Questionary

Release 2.0.1

Questionary

Sep 08, 2023

CONTENTS

Questionary is a Python library for effortlessly building pretty command line interfaces

It makes it very easy to query your user for input. You need your user to confirm a destructive action or enter a file path? We've got you covered:

Creating your first prompt is just a few key strokes away

```
import questionary

first_name = questionary.text("What's your first name").ask()
```

This prompt will ask the user to provide free text input and the result is stored in `first_name`.

You can install Questionary using pip (for details, head over to the *Installation* page):

```
$ pip install questionary
```

Ready to go? Check out the *Quickstart*.

CHAPTER

ONE

LICENSE

Licensed under the MIT License. Copyright 2020 Tom Bocklisch.

1.1 Installation

1.1.1 Use a Published Release

To install Questionary, simply run this command in your terminal of choice:

```
$ pip install questionary
```

1.1.2 Build from Source

Installing from source code is only necessary, if you want to make changes to the Questionary source code. Questionary is actively developed on GitHub.

You can either clone the public repository:

```
$ git clone git@github.com:tmbo/questionary.git
```

Or, download the tarball:

```
$ curl -OL https://github.com/tmbo/questionary/tarball/master
```

Note: If you are using windows, you can also download a zip instead:

```
$ curl -OL https://github.com/tmbo/questionary/zipball/master
```

Questionary uses Poetry for packaging and dependency management. If you want to build Questionary from source, you must install Poetry first:

```
$ curl -sSL https://raw.githubusercontent.com/python-poetry/poetry/master/get-poetry.py␣
↪| python3
```

There are several other ways to install Poetry, as seen in the official guide.

To install Questionary and its dependencies in editable mode, execute

```
$ make install
```

1.2 Quickstart

Questionary supports two different concepts:

- creating a **single question** for the user

  ```
  questionary.password("What's your secret?").ask()
  ```

- creating a **form with multiple questions** asked one after another

  ```
  answers = questionary.form(
      first = questionary.confirm("Would you like the next question?", default=True),
      second = questionary.select("Select item", choices=["item1", "item2", "item3"])
  ).ask()
  ```

1.2.1 Asking a Single Question

Questionary ships with a lot of different *Question Types* to provide the right prompt for the right question. All of them work in the same way though. Firstly, you create a question:

```
import questionary

question = questionary.text("What's your first name")
```

and secondly, you need to prompt the user to answer it:

```
answer = question.ask()
```

Since our question is a `text` prompt, `answer` will contain the text the user typed after they submitted it.

You can concatenate creating and asking the question in a single line if you like, e.g.

```
import questionary

answer = questionary.text("What's your first name").ask()
```

Note: There are a lot more question types apart from `text`. For a description of the different question types, head over to the *Question Types*.

1.2.2 Asking Multiple Questions

You can use the *form()* function to ask a collection of *Questions*. The questions will be asked in the order they are passed to *:meth:`~questionary.form`*.

```
import questionary

answers = questionary.form(
    first = questionary.confirm("Would you like the next question?", default=True),
    second = questionary.select("Select item", choices=["item1", "item2", "item3"])
).ask()

print(answers)
```

The printed output will have the following format:

```
{'first': True, 'second': 'item2'}
```

The *prompt()* function also allows you to ask a collection of questions, however instead of taking *Question* instances, it takes a dictionary:

```
import questionary

questions = [
  {
    "type": "confirm",
    "name": "first",
    "message": "Would you like the next question?",
    "default": True,
  },
  {
    "type": "select",
    "name": "second",
    "message": "Select item",
    "choices": ["item1", "item2", "item3"],
  },
]

questionary.prompt(questions)
```

The format of the returned answers is the same as the one for *form()*. You can find more details on the configuration dictionaries in *Create Questions from Dictionaries*.

1.3 Question Types

The different question types are meant to cover different use cases. The parameters and configuration options are explained in detail for each type. But before we get into to many details, here is a **cheatsheet with the available question types**:

- use *Text* to ask for **free text** input
- use *Password* to ask for free text where the **text is hidden**
- use *File Path* to ask for a **file or directory** path with autocompletion

- use *Confirmation* to ask a **yes or no** question
- use *Select* to ask the user to select **one item** from a beautiful list
- use *Raw Select* to ask the user to select **one item** from a list
- use *Checkbox* to ask the user to select **any number of items** from a list
- use *Autocomplete* to ask for free text with **autocomplete help**
- use *Press Any Key To Continue* to ask the user to **press any key to continue**

1.3.1 Text

`questionary.text`(*default=''*, *validate=None*, *qmark='?'*, *style=None*, *multiline=False*, *instruction=None*, *lexer=None*, ***kwargs*)

Prompt the user to enter a free text message.

This question type can be used to prompt the user for some text input.

Example

```
>>> import questionary
>>> questionary.text("What's your first name?").ask()
? What's your first name? Tom
'Tom'
```

This is just a really basic example, the prompt can be customised using the parameters.

Parameters

- **message** (str) – Question text.
- **default** (str) – Default value will be returned if the user just hits enter.
- **validate** (Any) – Require the entered value to pass a validation. The value can not be submitted until the validator accepts it (e.g. to check minimum password length).

  This can either be a function accepting the input and returning a boolean, or an class reference to a subclass of the prompt toolkit Validator class.
- **qmark** (str) – Question prefix displayed in front of the question. By default this is a `?`.
- **style** (Optional[Style]) – A custom color and style for the question parts. You can configure colors as well as font types for different elements.
- **multiline** (bool) – If `True`, multiline input will be enabled.
- **instruction** (Optional[str]) – Write instructions for the user if needed. If `None` and `multiline=True`, some instructions will appear.
- **lexer** (Optional[Lexer]) – Supply a valid lexer to style the answer. Leave empty to use a simple one by default.
- **kwargs** (Any) – Additional arguments, they will be passed to prompt toolkit.

Returns

Question instance, ready to be prompted (using `.ask()`).

Return type

Question

1.3.2 Password

`questionary.`**`password`**`(`*default=''*, *validate=None*, *qmark='?'*, *style=None*, ***kwargs*`)`

A text input where a user can enter a secret which won't be displayed on the CLI.

This question type can be used to prompt the user for information that should not be shown in the command line. The typed text will be replaced with `*`.

Example

```
>>> import questionary
>>> questionary.password("What's your secret?").ask()
? What's your secret? ********
'secret42'
```

This is just a really basic example, the prompt can be customised using the parameters.

Parameters

- **`message`** (`str`) – Question text.
- **`default`** (`str`) – Default value will be returned if the user just hits enter.
- **`validate`** (`Any`) – Require the entered value to pass a validation. The value can not be submitted until the validator accepts it (e.g. to check minimum password length).

  This can either be a function accepting the input and returning a boolean, or an class reference to a subclass of the prompt toolkit Validator class.
- **`qmark`** (`str`) – Question prefix displayed in front of the question. By default this is a `?`.
- **`style`** (`Optional[Style]`) – A custom color and style for the question parts. You can configure colors as well as font types for different elements.
- **`kwargs`** (*`Any`*) –

Returns

Question instance, ready to be prompted (using `.ask()`).

Return type

`Question`

1.3.3 File Path

`questionary.`**`path`**`(`*default=''*, *qmark='?'*, *validate=None*, *completer=None*, *style=None*, *only_directories=False*, *get_paths=None*, *file_filter=None*, *complete_style=CompleteStyle.MULTI_COLUMN*, ***kwargs*`)`

A text input for a file or directory path with autocompletion enabled.

Example

```
>>> import questionary
>>> questionary.path(
>>>    "What's the path to the projects version file?"
>>> ).ask()
? What's the path to the projects version file? ./pyproject.toml
'./pyproject.toml'
```

This is just a really basic example, the prompt can be customized using the parameters.

Parameters

- **message** (`str`) – Question text.
- **default** (`str`) – Default return value (single value).
- **qmark** (`str`) – Question prefix displayed in front of the question. By default this is a `?`.
- **complete_style** (`CompleteStyle`) – How autocomplete menu would be shown, it could be `COLUMN` `MULTI_COLUMN` or `READLINE_LIKE` from `prompt_toolkit.shortcuts.CompleteStyle`.
- **validate** (`Any`) – Require the entered value to pass a validation. The value can not be submitted until the validator accepts it (e.g. to check minimum password length).

  This can either be a function accepting the input and returning a boolean, or an class reference to a subclass of the prompt toolkit Validator class.
- **completer** (`Optional[Completer]`) – A custom completer to use in the prompt. For more information, see this.
- **style** (`Optional[Style]`) – A custom color and style for the question parts. You can configure colors as well as font types for different elements.
- **only_directories** (`bool`) – Only show directories in auto completion. This option does not do anything if a custom `completer` is passed.
- **get_paths** (`Optional[Callable[[], List[str]]]`) – Set a callable to generate paths to traverse for suggestions. This option does not do anything if a custom `completer` is passed.
- **file_filter** (`Optional[Callable[[str], bool]]`) – Optional callable to filter suggested paths. Only paths where the passed callable evaluates to `True` will show up in the suggested paths. This does not validate the typed path, e.g. it is still possible for the user to enter a path manually, even though this filter evaluates to `False`. If in addition to filtering suggestions you also want to validate the result, use `validate` in combination with the `file_filter`.
- **kwargs** (*`Any`*) –

Returns

Question instance, ready to be prompted (using `.ask()`).

Return type

`Question`

1.3.4 Confirmation

`questionary.confirm`(*default=True*, *qmark='?'*, *style=None*, *auto_enter=True*, *instruction=None*, ***kwargs*)

A yes or no question. The user can either confirm or deny.

This question type can be used to prompt the user for a confirmation of a yes-or-no question. If the user just hits enter, the default value will be returned.

Example

```
>>> import questionary
>>> questionary.confirm("Are you amazed?").ask()
? Are you amazed? Yes
True
```

This is just a really basic example, the prompt can be customised using the parameters.

Parameters

- **message** (`str`) – Question text.
- **default** (`bool`) – Default value will be returned if the user just hits enter.
- **qmark** (`str`) – Question prefix displayed in front of the question. By default this is a ?.
- **style** (`Optional[Style]`) – A custom color and style for the question parts. You can configure colors as well as font types for different elements.
- **auto_enter** (`bool`) – If set to *False*, the user needs to press the 'enter' key to accept their answer. If set to *True*, a valid input will be accepted without the need to press 'Enter'.
- **instruction** (`Optional[str]`) – A message describing how to proceed through the confirmation prompt.
- **kwargs** (*Any*) –

Returns
Question instance, ready to be prompted (using *.ask()*).

Return type
`Question`

1.3.5 Select

`questionary.select`(*choices*, *default=None*, *qmark='?'*, *pointer='»'*, *style=None*, *use_shortcuts=False*, *use_arrow_keys=True*, *use_indicator=False*, *use_jk_keys=True*, *use_emacs_keys=True*, *show_selected=False*, *instruction=None*, ***kwargs*)

A list of items to select **one** option from.

The user can pick one option and confirm it (if you want to allow the user to select multiple options, use `questionary.checkbox()` instead).

Example

```
>>> import questionary
>>> questionary.select(
...     "What do you want to do?",
...     choices=[
...         "Order a pizza",
...         "Make a reservation",
...         "Ask for opening hours"
...     ]).ask()
? What do you want to do? Order a pizza
'Order a pizza'
```

This is just a really basic example, the prompt can be customised using the parameters.

Parameters

- **message** (str) – Question text
- **choices** (Sequence[Union[str, *Choice*, Dict[str, Any]]]) – Items shown in the selection, this can contain *Choice* or or *Separator* objects or simple items as strings. Passing *Choice* objects, allows you to configure the item more (e.g. preselecting it or disabling it).
- **default** (Union[str, *Choice*, Dict[str, Any], None]) – A value corresponding to a selectable item in the choices, to initially set the pointer position to.
- **qmark** (str) – Question prefix displayed in front of the question. By default this is a ?.
- **pointer** (Optional[str]) – Pointer symbol in front of the currently highlighted element. By default this is a ». Use None to disable it.
- **instruction** (Optional[str]) – A hint on how to navigate the menu. It's (Use shortcuts) if only use_shortcuts is set to True, (Use arrow keys or shortcuts) if use_arrow_keys & use_shortcuts are set and (Use arrow keys) by default.
- **style** (Optional[Style]) – A custom color and style for the question parts. You can configure colors as well as font types for different elements.
- **use_indicator** (bool) – Flag to enable the small indicator in front of the list highlighting the current location of the selection cursor.
- **use_shortcuts** (bool) – Allow the user to select items from the list using shortcuts. The shortcuts will be displayed in front of the list items. Arrow keys, j/k keys and shortcuts are not mutually exclusive.
- **use_arrow_keys** (bool) – Allow the user to select items from the list using arrow keys. Arrow keys, j/k keys and shortcuts are not mutually exclusive.
- **use_jk_keys** (bool) – Allow the user to select items from the list using *j* (down) and *k* (up) keys. Arrow keys, j/k keys and shortcuts are not mutually exclusive.
- **use_emacs_keys** (bool) – Allow the user to select items from the list using *Ctrl+N* (down) and *Ctrl+P* (up) keys. Arrow keys, j/k keys, emacs keys and shortcuts are not mutually exclusive.
- **show_selected** (bool) – Display current selection choice at the bottom of list.
- **kwargs** (*Any*) –

Returns

Question instance, ready to be prompted (using .ask()).

Return type
Question

1.3.6 Raw Select

`questionary.rawselect`(*choices*, *default=None*, *qmark='?'*, *pointer='»'*, *style=None*, ***kwargs*)

Ask the user to select one item from a list of choices using shortcuts.

The user can only select one option.

Example

```
>>> import questionary
>>> questionary.rawselect(
...     "What do you want to do?",
...     choices=[
...         "Order a pizza",
...         "Make a reservation",
...         "Ask for opening hours"
...     ]).ask()
? What do you want to do? Order a pizza
'Order a pizza'
```

This is just a really basic example, the prompt can be customised using the parameters.

Parameters

- **message** (str) – Question text.
- **choices** (Sequence[Union[str, *Choice*, Dict[str, Any]]]) – Items shown in the selection, this can contain *Choice* or or *Separator* objects or simple items as strings. Passing *Choice* objects, allows you to configure the item more (e.g. preselecting it or disabling it).
- **default** (Optional[str]) – Default return value (single value).
- **qmark** (str) – Question prefix displayed in front of the question. By default this is a `?`.
- **pointer** (Optional[str]) – Pointer symbol in front of the currently highlighted element. By default this is a `»`. Use `None` to disable it.
- **style** (Optional[Style]) – A custom color and style for the question parts. You can configure colors as well as font types for different elements.
- **kwargs** (*Any*) –

Returns
Question instance, ready to be prompted (using `.ask()`).

Return type
Question

1.3.7 Checkbox

questionary.**checkbox**(*choices*, *default=None*, *validate=<function <lambda>>*, *qmark='?'*, *pointer='»'*, *style=None*, *initial_choice=None*, *use_arrow_keys=True*, *use_jk_keys=True*, *use_emacs_keys=True*, *instruction=None*, ***kwargs*)

Ask the user to select from a list of items.

This is a multiselect, the user can choose one, none or many of the items.

Example

```
>>> import questionary
>>> questionary.checkbox(
...     'Select toppings',
...     choices=[
...         "Cheese",
...         "Tomato",
...         "Pineapple",
...     ]).ask()
? Select toppings done (2 selections)
['Cheese', 'Pineapple']
```

This is just a really basic example, the prompt can be customised using the parameters.

Parameters

- **message** (str) – Question text
- **choices** (Sequence[Union[str, *Choice*, Dict[str, Any]]]) – Items shown in the selection, this can contain *Choice* or or *Separator* objects or simple items as strings. Passing *Choice* objects, allows you to configure the item more (e.g. preselecting it or disabling it).
- **default** (Optional[str]) – Default return value (single value). If you want to preselect multiple items, use `Choice("foo", checked=True)` instead.
- **validate** (Callable[[List[str]], Union[bool, str]]) – Require the entered value to pass a validation. The value can not be submitted until the validator accepts it (e.g. to check minimum password length).

  This should be a function accepting the input and returning a boolean. Alternatively, the return value may be a string (indicating failure), which contains the error message to be displayed.
- **qmark** (str) – Question prefix displayed in front of the question. By default this is a ?.
- **pointer** (Optional[str]) – Pointer symbol in front of the currently highlighted element. By default this is a ». Use `None` to disable it.
- **style** (Optional[Style]) – A custom color and style for the question parts. You can configure colors as well as font types for different elements.
- **initial_choice** (Union[str, *Choice*, Dict[str, Any], None]) – A value corresponding to a selectable item in the choices, to initially set the pointer position to.
- **use_arrow_keys** (bool) – Allow the user to select items from the list using arrow keys.
- **use_jk_keys** (bool) – Allow the user to select items from the list using *j* (down) and *k* (up) keys.

- **use_emacs_keys** (bool) – Allow the user to select items from the list using *Ctrl+N* (down) and *Ctrl+P* (up) keys.
- **instruction** (Optional[str]) – A message describing how to navigate the menu.
- **kwargs** (*Any*) –

Returns
Question instance, ready to be prompted (using `.ask()`).

Return type
Question

1.3.8 Autocomplete

`questionary.autocomplete`(*choices*, *default=''*, *qmark='?'*, *completer=None*, *meta_information=None*, *ignore_case=True*, *match_middle=True*, *complete_style=CompleteStyle.COLUMN*, *validate=None*, *style=None*, ***kwargs*)

Prompt the user to enter a message with autocomplete help.

Example

```
>>> import questionary
>>> questionary.autocomplete(
...    'Choose ant specie',
...    choices=[
...         'Camponotus pennsylvanicus',
...         'Linepithema humile',
...         'Eciton burchellii',
...         "Atta colombica",
...         'Polyergus lucidus',
...         'Polyergus rufescens',
...    ]).ask()
? Choose ant specie Atta colombica
'Atta colombica'
```

This is just a really basic example, the prompt can be customised using the parameters.

Parameters

- **message** (str) – Question text
- **choices** (List[str]) – Items shown in the selection, this contains items as strings
- **default** (str) – Default return value (single value).
- **qmark** (str) – Question prefix displayed in front of the question. By default this is a ?
- **completer** (Optional[Completer]) – A prompt_toolkit `prompt_toolkit.completion.Completion` implementation. If not set, a questionary completer implementation will be used.
- **meta_information** (Optional[Dict[str, Any]]) – A dictionary with information/anything about choices.
- **ignore_case** (bool) – If true autocomplete would ignore case.

- **match_middle** (bool) – If true autocomplete would search in every string position not only in string begin.
- **complete_style** (CompleteStyle) – How autocomplete menu would be shown, it could be `COLUMN` `MULTI_COLUMN` or `READLINE_LIKE` from `prompt_toolkit.shortcuts.CompleteStyle`.
- **validate** (Any) – Require the entered value to pass a validation. The value can not be submitted until the validator accepts it (e.g. to check minimum password length).

  This can either be a function accepting the input and returning a boolean, or an class reference to a subclass of the prompt toolkit Validator class.
- **style** (Optional[Style]) – A custom color and style for the question parts. You can configure colors as well as font types for different elements.
- **kwargs** (*Any*) –

Returns
Question instance, ready to be prompted (using `.ask()`).

Return type
Question

1.3.9 Printing Formatted Text

`questionary.print(`*style=None*, ***kwargs*`)`

Print formatted text.

Sometimes you want to spice up your printed messages a bit, *questionary.print()* is a helper to do just that.

Example

```
>>> import questionary
>>> questionary.print("Hello World ", style="bold italic fg:darkred")
Hello World
```

Parameters

- **text** (str) – Text to be printed.
- **style** (Optional[str]) – Style used for printing. The style argument uses the prompt toolkit style strings.
- **kwargs** (*Any*) –

Return type
None

1.3.10 Press Any Key To Continue

`questionary.press_any_key_to_continue`(*style=None*, ***kwargs*)

Wait until user presses any key to continue.

Example

```
>>> import questionary
>>> questionary.press_any_key_to_continue().ask()
 Press any key to continue...
''
```

Parameters

- **message** (Optional[str]) – Question text. Defaults to `"Press any key to continue...“`
- **style** (Optional[Style]) – A custom color and style for the question parts. You can configure colors as well as font types for different elements.
- **kwargs** (*Any*) –

Returns
Question instance, ready to be prompted (using `.ask()`).

Return type
Question

1.4 Advanced Concepts

This page describes some of the more advanced uses of Questionary.

1.4.1 Validation

Many of the prompts support a `validate` argument, which allows the answer to be validated before being submitted. A user can not submit an answer if it doesn't pass the validation.

The example below shows *text()* input with a validation:

```
import questionary
from questionary import Validator, ValidationError, prompt

class NameValidator(Validator):
    def validate(self, document):
        if len(document.text) == 0:
            raise ValidationError(
                message="Please enter a value",
                cursor_position=len(document.text),
            )

questionary.text("What's your name?", validate=NameValidator).ask()
```

In this example, the user can not enter a non empty value. If the prompt is submitted without a value. Questionary will show the error message and reject the submission until the user enters a value.

Alternatively, we can replace the `NameValidator` class with a simple function, as seen below:

```python
import questionary

print(questionary.text(
  "What's your name?",
  validate=lambda text: True if len(text) > 0 else "Please enter a value"
).ask())
```

Finally, if we do not care about the error message being displayed, we can omit the error message from the final example to use the default:

```python
import questionary

print(questionary.text("What's your name?", validate=lambda text: len(text) > 0).ask())
```

example

The *`checkbox()`* prompt does not support passing a `Validator`. See the *API Reference* for all the prompts which support the `validate` parameter.

A Validation Example using the Password Question

Here we see an example of `validate` being used on a *`password()`* prompt to enforce complexity requirements:

```python
import re
import questionary

def password_validator(password):

    if len(password) < 10:
        return "Password must be at least 10 characters"

    elif re.search("[0-9]", password) is None:
        return "Password must contain a number"

    elif re.search("[a-z]", password) is None:
        return "Password must contain an lower-case letter"

    elif re.search("[A-Z]", password) is None:
        return "Password must contain an upper-case letter"

    else:
        return True

print(questionary.password("Enter your password", validate=password_validator).ask())
```

1.4.2 Keyboard Interrupts

Prompts can be invoked in either a 'safe' or 'unsafe' way. The safe way captures keyboard interrupts and handles them by catching the interrupt and returning `None` for the asked question. If a question is asked using unsafe functions, the keyboard interrupts are not caught.

Safe

The following are safe (capture keyboard interrupts):

- *prompt()*;
- *ask* on *Form* (returned by *form()*);
- *ask* on *Question*, which is returned by the various prompt functions (e.g. *text()*, *checkbox()*).

When a keyboard interrupt is captured, the message `"Cancelled by user"` is displayed (or a custom message, if one is given) and `None` is returned. Here is an example:

```
# Questionary handles keyboard interrupt and returns `None` if the
# user hits e.g. `Ctrl+C`
prompt(...)
```

Unsafe

The following are unsafe (do not catch keyboard interrupts):

- *unsafe_prompt()*;
- *unsafe_ask* on *Form* (returned by *form()*);
- *unsafe_ask* on *Question*, which is returned by the various prompt functions (e.g. *text()*, *checkbox()*).

As a caller you must handle keyboard interrupts yourself when calling these methods. Here is an example:

```
try:
    unsafe_prompt(...)

except KeyboardInterrupt:
    # your chance to handle the keyboard interrupt
    print("Cancelled by user")
```

1.4.3 Asynchronous Usage

If you are running asynchronous code and you want to avoid blocking your async loop, you can ask your questions using `await`. *questionary.Question* and *questionary.Form* have `ask_async` and `unsafe_ask_async` methods to invoke the question using `asyncio`:

```
import questionary

answer = await questionary.text("What's your name?").ask_async()
```

1.4.4 Themes & Styling

You can customize all the colors used for the prompts. Every part of the prompt has an identifier, which you can use to style it. Let's create your own custom style:

```
from questionary import Style

custom_style_fancy = Style([
    ('qmark', 'fg:#673ab7 bold'),       # token in front of the question
    ('question', 'bold'),               # question text
    ('answer', 'fg:#f44336 bold'),      # submitted answer text behind the question
    ('pointer', 'fg:#673ab7 bold'),     # pointer used in select and checkbox prompts
    ('highlighted', 'fg:#673ab7 bold'), # pointed-at choice in select and checkbox␣
↪prompts
    ('selected', 'fg:#cc5454'),         # style for a selected item of a checkbox
    ('separator', 'fg:#cc5454'),        # separator in lists
    ('instruction', ''),                # user instructions for select, rawselect,␣
↪checkbox
    ('text', ''),                       # plain text
    ('disabled', 'fg:#858585 italic')   # disabled choices for select and checkbox␣
↪prompts
])
```

To use the custom style, you need to pass it to the question as a parameter:

```
questionary.text("What's your phone number", style=custom_style_fancy).ask()
```

Note: Default values will be used for any token types not specified in your custom style.

Styling Choices in Select & Checkbox Questions

It is also possible to use a list of token tuples as a `Choice` title to change how an option is displayed in *`questionary.select`* and *`questionary.checkbox`*. Make sure to define any additional styles as part of your custom style definition.

```
import questionary
from questionary import Choice, Style

custom_style_fancy = questionary.Style([
    ("highlighted", "bold"),  # style for a token which should appear highlighted
])

choices = [Choice(title=[("class:text", "order a "),
                         ("class:highlighted", "big pizza")])]

questionary.select(
   "What do you want to do?",
   choices=choices,
   style=custom_style_fancy).ask()
```

1.4.5 Conditionally Skip Questions

Sometimes it is helpful to be able to skip a question based on a condition. To avoid the need for an `if` around the question, you can pass the condition when you create the question:

```
import questionary

DISABLED = True
response = questionary.confirm("Are you amazed?").skip_if(DISABLED, default=True).ask()
```

If the condition (in this case `DISABLED`) is `True`, the question will be skipped and the default value gets returned, otherwise the user will be prompted as usual and the default value will be ignored.

1.4.6 Create Questions from Dictionaries

Instead of creating questions using the Python functions, you can also create them using a configuration dictionary:

```
from questionary import prompt

questions = [
    {
        'type': 'text',
        'name': 'phone',
        'message': "What's your phone number",
    },
    {
        'type': 'confirm',
        'message': 'Do you want to continue?',
        'name': 'continue',
        'default': True,
    }
]

answers = prompt(questions)
```

The questions will be prompted one after another and `prompt` will return as soon as all of them are answered. The returned `answers` will be a dictionary containing the responses, e.g.

```
{"phone": "0123123", "continue": False}.
```

Each configuration dictionary for a question must contain the following keys:

`type` (required)
: The type of the question.

`name` (required)
: The name of the question (will be used as key in the `answers` dictionary).

`message` (required)
: Message that will be shown to the user.

In addition to these required configuration parameters, you can add the following optional parameters:

`qmark` (optional)
: Question mark to use - defaults to `?`.

`default` (optional)
Preselected value.

`choices` (optional)
List of choices (applies when `'type': 'select'`) or function returning a list of choices.

`when` (optional)
Function checking if this question should be shown or skipped (same functionality as *`skip_if`*).

`validate` (optional)
Function or Validator Class performing validation (will be performed in real time as users type).

`filter` (optional)
Receive the user input and return the filtered value to be used inside the program.

Further information can be found at the *`questionary.prompt`* documentation.

A Complex Example using a Dictionary Configuration

Questionary allows creating quite complex workflows when combining all of the above concepts:

```
from pprint import pprint

from questionary import Separator
from questionary import prompt


def ask_dictstyle(**kwargs):
    questions = [
        {
            # just print a message, don't ask a question
            # does not require a name (but if provided, is ignored) and does not return␣
↪a value
            "type": "print",
            "name": "intro",
            "message": "This example demonstrates advanced features! ",
            "style": "bold italic",
        },
        {
            "type": "confirm",
            "name": "conditional_step",
            "message": "Would you like the next question?",
            "default": True,
        },
        {
            "type": "text",
            "name": "next_question",
            "message": "Name this library?",
            # Validate if the first question was answered with yes or no
            "when": lambda x: x["conditional_step"],
            # Only accept questionary as answer
            "validate": lambda val: val == "questionary",
        },
        {
            "type": "select",
```

(continues on next page)

(continued from previous page)

```
            "name": "second_question",
            "message": "Select item",
            "choices": ["item1", "item2", Separator(), "other"],
        },
        {
            # just print a message, don't ask a question
            # does not require a name and does not return a value
            "type": "print",
            "message": "Please enter a value for 'other'",
            "style": "bold italic fg:darkred",
            "when": lambda x: x["second_question"] == "other",
        },
        {
            "type": "text",
            # intentionally overwrites result from previous question
            "name": "second_question",
            "message": "Insert free text",
            "when": lambda x: x["second_question"] == "other",
        },
    ]
    return prompt(questions, **kwargs)


if __name__ == "__main__":
    pprint(ask_dictstyle())
```

The above workflow will show to the user the following prompts:

1. Yes/No question `"Would you like the next question?"`.
2. `"Name this library?"` - only shown when the first question is answered with yes.
3. A question to select an item from a list.
4. Free text input if `"other"` is selected in step 3.

Depending on the route the user took, the result will look like the following:

```
{
    'conditional_step': False,
    'second_question': 'Test input'   # Free form text
}
```

```
{
    'conditional_step': True,
    'next_question': 'questionary',
    'second_question': 'Test input'   # Free form text
}
```

You can test this workflow yourself by running the advanced_workflow.py example.

1.5 API Reference

`class questionary.Choice(title, value=None, disabled=None, checked=False, shortcut_key=True)`

One choice in a *select()*, *rawselect()* or *checkbox()*.

Parameters

- **title** (Union[str, List[Tuple[str, str]], List[Tuple[str, str, Callable[[Any], None]]], None]) – Text shown in the selection list.
- **value** (Optional[Any]) – Value returned, when the choice is selected. If this argument is *None* or unset, then the value of *title* is used.
- **disabled** (Optional[str]) – If set, the choice can not be selected by the user. The provided text is used to explain, why the selection is disabled.
- **checked** (Optional[bool]) – Preselect this choice when displaying the options.
- **shortcut_key** (Union[str, bool, None]) – Key shortcut used to select this item.

`static build(c)`

Create a choice object from different representations.

Parameters
c (Union[str, *Choice*, Dict[str, Any]]) – Either a `str`, *Choice* or `dict` with `name`, `value`, `disabled`, `checked` and `key` properties.

Return type
Choice

Returns
An instance of the *Choice* object.

`checked: Optional[bool]`

Whether the choice is initially selected

`disabled: Optional[str]`

Whether the choice can be selected

`shortcut_key: Optional[str]`

A shortcut key for the choice

`title: Union[str, List[Tuple[str, str]], List[Tuple[str, str, Callable[[Any], None]]], None]`

Display string for the choice

`value: Optional[Any]`

Value of the choice

`class questionary.Form(*form_fields)`

Multi question prompts. Questions are asked one after another.

All the answers are returned as a dict with one entry per question.

This class should not be invoked directly, instead use *form()*.

Parameters
form_fields (*Sequence*[FormField]) –

`ask`(*patch_stdout=False*, *kbi_msg='Cancelled by user'*)

Ask the questions synchronously and return user response.

Parameters

- **patch_stdout** (`bool`) – Ensure that the prompt renders correctly if other threads are printing to stdout.
- **kbi_msg** (`str`) – The message to be printed on a keyboard interrupt.

Return type

`Dict`[`str`, `Any`]

Returns

The answers from the form.

`async ask_async`(*patch_stdout=False*, *kbi_msg='Cancelled by user'*)

Ask the questions using asyncio and return user response.

Parameters

- **patch_stdout** (`bool`) – Ensure that the prompt renders correctly if other threads are printing to stdout.
- **kbi_msg** (`str`) – The message to be printed on a keyboard interrupt.

Return type

`Dict`[`str`, `Any`]

Returns

The answers from the form.

`unsafe_ask`(*patch_stdout=False*)

Ask the questions synchronously and return user response.

Does not catch keyboard interrupts.

Parameters

patch_stdout (`bool`) – Ensure that the prompt renders correctly if other threads are printing to stdout.

Return type

`Dict`[`str`, `Any`]

Returns

The answers from the form.

`async unsafe_ask_async`(*patch_stdout=False*)

Ask the questions using asyncio and return user response.

Does not catch keyboard interrupts.

Parameters

patch_stdout (`bool`) – Ensure that the prompt renders correctly if other threads are printing to stdout.

Return type

`Dict`[`str`, `Any`]

Returns

The answers from the form.

class questionary.Question(*application*)

A question to be prompted.

This is an internal class. Questions should be created using the predefined questions (e.g. text or password).

Parameters

application (*Application[Any]*) –

ask(*patch_stdout=False*, *kbi_msg='Cancelled by user'*)

Ask the question synchronously and return user response.

Parameters

- **patch_stdout** (bool) – Ensure that the prompt renders correctly if other threads are printing to stdout.
- **kbi_msg** (str) – The message to be printed on a keyboard interrupt.

Returns

The answer from the question.

Return type

Any

async ask_async(*patch_stdout=False*, *kbi_msg='Cancelled by user'*)

Ask the question using asyncio and return user response.

Parameters

- **patch_stdout** (bool) – Ensure that the prompt renders correctly if other threads are printing to stdout.
- **kbi_msg** (str) – The message to be printed on a keyboard interrupt.

Returns

The answer from the question.

Return type

Any

skip_if(*condition*, *default=None*)

Skip the question if flag is set and return the default instead.

Parameters

- **condition** (bool) – A conditional boolean value.
- **default** (Any) – The default value to return.

Returns

self.

Return type

Question

unsafe_ask(*patch_stdout=False*)

Ask the question synchronously and return user response.

Does not catch keyboard interrupts.

Parameters

patch_stdout (bool) – Ensure that the prompt renders correctly if other threads are printing to stdout.

Returns
The answer from the question.

Return type
Any

`async unsafe_ask_async(patch_stdout=False)`

Ask the question using asyncio and return user response.

Does not catch keyboard interrupts.

Parameters
patch_stdout (`bool`) – Ensure that the prompt renders correctly if other threads are printing to stdout.

Returns
The answer from the question.

Return type
Any

`class questionary.Separator(line=None)`

Used to space/separate choices group.

Parameters
line (*str*) –

`default_separator: str = '---------------'`

The default separator used if none is specified

`line: str`

The string being used as a separator

`class questionary.FormField(key, question)`

Represents a question within a form

Parameters

- **key** (*str*) – The name of the form field.
- **question** (`Question`) – The question to ask in the form field.

`key: str`

Alias for field number 0

`question: Question`

Alias for field number 1

`questionary.form()`

Create a form with multiple questions.

The parameter name of a question will be the key for the answer in the returned dict.

Parameters
kwargs (*Question*) – Questions to ask in the form.

Return type
Form

`questionary.prompt(answers=None, patch_stdout=False, true_color=False, kbi_msg='Cancelled by user', **kwargs)`

Prompt the user for input on all the questions.

Catches keyboard interrupts and prints a message.

See *unsafe_prompt()* for possible question configurations.

Parameters

- **questions** (Union[Dict[str, Any], Iterable[Mapping[str, Any]]]) – A list of question configs representing questions to ask. A question config may have the following options:
  - type - The type of question.
  - name - An ID for the question (to identify it in the answers dict).
  - when - Callable to conditionally show the question. This function takes a dict representing the current answers.
  - filter - Function that the answer is passed to. The return value of this function is saved as the answer.

  Additional options correspond to the parameter names for particular question types.
- **answers** (Optional[Mapping[str, Any]]) – Default answers.
- **patch_stdout** (bool) – Ensure that the prompt renders correctly if other threads are printing to stdout.
- **kbi_msg** (str) – The message to be printed on a keyboard interrupt.
- **true_color** (bool) – Use true color output.
- **color_depth** – Color depth to use. If `true_color` is set to true then this value is ignored.
- **type** – Default `type` value to use in question config.
- **filter** – Default `filter` value to use in question config.
- **name** – Default `name` value to use in question config.
- **when** – Default `when` value to use in question config.
- **default** – Default `default` value to use in question config.
- **kwargs** (Any) – Additional options passed to every question.

Return type

Dict[str, Any]

Returns

Dictionary of question answers.

questionary.**unsafe_prompt**(*answers=None*, *patch_stdout=False*, *true_color=False*, ***kwargs*)

Prompt the user for input on all the questions.

Won't catch keyboard interrupts.

Parameters

- **questions** (Union[Dict[str, Any], Iterable[Mapping[str, Any]]]) – A list of question configs representing questions to ask. A question config may have the following options:
  - type - The type of question.
  - name - An ID for the question (to identify it in the answers dict).

  - when - Callable to conditionally show the question. This function takes a `dict` representing the current answers.
  - filter - Function that the answer is passed to. The return value of this function is saved as the answer.

  Additional options correspond to the parameter names for particular question types.

- **answers** (`Optional[Mapping[str, Any]]`) – Default answers.
- **patch_stdout** (`bool`) – Ensure that the prompt renders correctly if other threads are printing to stdout.
- **true_color** (`bool`) – Use true color output.
- **color_depth** – Color depth to use. If `true_color` is set to true then this value is ignored.
- **type** – Default `type` value to use in question config.
- **filter** – Default `filter` value to use in question config.
- **name** – Default `name` value to use in question config.
- **when** – Default `when` value to use in question config.
- **default** – Default `default` value to use in question config.
- **kwargs** (`Any`) – Additional options passed to every question.

Return type
`Dict[str, Any]`

Returns
Dictionary of question answers.

Raises
`KeyboardInterrupt` – raised on keyboard interrupt

1.6 Support

Please open an issue with enough information for us to reproduce your problem. A minimal, reproducible example would be very helpful.

1.7 Contributor's Guide

1.7.1 Steps for Submitting Code

Contributions are very much welcomed and appreciated. Every little bit of help counts, so do not hesitate!

1. Check for open issues, or open a new issue to start some discussion around a feature idea or bug. There is a contributor friendly tag for issues that should be ideal for people who are not familiar with the codebase yet.
2. Fork the repository on GitHub to start making your changes.
3. Install Poetry.
4. Configure development environment.

```
make develop
```

5. Write some tests that show the bug is fixed or that the feature works as expected.
6. Ensure your code passes the code quality checks by running

   ```
   $ make lint
   ```

7. Check all of the unit tests pass by running

   ```
   $ make test
   ```

8. Check the type checks pass by running

   ```
   $ make types
   ```

9. Send a pull request and bug the maintainer until it gets merged and published

1.7.2 Bug Reports

Bug reports should be made to the issue tracker. Please include enough information to reproduce the issue you are having. A minimal, reproducible example would be very helpful.

1.7.3 Feature Requests

Feature requests should be made to the issue tracker.

1.7.4 Other

Create a New Release

1. Update the version number in `questionary/version.py` and `pyproject.toml`.
2. Add a new section for the release to *Changelog*.
3. Commit these changes.
4. `git tag` the commit with the release version number.

GitHub Actions will build and push the updated library to PyPi.

Create a Command Line Recording

1. Install the following tools:

   ```
   $ brew install asciinema
   $ npm install --global asciicast2gif
   ```

2. Start the recording with `asciinema`:

   ```
   $ asciinema rec
   ```

3. Do the thing you want to record.
4. Convert to gif using `asciicast2gif`:

```
$ asciicast2gif -h 7 -w 120 -s 2 <recording> output.gif
```

1.8 Changelog

1.8.1 2.0.1 (2023-09-08)

- Updated dependencies.
- Fixed broken documentation build.

1.8.2 2.0.0 (2023-07-25)

- Updated dependencies.
- Modified default choice selection based on the `Choice` value. Now, it is not necessary to pass the same instance of the `Choice` object: the same `value` may be used.
- Fixed various minor bugs in development scripts and continuous integration.
- Improved continuous integration and testing process.
- Added pull request and issue templates to the GitHub repository.
- Implemented lazy function call for obtaining choices.
- Expanded the test matrix to include additional Python versions.
- Added the ability to specify the start point of a file path.
- Enabled displaying arbitrary paths in file path input.
- Allowed skipping of questions in the `unsafe_ask` function.
- Resolved typing bugs.
- Included a password confirmation example.
- Now returning selected choices even if they are disabled.
- Added support for Emacs keys (`Ctrl+N` and `Ctrl+P`).
- Fixed rendering errors in the documentation.
- Introduced a new `print` question type.
- Deprecated support for Python 3.6 and 3.7.
- Added dynamic instruction messages for `checkbox` and `confirm`.
- Removed the upper bound from the Python version specifier.
- Added a `press_any_key_to_continue` prompt.

1.8.3 1.10.0 (2021-07-10)

- Use direct image URLs in `README.md`.
- Switched to `poetry-core`.
- Relax Python version constraint.
- Add `pointer` option to `checkbox` and `select`.
- Change enter instruction for multiline input.
- Removed unnecessary Poetry includes.
- Minor updates to documentation.
- Added additional unit tests.
- Added `use_arrow_keys` and `use_jk_keys` options to `checkbox`.
- Added `use_jk_keys` and `show_selected` options to `select`.
- Fix highlighting bug when using `default` parameter for `select`.

1.8.4 1.9.0 (2020-12-20)

- Added brand new documentation https://questionary.readthedocs.io/ (thanks to @kiancross)

1.8.5 1.8.1 (2020-11-17)

- Fixed regression for checkboxes where all values are returned as strings fixes #88.

1.8.6 1.8.0 (2020-11-08)

- Added additional question type `questionary.path`
- Added possibility to validate select and checkboxes selections before submitting them.
- Added a helper to print formatted text `questionary.print`.
- Added API method to call prompt in an unsafe way.
- Hide cursor on select only showing the item marker.

1.8.7 1.7.0 (2002-10-15)

- Added support for Python 3.9.
- Better UX for multiline text input.
- Allow passing custom lexer.

1.8.8 1.6.0 (2020-10-04)

- Updated black code style formatting and fixed version.
- Fixed colour of answer for some prompts.
- Added `py.typed` marker file.
- Documented multiline input for devs and users and added tests.
- Accept style tuples in `title` argument annotation of `Choice`.
- Added `default` for select and `initial_choice` for checkbox prompts.
- Removed check for choices if completer is present.

1.8.9 1.5.2 (2020-04-16)

Bug fix release.

- Added `.ask_async` support for forms.

1.8.10 1.5.1 (2020-01-22)

Bug fix release.

- Fixed `.ask_async` for questions on `prompt_toolkit==2.*`. Added tests for it.

1.8.11 1.5.0 (2020-01-22)

Feature release.

- Added support for `prompt_toolkit` 3.
- All tests will be run against `prompt_toolkit` 2 and 3.
- Removed support for Python 3.5 (`prompt_toolkit` 3 does not support that any more).

1.8.12 1.4.0 (2019-11-10)

Feature release.

- Added additional question type `autocomplete`.
- Allow pointer and highlight in select question type.

1.8.13 1.3.0 (2019-08-25)

Feature release.

- Add additional options to style checkboxes and select prompts #14.

1.8.14 1.2.1 (2019-08-19)

Bug fix release.

- Fixed compatibility with Python 3.5.2 by removing `Type` annotation (this time for real).

1.8.15 1.2.0 (2019-07-30)

Feature release.

- Allow a user to pass in a validator as an instance #10.

1.8.16 1.1.1 (2019-04-21)

Bug fix release.

- Fixed compatibility with python 3.5.2 by removing `Type` annotation.

1.8.17 1.1.0 (2019-03-10)

Feature release.

- Added `skip_if` to questions to allow skipping questions using a flag.

1.8.18 1.0.2 (2019-01-23)

Bug fix release.

- Fixed odd behaviour if select is created without providing any choices instead, we will raise a `ValueError` now #6.

1.8.19 1.0.1 (2019-01-12)

Bug fix release, adding some convenience shortcuts.

- Added shortcut keys `j` (move down the list) and `k` (move up) to the prompts `select` and `checkbox` (fixes #2).
- Fixed unclosed file handle in `setup.py`.
- Fixed unnecessary empty lines moving selections to far down (fixes #3).

1.8.20 1.0.0 (2018-12-14)

Initial public release of the library.

- Added python interface.
- Added dict style question creation.
- Improved the documentation.
- More tests and automatic Travis test execution.

INDEX