Conventions, tips, and pitfalls

Topics

- Scoping your module(s)
- Designing module interfaces
- General guidelines & tips
- Functions and Methods
- Python tips
- Importing and using shared code
- Handling module failures
- Handling exceptions (bugs) gracefully
- Creating correct and informative module output
- Following Ansible conventions
- Module Security

As you design and develop modules, follow these basic conventions and tips for clean, usable code:

Scoping your module(s)

Especially if you want to contribute your module(s) to an existing Ansible Collection, make sure each module includes enough logic and functionality, but not too much. If these guidelines seem confusing, consider :ref:`whether you really need to write a module <module_dev_should_you>` at all.

> **System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\ansible-devel\docs\docsite\rst\dev_guide\(ansible-devel)(docs)(docsite)(rst)(dev_guide)developing_modules_best_practices.rst`, **line 16**); *backlink*
>
> Unknown interpreted text role "ref".

- Each module should have a concise and well-defined functionality. Basically, follow the UNIX philosophy of doing one thing well.
- Do not add `get`, `list` or `info` state options to an existing module - create a new `_info` or `_facts` module.
- Modules should not require that a user know all the underlying options of an API/tool to be used. For instance, if the legal values for a required module option cannot be documented, the module does not belong in Ansible Core.
- Modules should encompass much of the logic for interacting with a resource. A lightweight wrapper around a complex API forces users to offload too much logic into their playbooks. If you want to connect Ansible to a complex API, :ref:`create multiple modules <developing_modules_in_groups>` that interact with smaller individual pieces of the API.

> **System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\ansible-devel\docs\docsite\rst\dev_guide\(ansible-devel)(docs)(docsite)(rst)(dev_guide)developing_modules_best_practices.rst`, **line 21**); *backlink*
>
> Unknown interpreted text role "ref".

- Avoid creating a module that does the work of other modules; this leads to code duplication and divergence, and makes things less uniform, unpredictable and harder to maintain. Modules should be the building blocks. If you are asking 'how can I have a module execute other modules' ... you want to write a role.

Designing module interfaces

- If your module is addressing an object, the option for that object should be called `name` whenever possible, or accept `name` as an alias.
- Modules accepting boolean status should accept `yes`, `no`, `true`, `false`, or anything else a user may likely throw at them. The AnsibleModule common code supports this with `type='bool'`.
- Avoid `action`/`command`, they are imperative and not declarative, there are other ways to express the same thing.

General guidelines & tips

- Each module should be self-contained in one file, so it can be auto-transferred by `ansible-core`.
- Module name MUST use underscores instead of hyphens or spaces as a word separator. Using hyphens and spaces will prevent `ansible-core` from importing your module.
- Always use the `hacking/test-module.py` script when developing modules - it will warn you about common pitfalls.

- If you have a local module that returns information specific to your installations, a good name for this module is `site_info`.
- Eliminate or minimize dependencies. If your module has dependencies, document them at the top of the module file and raise JSON error messages when dependency import fails.
- Don't write to files directly; use a temporary file and then use the `atomic_move` function from `ansible.module_utils.basic` to move the updated temporary file into place. This prevents data corruption and ensures that the correct context for the file is kept.
- Avoid creating caches. Ansible is designed without a central server or authority, so you cannot guarantee it will not run with different permissions, options or locations. If you need a central authority, have it on top of Ansible (for example, using bastion/cm/ci server, AWX, or the Red Hat Ansible Automation Platform); do not try to build it into modules.
- If you package your module(s) in an RPM, install the modules on the control machine in `/usr/share/ansible`. Packaging modules in RPMs is optional.

Functions and Methods

- Each function should be concise and should describe a meaningful amount of work.
- "Don't repeat yourself" is generally a good philosophy.
- Function names should use underscores: `my_function_name`.
- The name of each function should describe what the function does.
- Each function should have a docstring.
- If your code is too nested, that's usually a sign the loop body could benefit from being a function. Parts of our existing code are not the best examples of this at times.

Python tips

- Include a `main` function that wraps the normal execution.
- Call your `main` function from a conditional so you can import it into unit tests - for example:

```
if __name__ == '__main__':
    main()
```

Importing and using shared code

- Use shared code whenever possible - don't reinvent the wheel. Ansible offers the `AnsibleModule` common Python code, plus :ref:`utilities <developing_module_utilities>` for many common use cases and patterns. You can also create documentation fragments for docs that apply to multiple modules.

> **System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\ansible-devel\docs\docsite\rst\dev_guide\(ansible-devel)(docs)(docsite)(rst)(dev_guide)developing_modules_best_practices.rst`, **line 69**); *backlink*
>
> Unknown interpreted text role "ref".

- Import `ansible.module_utils` code in the same place as you import other libraries.
- Do NOT use wildcards (*) for importing other python modules; instead, list the function(s) you are importing (for example, `from some.other_python_module.basic import otherFunction`).
- Import custom packages in `try/except`, capture any import errors, and handle them with `fail_json()` in `main()`. For example:

```
import traceback

from ansible.module_utils.basic import missing_required_lib

LIB_IMP_ERR = None
try:
    import foo
    HAS_LIB = True
except:
    HAS_LIB = False
    LIB_IMP_ERR = traceback.format_exc()
```

Then in `main()`, just after the argspec, do

```
if not HAS_LIB:
    module.fail_json(msg=missing_required_lib("foo"),
                     exception=LIB_IMP_ERR)
```

And document the dependency in the `requirements` section of your module's :ref:`documentation_block`.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\ansible-devel\docs\docsite\rst\dev_guide\(ansible-devel)(docs)(docsite)(rst)(dev_guide)developing_modules_best_practices.rst, line 98); *backlink*

Unknown interpreted text role "ref".

Handling module failures

When your module fails, help users understand what went wrong. If you are using the `AnsibleModule` common Python code, the `failed` element will be included for you automatically when you call `fail_json`. For polite module failure behavior:

- Include a key of `failed` along with a string explanation in `msg`. If you don't do this, Ansible will use standard return codes: 0=success and non-zero=failure.
- Don't raise a traceback (stacktrace). Ansible can deal with stacktraces and automatically converts anything unparsable into a failed result, but raising a stacktrace on module failure is not user-friendly.
- Do not use `sys.exit()`. Use `fail_json()` from the module object.

Handling exceptions (bugs) gracefully

- Validate upfront--fail fast and return useful and clear error messages.
- Use defensive programming--use a simple design for your module, handle errors gracefully, and avoid direct stacktraces.
- Fail predictably--if we must fail, do it in a way that is the most expected. Either mimic the underlying tool or the general way the system works.
- Give out a useful message on what you were doing and add exception messages to that.
- Avoid catchall exceptions, they are not very useful unless the underlying API gives very good error messages pertaining the attempted action.

Creating correct and informative module output

Modules must output valid JSON only. Follow these guidelines for creating correct, useful module output:

- Make your top-level return type a hash (dictionary).
- Nest complex return values within the top-level hash.
- Incorporate any lists or simple scalar values within the top-level return hash.
- Do not send module output to standard error, because the system will merge standard out with standard error and prevent the JSON from parsing.
- Capture standard error and return it as a variable in the JSON on standard out. This is how the command module is implemented.
- Never do `print("some status message")` in a module, because it will not produce valid JSON output.
- Always return useful data, even when there is no change.
- Be consistent about returns (some modules are too random), unless it is detrimental to the state/action.
- Make returns reusable--most of the time you don't want to read it, but you do want to process it and re-purpose it.
- Return diff if in diff mode. This is not required for all modules, as it won't make sense for certain ones, but please include it when applicable.
- Enable your return values to be serialized as JSON with Python's standard JSON encoder and decoder library. Basic python types (strings, int, dicts, lists, and so on) are serializable.
- Do not return an object using exit_json(). Instead, convert the fields you need from the object into the fields of a dictionary and return the dictionary.
- Results from many hosts will be aggregated at once, so your module should return only relevant output. Returning the entire contents of a log file is generally bad form.

If a module returns stderr or otherwise fails to produce valid JSON, the actual output will still be shown in Ansible, but the command will not succeed.

Following Ansible conventions

Ansible conventions offer a predictable user interface across all modules, playbooks, and roles. To follow Ansible conventions in your module development:

- Use consistent names across modules (yes, we have many legacy deviations - don't make the problem worse!).
- Use consistent options (arguments) within your module(s).
- Do not use 'message' or 'syslog_facility' as an option name, because this is used internally by Ansible.
- Normalize options with other modules - if Ansible and the API your module connects to use different names for the same option, add aliases to your options so the user can choose which names to use in tasks and playbooks.
- Return facts from `*_facts` modules in the `ansible_facts` field of the :ref:`result dictionary<common_return_values>` so

other modules can access them.

> **System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\ansible-devel\docs\docsite\rst\dev_guide\(ansible-devel)(docs)(docsite)(rst)(dev_guide)developing_modules_best_practices.rst`, **line 154**); *backlink*
>
> Unknown interpreted text role "ref".

- Implement `check_mode` in all `*_info` and `*_facts` modules. Playbooks which conditionalize based on fact information will only conditionalize correctly in `check_mode` if the facts are returned in `check_mode`. Usually you can add `supports_check_mode=True` when instantiating `AnsibleModule`.
- Use module-specific environment variables. For example, if you use the helpers in `module_utils.api` for basic authentication with `module_utils.urls.fetch_url()` and you fall back on environment variables for default values, use a module-specific environment variable like `API_<MODULENAME>_USERNAME` to avoid conflicts between modules.
- Keep module options simple and focused - if you're loading a lot of choices/states on an existing option, consider adding a new, simple option instead.
- Keep options small when possible. Passing a large data structure to an option might save us a few tasks, but it adds a complex requirement that we cannot easily validate before passing on to the module.
- If you want to pass complex data to an option, write an expert module that allows this, along with several smaller modules that provide a more 'atomic' operation against the underlying APIs and services. Complex operations require complex data. Let the user choose whether to reflect that complexity in tasks and plays or in vars files.
- Implement declarative operations (not CRUD) so the user can ignore existing state and focus on final state. For example, use `started/stopped`, `present/absent`.
- Strive for a consistent final state (aka idempotency). If running your module twice in a row against the same system would result in two different states, see if you can redesign or rewrite to achieve consistent final state. If you can't, document the behavior and the reasons for it.
- Provide consistent return values within the standard Ansible return structure, even if NA/None are used for keys normally returned under other options.
- Follow additional guidelines that apply to families of modules if applicable. For example, AWS modules should follow the :ref:`Amazon development checklist <AWS_module_development>`.

> **System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\ansible-devel\docs\docsite\rst\dev_guide\(ansible-devel)(docs)(docsite)(rst)(dev_guide)developing_modules_best_practices.rst`, **line 163**); *backlink*
>
> Unknown interpreted text role "ref".

Module Security

- Avoid passing user input from the shell.
- Always check return codes.
- You must always use `module.run_command`, not `subprocess` or `Popen` or `os.system`.
- Avoid using the shell unless absolutely necessary.
- If you must use the shell, you must pass `use_unsafe_shell=True` to `module.run_command`.
- If any variables in your module can come from user input with `use_unsafe_shell=True`, you must wrap them with `pipes.quote(x)`.
- When fetching URLs, use `fetch_url` or `open_url` from `ansible.module_utils.urls`. Do not use `urllib2`, which does not natively verify TLS certificates and so is insecure for https.
- Sensitive values marked with `no_log=True` will automatically have that value stripped from module return values. If your module could return these sensitive values as part of a dictionary key name, you should call the `ansible.module_utils.basic.sanitize_keys()` function to strip the values from the keys. See the `uri` module for an example.