inquirer

Miguel Ángel García

Dec 28, 2021

CONTENTS

CHAPTER

ONE

OVERVIEW

Born as a Inquirer.js clone, it shares part of the goals and philosophy.

So, **Inquirer** should ease the process of asking end user **questions**, **parsing**, **validating** answers, managing **hierarchical prompts** and providing **error feedback**.

You can *download the code from GitHub*.

CHAPTER

TWO

CONTENTS

2.1 Installation

To install it, just execute:

```
pip install inquirer
```

Usage example:

```
import inquirer

if __name__ == "__main__":

    questions = [
        inquirer.Text("user", message="Please enter your github username",␣
↪validate=lambda _, x: x != "."),
        inquirer.Password("password", message="Please enter your password"),
        inquirer.Text("repo", message="Please enter the repo name", default="default"),
        inquirer.Checkbox(
            "topics",
            message="Please define your type of project?",
            choices=["common", "backend", "frontend"],
        ),
        inquirer.Text(
            "organization",
            message=(
                "If this is a repo from a organization please enter the organization␣
↪name,"
                " if not just leave this blank"
            ),
        ),
        inquirer.Confirm(
            "correct",
            message="This will delete all your current labels and create a new ones.␣
↪Continue?",
            default=False,
        ),
    ]

    answers = inquirer.prompt(questions)
```

(continues on next page)

(continued from previous page)

```
    print(answers)
```

2.2 Usage

The idea is quite simple:

1. Create an array of `Questions`
2. Call the prompt render.

Each `Question` require some common arguments. So, you just need to know which kind of `Questions` and `Arguments` are available.

2.2.1 Question types

TEXT	Expects a text answer.
EDITOR	Expects a text answer, entered through external editor.
PASSWORD	Do not prompt the answer.
CONFIRM	Requires a boolean answer.
LIST	Show a list and allow to select just one answer.
CHECKBOX	Show a list and allow to select a bunch of them.
PATH	Requires valid path and allows additional validations.

There are pictures of some of them in the Examples section.

2.2.2 Question Arguments

The main object is `Question`, but it should not be instantiated. You must use any of the subclasses, listed below. All of them have the next attributes that can be set in the initialization:

name

It will be the key in the hash of answers. So, it is **mandatory**.

You can use any `String` or `hashable` code as value.

message

Contains the prompt to be shown to the user, and is **mandatory** too.

You can use a new style formatted string, using the previous answers, and it will be replaced automatically:

```
questions = [
    Text(name='name', message="What's your name?"),
    Text(name='surname', message="What's your surname, {name}")
]
```

The value can be a `function`, with the next sign:

```
def get_message(answers): return str()
```

Example:

```
def get_message(answers):
    return "What's your name?"

Text(name='name', message= get_message)
```

Where `answers` is the dictionary with previous answers.

If the `message` is too long for the terminal, it will be cut to fit.

default

Stores the default value to be used as answer. This allow the user just to press *Enter* to use it. It is optional, using `None` if there is no input and no default value.

As in `message`, you can use a new format string or a function with the sign:

```
def get_default(answers): return str()
```

Where `answers` is a `dict` containing all previous answers.

Remember that it should be a list for *Checkbox* questions.

choices

Mandatory just for `Checkbox` and `List` questions; the rest of them do not use it.

It contains the list of selectable answers.

Its value can be a `list` of strings, new format style strings or pairs(tuples) or a *function* that returns that list, with the sign:

```
def get_choices(answers): return list(str())
```

If any of the list values is a pair, it should be a tuple like: `(label, value)`. Then the `label` will be shown but the `value` will be returned.

As before, the `answers` is a *dict* containing the previous answers.

validate

Optional attribute that allows the program to check if the answer is valid or not. It requires a *boolean* value or a *function* with the sign:

```
def validate(answers, current): return boolean()
```

Where `answers` is a *dict* with previous answers again and `current` is the current answer. If you want to customize the validation message, you can raise your own error with specific reason: `inquirer.errors.ValidationError('', reason='your reason that will be displayed to the user')` inside the validation function, but be aware that if the validation passes you still have to return *True*!

Example:

```
from inquirer import errors
import random

def validation_function(answers, current):
   if random.random() > 0.5:
      raise errors.ValidationError('', reason='Sorry, just have bad mood.')

   return True


Text('nothing', "Moody question", validate=validation_function)
Text('age', "How old are you?", validate=lambda _, c: 0 <= c < 120)
```

ignore

Questions are statically created and some of them may be optional depending on other answers. This attribute allows to control this by hiding the question.

It's value is *boolean* or a *function* with the sign:

```
def ignore(answers): return boolean()
```

where `answers` contains the *dict* of previous answers again.

Example:

```
questions = [
    inquirer.Text("name", message="What's your name?"),
    inquirer.Text(
        "surname",
        message="What's your surname, {name}?",
        ignore=lambda x: x["name"].lower() == "anonymous"
    ),
    inquirer.Confirm("married", message="Are you married?"),
    inquirer.Text(
        "time_married",
        message="How long have you been married?",
        ignore=lambda x: not x["married"]
    )
]
```

2.2.3 Path Question

Path Question accepts any valid path which can be both absolute or relative. By default it only validates the validity of the path. Except of validation it return normalized path and it expands home alias (~).

The Path Question have additional arguments for validating paths.

path_type

Validation argument that enables to enforce if the path should be aiming to file (`Path.FILE`) or directory (`Path.DIRECTORY`).

By default nothing is enforced (`Path.ANY`).

```
Path('log_file', 'Where should be log files located?', path_type=Path.DIRECTORY)
```

exists

Validation argument that enables to enforce if the provided path should or should not exists. Expects `True` if the path should exists, or `False` if the path should not exists.

By default nothing is enforced (`None`)

```
Path('config_file', 'Point me to your configuration file.', exists=True, path_type=Path.
↪File)
```

normalize_to_absolute_path

Argument which will enable normalization on the provided path. When enabled, in case of relative path would be provided the Question will normalize it to absolute path.

Expects `bool` value. Default `False`.

```
Path('config_file', 'Point me to your configuration file.', normalize_to_absolute_
↪path=True)
```

2.2.4 Creating the Question object

With this information, it is easy to create a `Question` object:

```
Text('name', "What's your name?")
```

It's possible to load the `Question` objects from a `dict`, or even the whole list of them, with the method `load_from_dict` and `load_from_list`, respectively.

The method `load_from_json` has been added as commodity to use JSON inputs instead. Here you have an example:

```
import os
import sys
from pprint import pprint

sys.path.append(os.path.realpath("."))
import inquirer  # noqa

with open("examples/test_questions.json") as fd:
    questions = inquirer.load_from_json(fd.read())

answers = inquirer.prompt(questions)

pprint(answers)
```

2.2.5 The prompter

The last step is to call the *prompter* With the list of `Question`:

```
answers = inquirer.prompt(questions)
```

This line will ask the user for information and will store the answeres in a dict, using the question name as **key** and the user response as **value**.

Remember the `prompt` always require a list of `Question` as input.

2.2.6 Themes

You can change the colorscheme and some icons passing a theme object defined in inquirer.themes There are Default and GreenPassion themes, but you can define your own via class, dict or json!

```
import inquirer
from inquirer.themes import GreenPassion

q = [
    inquirer.Text("name", message="Whats your name?", default="No one"),
    inquirer.List("jon", message="Does Jon Snow know?", choices=["yes", "no"], default=
↪"no"),
    inquirer.Checkbox(
        "kill_list", message="Who you want to kill?", choices=["Cersei", "Littlefinger",
↪"The Mountain"]
    ),
]

inquirer.prompt(q, theme=GreenPassion())
```

Result:

2.2.7 Shortcut functions

For one-off prompts, you can use the shortcut functions.

```
text = inquirer.text(message="Enter your username")
password = inquirer.password(message='Please enter your password'),
choice = inquirer.list_input("Public or private?",
                             choices=['public', 'private'])
correct = inquirer.confirm("This will delete all your current labels and "
                           "create a new ones. Continue?", default=False)
```

2.3 Examples

You can find all these examples at `examples` directory.

2.3.1 text.py

```
import os
import re
import sys
from pprint import pprint

sys.path.append(os.path.realpath("."))
import inquirer  # noqa


def phone_validation(answers, current):
    if not re.match(r"\+?\d[\d ]+\d", current):
        raise inquirer.errors.ValidationError("", reason="I don't like your phone number!
↪")

    return True


questions = [
    inquirer.Text("name", message="What's your name?"),
    inquirer.Text("surname", message="What's your surname, {name}?"),
    inquirer.Text(
        "phone",
        message="What's your phone number",
        validate=phone_validation,
    ),
]

answers = inquirer.prompt(questions)

pprint(answers)
```

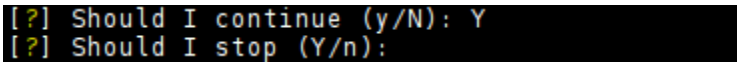
Result on something like:

```
[?] What's your name: Miguel
[?] What's your surname: Garcia
[?] What's your phone number: abc
>> Invalid value.
```

2.3.2 confirm.py

```
import os
import sys
from pprint import pprint

sys.path.append(os.path.realpath("."))
import inquirer  # noqa

questions = [
    inquirer.Confirm("continue", message="Should I continue"),
    inquirer.Confirm("stop", message="Should I stop", default=True),
]

answers = inquirer.prompt(questions)

pprint(answers)
```

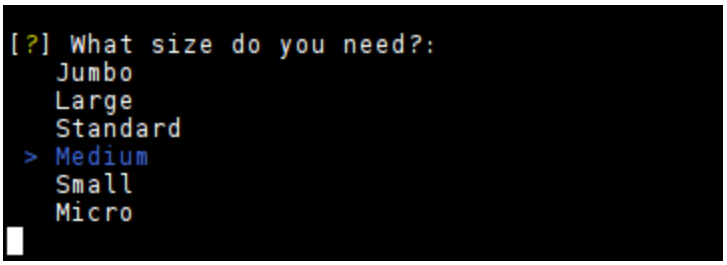
Result on something like:



2.3.3 list.py

```
import os
import sys
from pprint import pprint

sys.path.append(os.path.realpath("."))
import inquirer  # noqa

questions = [
    inquirer.List(
        "size",
        message="What size do you need?",
        choices=["Jumbo", "Large", "Standard", "Medium", "Small", "Micro"],
    ),
]

answers = inquirer.prompt(questions)

pprint(answers)
```

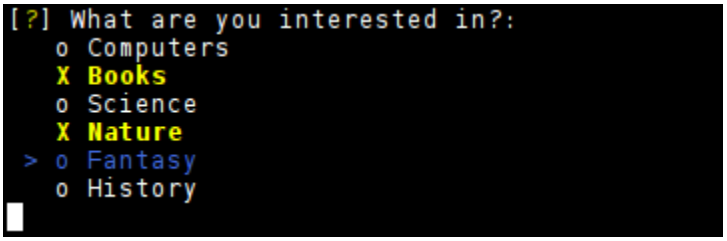
Result on something like:



2.3.4 checkbox.py

```
import os
import sys
from pprint import pprint

sys.path.append(os.path.realpath("."))
import inquirer  # noqa

questions = [
    inquirer.Checkbox(
        "interests",
        message="What are you interested in?",
        choices=["Computers", "Books", "Science", "Nature", "Fantasy", "History"],
        default=["Computers", "Books"],
    ),
]

answers = inquirer.prompt(questions)

pprint(answers)
```

Result on something like:



The `choices` list can also be a list of tuples. The first value in each tuple should be the label displayed to the user. The second value in each tuple should be the actual value for that option. This allows you to have the user choose options that are not plain strings in the code.

```
import os
import sys
from pprint import pprint

sys.path.append(os.path.realpath("."))
import inquirer  # noqa
```

(continues on next page)

(continued from previous page)

```
questions = [
    inquirer.Checkbox(
        "interests",
        message="What are you interested in?",
        choices=[
            ("Computers", "c"),
            ("Books", "b"),
            ("Science", "s"),
            ("Nature", "n"),
            ("Fantasy", "f"),
            ("History", "h"),
        ],
        default=["c", "b"],
    ),
]

answers = inquirer.prompt(questions)

pprint(answers)
```

2.3.5 theme.py

```
import inquirer
from inquirer.themes import GreenPassion

q = [
    inquirer.Text("name", message="Whats your name?", default="No one"),
    inquirer.List("jon", message="Does Jon Snow know?", choices=["yes", "no"], default=
↪"no"),
    inquirer.Checkbox(
        "kill_list", message="Who you want to kill?", choices=["Cersei", "Littlefinger",
↪"The Mountain"]
    ),
]

inquirer.prompt(q, theme=GreenPassion())
```

Result on something like:

2.4 Contributor Guide

Thank you for your interest in improving this project. This project is open-source under the MIT license and welcomes contributions in the form of bug reports, feature requests, and pull requests.

Here is a list of important resources for contributors:

- Source Code
- Documentation

- Issue Tracker
- Code of Conduct

2.4.1 How to report a bug

Report bugs on the Issue Tracker.

When filing an issue, make sure to answer these questions:

- Which operating system and Python version are you using?
- Which version of this project are you using?
- What did you do?
- What did you expect to see?
- What did you see instead?

The best way to get your bug fixed is to provide a test case, and/or steps to reproduce the issue.

2.4.2 How to request a feature

Request features on the Issue Tracker.

2.4.3 How to set up your development environment

You need Python 3.7+ and the following tools:

- Poetry
- Nox
- nox-poetry

Install the package with development requirements:

```
$ poetry install
```

You can now run an interactive Python session, or the command-line interface:

```
$ poetry run python
```

2.4.4 How to test the project

Run the full test suite:

```
$ nox
```

List the available Nox sessions:

```
$ nox --list-sessions
```

You can also run a specific Nox session. For example, invoke the unit test suite like this:

```
$ nox --session=tests
```

Unit tests are located in the `tests` directory, and are written using the pytest testing framework.

2.4.5 How to submit changes

Open a pull request to submit changes to this project.

Your pull request needs to meet the following guidelines for acceptance:

- The Nox test suite must pass without errors and warnings.
- Include unit tests. This project maintains 100% code coverage.
- If your changes add functionality, update the documentation accordingly.

Feel free to submit early, though—we can always iterate on this.

To run linting and code formatting checks before committing your change, you can install pre-commit as a Git hook by running the following command:

```
$ nox --session=pre-commit -- install
```

It is recommended to open an issue before starting work on anything. This will allow a chance to talk it over with the owners and validate your approach.

2.5 Contributor Covenant Code of Conduct

2.5.1 Our Pledge

We as members, contributors, and leaders pledge to make participation in our community a harassment-free experience for everyone, regardless of age, body size, visible or invisible disability, ethnicity, sex characteristics, gender identity and expression, level of experience, education, socio-economic status, nationality, personal appearance, race, religion, or sexual identity and orientation.

We pledge to act and interact in ways that contribute to an open, welcoming, diverse, inclusive, and healthy community.

2.5.2 Our Standards

Examples of behavior that contributes to a positive environment for our community include:

- Demonstrating empathy and kindness toward other people
- Being respectful of differing opinions, viewpoints, and experiences
- Giving and gracefully accepting constructive feedback
- Accepting responsibility and apologizing to those affected by our mistakes, and learning from the experience
- Focusing on what is best not just for us as individuals, but for the overall community

Examples of unacceptable behavior include:

- The use of sexualized language or imagery, and sexual attention or advances of any kind
- Trolling, insulting or derogatory comments, and personal or political attacks
- Public or private harassment

- Publishing others' private information, such as a physical or email address, without their explicit permission
- Other conduct which could reasonably be considered inappropriate in a professional setting

2.5.3 Enforcement Responsibilities

Community leaders are responsible for clarifying and enforcing our standards of acceptable behavior and will take appropriate and fair corrective action in response to any behavior that they deem inappropriate, threatening, offensive, or harmful.

Community leaders have the right and responsibility to remove, edit, or reject comments, commits, code, wiki edits, issues, and other contributions that are not aligned to this Code of Conduct, and will communicate reasons for moderation decisions when appropriate.

2.5.4 Scope

This Code of Conduct applies within all community spaces, and also applies when an individual is officially representing the community in public spaces. Examples of representing our community include using an official e-mail address, posting via an official social media account, or acting as an appointed representative at an online or offline event.

2.5.5 Enforcement

Instances of abusive, harassing, or otherwise unacceptable behavior may be reported to the community leaders responsible for enforcement at miguelangel.garcia@gmail.com. All complaints will be reviewed and investigated promptly and fairly.

All community leaders are obligated to respect the privacy and security of the reporter of any incident.

2.5.6 Enforcement Guidelines

Community leaders will follow these Community Impact Guidelines in determining the consequences for any action they deem in violation of this Code of Conduct:

1. Correction

Community Impact: Use of inappropriate language or other behavior deemed unprofessional or unwelcome in the community.

Consequence: A private, written warning from community leaders, providing clarity around the nature of the violation and an explanation of why the behavior was inappropriate. A public apology may be requested.

2. Warning

Community Impact: A violation through a single incident or series of actions.

Consequence: A warning with consequences for continued behavior. No interaction with the people involved, including unsolicited interaction with those enforcing the Code of Conduct, for a specified period of time. This includes avoiding interactions in community spaces as well as external channels like social media. Violating these terms may lead to a temporary or permanent ban.

3. Temporary Ban

Community Impact: A serious violation of community standards, including sustained inappropriate behavior.

Consequence: A temporary ban from any sort of interaction or public communication with the community for a specified period of time. No public or private interaction with the people involved, including unsolicited interaction with those enforcing the Code of Conduct, is allowed during this period. Violating these terms may lead to a permanent ban.

4. Permanent Ban

Community Impact: Demonstrating a pattern of violation of community standards, including sustained inappropriate behavior, harassment of an individual, or aggression toward or disparagement of classes of individuals.

Consequence: A permanent ban from any sort of public interaction within the community.

2.5.7 Attribution

This Code of Conduct is adapted from the Contributor Covenant, version 2.0, available at https://www.contributor-covenant.org/version/2/0/code_of_conduct/.

Community Impact Guidelines were inspired by Mozilla's code of conduct enforcement ladder.

For answers to common questions about this code of conduct, see the FAQ at https://www.contributor-covenant.org/faq. Translations are available at https://www.contributor-covenant.org/translations.

2.6 The MIT License (MIT)

OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

License taken from MIT license.

2.7 inquirer

2.7.1 inquirer package

Subpackages

inquirer.render package

Subpackages

inquirer.render.console package

Submodules

inquirer.render.console.base module

class inquirer.render.console.base.**BaseConsoleRender**(*question*, *theme=None*, *terminal=None*, *show_default=False*, **args*, ***kwargs*)

Bases: object

get_current_value()

get_header()

get_options()

handle_validation_error(*error*)

process_input(*pressed*)

title_inline = False

Module contents

class inquirer.render.console.**ConsoleRender**(*event_generator=None*, *theme=None*, **args*, ***kwargs*)

Bases: object

clear_bottombar()

clear_eos()

property height

print_line(*base*, *lf=True*, ***kwargs*)

print_str(*base*, *lf=False*, ***kwargs*)

render(*question*, *answers=None*)

render_error(*message*)

render_factory(*question_type*)

render_in_bottombar(*message*)

property width

Module contents

class inquirer.render.**Render**(*impl=<class 'inquirer.render.console.ConsoleRender'>*)
Bases: object

render(*question*, *answers*)

Submodules

inquirer.errors module

exception inquirer.errors.**Aborted**
Bases: *inquirer.errors.InquirerError*

exception inquirer.errors.**EndOfInput**(*selection*, **args*)
Bases: *inquirer.errors.InquirerError*

exception inquirer.errors.**InquirerError**
Bases: Exception

exception inquirer.errors.**ThemeError**
Bases: AttributeError

exception inquirer.errors.**UnknownQuestionTypeError**
Bases: *inquirer.errors.InquirerError*

exception inquirer.errors.**ValidationError**(*value*, *reason=None*, **args*)
Bases: *inquirer.errors.InquirerError*

inquirer.events module

class inquirer.events.**Event**
Bases: object

class inquirer.events.**KeyEventGenerator**(*key_generator=None*)
Bases: object

next()

class inquirer.events.**KeyPressed**(*value*)
Bases: *inquirer.events.Event*

class inquirer.events.**Repaint**
Bases: *inquirer.events.Event*

inquirer.prompt module

inquirer.prompt.**prompt**(*questions*, *render=None*, *answers=None*, *theme=<inquirer.themes.Default object>*, *raise_keyboard_interrupt=False*)

inquirer.questions module

Module that implements the questions types.

class inquirer.questions.**Checkbox**(*name*, *message=''*, *choices=None*, *default=None*, *ignore=False*, *validate=True*, *show_default=False*)

Bases: *inquirer.questions.Question*

kind = 'checkbox'

class inquirer.questions.**Confirm**(*name*, *default=False*, ***kwargs*)

Bases: *inquirer.questions.Question*

kind = 'confirm'

class inquirer.questions.**Editor**(*name*, *message=''*, *default=None*, ***kwargs*)

Bases: *inquirer.questions.Text*

kind = 'editor'

class inquirer.questions.**List**(*name*, *message=''*, *choices=None*, *default=None*, *ignore=False*, *validate=True*, *carousel=False*)

Bases: *inquirer.questions.Question*

kind = 'list'

class inquirer.questions.**Password**(*name*, *echo='*'*, ***kwargs*)

Bases: *inquirer.questions.Text*

kind = 'password'

class inquirer.questions.**Path**(*name*, *default=None*, *path_type='any'*, *exists=None*, *normalize_to_absolute_path=False*, ***kwargs*)

Bases: *inquirer.questions.Text*

ANY = 'any'

DIRECTORY = 'directory'

FILE = 'file'

kind = 'path'

normalize_value(*value*)

validate(*current*)

class inquirer.questions.**Question**(*name*, *message=''*, *choices=None*, *default=None*, *ignore=False*, *validate=True*, *show_default=False*)

Bases: `object`

property choices

property choices_generator

property default

property ignore

kind = 'base question'

property message

validate(*current*)

class inquirer.questions.**TaggedValue**(*label*, *value*)
Bases: object

class inquirer.questions.**Text**(*name*, *message=''*, *default=None*, ***kwargs*)
Bases: *inquirer.questions.Question*

kind = 'text'

inquirer.questions.**is_pathname_valid**(*pathname*)
True if the passed pathname is a valid pathname for the current OS; *False* otherwise.

inquirer.questions.**load_from_dict**(*question_dict*)
Load one question from a dict.

It requires the keys 'name' and 'kind'.

Returns The Question object with associated data.

Return type *inquirer.questions.Question*

inquirer.questions.**load_from_json**(*question_json*)
Load Questions from a JSON string.

Returns A list of Question objects with associated data if the JSON contains a list or a Question if the JSON contains a dict.

Return type list | dict

inquirer.questions.**load_from_list**(*question_list*)
Load a list of questions from a list of dicts.

It requires the keys 'name' and 'kind' for each dict.

Returns A list of Question objects with associated data.

Return type list[*inquirer.questions.Question*]

inquirer.questions.**question_factory**(*kind*, **args*, ***kwargs*)

inquirer.shortcuts module

inquirer.shortcuts.**checkbox**(*message*, *render=None*, ***kwargs*)

inquirer.shortcuts.**confirm**(*message*, *render=None*, ***kwargs*)

inquirer.shortcuts.**editor**(*message*, *render=None*, ***kwargs*)

inquirer.shortcuts.**list_input**(*message*, *render=None*, ***kwargs*)

inquirer.shortcuts.**password**(*message*, *render=None*, ***kwargs*)

inquirer.shortcuts.**path**(*message*, *render=None*, ***kwargs*)

inquirer.shortcuts.**text**(*message*, *render=None*, ***kwargs*)

inquirer.themes module

class inquirer.themes.**Default**
Bases: *inquirer.themes.Theme*

class inquirer.themes.**GreenPassion**
Bases: *inquirer.themes.Theme*

class inquirer.themes.**Theme**
Bases: object

inquirer.themes.**load_theme_from_dict**(*dict_theme*)
Load a theme from a dict.

Expected format:

```
>>> {
...     "Question": {
...         "mark_color": "yellow",
...         "brackets_color": "normal",
...         ...
...     },
...     "List": {
...         "selection_color": "bold_blue",
...         "selection_cursor": "->"
...     }
... }
```

Color values should be string representing valid blessings.Terminal colors.

inquirer.themes.**load_theme_from_json**(*json_theme*)
Load a theme from a json.

Expected format:

```
>>> {
...     "Question": {
...         "mark_color": "yellow",
...         "brackets_color": "normal",
...         ...
...     },
...     "List": {
...         "selection_color": "bold_blue",
...         "selection_cursor": "->"
...     }
... }
```

Color values should be string representing valid blessings.Terminal colors.

CHAPTER

THREE

INDICES AND TABLES

- genindex
- modindex

PYTHON MODULE INDEX

i

INDEX

L

M

N

P

Q

R

T

U

V

W