionary

Example of what this registration does:

- When playbook has: `{{ 1024 | to_megabytes }}`
- Ansible looks up 'to_megabytes' in this dictionary
- Finds it points to `self.to_megabytes` method
- Calls that method with 1024 as the argument

Phase 4: First Filter Implementation - to_megabytes

```python
```

```
def to_megabytes(self, kilobytes):
```

Flow:

1. Define a method named `to_megabytes`
2. Takes one parameter: `kilobytes` (the value being filtered)
3. When used as `{{ value | to_megabytes }}`, the `value` becomes the `kilobytes` parameter
4. Method name can be anything - it's the dictionary key in `filters()` that matters

```
python

"""Convert KB to MB"""
```

Flow:

1. Docstring explaining what this filter does
2. Helps users understand the filter's purpose

```
python

try:
```

Flow:

1. Start a try-except block for error handling
2. Protects against invalid input (non-numeric values, None, empty strings)
3. Ensures the filter doesn't crash the entire playbook

```
python

kb = int(kilobytes)
```

Flow:

1. Convert the input to an integer
2. Handles cases where input might be a string like "1024"
3. If conversion fails, jumps to the except block
4. Example: "1024" → 1024 (integer)

```
python
```

```
mb = kb / 1024.0
```

Flow:

1. Divide kilobytes by 1024 to convert to megabytes
2. Use 1024.0 (float) instead of 1024 (int) to ensure decimal result
3. Mathematical conversion: 1 MB = 1024 KB
4. Result stored in `mb` variable
5. Example: 2048 KB ÷ 1024 = 2.0 MB

```
python

return f"{mb:.2f} MB"
```

Flow:

1. Use f-string (formatted string literal) to create output
2. `{mb:.2f}` formats the number:
   - `mb` - the variable to format
   - `:` - start of format specification
   - `.2f` - show 2 decimal places in floating-point format
3. Append " MB" text to the number
4. Returns a string like "2.00 MB" or "1536.75 MB"
5. This returned string replaces the filter expression in the playbook

Example flow:

- Input: `{{ 1536 | to_megabytes }}`
- `kilobytes` = 1536
- `kb` = 1536
- `mb` = 1536 / 1024.0 = 1.5
- Returns: "1.50 MB"
- Playbook sees: "1.50 MB"

```
python
```

```
except:
    return "0 MB"
```

Flow:

1. Catches ANY exception that occurs in the try block
2. Prevents the filter from crashing the playbook
3. Returns a safe default value "0 MB"
4. Triggers when:
   - Input is None
   - Input can't be converted to integer (e.g., "abc")
   - Input is empty string
   - Any other error occurs

Phase 5: Second Filter Implementation - service_status

```
python

def service_status(self, status_code):
```

Flow:

1. Define a method named `service_status`
2. Takes one parameter: `status_code` (the value being filtered)
3. When used as `{{ code | service_status }}`, the `code` becomes the `status_code` parameter
4. Designed to convert numeric systemd status codes to human-readable text

```
python

"""Convert systemd status code to readable text"""
```

Flow:

1. Docstring explaining the filter's purpose
2. Indicates this is specifically for systemd service status codes

```
python
```

```
status_map = {
    '0': 'Running',
    '3': 'Stopped',
    '1': 'Failed'
}
```

Flow:

1. Create a dictionary (lookup table) for status code translation
2. KEY: Status code as a string (e.g., '0')
3. VALUE: Human-readable status text (e.g., 'Running')
4. This is a LOCAL variable - created fresh each time the filter is called
5. Based on common systemd return codes:
   - 0 = service is active/running
   - 3 = service is stopped/inactive
   - 1 = service has failed
6. Stored in memory only while this method executes

```
python

return status_map.get(str(status_code), 'Unknown')
```

Flow:

1. Use the dictionary's `.get()` method to look up the status code
2. `str(status_code)` - Convert input to string first (handles integer input)
   - If input is 0 (integer), converts to '0' (string)
   - If input is already '0' (string), stays as '0'
3. `status_map.get(key, default)` method:
   - Looks up `key` in the dictionary
   - If found: returns the corresponding value
   - If NOT found: returns the `default` value ('Unknown')
4. Returns a string like 'Running', 'Stopped', 'Failed', or 'Unknown'

Example flow:

- Input: `{{ 0 | service_status }}`

- `status_code` = 0
- `str(status_code)` = '0'
- `status_map.get('0', 'Unknown')` looks up '0' in dictionary
- Finds '0' → 'Running'
- Returns: 'Running'
- Playbook sees: 'Running'

Example with unknown code:

- Input: `{{ 99 | service_status }}`
- `status_code` = 99
- `str(status_code)` = '99'
- `status_map.get('99', 'Unknown')` looks up '99' in dictionary
- '99' not found in dictionary
- Returns default: 'Unknown'
- Playbook sees: 'Unknown'

Complete Execution Flow Timeline

When you run: **`ansible-playbook test_plugin.yml`**

Step 1: Ansible Startup

- Ansible reads the playbook file
- Identifies the current working directory
- Looks for a `filter_plugins/` directory

Step 2: Plugin Discovery

- Finds `filter_plugins/system_filters.py`
- Imports the Python file as a module
- Python interpreter loads the file into memory

Step 3: Plugin Loading

- Ansible looks for a class named `FilterModule` (REQUIRED name)
- Finds: `class FilterModule(object):`
- Creates an instance of the class: `plugin_instance = FilterModule()`

Step 4: Filter Registration

- Ansible calls: `plugin_instance.filters()`
- Method returns:

```python
{
    'to_megabytes': <function to_megabytes>,
    'service_status': <function service_status>
}
```

- Ansible registers these two filters as available for use
- Filters are now ready to be used in the playbook

Step 5: Playbook Parsing

- Ansible parses the playbook YAML
- Encounters: `{{ ansible_memtotal_mb | to_megabytes }}`
- Recognizes `to_megabytes` as a registered filter
- Prepares to call the associated function

Step 6: First Filter Usage

```yaml
msg: "Total Memory: {{ ansible_memtotal_mb | to_megabytes }}"
```

Execution steps:

1. Ansible gathers facts: `ansible_memtotal_mb = 7812` (integer)
2. Sees the pipe operator `|` - indicates a filter
3. Looks up 'to_megabytes' in registered filters
4. Finds it points to `FilterModule.to_megabytes` method
5. Calls: `plugin_instance.to_megabytes(7812)`
6. Inside the method:
   - `kb = int(7812)` → `kb = 7812`
   - `mb = 7812 / 1024.0` → `mb = 7.62890625`
   - `return f"{7.62890625:.2f} MB"` → returns "7.63 MB"

7. Ansible replaces `{{ ansible_memtotal_mb | to_megabytes }}` with "7.63 MB"
8. Final message: "Total Memory: 7.63 MB"
9. Displays to user

Step 7: Second Filter Usage (Shell Command Result)

```yaml
- shell: df -k / | tail -1 | awk '{print $3}'
  register: disk_used

- debug:
    msg: "Disk used: {{ disk_used.stdout | to_megabytes }}"
```

Execution steps:

1. Shell command executes: returns "15728640" (as string)
2. Result stored in `disk_used.stdout = "15728640"`
3. Ansible encounters: `{{ disk_used.stdout | to_megabytes }}`
4. Calls: `plugin_instance.to_megabytes("15728640")`
5. Inside the method:
   - `kb = int("15728640")` → `kb = 15728640`
   - `mb = 15728640 / 1024.0` → `mb = 15360.0`
   - `return f"{15360.0:.2f} MB"` → returns "15360.00 MB"
6. Final message: "Disk used: 15360.00 MB"
7. Displays to user

Step 8: Third Filter Usage (Loop with service_status)

```yaml
- debug:
    msg: "Service is {{ item | service_status }}"
  loop:
    - 0
    - 3
    - 1
```

Execution steps for EACH iteration:

Iteration 1:

1. `item = 0` (integer from loop list)
2. Encounters: `{{ item | service_status }}`
3. Calls: `plugin_instance.service_status(0)`
4. Inside the method:
   - `status_map = {'0': 'Running', '3': 'Stopped', '1': 'Failed'}`
   - `str(0)` → '0'
   - `status_map.get('0', 'Unknown')` → 'Running'
   - Returns: 'Running'
5. Final message: "Service is Running"
6. Displays to user

Iteration 2:

1. `item = 3`
2. Calls: `plugin_instance.service_status(3)`
3. Inside: `str(3)` → '3' → finds '3' in map → 'Stopped'
4. Returns: 'Stopped'
5. Final message: "Service is Stopped"

Iteration 3:

1. `item = 1`
2. Calls: `plugin_instance.service_status(1)`
3. Inside: `str(1)` → '1' → finds '1' in map → 'Failed'
4. Returns: 'Failed'
5. Final message: "Service is Failed"

Step 9: Playbook Completion

- All tasks finish executing
- Plugin instance remains in memory (might be reused)
- Playbook exits

Key Concepts

1. Filter Plugin Structure

Required Components:

```python
class FilterModule(object):          # MUST be named FilterModule
    def filters(self):               # MUST be named filters
        return {                     # MUST return a dictionary
            'name': self.method      # Maps name to function
        }
```

Optional Components:

- Helper methods (any name)
- Additional filter methods
- Error handling
- Documentation strings

2. How Filters Are Called

In Playbook:

```yaml
{{ variable | filter_name }}
```

What Happens:

1. Ansible evaluates `variable` first
2. Passes the result to `filter_name`
3. The filter receives the value as its first parameter (after `self`)
4. Filter returns a transformed value
5. Ansible uses the returned value in place of the original expression

Multi-filter Chain:

```yaml
{{ variable | filter1 | filter2 }}
```

1. `variable` → passed to `filter1`

2. `filter1` returns a value → passed to `filter2`
3. `filter2` returns final value → used in playbook

3. Parameter Flow

```
python

def to_megabytes(self, kilobytes):
                 ↑
                 |
                 This receives the piped value
```

Example:

- Playbook: `{{ 2048 | to_megabytes }}`
- Python: `to_megabytes(self, 2048)`
- The value 2048 flows into the `kilobytes` parameter

4. Return Value Usage

Whatever your filter returns becomes the replacement text:

```
python

return "7.63 MB"
```

↓

```
yaml

msg: "Total Memory: {{ ansible_memtotal_mb | to_megabytes }}"
```

↓

```
msg: "Total Memory: 7.63 MB"
```

5. Error Handling Strategy

Without error handling:

```
python

def to_megabytes(self, kilobytes):
    return f"{int(kilobytes) / 1024.0:.2f} MB"
```

- If `kilobytes` is None → playbook CRASHES

- If `kilobytes` is "abc" → playbook CRASHES

With error handling:

```python
def to_megabytes(self, kilobytes):
    try:
        return f"{int(kilobytes) / 1024.0:.2f} MB"
    except:
        return "0 MB"
```

- If `kilobytes` is invalid → returns "0 MB", playbook CONTINUES
- More robust and production-ready

6. Dictionary vs Direct Conversion

Using dictionary (service_status approach):

```python
status_map = {'0': 'Running', '3': 'Stopped'}
return status_map.get(str(status_code), 'Unknown')
```

✅ Advantages:

- Easy to read and maintain
- Easy to add new mappings
- Handles unknown values gracefully with default

Using if-elif (alternative approach):

```python
if status_code == '0':
    return 'Running'
elif status_code == '3':
    return 'Stopped'
else:
    return 'Unknown'
```

❌ More verbose, less maintainable

7. Why Filter Plugins?

Advantages:

- **Reusable:** Write once, use in any playbook

- **Clean Playbooks:** Logic moves to Python, playbooks stay simple
- **Type Safe:** Python handles type conversion and validation
- **Testable:** Can test filter functions independently
- **Maintainable:** One place to update formatting logic

Example - Without Filter:

```yaml
- debug:
    msg: "Memory: {{ (ansible_memtotal_mb | int / 1024.0) | round(2) }} MB"
```

- Complex Jinja2 logic in playbook
- Hard to read
- Repeated in multiple places

Example - With Filter:

```yaml
- debug:
    msg: "Memory: {{ ansible_memtotal_mb | to_megabytes }}"
```

- Simple and readable
- Reusable across playbooks
- Logic centralized in Python

Variable Scope and Lifecycle

Plugin Instance Lifecycle

```python
```

```
# Step 1: Ansible creates instance (once per playbook run)
plugin = FilterModule()

# Step 2: Ansible calls filters() to register available filters
available_filters = plugin.filters()

# Step 3: Playbook uses filters multiple times
result1 = plugin.to_megabytes(1024)      # Call 1
result2 = plugin.to_megabytes(2048)      # Call 2
result3 = plugin.service_status(0)       # Call 3

# Step 4: Playbook ends, plugin instance may be destroyed
```

Local vs Instance Variables

Local Variable (created each call):

```
python

def service_status(self, status_code):
    status_map = {...}              # Created fresh each time
    return status_map.get(...)      # Destroyed when method ends
```

Instance Variable (persists across calls):

```
python

def __init__(self):
    self.status_map = {...}         # Created once at initialization

def service_status(self, status_code):
    return self.status_map.get(...)  # Reuses same dictionary
```

For this simple plugin, local variables are fine. Instance variables would be useful for:

- Caching expensive computations
- Maintaining counters
- Storing configuration

Adding More Filters - Step by Step

Step 1: Add to Registration Dictionary

```
python
```

```
def filters(self):
    return {
        'to_megabytes': self.to_megabytes,
        'service_status': self.service_status,
        'to_gigabytes': self.to_gigabytes,       # NEW
        'format_uptime': self.format_uptime      # NEW
    }
```

What this does:

- Makes 'to_gigabytes' available in playbooks as `{{ value | to_gigabytes }}`
- Makes 'format_uptime' available as `{{ value | format_uptime }}`

Step 2: Implement the Filter Method

python

```
def to_gigabytes(self, megabytes):
    """Convert MB to GB"""
    try:
        mb = float(megabytes)        # Accept decimal MB values
        gb = mb / 1024.0             # Convert to GB
        return f"{gb:.2f} GB"        # Format with 2 decimals
    except:
        return "0 GB"                # Safe default
```

Execution example:

- Input: `{{ 2048 | to_gigabytes }}`
- `megabytes = 2048`
- `mb = float(2048)` → `2048.0`
- `gb = 2048.0 / 1024.0` → `2.0`
- `return "2.00 GB"`
- Output: "2.00 GB"

Step 3: Chain Filters

yaml

```
# Convert KB → MB → GB
{{ 2097152 | to_megabytes | to_gigabytes }}
```

Flow:

1. `2097152` passed to `to_megabytes(2097152)`
2. Returns: "2048.00 MB"
3. "2048.00 MB" passed to `to_gigabytes("2048.00 MB")`
4. Wait! This will FAIL because "2048.00 MB" is a string with " MB" at the end
5. `float("2048.00 MB")` throws an error
6. Returns: "0 GB" (from except block)

To fix this, modify to_megabytes to return a number when chaining:

```python
def to_megabytes(self, kilobytes, format_output=True):
    try:
        kb = int(kilobytes)
        mb = kb / 1024.0
        if format_output:
            return f"{mb:.2f} MB"
        else:
            return mb               # Return raw number for chaining
    except:
        return "0 MB" if format_output else 0
```

Real-World Usage Patterns

Pattern 1: Converting Units

```yaml
- name: Get memory statistics
  debug:
    msg: |
      Total Memory: {{ ansible_memtotal_mb | to_megabytes }}
      Free Memory: {{ ansible_memfree_mb | to_megabytes }}
      Used Memory: {{ (ansible_memtotal_mb - ansible_memfree_mb) | to_megabytes }}
```

Pattern 2: Status Translation

```yaml
```

```yaml
- name: Check multiple services
  shell: systemctl is-active {{ item }} && echo 0 || echo 3
  register: service_check
  loop:
    - sshd
    - nginx
    - mysql

- name: Display service statuses
  debug:
    msg: "{{ item.item }} is {{ item.stdout | service_status }}"
  loop: "{{ service_check.results }}"
```

Pattern 3: Conditional Formatting

```yaml
- name: Get disk usage
  shell: df -k /var | tail -1 | awk '{print $3}'
  register: var_used

- name: Show warning if needed
  debug:
    msg: "WARNING: /var is using {{ var_used.stdout | to_megabytes }}"
  when: (var_used.stdout | int) > 10485760  # More than 10 GB in KB
```

Why This Plugin Design?

Design Choice 1: Simple Error Handling

```python
except:
    return "0 MB"
```

Pros:

- Playbook never crashes
- Always returns something
- Simple to implement

Cons:

- Hides the real error

- "0 MB" might be misleading

Better approach for production:

```python
except ValueError as e:
    return f"ERROR: Invalid input - {e}"
except Exception as e:
    return f"ERROR: {e}"
```

Design Choice 2: String Formatting in Filter

```python
return f"{mb:.2f} MB"
```

Why include " MB" in the filter?

- Convenience: Users don't need to add "MB" in every playbook
- Consistency: All outputs formatted the same way
- Readability: `{{ value | to_megabytes }}` is cleaner than `{{ value | to_megabytes }} MB`

Alternative (return number only):

```python
return round(mb, 2)
```

Then in playbook: `msg: "Memory: {{ value | to_megabytes }} MB"`

Design Choice 3: Dictionary for Status Mapping

```python
status_map = {'0': 'Running', ...}
```

Why a dictionary?

- O(1) lookup time (constant, very fast)
- Easy to add new status codes
- Clear mapping of inputs to outputs
- `.get()` method provides default value

Complete Plugin Anatomy

```python
#!/usr/bin/python                    # 1. Shebang

class FilterModule(object):              # 2. Required class name
    """Documentation"""                  # 3. Plugin description

    def filters(self):                   # 4. Required registration method
        return {                         # 5. Return filter dictionary
            'filter1': self.method1,     # 6. Map names to methods
            'filter2': self.method2
        }

    def method1(self, input_value):      # 7. Filter implementation
        try:                             # 8. Error handling
            # Process input              # 9. Business logic
            return formatted_output      # 10. Return result
        except:
            return safe_default          # 11. Fallback value

    def method2(self, input_value):      # 12. Additional filters...
        # Implementation
        pass
```

Summary

✅ **Filter Plugin Structure:**

- Class must be named `FilterModule`
- Must have a `filters()` method
- `filters()` returns a dictionary mapping names to methods

✅ **Filter Methods:**

- Receive filtered value as first parameter (after `self`)
- Can have additional parameters
- Must return a value (any type)
- Should handle errors gracefully

✅ **Execution Flow:**

1. Ansible loads plugin → creates instance
2. Calls `filters()` → registers available filters
3. Playbook uses filter → calls corresponding method
4. Method processes input → returns result
5. Result replaces filter expression in playbook

✅ **Best Practices:**

- Always use try-except blocks
- Return safe defaults on error
- Keep filter logic simple and focused
- Document what each filter does
- Test with various input types

Now you understand how Ansible filter plugins work at a deep, line-by-line level!