cooked_input Documentation

Release 0.5.3

Len Wanger

Oct 13, 2020

Contents:

1 Cooked Input Project **3**
- 1.1 Documentation . . . 3
- 1.2 Python 2/3 Support . . . 4
- 1.3 Installation . . . 4
- 1.4 Project Page . . . 4
- 1.5 Tutorial . . . 4
- 1.6 Change log . . . 4

2 Cooked Input Quick Start **5**
- 2.1 Installing cooked_input: . . . 5
- 2.2 Getting Simple Values: . . . 5
  - 2.2.1 Getting Strings: . . . 5
  - 2.2.2 Getting Integers: . . . 6
  - 2.2.3 Getting Dates: . . . 7
  - 2.2.4 More Convenience Functions: . . . 7
  - 2.2.5 Menus: . . . 7
  - 2.2.6 Further Reading: . . . 8

3 Cooked Input Tutorial **9**
- 3.1 Introduction: . . . 9
- 3.2 The CCV paradigm: . . . 10
- 3.3 Using CCV in Convenience Functions: . . . 11
- 3.4 Breaking down get_input: . . . 11
- 3.5 Custom Cleaners, Convertors and Validators: . . . 13
- 3.6 The GetInput class: . . . 14
- 3.7 Using get_list: . . . 15
- 3.8 From Here: . . . 16

4 Cooked Input Tutorial, Part Two **17**
- 4.1 Introduction: . . . 17
- 4.2 Commands: . . . 17
- 4.3 Tables: . . . 18
- 4.4 Menus: . . . 21
- 4.5 Running the application: . . . 22
- 4.6 From Here: . . . 22

5 How-to / FAQ **23**

5.1 Getting yes or no . . . 23
5.2 Restricting to a list of choices . . . 23
5.3 Excluding a list of choices . . . 24
5.4 Composing Multiple Validators . . . 24
5.5 More Examples . . . 24
5.6 TODO . . . 25

6 Convenience Functions **27**
6.1 GetInput Convenience Functions . . . 27
6.1.1 get_string . . . 27
6.1.2 get_int . . . 28
6.1.3 get_float . . . 28
6.1.4 get_boolean . . . 29
6.1.5 get_date . . . 29
6.1.6 get_yes_no . . . 29
6.1.7 get_list . . . 30
6.1.8 get_input . . . 30
6.1.9 process_value . . . 31
6.1.10 validate . . . 31
6.2 Table Convenience Functions . . . 32
6.2.1 get_menu . . . 32
6.2.2 create_rows . . . 32
6.2.3 create_table . . . 34
6.2.4 show_table . . . 35
6.2.5 get_table_input . . . 35

7 Cleaners **37**
7.1 Creating Cleaners . . . 37
7.2 Cleaners . . . 37
7.2.1 CapitalizationCleaner . . . 37
7.2.2 ChoiceCleaner . . . 38
7.2.3 RegexCleaner . . . 38
7.2.4 RemoveCleaner . . . 39
7.2.5 ReplaceCleaner . . . 39
7.2.6 StripCleaner . . . 40

8 Convertors **41**
8.1 Creating Convertors . . . 41
8.2 Convertors . . . 42
8.2.1 BooleanConvertor . . . 42
8.2.2 DateConvertor . . . 42
8.2.3 FloatConvertor . . . 42
8.2.4 IntConvertor . . . 43
8.2.5 ListConvertor . . . 43
8.2.6 YesNoConvertor . . . 43
8.2.7 ChoiceConvertor . . . 44

9 Validators **45**
9.1 Creating Validators . . . 45
9.2 Validators . . . 46
9.2.1 AnyOfValidator . . . 46
9.2.2 ChoiceValidator . . . 46
9.2.3 EqualToValidator . . . 47
9.2.4 IsFileValidator . . . 47
9.2.5 LengthValidator . . . 47

9.2.6 ListValidator . . . 47
9.2.7 NoneOfValidator . . . 48
9.2.8 PasswordValidator . . . 49
9.2.9 RangeValidator . . . 49
9.2.10 RegexValidator . . . 50
9.2.11 SimpleValidator . . . 50

10 GetInput **51**
10.1 GetInput: . . . 51

11 Tables and Menus **53**
11.1 TableStyle: . . . 53
11.2 TableItem: . . . 54
11.3 Table: . . . 55
11.4 Table Action Functions: . . . 59
11.4.1 return_table_item_action . . . 59
11.4.2 return_row_action . . . 59
11.4.3 return_tag_action . . . 59
11.4.4 return_first_col_action . . . 60

12 Exceptions **61**
12.1 ConvertorError: . . . 61
12.2 MaxRetriesError: . . . 61
12.3 ValidationError: . . . 61
12.4 GetInputInterrupt: . . . 61
12.5 RefreshScreenInterrupt: . . . 62
12.6 PageUpRequest: . . . 62
12.7 PageDownRequest: . . . 62
12.8 FirstPageRequest: . . . 62
12.9 LastPageRequest: . . . 62
12.10 UpOneRowRequest: . . . 62
12.11 DownOneRowRequest: . . . 62

13 error_callbacks **63**
13.1 Creating Error functions: . . . 63
13.2 error_callbacks: . . . 63
13.2.1 log_error . . . 63
13.2.2 print_error . . . 64
13.2.3 silent_error . . . 64

14 Commands **65**
14.1 GetInputCommand: . . . 65
14.2 Command Action Functions: . . . 67
14.2.1 first_page_cmd_action . . . 67
14.2.2 last_page_cmd_action . . . 67
14.2.3 prev_page_cmd_action . . . 67
14.2.4 next_page_cmd_action . . . 68
14.2.5 scroll_up_one_row_cmd_action . . . 68
14.2.6 scroll_down_one_row_cmd_action . . . 68

15 Indices and tables **69**

Python Module Index **71**

Index **73**

This is documentation for cooked_input v0.5.3, generated Oct 13, 2020.

CHAPTER 1

Cooked Input Project

`cooked_input` is a Python package for getting, cleaning, converting, and validating command line input. If you think of input (raw_input in legacy Python) as raw input, then this is cooked input.

`cooked_input` provides a simple and safe way to get validated input that ranges from the simplest of Python programs to complex command line applications using menus and tables. Beginner's can use the provided convenience classes to get simple inputs from the user. Following the tutorial you can be up and running in minutes. More advanced users can easily create custom classes for sophisticated cleaning and validation of inputs.

More complicated command line application (CLI) input can take advantage of `cooked_input`'s ability to create commands, menus and data tables. The latter tutorials and examples show several examples of more advanced usage.

`Cooked_input` also provides a pathway to use the same cleaning and validation logic used in the command line for processing and validating web or GUI based inputs. This allows seamless transition from command line to GUI applications.

1.1 Documentation

Read the full documentation at readthedocs.org:

- cooked_input documentation at: http://cooked-input.readthedocs.io/en/latest/

1.2 Python 2/3 Support

- Python 2.7
- Python 3.3/3.4/3.5/3.6

1.3 Installation

From pypi:

```
pip install cooked_input
```

1.4 Project Page

Source code and other project information available at: https://github.com/lwanger/cooked_input

1.5 Tutorial

The best way to get started is to read the quick start at: http://cooked-input.readthedocs.io/en/latest/quick_start.html

After that, more advanced usage can be learned from the tutorial at: http://cooked-input.readthedocs.io/en/latest/tutorial.html

Finally,, part two of the tutorial can be found at: http://cooked-input.readthedocs.io/en/latest/tutorial2.html

1.6 Change log

See the CHANGELOG for a list of changes.

CHAPTER 2

Cooked Input Quick Start

Getting started in *cooked_input* is very easy. This quick start shows how to use *cooked_input* to get simple keyboard input from the user. For more advanced appplications see the tutorial and how to sections.

2.1 Installing cooked_input:

To install *cooked_input*:

```
pip install cooked_input
```

2.2 Getting Simple Values:

All *cooked_input* applications start with importing the *cooked_input* library:

```
import cooked_input as ci
```

2.2.1 Getting Strings:

The simplest usage of *cooked_input* is to use get_string to get a string value:

```
ci.get_string(prompt="What is your name?")
```

Running this code produces:

```
>>> ci.get_string(prompt="What is your name?")

What is your name?: john cleese
'john cleese'
```

This acts just like the Python input command (or *raw_input* in legacy Python.) Unlike *input cooked_input* will keep on asking until you enter valid input. *Get_string* will not accept a blank line.

Note: To allow *get_string* to accept a blank line, like input, set the `required` parameter to **False**:

```
ci.get_string(prompt="What is your name?", required=False)
```

Since we are entering a name, we want to make sure the value is capitalized. *Cooked_input* provides a number of cleaners that can be used to clean up the input value. *CapitalizationCleaner* can be used to change the case of the value. It takes a `style` parameter to say how you want the value changed. In this case we want use use `ALL_WORDS_CAP_STYLE` to capitalize the first letter of each of the words in the value:

```
cap_cleaner = ci.CapitalizationCleaner(style=ci.ALL_WORDS_CAP_STYLE)
ci.get_string(prompt="What is your name?", cleaners=[cap_cleaner])
```

Now the input will be cleaned up with the proper capitalization:

```
>>> ci.get_string(prompt="What is your name?", cleaners=[cap_cleaner])

What is your name?: jOhn CLeEse
'John Cleese'
```

This is just one of the capitalization options. For the full list see the documentation for *CapitalizationCleaner*.

2.2.2 Getting Integers:

Cooked_input has a number of convenience functions to get different input types. Integers can be fetched using the get_int function:

```
ci.get_int(prompt="How old are you?", minimum=1)
```

get_int requests the input and returns a whole number (i.e. an integer.) If the input cannot be converted to an integer it will print an error message and ask the user again. *Get_int* can also take parameters for the `minimum` and `maximum` values allowed. Since we are asking for a person's age, we want to make sure the number is a positive number (i.e. the person is at least 1 year old.) Since no `maximum` value is given in this example there is no maximum age for this input:

```
>>>  ci.get_int(prompt="How old are you?", minimum=1)

How old are you?: abc
"abc" cannot be converted to an integer number
How old are you?: 0
"0" too low (min_val=1)
How old are you?: 67
67
```

This is just the tip of the iceberg of what *get_int* can do. There are a lot of examples of using *get_int* in *get_ints.py* in the *cooked_input* examples directory. *Get_ints.py* shows examples of more complicated restrictions on the allowed number (validators), suggesting default values, and other more advanced usages of *get_int*. The *examples* directory is a good place to look to see how to use many of *cooked_input*'s features.

2.2.3 Getting Dates:

A good example of how *cooked_input* can be helpful is get_date. *Get_date* is used to get dates and times (more specifically a Python datetime value.) The following code shows how to ask the user for their birthday. Today's date is used in the example below as the `maximum` beause it doesn't make sense that the user's birthday is in the future:

```
import datetime
today = datetime.datetime.today()
birthday = ci.get_date(prompt="What is your birthday?", maximum=today)
print(birthday)
```

Running this returns a datetime:

```
What is your birthday?: 4/1/1957
1957-04-01 00:00:00
```

Get_date is very flexible regarding how you can type times and dates. For instance, `"April 4, 1967"`, `"today"`, `"1 hour from now"`, `"9:30 am yesterday"` and `"noon 3 days ago"` are all valid date values. *Cooked_input* functions can also take a `default` value. For instance, using `"today"` as the default value will use today's date if the user hits enter without entering a value:

```
appt_date = ci.get_date(prompt="Appointment date?", default="today")
print(appt_date.strftime("%x"))
```

Produces:

```
Appointment date? (enter for: today):
4/14/2018
```

2.2.4 More Convenience Functions:

Cooked_input provides several more convenience functions for getting different input types:

Function	Return type
get_string	string
get_int	int
get_float	float
get_date	datetime
get_boolean	boolean (*True* or *False*)
get_yes_no	string (*"yes"* or *"no"*)

There are also convenience functions for a number of other *cooked_input* features, such as: getting lists, choosing values from a table or showing a menu. For more information see the convenience functions documentation,

---

Note: It is easy to add support for input types not included with *cooked_input*. See the tutorial for examples of adding custom types (convertors) and validators.

2.2.5 Menus:

To give you a teaser of some of the further features in *Cooked_input* let's look a quick look at menus. *Cooked_input* has a lot more functionality for menus and tables, but this will show you a little of what you have to look forward to.

Cooked_input menus have two parts: a *tag* (the thing you type in to choose the menu item) and the *item data* (the data for the menu item.) The simplest way to make a menu is to use the *get_menu* convenience function. This creates numbered menu items for each item in a list of choices and returns the number of the item picked:

```
choices = ['red', 'blue', 'green']
result = ci.get_menu(choices)
print('result={}'.format(result))
```

produces:

```
1  red
2  blue
3  green

Choose a table item: 1
result=1
```

There are a lot of options that can be passed to *get_menu*. For instance, the *prompt* option sets the prompt string presented to the user. Setting *default_action* to **TABLE_RETURN_FIRST_VAL** option can be used to return the choice item_data instead of the tag:

```
choices = ['red', 'blue', 'green']
result = ci.get_menu(choices, prompt="Choose a color", default_action=ci.TABLE_RETURN_
↪FIRST_VAL)
print('result={}'.format(result))
```

produces:

```
1  red
2  blue
3  green

Choose a color: 1
result=red
```

This just scratches the surface of what can be done with *Cooked_input* menus and tables. For more examples, see *get_menu.py*, *get_table_input.py*, and *get_unicode.py* in the examples folder.

2.2.6 Further Reading:

Cooked_input has a lot more features and capabilities for getting, cleaning and validating inputs. *Cooked_input* also has capabilities for user commands and data tables. To learn about the advanced capabilities of *cooked_input*, see the tutorial and how to.

CHAPTER 3

Cooked Input Tutorial

Note: Many users will to start with the quick start guide to get up and running with *cooked_input* quickly.

3.1 Introduction:

Command line tools and terminal input are very useful. I end up writing lots of programs that get some simple input from the user and process it. It may be a creating a report from a database or a simple text-based game. While it sounds trivial, handling all of the exceptions and corner cases of command line input is not easy, and the Python standard library doesn't have anything to make it easier. Let's start with a simple example.

The first program in an introduction to Python usually looks something like this:

```
import random
number = random.randint(1, 10)
print('I am thinking of a number between 1 and 10.')
guess_str = input('Guess what number I am thinking of: ')
guess = int(guess_str)

if guess < number:
    print('Buzz.... you guessed too low.')
elif guess > number:
    print('Buzz.... you guessed too high.')
else:
    print('Ding ding... you guessed it!')
```

Looks simple enough right? We get input from the user and convert it to an integer. But what happens when the user responds with 'a':

```
I am thinking of a number between 1 and 10.
Guess what number I am thinking of: a
Traceback (most recent call last):
```

(continues on next page)

(continued from previous page)

```
  File "C:/apps/simple_input.py", line 67, in <module>
    simple_guess()
  File "C:/apps/simple_input.py", line 13, in simple_guess
    guess = int(input('Guess what number I am thinking of: '))
ValueError: invalid literal for int() with base 10: 'a'
```

Let's look at what it takes to make the input robust. Checking that the input is: a number, within the correct range, works with legacy Python (ie. version 2) and 3, etc. becomes:

```
while True:
    if sys.version_info.major > 2:
        result = input('Guess what number I am thinking of: ')
    else:
        result = raw_input('Guess what number I am thinking of: ')

    try:
        guess = int(result)
        if guess < 1 or guess > 10:
            print('not between 1 and 10, try again.')
            continue
        break
    except ValueError:
        print('That is not an integer, try again.')
```

That's a lot of code to handle the simplest of inputs. This also forces a beginer to deal with advanced concepts such as exceptions and versions of Python. This boiler plate code is replicated and expanded for each input from the command line. Just think of how much code you would need to get and validate a new password from a user – making sure the input is hidden, is at least 8 characters long, has at least 2 upper case letters, has at least 2 punctuation marks, has at least 1 number, doesn't use the characters '[', ']', or '&', and exits after 3 failed attempts.

The purpose of `cooked_input` is to make it easier to get command line input from the user. It takes care of cleaning, converting, and validating the input. It also helps put together the prompt message and error messages. In cooked_input, safely getting the value from the user in the guessing game becomes:

```
prompt_str ='Guess what number I am thinking of'
guess = get_int(prompt=prompt_str, minimum=0, maximum=10)
```

For a complete listing of the guessing game code using cooked_input, see simple_input.py in the examples directory.

In case your curious, the password example above can be written in cooked_input as:

```
prompt_str = 'Enter a new password'
good_password = PasswordValidator(min_len=8, min_upper=2, min_digits=1, min_puncts=2,␣
↪disallowed='[]&')
password = get_input(prompt=prompt_str, validators=[good_password], hidden=True,␣
↪retries=3)
```

3.2 The CCV paradigm:

`Cooked_input` processes input using the Clean, Convert, Validate (CCV) paradigm. This is the underlying principal used for getting input in `cooked_input`.

Fig. 1: The "clean, convert, validate" (CCV) paradigm

In CCV, an input string is processed by first running it through a set of `cleaners`. This is a set of zero or more functions which clean the input string. For instance a cleaning function may convert a string from to upper case or strip off white space from the ends of the string.

Next the cleaned string is run through a `convertor` function, which converts the string into the desired output type. `Cooked_input` includes convertors for many types, such as integers, floating point numbers, Booleans, and dates.

Finally the cleaned, converted value is run through a set of `validators`. This is a set of zero or more validation functions which verify the value is acceptable. For instance, a validator may check that a value in a specified range.

If the value gets through the CCV pipeline without generating an error, the cleaned, converted value is returned. Otherwise an error message is presented and the user is prompted to re-enter the value.

By combining `cooked_input's` rich set of cleaners, convertors and validators complicated inputs can be processed easily. It is also easy to create your own custom cleaners, convertors and validators to expand `cooked_input's` functionality.

3.3 Using CCV in Convenience Functions:

In the quick start we saw how to use the `cooked_input` convenience functions to process simple values. The convenience functions are wrappers that set useful default values for the cleaners, convertor and validators. For instance, the *`get_int()`* sets the convertor to *`IntConvertor`* and has parameters for setting the minimum and maximum values for validating the value is within a range *`RangeValidator`*.

All of the convenience functions also take an optional set of cleaners and validators. For instance, a *`NoneOfValidator`* can be sent to `get_int` to get a value between -10 and 10, but not zero:

```
not_zero_validator = NoneOfValidator([0])
get_int(minimum=-10, maximum=10, validators=[not_zero_validator])
```

3.4 Breaking down get_input:

The convenience functions are wrappers around the *get_input* function. For instance, `get_int` is a wrapper around *`get_input()`* that automatically fills in some of the values required to get an integer (e.g. set the convertor function to *`IntConvertor`*). Similarly, all of the other convenience functions (such as *`get_float()`*, *`get_boolean()`*, *`get_date()`*, etc.) are just wrappers around get_input too.

The simplest call to *get_input* is:

```
result = get_input('What is your name')
```

This will prompt the user for their name and return a non-blank string. If the user types in a blank value (a zero length string) they will receive an error message and be prompted to re-enter the value until a non-zero length string is entered

---

Note: By default get_input does not strip out white space, so typing a space or other whitespace characters would not be considered a blank string. This differs from `get_string` which defaults to stripping whitespace meaning a space would be cleaned to be a blank string

---

Let's look at a more complicated example. The get_int call in the guessing game makes a call to get_input that looks something like this:

```
prompt_str = 'Guess what number I am thinking of'
range_validator = RangeValidator(min_val=1, max_val=10)
result = get_input(prompt=prompt_str, convertor=IntConvertor(),
    validators=range_validator, retries=3)
```

This call passes several parameters to `get_input`:

prompt: the string to print to prompt the user.

convertor: the *Convertor* is called to convert the string entered into the type of value we want. *IntConvertor* converts the value to an integer (*int*).

validators: the *Validator* function (or list of *Validator* functions) are used to check the entered string meets the criteria we want. If the input doesn't pass the validation, an error message is produced, and the user is prompted to re-enter a value.

`RangeValidator` takes a minimum and maximum value and checks that the input value is in the interval between the two. For example, *RangeValidator(min_val=1, max_val=10)* would make sure the value is between *1* and *10*. (i.e. *1<=value<=10*). In the case above, *max_val* is set to *None*, so no maximum value is applied (i.e. checks *1<=value*)

options: there are a number of optional parameters that get_input can take (see *get_input* for more information). By default, get_input will keep asking for values until a valid value is entered. The *retries* option specifies the maximum number of times to ask. If no valid input is entered within the maximum number of tries a `MaxRetriesError` exception is raised.

return value: the cleaned, converted, validated value is returned. The returned value is safe to use as we know it meets all the criteria we requested. In this case, an integer value that is greater than or equal to *0*.

The general flow of *get_input* is:

1) Prompt the user and get the input from the keyboard (sys.stdin)
2) Apply the cleaner functions to the string.
3) Apply the convertor function to the cleaned string.
4) **Apply the validator functionss to the converted value. The converted value needs to pass all of the** validators (i.e. they are AND'd together). Other combinations of validators can be achieved by using the `AnyOfValidator` (OR) and `NoneOfValidator` (NOT) validators.
5) Return the cleaned, converted, validated value.

Note: The order of the cleaners and validators is maintained and applied from left-to-right. For example, if the list of cleaners is *cleaners=[StripCleaner(), CapitalizationCleaner(style='lower')]*, then the strip operation is performed before conversion to lower case. Applying these cleaners to the value *" Yes "* is equivalent to the Python statement: *" Yes ".strip().lower()* (strip, then convert to lower case), which would produce the cleaned value: *"yes"*.

Note: The `process_value()` function runs all of the *get_input* processing steps on a value (i.e. runs steps 2–5 above.) This is useful for applying the same cooked_input cleaning, conversion and validation to values received from GUI forms, web forms or for data cleaning (from files or databases.) The *validate_tk* example shows how *process* can be used to validate an input in a GUI.

Similarly, the `validate()` function can be used to just perform the validation step on a value. These two functions are very handy as they allow the same business logic (i.e. `cooked_input` code) to be used between input from the command line and other sources.

3.5 Custom Cleaners, Convertors and Validators:

Writing custom cleaners, convertors and validators is easy in `cooked_input`. To demonstrate, we will create a custom convertor and validator for a simple class to represent an interval (i.e. a range of numbers):

```
class Interval(object):
    def __init__(self, min_val, max_val):
        # represent the interval as a complex number with the minimum
        # and maximum values as the real imaginary parts respectively
        self.interval = complex(min_val, max_val)
```

We can create a convertor function to convert a string with the format "x:y" (where x and y are integers) to an Interval. Cooked_input convertor functions are inherited from the `Convertor` baseclass and implement two methods. The `__init__` method sets up any context variables for the convertor (none in this case) and calls super on the Convertor baseclass. All cleaners, convertors and validators are callable objects in Python, meaning they define the `__call__` method. The __call__ method is called to convert the value, and takes three parameters:

- **value**: the value string to convert
- **error_callback**: an error callback function used to report conversion problems
- **convertor_fmt_str**: a format string used by the error callback function

---

Note: The `error_callback` function and `convertor_fmt string` are used to set how errors are reported. For more information see: error_callback

---

If an error occurs during the conversion, the `__call__` method should call the error callback function (i.e. `error_callback`) and raise a *`ConvertorError`* exception. If the conversion is sucessful, the __call__ methods returns the converted Value. The following code implements our Interval convertor:

```
class IntervalConvertor(Convertor):
    def __init__(self, value_error_str='a range of numbers("x:y")'):
        super(IntervalConvertor, self).__init__(value_error_str)

    def __call__(self, value, error_callback, convertor_fmt_str):
        # convert an interval formatted as "min : max"
        use_val = value.strip()
        dash_idx = use_val.find(':')

        if dash_idx == -1:  # ":" not found
            error_callback(convertor_fmt_str, value,
                'an interval -- ":" not found to separate values')
            raise ConvertorError
        else:
            try:
                min_val = int(value[:dash_idx])
            except (IndexError, TypeError, ValueError):
                error_callback(convertor_fmt_str, value,
                    'an interval -- invalid minimum value')
                raise ConvertorError

            try:
                max_val = int(value[dash_idx + 1:])
            except (IndexError, TypeError, ValueError):
                error_callback(convertor_fmt_str, value,
                    'an interval -- invalid maximum value')
```

(continues on next page)

(continued from previous page)

```
                raise ConvertorError

        if min_val > max_val:
            error_callback(convertor_fmt_str, value,
                'an interval -- low value is higher than the high value')
            raise ConvertorError

        return Interval(min_val, max_val)
```

Validators have a similar structure to convertors. They are inherited from the `Validator` base class and implement the same two methods: `__init__` and `__call__`. The `__init__` method has parameters defining the attributes for valid values. The Interval validator is initialized with the minimum and maximum values (i.e. an interval) and saves it as class member.

Note: validators do not need to call to super on the Validator base class.

The `__call__` method verifies that the value is an Instance of the Interval class and that it's minimum and maximum values are within the valid range specified in the `__init__` function. `__call__` takes the same three parameters as convertors (`value`, `error_callback`, `validator_fmt_str`.) `True` is returned if the value passes validation. If the value fails validation, `error_callback` is called to report the error and `False` is returned. The Interval validator looks like:

```
class IntervalValidator(Validator):
    # validate an interval is within a specified range
    def __init__(self, range_interval):
        # range_interval specifies minimum and maximum input values
        self.range = range_interval

    def __call__(self, value, error_callback, validator_fmt_str):
        if not isinstance(value, Interval):
            error_callback(validator_fmt_str, value, 'Not an interval')
            return False

        if value.interval.real < self.range.interval.real:
            err_string = 'Low end below the minimum ({})'.format(
                self.range.interval.real)
            error_callback(validator_fmt_str, value.interval.real,
                err_string)
            return False

        if value.interval.imag > self.range.interval.imag:
            err_string = 'High end above the maximum ({})'.format(
                self.range.interval.imag)
            error_callback(validator_fmt_str, value.interval.imag,
                err_string)
            return False

        return True # passed validation!
```

3.6 The GetInput class:

Peeling back the final layer the onion, the *`get_input()`* convenience function just creates an instance of the *`GetInput`* class and calls the `get_input` method on the instance. There are two times you might may want to

skip the `get_input` convenience function and create an instance of `GetInput` directly:

1) **If you are going to ask for the same input repeatedly, you can save the overhead of re-creating** the instance by creating the `GetInput` instance once and calling its `get_input` and `process_value` methods directly.
2) To use the *`get_list()`* convenience function.

Creating an instance of teh GetInput class is easy; it takes the same parameters at the *`get_input()`* function:

- **cleaners**: list of cleaners to apply to clean the value
- **convertor**: the convertor to apply to the cleaned value
- **validators**: list of validators to apply to validate the cleaned and converted value
- ****options**: optional values. see the *`GetInput`* documentation for details.

3.7 Using get_list:

The *`get_list()`* convenience function is used to get a list of values from the user. The `cleaners`, `convertor`, and `validators` parameters are applied to the list returned. For example, using *`LengthValidator`* as the validator will check the length of the list, not the length of an element. There are two additional parameters used by `get_list`:

- **delimiter: the string (generally a single character) used to separate the elements of the list. By** default a comma (',') is used as the delimiter.
- **elem_get_input**: an instance of *`GetInput`* to apply to each element of the list.

`Get_list` cleans the input string, then calls *`ListConvertor`* to split the input string by the delimiter and runs the CCV process defined in the `elem_get_input` parameter on each each element of the list. Finally the validator functions are run on the converted (list) value.

Fig. 2: The get_list pipeline

For example, to get a list of exactly three integers:

```
prompt_str = 'Enter a list of 3 integers'
elem_gi = GetInput(convertor=IntConvertor())
len_validator = LengthValidator(min_len=3, max_len=3)
result = get_list(prompt=prompt_str, elem_get_input=elem_gi,
    validators=len_validator)
```

A second example is to get a list of at least two integers between *-5* and *5*

```
prompt_str = 'Enter a list of at least 2 integers (between -5 and 5)'
elem_gi = GetInput(convertor=IntConvertor(),
    validators=[RangeValidator(min_val=-5, max_val=5)])
length_validator = LengthValidator(min_len=2)
result = get_list(prompt=prompt_str, elem_get_input=elem_gi,
    validators=length_validator)
```

Note: The use of `elem_get_input` is a little strange to think about at first, but allows a lot of flexibility for getting list input. For instance, you can create a list where each element is a list itself by using *`ListConvertor`* as the `convertor` to `elem_get_input`.

However, `get_list()` is currently limited to creating homogenous lists (i.e. each element must pass the same CCV chain) as *get_elem_input'* takes a single instance of `GetInput`.

3.8 From Here:

That completes the first part of the `cooked_input` tutorial. You can continue with the second part of the tutorial. Or, for more information take a look at the how-to/FAQ section of the documentation. You can also look at the various examples.

CHAPTER 4

Cooked Input Tutorial, Part Two

4.1 Introduction:

This is the second part of the tutorial for `cooked_input`. It assumes that you have completed the first part of the tutorial. In this tutorial you will create a simple, menu driven, application that demonstrates some of cooked_input's advanced features: tables, menus, and commands.

To start we will import cooked_input and create lists to hold event types and events

```
from collections import namedtuple
import cooked_input as ci

EventType = namedtuple('EventType', 'id name desc')
Event = namedtuple('Event', 'id date desc type')

event_types = [
    EventType(1, 'birthday' ,'a birthday event'),
    EventType(2, 'anniversary' ,'an anniversary event'),
    EventType(3, 'meeting', 'a meeting event')
]
events = []
```

4.2 Commands:

Next we will create some commands for the application. A cooked_input command is a string the user can enter during a cooked_input input request that calls a callback function.

Commands are specified by by passing a dictionary, where the key is the string and the value is a *`GetInputCommand`* instance, to the `commands` parameter of a cooked_input input call. `GetInputCommand` objects consist of a callback function and an optional dictionary of values to be sent to the callback (`cmd_dict`). Command callbacks take three input parameters (`cmd_str`, `cmd_vars`, and `cmd_dict`) and return a `CommandResponse` tuple. `cmd_str` and `cmd_vars` are the command and any parameters after the command, and `cmd_dict` is the dictionary specified in

the GetInputCommand. CommandResponse is a tuple where the first value is the return type of the command and the second value the value returned by the command.

For example, the following code creates a /in_to_mm command to convert from inches to millimeters and supplies the result as the input:

```
def in_to_mm_cmd_action(cmd_str, cmd_vars, cmd_dict):
    mm = int(cmd_vars) * 25.4  # 1 inch is 25.4 millimeters
    return ci.CommandResponse(ci.COMMAND_ACTION_USE_VALUE, str(mm))

cmds = {'/in_to_mm': ci.GetInputCommand(in_to_mm_cmd_action)}
v = ci.get_float(prompt="How long is it (in mm)", commands=cmds)
print(v)
```

Running the code, we can use the command to use 2 inches as the input:

```
>>> How long is it (in mm): /in_to_mm 2
>>> 50.8
```

In addition to commands that return values, there are CommandResponses for purely informational commands (**COMMAND_ACTION_NOP**) and cancellation commands (**COMMAND_ACTION_CANCEL**).

Let's go ahead and add two commands to the event manager application. The first displays a help message help, and the second to cancel the current operation:

```
def help_cmd_action(cmd_str, cmd_vars, cmd_dict):
    help_str = """
        Commands:
            /?, /help   Display this help message
            /cancel     Cancel the current operation
    """
    print(help_str)
    return ci.CommandResponse(ci.COMMAND_ACTION_NOP, None)

def cancel_cmd_action(cmd_str, cmd_vars, cmd_dict):
    if ci.get_yes_no(prompt='Are you sure?', default='no') == 'yes':
        print('\nCommand cancelled...')
        return ci.CommandResponse(ci.COMMAND_ACTION_CANCEL, None)
     else:
        return ci.CommandResponse(ci.COMMAND_ACTION_NOP, None)

help_cmd = ci.GetInputCommand(help_cmd_action)
cancel_cmd = ci.GetInputCommand(cancel_cmd_action)
commands_std = { '/?': help_cmd, '/h': help_cmd, '/cancel': cancel_cmd }
```

Note: As was done in the help command, you can have more than one command point to the same command GetInputCommand. Also, as was done in the cancel commands, you can call cooked_input functions within a command callback.

More detail about cooked_input commands can be found at: command

4.3 Tables:

cooked_input can also display tables of data. The easiest way to create a table is to use the *create_table()* convenience function. create_table can create a table any iterable where the fields in each item can be fetched

by field or attribute name, including: objects, dictionaries, and namedtuples.

`create_table` requires two parameters: `items`, an iterable containing row data for the table, and `fields`, the list of which fields/attributes from the items to show as columns in the table.

Once created, the table can be displayed using its `show_table` method:

```
flds = ['name', 'desc']
tbl = ci.create_table(items=event_types, fields=flds)
tbl.show_table()
```

will display:

```
+-------------+----------------------+
|        name | desc                 |
+-------------+----------------------+
|    birthday | a birthday event     |
| anniversary | an anniversary event |
|     meeting | a meeting event      |
+-------------+----------------------+
```

There are several other common parameters for `create_table`. `field_names` is a list of strings to use for the column headers, *title* sets a title for the table, and `style` specified a *`TableStyle`* for used when displaying the table.

You can also use tables for input by either calling the the *`get_table_input()`* convenience function, or the table's `get_table_choice` method. You can see this in action by trying the following:

```
>>> v = ci.get_table_input(tbl, prompt='event type')
>>>
>>> +-------------+----------------------+
>>> |        name | desc                 |
>>> +-------------+----------------------+
>>> |    birthday | a birthday event     |
>>> | anniversary | an anniversary event |
>>> |     meeting | a meeting event      |
>>> +-------------+----------------------+
>>>
>>> event type: m
>>> >>> print(v)
>>> TableItem(col_values=['a meeting event'], tag=meeting, action=default, item_
↪data=None, hidden=False, enabled=True)
```

Note: Notice that we only had to type in 'm' to choose `meeting`, `get_table_choice` automatically adds a ChoiceCleaner as a convenience!

Each row in a cooked_input table is a *`TableItem`*, and by default `get_table_choice` will return the *TableItem* for the choice entered. This can be modified by setting the `default_action` parameter, For instance, setting `default_action` to **TABLE_RETURN_FIRST_VAL** would have returned "meeting" in the example above.

Note: `create_table` can create a table by passing in the query from an ORM-managed database such as SQLAlchemy. When doing so it's useful to set the `add_item_to_item_data` parameter in `create_table` to **True**. This will automatically attach the full data for the item to the row's item_data. In the example above the `id` for the event_type is not in the table. By setting `add_item_to_item_data` = **True** it could be accessed by through `v.item_data['item']['id']`. This makes it easy to get the primary key of the choice's database entry

even though it's not shown in the table.

The default action for the table can also be set to a function or callable. The action callback receives two input parameters. `Row` contains the `TableItem` for the row chosen, which includes a `values` field containing the columns values for the row. The callback function also receives `action_data` which is an optional dictionary of values to send to the action. For instance, we can send a capitalized version of the event type with the following default action callback

```
def cap_action(row, action_item):
    val = row.tag.upper()
    return val

tbl = ci.create_table(event_types, flds, default_action=cap_action)
```

would produce:

```
>>> +-------------+----------------------+
>>> |        name | desc                 |
>>> +-------------+----------------------+
>>> |    birthday | a birthday event     |
>>> | anniversary | an anniversary event |
>>> |     meeting | a meeting event      |
>>> +-------------+----------------------+
>>>
>>> event type: a
>>> ANNIVERSARY
```

Lets create some action callbacks to out application to: add an event, list all of the events and delete all of the events:

```
def reset_db_action(row, action_item):
    cmds = action_dict['commands']
    if ci.get_yes_no(prompt='Delete all events? ', default='no',
                     commands=cmds) == 'yes':
        action_dict['events'] = []

def add_event_action(row, action_item):
    events = action_dict['events']
    event_types = action_dict['event_types']
    cmds = action_dict['commands']
    desc = ci.get_string(prompt="Event description? ", commands=cmds)
    tbl = ci.create_table(event_types, ["name", "desc"], ["Name", "Desc"],
        add_item_to_item_data=True)
    event_type = tbl.get_table_choice(prompt='Type? ', commands=cmds'])
    date = ci.get_date(prompt='Date? ', default='today', commands=cmds)
    type_id = event_type.item_data['item'].id
    events.append(Event(len(events)+1, date, desc, type_id))

def list_event_action(row, action_item):
    events = action_dict['events']
    event_types = action_dict['event_types']

    if len(events) == 0:
        print('\nno events\n')
        return

    et_dict = {item.id: item.name for item in event_types}
    items = []
```

(continues on next page)

(continued from previous page)

```
    for e in events:
        date = e.date.strftime('%x')
        etype = et_dict[e.type]
        items.append({'id': e.id, 'date': date, 'desc': e.desc, 'type': etype})

    fields = ['date', 'desc', 'type']
    field_names = ['Date', 'Desc', 'Type']
    tbl = ci.create_table(items, fields, field_names, title='Events')
    print('\n')
    tbl.show_table()
    print('\n')
```

Note: These action callbacks depend on receiving the commands, events and event_types lists in `action_dict`. Action_dict provides a method of sending data to and from the callback without using global variables. This mechanism is useful to provide context such as: database sessions/connections, user information, etc.

4.4 Menus:

The final piece of the application is to add menus. In cooked_input menus are tables where the action callback performs the action for the menu item. The `run` method on tables loops calling `get_table_choice`. By default the menu will loop indefinately. A menu option to exit the menu can be added by setting the `add_exit` parameter to **TABLE_ADD_EXIT** when creating the table.

Submenus can be created by running a table from a menu item action callback and setting the `add_exit` parameter to **TABLE_ADD_RETURN** when creating the table.

Let's finish the application by adding a main menu and database submenu:

```
def db_submenu_action(row, action_item):
    style = action_dict['menu_style']
    items = [ ci.TableItem('Delete all events', action=reset_db_action) ]
    menu = ci.Table(rows=items, add_exit=ci.TABLE_ADD_RETURN, style=style,
        action_dict=action_dict)
    menu.run()

if __name__ == '__main__':
    style = ci.TableStyle(show_cols=False, show_border=False)
    action_dict = { 'events': events, 'event_types': event_types,
        'commands': commands_std, , 'style': style }

    items = [
            ci.TableItem('Add an event', action=add_event_action),
            ci.TableItem('List events', action=list_event_action),
            ci.TableItem('Database submenu', action=db_submenu_action)
        ]
    menu = ci.Table(rows=items, add_exit=ci.TABLE_ADD_EXIT, style=style,
        action_dict=action_dict)
    menu.run()
```

4.5 Running the application:

Try running the application (the full code is available at events.py). You can test adding events and listing them. Don't forget to try the commands. For example, start adding event and type **\cancel** to stop in the middle of the operation without adding the event.

You may also want to try adding a new menu item to the database menu to add an event type. You'll see just how easy it is to add new functionality to an application with cooked_input.

4.6 From Here:

That completes the second part of the cooked_input tutorial. For more information take a look at the how-to/FAQ section of the documentation. You can also look at the various examples.

CHAPTER 5

How-to / FAQ

Some examples of how to. . . .

5.1 Getting yes or no

To get 'yes' or 'no':

```
get_yes_no()
```

Adding the *default* option, specifies to return 'yes' if a blank value is entered:

```
get_yes_no(default='Y')
```

which is equivalent to writing:

```
get_input(prompt="Enter yes or no?", cleaners=StripCleaner(),␣
↪convertor=YesNoConvertor(), default='Y')
```

5.2 Restricting to a list of choices

To get that is restricted to a value from a list of choices:

```
colors = ['red', 'green', 'blue']
color_validator = ChoiceValidator(colors)
prompt_str = 'What is your favorite color (%s)' % ', '.join(colors)
result = get_input(prompt=prompt_str, cleaners=[StripCleaner(),␣
↪CapitalizationCleaner(style='lower')], validators=color_validator)
```

Note: Validator functions compare the exact value sent from the cleaned input. Without the specified cleaners in the example above any leading or trailing white space characters or capital letters would prevent matches on the ChoiceValidator.

Adding a *ChoiceCleaner*, allows the user to just input the first few letters of the choice (enough to differentiate to a single choice.):

```
colors = ['red', 'green', 'blue', 'black']
color_cleaner = ChoiceCleaner(colors)
color_validator = ChoiceValidator(colors)
prompt_str = 'What is your favorite color (%s)' % ', '.join(colors)
result = get_input(prompt=prompt_str, cleaners=[StripCleaner(), CapitalizationCleaner(
↪'lower'), color_cleaner], validators=color_validator)
```

Typing *"r"* or *"g"* would be enough to match *red* or *green* respectively, but three letters (e.g. *"blu"*) would be required to differentiate between *black* and *blue*.

5.3 Excluding a list of choices

The following example will except any string value, except *"licorice"* or *"booger"*:

```
bad_flavors = ['licorice', 'booger']
not_in_choices_validator = NoneOfValidator(bad_flavors)

prompt_str = "What is your favorite flavor of jelly bean (don't say: %s)?" % ' or '.
↪join(bad_flavors)
response = get_input(prompt=prompt_str, cleaners=[StripCleaner(),␣
↪CapitalizationCleaner(style='lower')], validators=not_in_choices_validator)
```

Note: The *AnyOf* and *NoneOf* validators can take either values or validator functions as validators. For instance, in the example above you could also use: not_in_choices_validator = NoneOfValidator(ChoiceValidator(bad_flavors))

5.4 Composing Multiple Validators

Of course you can compose an arbitrary number of these together. For instance, to get a number from *-10* to *10*, defaulting to *5*, and not allowing *0*:

```
prompt_str = "Enter a number between -1 and 10, but not 0"
validators = [RangeValidator(-10, 10), NoneOfValidator(0)]
response = get_input(prompt=prompt_str, convertor=IntConvertor(),␣
↪validators=validators, default=5)
```

5.5 More Examples

Cooked_input has a lot more of functionality for getting input of different types (floats, Booleans, Dates, lists, passwords, etc.), as well as lots of validators and cleaners. It also has a number of features for getting input from tables (which is nice for working with values in database tables). There are a lot of examples in the examples directory.

5.6 TODO

- error callback, convertor_error_fmt and validator_error_fmt
- Menus
- Actions/callbacks/action_dicts
- TableItems/Tables
- item filters, enabled, hidden items
- pagination
- Commands

example commands:

```
def user_reverse_action(cmd_str, cmd_vars, cmd_dict):
        user = cmd_dict['user']
        return (COMMAND_ACTION_USE_VALUE, user[::-1])

def to_mm_cmd_action(cmd_str, cmd_vars, cmd_dict):
    # Convert a value from inches to mm and use the resulting value for the input
    try:
        inches = float(cmd_vars)
        return (COMMAND_ACTION_USE_VALUE, str(inches * 25.4))
    except(ValueError):
        print(f'Invalid number of inches provided to {} command.'.format(cmd_str)
        return (COMMAND_ACTION_NOP, None)
```

- Exceptions

CHAPTER 6

Convenience Functions

Using these convenience functions you can get up and running in `cooked_input` very quickly. Most users can just use these convenience functions and never have to go deeper into the library.

The convenience functions can also take advantage of the rich set up Cleaners, Convertors, and Validators in the `cooked_input` library.

6.1 GetInput Convenience Functions

These functions create a *`GetInput`* object with parameter values for the type desired (e.g. the convertor and a reasonable prompt and cleaners.) The convenience functions are just syntactic sugar for calls to *`GetInput`*, but simpler to use. For instance, the following two versions calls do the same thing:

```
# GetInput version:
gi = GetInput(prompt='Enter a whole number', convertor=IntConvertor())
result = gi.get_input()

# Convenience function:
result = get_int(prompt='Enter a whole number')
```

6.1.1 get_string

`cooked_input.`**`get_string`**(*cleaners=StripCleaner(lstrip=True, rstrip=True), validators=None, min_len=None, max_len=None, **options*)

Parameters

- **`cleaners`** (*`List[Cleaner]`*) – list of cleaners to apply to clean the value.
- **`validators`** (*`List[Validator]`*) – list of validators to apply to validate the cleaned and converted value
- **`min_len`** (*`int`*) – the minimum allowable length for the string. No minimum length if None (default)

- **max_len** (*int*) – the maximum allowable length for the string. No maximum length if None (default)
- **options** – all *GetInput* options supported, see *GetInput* documentation for details.

Returns the cleaned, converted, validated string

Return type str

Convenience function to get a string value.

6.1.2 get_int

cooked_input.**get_int**(*cleaners=None*, *validators=None*, *minimum=None*, *maximum=None*, *base=10*, ***options*)

Parameters

- **cleaners** (*List[Cleaner]*) – list of cleaners to apply to clean the value.
- **validators** (*List[Validator]*) – list of validators to apply to validate the cleaned and converted value
- **minimum** (*int*) – minimum value allowed. Use None (default) for no minimum value.
- **maximum** (*int*) – maximum value allowed. Use None (default) for no maximum value.
- **base** (*int*) – Convert a string in radix base to an integer. Base defaults to 10.
- **options** – all *GetInput* options supported, see *GetInput* documentation for details.

Returns the cleaned, converted, validated int value

Return type int

Convenience function to get an integer value. See the documentation for the Python int builtin function for further description of the *base* parameter.

6.1.3 get_float

cooked_input.**get_float**(*cleaners=None*, *validators=None*, *minimum=None*, *maximum=None*, ***options*)

Parameters

- **cleaners** (*List[Cleaner]*) – list of cleaners to apply to clean the value.
- **validators** (*List[Validator]*) – list of validators to apply to validate the cleaned and converted value
- **minimum** (*float*) – minimum value allowed. Use None (default) for no minimum value.
- **maximum** (*float*) – maximum value allowed. Use None (default) for no maximum value.
- **options** – all *GetInput* options supported, see *GetInput* documentation for details.

Returns the cleaned, converted, validated float value

Return type float

Convenience function to get an float value.

6.1.4 get_boolean

cooked_input.**get_boolean**(*cleaners=StripCleaner(lstrip=True, rstrip=True), validators=None, **options*)

Parameters

- **cleaners** (*List[Cleaner]*) – list of cleaners to apply to clean the value.
- **validators** (*List[Validator]*) – list of validators to apply to validate the cleaned and converted value
- **options** – all *GetInput* options supported, see *GetInput* documentation for details.

Returns the cleaned, converted, validated boolean value

Return type bool

Convenience function to get a Boolean value. See *BooleanConvertor* for a list of values accepted for *True* and *False*.

6.1.5 get_date

cooked_input.**get_date**(*cleaners=StripCleaner(lstrip=True, rstrip=True), validators=None, minimum=None, maximum=None, **options*)

Parameters

- **cleaners** (*List[Cleaner]*) – list of cleaners to apply to clean the value. Not needed in general.
- **validators** (*List[Validator]*) – list of validators to apply to validate the cleaned and converted value
- **minimum** (*datetime*) – earliest date allowed. Use None (default) for no minimum value.
- **maximum** (*datetime*) – latest date allowed. Use None (default) for no maximum value.
- **options** – all *GetInput* options supported, see *GetInput* documentation for details.

Returns the cleaned, converted, validated date value

Return type datetime

Convenience function to get a date value. See *DateConvertor* for more information on converting dates. Get_date can be used to get both times and dates.

6.1.6 get_yes_no

cooked_input.**get_yes_no**(*cleaners=StripCleaner(lstrip=True, rstrip=True), validators=None, **options*)

Parameters

- **cleaners** (*List[Cleaner]*) – list of cleaners to apply to clean the value. Not needed in general.
- **validators** (*List[Validator]*) – list of validators to apply to validate the cleaned and converted value
- **options** – all *GetInput* options supported, see *GetInput* documentation for details.

Returns the cleaned, converted, validated yes/no value

Return type str (**"yes"** or **"no"**)

Convenience function to get an yes/no value. See *YesNoConvertor* for a list of values accepted for *yes* and *no*.

6.1.7 get_list

cooked_input.**get_list**(*elem_get_input=None*, *cleaners=None*, *validators=None*, *value_error_str=u'list of values'*, *delimiter=u','*, ***options*)

Parameters

- **elem_get_input** (*GetInput*) – an instance of a *GetInput* to apply to each element
- **cleaners** (*List[Cleaner]*) – cleaners to be applied to the input line before the *ListConvertor* is applied.
- **validators** (*List[Validator]*) – list of validators to apply to validate the converted list
- **value_error_str** (*str*) – the error string for improper value inputs
- **delimiter** (*str*) – the delimiter to use between values
- **options** – all get_input options supported, see get_input documentation for details.

Returns the cleaned, converted, validated list of values. For more information on the *value_error_str*, *delimeter*, *elem_convertor*, and elem_valudator' parameters see *ListConvertor*.

Return type List[Any]

Get a homogenous list of values. The *GetInput.process_value()* method on the elem_get_input *GetInput* instance is called for each element in the list.

Example usage - get a list of integers between 3 and 5 numbers long, separated by colons (:):

```
elem_gi = GetInput(convertor=IntConvertor())
length_validator = RangeValidator(min_val=3, max_val=5)
list_validator = ListValidator(len_validator=length_validator)
prompt_str = 'Enter a list of integers, each between 3 and 5, separated by ":"'
result = get_list(prompt=prompt_str, elem_get_input=elem_gi, validators=list_
↪validator, delimiter=":")
```

6.1.8 get_input

cooked_input.**get_input**(*cleaners=None*, *convertor=None*, *validators=None*, ***options*)

Parameters

- **cleaners** (*List[Cleaner]*) – list of cleaners to apply to clean the value. Not needed in general.
- **convertor** (*Convertor*) – the convertor to apply to the cleaned value
- **validators** (*List[Validator]*) – list of validators to apply to validate the cleaned and converted value
- **options** – all *GetInput* options supported, see *GetInput* documentation for details.

Returns the cleaned, converted, validated input string

Return type Any (returned value is dependent on type returned from `convertor`)

Convenience function to create a *GetInput* instance and call its *get_input* function. See *GetInput.get_input()* for more details.

6.1.9 process_value

cooked_input.**process_value**(*value*, *cleaners=None*, *convertor=None*, *validators=None*, *error_callback=<function print_error>*, *convertor_error_fmt='"{value}" cannot be converted to {error_content}'*, *validator_error_fmt='"{value}" {error_content}'*)

Parameters

- **value** (*str*) – the value to process
- **cleaners** (*List[Cleaner]*) – list of cleaners to apply to clean the value
- **convertor** (*Convertor*) – the convertor to apply to the cleaned value
- **validators** (*List[Validator]*) – list of validators to apply to validate the cleaned and converted value
- **error_callback** (*str*) – a callback function to call when an error is encountered. Defaults to *print_error()*
- **convertor_error_fmt** (*str*) – format string to use for convertor errors. Defaults to **DEFAULT_CONVERTOR_ERROR**
- **validator_error_fmt** (*str*) – format string to use for validator errors. Defaults to **DEFAULT_VALIDATOR_ERROR**

Returns the cleaned, converted validated input value.

Return type Any (returned value is dependent on type returned from `convertor`)

Convenience function to create a *GetInput* instance and call its process_value function. See *GetInput.process_value()* for more details. See *GetInput* for more information on the *error_callback*, *convertor_error_fmt*, and *validator_error_fmt* parameters.

6.1.10 validate

cooked_input.**validate**(*value*, *validators*, *error_callback=<function print_error>*, *validator_fmt_str='"{value}" {error_content}'*)

return **True** is a value passes validation.

Parameters

- **value** (*Any*) – the value to validate
- **validators** (*List[Validator]*) – an iterable (list or tuple) of validators to run on `value`
- **error_callback** (*Callable*) – a function called when an error occurs during validation
- **validator_fmt_str** (*str*) – format string to pass to the error callback routine for formatting the error

Returns **True** if the input passed validation, else **False**

Return type boolean

6.2 Table Convenience Functions

These functions create a *Table* object with everything needed to display a simple menu or table. The convenience functions are just syntactic sugar for calls to *Table*, but simpler to use. For instance, the following two versions do the same thing:

```
# GetInput version:
menu_choices = [ TableItem('red'), TableItem('green'), TableItem('blue') ]
menu = Table(rows=menu_choices, prompt='Pick a color')
result = menu.get_table_choice()

# Convenience function:
result = get_menu(['red', 'green', 'blue'], prompt='Pick a color')
```

6.2.1 get_menu

cooked_input.**get_menu**(*choices*, *title=None*, *prompt=None*, *default_choice=None*, *add_exit=False*, *style=None*, ***options*)

Parameters

- **choices** – the list of text strings to use for the menu items
- **title** – a title to use for the menu
- **prompt** – the prompt string used when asking the user for the menu selection
- **default_choice** – an optional default item to select
- **add_exit** – add an exit item if *True* or not if *False* (default)
- **style** – a *TableStyle* defining the look of the menu.
- **options** – all *Table* options supported, see *Table* documentation for details.

Returns the result of calling *Table.get_table_choice()* on the menu table. Will return the index (one based) of the choice selected, unless a different default action is provided in the options. Returns 'exit' if the input value is *None* or the menu was exited.

Return type int or str (dependent on default action specified)

This is a convenience function to create a Table that acts as a simple menu. It takes a list of text strings to use for the menu items, and returns the text string of the item picked. *get_menu* is just syntactic sugar for calls to the *Table* class, but simpler to use.

6.2.2 create_rows

cooked_input.**create_rows**(*items*, *fields*, *gen_tags=None*, *item_data=None*, *add_item_to_item_data=False*)

Create a list of TableItems from an iterable (items) of objects

Parameters

- **items** – iterable containing items for the table.
- **fields** (*List[str]*) – list of field/attribute names to use as column values for each item.
- **gen_tags** (*bool*) – if **True** will generate sequentially numbered tags for table items, if False (default) uses first column value of each item for the row's tag.

- **item_data** (*Dict*) – An optional dictionary to be copied and attached to the *TableItem* for the row.
- **add_item_to_item_data** (*bool*) – if **True** `item_data['item']` is set to *item*.

Returns List[TableItem] of table items (*TableItem*)

`create_rows` is a convenience function used to create a list of table items (*TableItem*) for a `cooked_input` *Table*. `create_rows` tries to make it easy to create the rows for a table from a list of data or a query.

`create_rows` takes an iterable of `items`, such as a list, dictionary or query. The items in `items` can be just about anything too: objects, lists, dictionaries, tuples, or namedtuples. `create_rows` also takes a list of `fields` with each item in the list the name of a field or attribute in the items. `create_rows` iterates through `items` and add the value for each field as a column value for the table row.

`create_rows` fetches the field data based on the following:

```
1. If ``hasattr`` for the fields returns **True** (**__getattr__** is defined),␣
↪uses ``getattr`` to retreive
    field values. This works nicely for class instances, named tuples, and␣
↪database query results from an
    object-relationship mapper (ORM).
2. If the items are dictionaries, uses ``get`` to retreive the value for the key␣
↪matching the field name.
3. If both of the previous methods fail, the first len(fields) values of the item␣
↪are used (requires
    **__get_item__** to be defined.)
```

Note: Care is taken to make a single pass through the `items` iterable as some iterables are non-reentrant (e.g. generators and some database queries)

Example usage - get a list of integers between 3 and 5 numbers long, separated by colons (:):

```
class Person(object):
    def __init__(self, first, last, age, shoe_size):
        self.first = first
        self.last = last
        self.age = age
        self.shoe_size = shoe_size

people = [
    Person('John', 'Cleese', 78, 14),
    Person('Terry', 'Gilliam', 77, 10),
    Person('Eric', 'Idle', 75, 12),
]

rows = create_rows(people, ['last', 'first', 'shoe_size'])
Table(rows, ['First', 'Shoe Size'], tag_str='Last').show_table()
```

`create_rows` is called by *create_table()* to create the table rows.

6.2.3 create_table

cooked_input.**create_table**(*items*, *fields*, *field_names=None*, *gen_tags=None*, *item_data=None*, *add_item_to_item_data=False*, *title=None*, *prompt=None*, *default_choice=None*, *default_str=None*, *default_action='table_item'*, *style=None*, ***options*)

Convenience function to create `cooked_input` a table.

Parameters

- **items** – iterable containing items for the table.
- **fields** (*List[str]*) – list of field/attribute names to use as column values for each item.
- **field_names** (*List[str]*) – a list of strings to use for the names of the table columns.
- **gen_tags** (*bool*) – if **True** will generate sequentially numbered tags for table items, if False (default) uses first column value of each item for the row's tag.
- **item_data** (*Dict*) – An optional dictionary to be copied and attached to the *TableItem* for the row.
- **add_item_to_item_data** (*bool*) – if **True** `item_data['item']` is set to *item*.
- **title** (*str*) – an optional string to use as the title for the table.
- **prompt** (*str*) – an optional string to use for the table prompt.
- **default_choice** (*str*) – an optional default value to use for when getting input from the table.
- **default_str** (*str*) – an optional string to display for the default choice value.
- **default_action** – the default action to take when a table item is picked. Defaults to **TABLE_RETURN_TABLE_ITEM***.
- **style** (`TableStyle`) – an optional *TableStyle* to use for the table.
- **options** – a dictionary of optional values for the table. See *Table* for details.

Returns an instance of a `cooked_input` *Table*

`create_table` is a convenience function used to create a `cooked_input` table (*Table*) from a list of data or a query.

`create_table` calls *create_rows()* to create the table rows. See *create_rows()* for an explanation of the: `items`, `fields`, `gen_tags`, `item_data`, and `add_item_to_item_data` parameters.

See *Table* for an explanation of the: `title`, `prompt`, `default_choice`, `default_str`, `default_action`, `style` and `options` parameters.

Note: all items are created with the same `default_action` and `item_data` (with exception of adding the item to `item_data['item']` if `add_item_to_item_data` is **True**.)

Note: By default all items (rows) are visible and enabled. Rows can be hidden or disabled by setting an `item_filter` value in the `options` dictionary.

Example usage - get a list of integers between 3 and 5 numbers long, separated by colons (:):

```
items = {
    1: {"episode": 1, "name": "Whither Canada?", "date": "5 October, 1969",
↪"season": 1},
    2: {"episode": 4, "name": "Owl Stretching Time", "date": "26 October, 1969",
↪"season": 1},
    3: {"episode": 15, "name": "The Spanish Inquisition", "date": "22 September,␣
↪1970", "season": 2},
    4: {"episode": 35, "name": "The Nude Organist", "date": "14 December, 1972",
↪"season": 2}
}

fields = 'episode name date'.split()
field_names = 'Episode Name Date'.split()
tbl = create_table(items, fields, field_names, add_item_to_item_data=True, title=
↪'Episode List')
choice = tbl.get_table_choice()
item = choice.item_data["item"]
print('{}: {}'.format(item['name'], item['season']))
```

6.2.4 show_table

cooked_input.**show_table**(*table*, ***options*)

Displays a table without asking for input from the user.

Parameters

- **table** – a *Table* instance
- **options** – all *Table* options supported, see *Table* documentation for details

Returns None

6.2.5 get_table_input

cooked_input.**get_table_input**(*table*, ***options*)

Get input value from a table of values.

Parameters

- **table** – a *Table* instance
- **options** – all *Table* options supported, see *Table* documentation for details

Returns the value from calling *Table.get_table_choice()* on the table

Return type Any (dependent on the action function of the *TableItem* selected)

CHAPTER 7

Cleaners

Cleaner classes for cleaning input before conversion and validation.

7.1 Creating Cleaners

Cleaner classes inherit from the `Cleaner` base class. They must be callable, with the `__call__` dunder method taking one parameter, the value. An example of a cleaner to change the input value to lower case looks like:

```
class LowerCleaner(Cleaner):
    def __init__(self, **kwargs):
        # initialize any specific state for the cleaner.
        pass

    def __call__(self, value):
        return value.lower()
```

7.2 Cleaners

7.2.1 CapitalizationCleaner

class cooked_input.**CapitalizationCleaner**(*style='lower'*)

param CAP_STYLE_STR style (optional) capitalization style to use. Defaults to `LOWER_CAP_STYLE`

return the cleaned (capitalized) value

rtype str

Capitalize the value using the specified style

The styles are equivalent to the following:

style	equivalent string function
LOWER_CAP_STYLE	lower
UPPER_CAP_STYLE	upper
FIRST_WORD_CAP_STYLE	capitalize
LAST_WORD_CAP_STYLE	–[1]
ALL_WORDS_CAP_STYLE	capwords

7.2.2 ChoiceCleaner

class cooked_input.**ChoiceCleaner**(*choices*, *case_sensitive=True*)

param List[str] choices the list of choices to match

param bool case_sensitive (optional) if **True** (default) matching the choice is case sensitive, otherwise matching is case insensitive

return the cleaned (matched choice from the `choices` list) value or the original value if no match is found

rtype str (type is dependent on the mapped value in `choices` but is generally *str*)

Note: The cleaned output uses the same capitalization as the item matched from the choices list regardless of the `case_sensitive` parameter.

ChoiceCleaner tries to replace the input value with a single element from a list of choices by finding the unique element starting with the input value. If no single element can be identified, the input value is returned (i.e. no cleaning is performed.) This is a complicated way of saying you can type in the first few letters of an input and the cleaner will return the choice that starts with those letters if it can determined which one it is.

For example:

```
ChoiceCleaner(choices=['blue', 'brown', 'green'], case_sensitive=True)
```

will with the following input values would return the following values:

value	returns	note
'g'	'green'	
'br'	'brown'	
'blu'	'blue'	
'b'	'b'	original value returned as can't tell between 'brown' and 'blue'
'BR'	'BR'	original value returned as case of the input does not match[2]

7.2.3 RegexCleaner

class cooked_input.**RegexCleaner**(*pattern*, *repl*, *count=0*, *flags=0*)

Parameters

[1] There is no standard library function for capitalizing the last word in a string. This was added to properly capitalize family names, (e.g. *"van Rossum"*.) This parameter is dedicated to Colleen, who will be happy to capitalize on getting the last word.

[2] Would return *"brown"* if `case_sensitive` is **False**

- **pattern** (*Pattern[str]*) – regular expression to search for
- **repl** (*str*) – string to substitute for occurrences of `pattern`
- **count** (*int*) – (optional) the maximum number of substitutions to perform on the input value. Default is to replace all occurrences
- **flags** – (optional) flags. Default is no flags. See below for details

Returns the cleaned (**pattern** replaced with **repl**) value

Return type str

Return the string obtained by replacing the leftmost non-overlapping occurrences of *pattern* in the input value by the replacement *repl*. If the pattern is not found in the input value, the value is returned unchanged. `Count` occurrences, from left to right, are replaced. If count is **0**, or not specified, all occurrences are replaced.

For more information on regular expressions and the meaning of count and flags. See the re.sub function in the re module in the Python standard library.

7.2.4 RemoveCleaner

class cooked_input.**RemoveCleaner**(*patterns*, *count=0*)

Parameters

- **patterns** (*List[str]*) – a list of strings to remove
- **count** (*int*) – (optional) the maximum number of substitutions to perform on the input value. Default is to remove all occurrences

Returns the cleaned (`patterns` removed) value

Return type str

Removes all occurrences of any of the strings in the `patterns` list from the input value.

7.2.5 ReplaceCleaner

class cooked_input.**ReplaceCleaner**(*old*, *new*, *count=0*)

Parameters

- **old** (*str*) – string to replace
- **new** (*str*) – string to substitute for occurrences of `old`
- **count** (*int*) – (optional) the maximum number of substitutions to perform on the input value. Default is to replace all occurrences

Returns the cleaned (`old` replaced with `new`) value

Return type str

Replaces occurrences of `old` string with `new` string from the input value. If *count* is specified the first `count` occurrences, from left to right, are replaced. If count is **0**, or not specified, all occurrences are replaced.

7.2.6 StripCleaner

class cooked_input.**StripCleaner**(*lstrip=True*, *rstrip=True*)

Parameters

- **lstrip** (*bool*) – (optional) strips white space from the left side of the value if **True** (default)
- **rstrip** (*bool*) – (optional) strips white space from the right side of the value if **True** (default)

Returns the cleaned (stripped) value

Return type str

Strips white space from the input value. Strips from the left side if lstrip=True, and from the right side if rstrip=True. Both are True by default (i.e. strips from both left and right).

CHAPTER 8

Convertors

Convertors classes for converting the string input to the desired output type. The GetInput class calls the Convertor after cleaning and before validation.

8.1 Creating Convertors

Convertor classes inherit from the Convertor base class. They must be callable, with the *__call__* dunder method taking three parameters: the value to convert, a function to call when an error occurs and the format string for the error function. The *__call__* method returns the converted value. Error conditions are handled by calling the *error_callback* function . See error_callbacks for more information on error functions and their format strings. The *__init__* method should use *super* to call the *__init__* method on the Convertor base class so *value_error_str* gets set.

An example of a convertor to change the input value to an integer looks like:

```
class IntConvertor(Convertor):
    # convert to a value to an integer
    def __init__(self, base=10, value_error_str='an integer number'):
        self._base = base
        super(IntConvertor, self).__init__(value_error_str)

    def __call__(self, value, error_callback, convertor_fmt_str):
        try:
            return int(value, self._base)
        except ValueError:
            error_callback(convertor_fmt_str, value, 'an int')
            raise   # re-raise the exception

    def __repr__(self):
        return 'IntConvertor(base={}, value_error_str={})'.format(self._base, self.
↪value_error_str)
```

8.2 Convertors

8.2.1 BooleanConvertor

class cooked_input.**BooleanConvertor**(*value_error_str='true or false'*)
convert to a boolean value (**True** or **False**.)

Parameters **value_error_str** (*str*) – (optional) the error string to use when an improper value is input

Returns value converted to a *boolean*

Return type boolean (**True** or **False**)

Raises ***ConvertorError*** – if value cannot be converted to *bool*

BooleanConvertor returns **True** for input values: 't', 'true', 'y', 'yes', and '1'. *BooleanConvertor* returns **False** for input values: 'f', 'false', 'n', 'no', '0'.

8.2.2 DateConvertor

class cooked_input.**DateConvertor**(*value_error_str='a date'*)
convert to a datetime value.

Parameters **value_error_str** (*str*) – (optional) the error string to use when an improper value is input

Returns

value converted to a datetime

Return type

datetime

Raises ***ConvertorError*** – if dateparser is unable to convert value to a datetime

Converts the cleaned input to an datetime value. Dateparser is used for the parsing, allowing a lot of flexibility in how date input is entered (e.g. '12/12/12', 'October 1, 2015', 'today', or 'next Tuesday'). For more information about dateparser see: https://dateparser.readthedocs.io/en/latest/

8.2.3 FloatConvertor

class cooked_input.**FloatConvertor**(*value_error_str='a float number'*)
convert to a floating point number.

Parameters **value_error_str** (*str*) – (optional) the error string to use when an improper value is input

Returns value converted to *float*

Return type float

Raises ***ConvertorError*** – if value cannot be converted to *float*

8.2.4 IntConvertor

class cooked_input.**IntConvertor**(*base=10*, *value_error_str='an integer number'*)

convert the cleaned input to an integer.

Parameters

- **base** (*int*) – the radix base to use for the int conversion (default=10)
- **value_error_str** (*str*) – (optional) the error string to use when an improper value is input

Returns value converted to *int*

Return type int

Raises ***ConvertorError*** – if value cannot be converted to *int*

Legal values for the *base* parameter are 0 and 2-36. See the Python int built-in function for more information.

8.2.5 ListConvertor

class cooked_input.**ListConvertor**(*elem_get_input=None*, *delimiter=','*, *value_error_str='list of values'*)

convert to a list.

Parameters

- **elem_get_input** (GetInput) – an instance of a *GetInput* to apply to each element. If None (default) each element in the list is a string
- **delimiter** (*str*) – (optional) the single character delimiter to use for parsing the list. If None, will sniff the value (ala CSV library.)
- **value_error_str** (*str*) – (optional) the error string for improper value inputs

Returns a *list* values, where each item in the list is of the type returned by elem_get_input

Return type List[Any] (element type of list determined by elem_get_input)

Raises ***ConvertorError*** – if elem_get_input's *GetInput.process_value()* fails

Converts to a homogenous list of values. The *GetInput.process_value()* method on the elem_get_input *GetInput* instance is called for each element in the list.

For example, the accept a list of integers separated by colons (':') and return it as a Python list of ints:

```
prompt_str = 'Enter a list of integers (separated by ":")'
lc = ListConvertor(delimiter=':', elem_get_
↪input=GetInput(convertor=IntConvertor()))
result = get_input(prompt=prompt_str, convertor=lc)
```

8.2.6 YesNoConvertor

class cooked_input.**YesNoConvertor**(*value_error_str='yes or no'*)

convert to 'yes' or 'no'.

Parameters **value_error_str** (*str*) – (optional) the error string to use when an improper value is input

Returns a string set to either **"yes"** or **"no"**

Return type str (**"yes"** or **"no"**)

Raises *ConvertorError* – if `value` cannot be converted to **"yes"** or **"no"**

YesNoConvertor returns *yes* for input values: 'y', 'yes', 'yeah', 'yup', 'aye', 'qui', 'si', 'ja', 'ken', 'hai', 'gee', 'da', 'tak', 'affirmative'. *YesNoConvertor* returns *no* for input values: 'n', 'no', 'nope', 'na', 'nae', 'non', 'negatory', 'nein', 'nie', 'nyet', 'lo'.

8.2.7 ChoiceConvertor

class cooked_input.**ChoiceConvertor**(*value_dict*, *value_error_str='a valid row number'*)

Convert a value to its mapped value in a dictionary.

Parameters

- **value_dict** (*Dict*) – a dictionary containing keys to map from and values to map to
- **value_error_str** (*str*) – (optional) the error string to use when an improper value is input

:raises ConvertorError if `value` key is not found in `value_dict`

Returns the *value* associated with the choice in `value_dict` (e.g. *value_dict[value]*)

Return type Any (type is dependent on mapped value in `value_dict`)

convert a value to it's return value in a dictionary (i.e. value_dict[value]). Can be used to map the row index from a table of values or to map multiple tags to a single choice.

For example, to use a number to pick from a list of colors:

```
value_map = {'1': 'red', '2': 'green', '3': 'blue'}
choice_convertor = ci.ChoiceConvertor(value_dict=value_map)
result = ci.get_input(convertor=choice_convertor, prompt='Pick a color (1 - red,␣
↪2 - green, 3 - blue)')
```

CHAPTER 9

Validators

The last step in cooked input is to validate that the entered input is valid. When called, Validators return True if the input passes the validation (i.e. is valid), and False otherwise.

9.1 Creating Validators

Validator classes inherit from the `Validator` base class. They must be a callable, and take three parameters: the `value` to validate, a function to call when an error occurs and a format string for the error function. See error callbacks for more information on error functions and their format strings.

An example of a validator to verify that the input is exactly a specified length looks like:

```
class LengthValidator(Validator):
    def __init__(self, min_len=None, max_len=None):
        self._min_len = min_len
        self._max_len = max_len

    def __call__(self, value, error_callback, validator_fmt_str):
        try:
            val_len = len(value)
        except (TypeError):
            print('LengthValidator: value "{}" does not support __len__.'.
↪format(value), file=sys.stderr)
            return False

        min_condition = (self._min_len is None or val_len >= self._min_len)
        max_condition = (self._max_len is None or val_len <= self._max_len)

        if min_condition and max_condition:
            return True
        elif not min_condition:
            error_callback(validator_fmt_str, value, 'too short (min_len={})'.
↪format(self._min_len))
```

(continues on next page)

(continued from previous page)

```
            return False
        else:
            error_callback(validator_fmt_str, value, 'too long (max_len={})'.
↪format(self._max_len))
            return False
```

Note: There are a large number of Boolean validation functions available from the validus project. These can be used as cooked_input validation functions by wrapping them in a `SimpleValidator`. For instance, to use `validus` to validate an email address:

```
from validus import isemail
email_validator = SimpleValidator(isemail, name='email')
email = get_input(prompt='enter a valid Email address', validators=email_validator)
```

9.2 Validators

9.2.1 AnyOfValidator

class cooked_input.**AnyOfValidator**(*validators*)

check if a value matches any of a set of validators (OR operation).

Parameters **validators** (*List[Validator]*) – a list of validators. Returns **True** once any of the validators passes.

Returns **True** if the input passed validation, else **False**

Return type boolean

Note: if `choices` is mutable, it can be changed after the instance is created.

Example:

```
rv1 = RangeValidator(min_val=1, max_val=3)
rv2 = EqualToValidator(7)
nv = AnyOfValidator([rv1, rv2])
prompt_str = "Enter a number (between 1 and 3, or 7)"
result = get_int(prompt=prompt_str, validators = nv)
```

9.2.2 ChoiceValidator

class cooked_input.**ChoiceValidator**(*choices*)

check if a value is in a set of choices.

Parameters **choices** (*Iterable*) – an iterable (tuple, list, or set) containing the allowed set of choices for the value.

Returns **True** if the input passed validation, else **False**

Return type boolean

Example:

```
colors = ["red", "green", "blue"]
cv = ChoiceValidator(colors]
result = get_string(prompt="Enter a color", validators=cv)
```

9.2.3 EqualToValidator

class cooked_input.**EqualToValidator**(*value*)

check if a value is equal to a specified value.

Parameters **value** (*Any*) – the value to match

Returns **True** if the input passed validation, else **False**

Return type boolean

9.2.4 IsFileValidator

class cooked_input.**IsFileValidator**

check is a string is the name of an existing filename

Parameters **value** (*str*) – the filename to verify

Returns **True** if the input passed validation, else **False**

Return type boolean

9.2.5 LengthValidator

class cooked_input.**LengthValidator**(*min_len=None*, *max_len=None*)

check the length of a value is in a range (open interval). For exact length match set `min_len` and `max_len` lengths to the same value.

Parameters

- **min_len** (*int*) – the minimum required length for the input. If **None** (default), no minimum length is checked.
- **max_len** (*int*) – the maximum required length for the input. If **None** (default), no maximum length is checked.

Returns **True** if the input passed validation

Return type boolean

Example:

```
lv = LengthValidator(min_len=3, max_len=5)
result = get_string(prompt="Enter a 3 to 5 char string", validators=lv)
```

9.2.6 ListValidator

class cooked_input.**ListValidator**(*len_validators=None*, *elem_validators=None*, *len_validator_fmt_str=None*)

Run a set of validators on a list.

Parameters

- **len_validators** (*List[Validator]*) – a list of validators to run on the length of the `value` list. if **None** (default) no validation is done on the list length.
- **elem_validators** (*List[Validator]*) – a list of validators to apply to the elements of the list.
- **len_validator_fmt_str** (*str*) – a format string to use as an error message is the length of the `value` string does not pass the length validation (`len_validators`). If **None** (default), `validator_fmt_str` is used from the call to the validator.

Returns **True** if the input passed validation, else **False**

Return type boolean

Note: `len_validators` is usually an instance of *EqualToValidator* for a list of a specific length, or *RangeValidator* for a list whose length is in a range. *LengthValidator* is not used as the `value` passed to the validator is the length of the list, not the list itself.

Example:

```
colors = ['red', 'green', 'blue']
len_validator = ci.RangeValidator(min_val=2, max_val=4)
lvfs="List length {error_content} ({value})"
elem_validator = ci.ChoiceValidator(colors)
prompt_str = "Enter a list of 2 to 4 colors"
lv = ci.ListValidator(len_validators=len_validator, elem_validators=elem_
↪validator, len_validator_fmt_str=lvfs)
result = ci.get_list(prompt=prompt_str, validators=lv)
```

9.2.7 NoneOfValidator

class cooked_input.**NoneOfValidator**(*validators*)

check if a value does not pass validation for a list of validators (NOT operation).

Parameters **validators** (*List[Validator]*) – a list of validators that should not pass validation on the input value

Returns **True** if the input passed validation, else **False**

Return type boolean

Note: if `choices` is mutable, it can be changed after the instance is created.

Example:

```
rv1 = RangeValidator(min_val=1, max_val=3)
rv2 = EqualToValidator(7)
nv = NoneOfValidator([rv1, rv2])
prompt_str = "Enter a number (not between 1 and 3, and not 7)"
result = get_int(prompt=prompt_str, validators = nv)
```

9.2.8 PasswordValidator

class cooked_input.**PasswordValidator**(*min_len=None*, *max_len=None*, *min_lower=0*, *min_upper=0*, *min_digits=0*, *min_puncts=0*, *allowed=None*, *disallowed=None*)

validate a password string.

Parameters

- **min_len** (*int*) – the minimum allowed password length (default=1)
- **max_len** (*int*) – the maximum password length (default=64)
- **min_lower** (*int*) – the minimum number of lower case letters (default=‘‘None‘‘)
- **min_upper** (*int*) – the minimum number of upper case letters (default=‘‘None‘‘)
- **min_digits** (*int*) – the minimum number of digits (default=‘‘None‘‘)
- **min_puncts** (*int*) – the minimum number of punctuation characters (default=‘‘None‘)
- **allowed** (*str*) – a string containing the allowed characters in the password. Default is upper and lower case ascii letters, plus digits, plus punctuation characters
- **disallowed** (*str*) – a string containing characters not allowed in the password (default=‘‘None‘‘)

Returns **True** if the input passed validation, else **False**

Return type boolean

Example:

```
pv = PasswordValidator(min_len=5, min_lower=2, min_upper=2, min_digits=1, min_
↪puncts=1)
result = ci.get_string(prompt="Enter a password", validators=pv, hidden=True)
```

9.2.9 RangeValidator

class cooked_input.**RangeValidator**(*min_val=None*, *max_val=None*)

check if a value is in a specified range (open interval.) The value can be of any type as long as the __ge__ and __le__ comparison functions are defined.

Parameters

- **min_val** (*Any*) – The minimum allowed value (i.e. `value` must be >= `min_val`). If **None** (the default), no minimum value is checked.
- **max_val** (*Any*) – The maximum allowed value (i.e. `value` must be <= max_val). If **None** (the default), no maximum value is checked.

Returns **True** if the input passed validation, else **False**

Return type boolean

Example:

```
rv = RangeValidator(min_val=1), max_val=10)
result = get_int(prompt="Enter a number (1 to 10)", validators=rv)
```

9.2.10 RegexValidator

class cooked_input.**RegexValidator**(*pattern*, *regex_desc=None*)

check if a value matches a regular expression.

Parameters

- **pattern** (*str*) – the regular expression to match
- **regex_desc** (*str*) – a human readable string to use for the regular expression (used for error messages)

Returns **True** if the input passed validation, else **False**

Return type boolean

Example:

```
rv = RegexValidator(pattern=r'^[2-9]\d{9}$', regex_desc='phone number')
result = get_string(prompt="Enter a phone number", validators = rv)
```

9.2.11 SimpleValidator

class cooked_input.**SimpleValidator**(*validator_func*, *name='SimpleValidator value'*)

use a simple function as a validator. `validator_func` is any callable that takes a single value as input and returns **True** if the value passes (and **False** otherwise.) Used to wrap functions (e.g. validus functions. Can also be used with func.partial to wrap validation functions that take more complex parameters.

Parameters

- **validator_func** (*Callable*) – a function (or other callable) called to validate the value
- **name** (*str*) – an optional string to use for the validator name in error messages

Returns **True** if the input passed validation, else **False**

Return type boolean

Example:

```
def is_even(value):
    return True if (value % 2) == 0 else False

sv = SimpleValidator(is_even, "EvenNumberValidator")
result = get_int(prompt="Enter an even number", validators = sv)
```

CHAPTER 10

GetInput

The *GetInput* class is the heart of the cooked_input library. Calls to *GetInput* objects perform the cleaning, conversion and validation of input data.

Note: Using the *GetInput* class is for more advanced users, Beginners can just use the convenience functions.

10.1 GetInput:

class cooked_input.**GetInput**(*cleaners=None*, *convertor=None*, *validators=None*, ***options*)

Class to get cleaned, converted, validated input from the command line. This is the central class used for cooked_input.

Parameters

- **cleaners** (*List[Cleaner]*) – list of cleaners to apply to clean the value
- **convertor** (*Convertor*) – the convertor to apply to the cleaned value
- **validators** (*List[Validator]*) – list of validators to apply to validate the cleaned and converted value
- **options** – see below

Options:

prompt: the string to use for the prompt. For example prompt="Enter your name"

required: **True** if a non-blank value is required, **False** if a blank response is OK.

default: the default value to use if a blank string is entered. This takes precedence over required (i.e. a blank response will return the default value.)

default_str: the string to use for the default value. In general just set the default option.

hidden: the input typed should not be displayed. This is useful for entering passwords.

retries: the maximum number of attempts to allow before raising a *MaxRetriesError* exception.

error_callback: a callback function to call when an error is encountered. Defaults to *print_error()*

convertor_error_fmt: format string to use for convertor errors. Defaults to DEFAULT_CONVERTOR_ERROR. Format string receives two variables - **{value}** the value that failed conversion, and **{error_content}** set by the convertor.

validator_error_fmt: format string to use for validator errors. Defaults to DEFAULT_VALIDATOR_ERROR. Format string receives two variables - **{value}** the value that failed conversion, and **{error_content}** set by the validator.

commands: an optional dictionary of commands. See below for more details.

Commands:

GetInput optionally takes a dictionary containing commands that can be run from the input prompt. The key for each command is the string used to call the command and the value is an instance of the *GetInputCommand* class for the command. For intance, the following dictionary sets two different command string (**/?** and **/help**) to call a function to show help information:

```
{
    "/?": GetInputCommand(show_help_action)),
    "/help": GetInputCommand(show_help_action)),
}
```

For more information see *GetInputCommand*

GetInput.**get_input**()

Get input from the command line and return a validated response.

Returns the cleaned, converted, validated input

Return type Any (dependent on the value returned from the *convertors*)

This method prompts the user for an input, and returns the cleaned, converted, and validated input.

GetInput.**process_value**(*value*)

Parameters **value** (*str*) – the value to process

Returns Return a **ProcessValueResponse** namedtuple (valid, converted_value)

Return type NamedTuple[bool, Any]

Run a value through cleaning, conversion, and validation. This allows the same processing used in *GetInput.get_input()* to be performed on a value. For instance, the same processing used for getting keyboard input can be applied to the value from a gui or web form input.

The **ProcessValueResponse** namedtuple has elements **valid** and **value**. If the value was successfully cleaned, converted and validated, **valid** is True and **value** is the converted and cleaned value. If not, **valid** is **False**, and **value** is **None**.

CHAPTER 11

Tables and Menus

The *TableItem* and *Table* support text-based tables and menus in *cooked_input*.

Note: Using the *Table* class is for more advanced users, Beginners can just use the *get_menu()* and *get_table_input()* convenience functions.

11.1 TableStyle:

class cooked_input.**TableStyle**(*show_cols=True*, *show_border=True*, *hrules=0*, *vrules=1*, *rows_per_page=20*)

TableStyle is used to define the visual style of a Cooked_Input table. *Table* objects take an instance of TableStyle as the style parameter.

Parameters

- **show_cols** (*Bool*) – if **True** (default) shows a the column names at the top of the table
- **show_border** (*Bool*) – if **True** (default) shows a border around the table
- **hrules** – whether to draw horizontal lines between rows. See below for allowed RULE values.
- **vrules** – whether to draw vertical lines between rows. See below for allowed RULE values.
- **rows_per_page** (*int*) – The maximum number of rows to display in the table. Used for paginated tables (None for no maximum).

hrules and vrules can use the following RULE values for the rows and columns respectively:

value	action
RULE_FRAME RULE_HEADER RULE_ALL RULE_NONE	Draw ruled lines around the outside (frame) of the table. Draw ruled lines around the header of the table. Draw ruled line around the table, header and between columns/rows. Do not draw any rules lines around columns/rows.

11.2 TableItem:

class cooked_input.**TableItem**(*col_values*, *tag=None*, *action='default'*, *item_data=None*, *hidden=False*, *enabled=True*)

TableItem is used to represent individual rows in a table. This is also often used for menu items.

Parameters

- **col_values** (*List*) – A list of values for the row's columns.
- **tag** – a value used to choose the item. If None, a default tag will be assigned by the *Table*.
- **action** (*Callable*) – an action function called when the item is selected.
- **item_data** (*Dict*) – a dictionary containing addtional contextual data for the table row. This is not displayed as part of the table item but can be used for processing actions. For example, item_data can store the database ID associated for the item. item_data is also used for item filters.
- **hidden** (*bool*) – The table row is hidden if **True**, or visible if **False** (default). Hidden table items are still selectable, unless the enabled attribute is **False**.
- **enabled** (*bool*) – The table row is selectable if **True **(default), and not selectable if **False**.

TableItem actions:

The table item action specifies what to do when a table item is selected. The action can be one of the default actions listed in the following table or a custom action can be provided:

value	action
TABLE_RETURN_TAG	return the selected row's tag.
TABLE_RETURN_FIRST_VAL	return the first data column value of the selected row.
TABLE_RETURN_ROW	return the list of column values for the selected row.
TABLE_RETURN_TABLE_ITEM	return the TableItem instance for the selected row.
TABLE_ITEM_DEFAULT	use default method to handle the table item (e.g. use the parent table's default_action handler function)
TABLE_ITEM_EXIT	selecting the table row should exit (used to exit a menu)
TABLE_ITEM__RETURN	selecting the table row should return (used to return from a submenu)

In addition to the values in the table above, the action can be any callable that takes the following parameters:

- **row** (TableItem) – The TableItem instance selected (i.e. this table item)

- **action_dict** (Dict) – The parent table's `action_dict`.

11.3 Table:

class cooked_input.**Table**(*rows*, *col_names=None*, *title=None*, *prompt=None*, *default_choice=None*, *default_str=None*, *default_action=None*, *style=None*, ***options*)

The Table class is used to display a table of data. Each row of data has the same number of columns (specified by the `col_name` parameter) as is represented by a *TableItem* instance. Tables are often used for menus.

Parameters

- **rows** (*List*) – The rows of the table. Each row is a *TableItem* instance.
- **col_names** (*List*) – An optional list of the column names (strings) for the table. If no list is given the number of columns is determined by the length of the data list for the first row (*TableItem*).
- **title** (*str*) – An optional title for the table.
- **prompt** (*str*) – The prompt for choosing a table value.
- **default_choice** (*str*) – An optional default tag value to use for the table selection.
- **default_str** (*str*) – An optional string to display for the default table selection.
- **default_action** (*Callable*) – The default action function to call a table item is selected. See below for details.
- **style** (TableStyle) – a *TableStyle* defining the look of the table.
- **options** (*Dict*) – see below for a list of valid options

Options:

required: requires an entry if **True**, exits the table on blank entry if **False**.

tag_str: string to use for the tag column name. Defaults to an empty string ("").

add_exit: automatically adds a *TableItem* to exit the table menu (TABLE_ITEM_EXIT) or return to the parent table/menu (TABLE_ADD_RETURN), or not to add a *TableItem* at all (**False**). Used to exit menus or return from sub-menus.

action_dict: a dictionary of values to pass to action functions. Used to provide context to the action. Helpful to provide items such as data base sessions, user credentials, etc.

case_sensitive: whether choosing table items should be case sensitive (**True**) or not (**False** - default)

commands: a dictionary of commands for the table. For each entry, the key is the command and the value the action to take for the command. See *GetInput* and *GetInputCommand* for further details

item_filter: a function used to determine which table items to display. Displays all items if None. See below for more details.

refresh: refresh table items each time the table is shown (True - default), or just when created (**False**). Useful for dynamic tables

header: a format string to print before the table, can use any value from action_dict as well as pagination information

footer: a format string to print after the table, can use any values from action_dict as well as pagination information

Table default actions:

Each table has a default action to take when an item is selected. The action can be a callable or a value from the table below. The Table's default action is called if the If the selected row (TableItem) has its action set to TABLE_DEFAULT_ACTION, otherwise the action for the selected TableItem is called. Standard values for the Table default action are:

value	action
TABLE_RETURN_TAG	return the selected row's tag.
TABLE_RETURN_FIRST_VAL	return the first data column value of the selected row.
TABLE_RETURN_ROW	return the list of column values for the selected row.
TABLE_RETURN_TABLE_ITEM	return the TableItem instance for the selected row.

In addition to the values in the table above, the action can be any callable that takes the following parameters:

- **row** (TableItem) – The *TableItem* instance selected (i.e. this table item)
- **action_dict** (Dict) – The parent table's action_dict.

For example:

```
def reverse_tag_action(row, action_dict):
    if action_dict['reverse'] is True:
        return row.tag[::-1]
    else
        return row.tag
```

item filters:

The item filter option provides a function that determines which table items are hidden and/or enabled in the table. It is a callable that takes the following input parameters:

- **item** (TableItem) – the *TableItem* instance
- **action_dict** (Dict) – the action_dict for the *Table*

The `item_filter` function returns a tuple of two Booleans `(hidden, enabled)` for the item (`TableItem`).

For example, a menu can have choices that are visible and enabled only for user's who are part of the administrator group:

```
def user_role_filter(row, action_dict):
    if row.item_data is None:
        return  (False, True)   # visible and enabled

    for role in action_dict['user_roles'].roles:
        if role in row.item_data['roles']:
            return  (False, True)   # visible and enabled

    return (True, False)    # hidden and disabled

action_dict = {'user_roles': ['admin', 'users']}
admin_only = {'roles': {'admin'} }

rows = [
    TableItem('Add a new user', action=user_add_action, item_data=admin_
↪only)
```

(continues on next page)

(continued from previous page)

```
    TableItem('list users', action=user_list_action) ]
menu = Table(menu_items=rows, action_dict=action_dict, item_filter=user_
↪role_filter)
```

Table.**show_table**()
Show the table without asking for input.

Returns None

Table.**get_table_choice**(***options*)
Prompts the user to choose a value from the table. This is the main method used to choose a value from a table.

Parameters **options** – See below for details.

Returns the result of performing the action (specified by the table or row) on the row. Returns **None** if no row is selected.

Options:

- **prompt** (str) – the prompt for choosing a table value.
- **required** (bool) – requires an entry if **True**, exits the table on blank entry if **False**.
- **default** (str) – the default value to use.
- **default_str** (str) – An optional string to display for the default table selection.
- **commands (Dict) – a dictionary of commands for the table. For each entry, the key is the** command and the value the action to take for the command. See *GetInput* and *GetInputCommand* for further details

Table.**run**()

Continue to get input from the table until a blank row (None) is returned, or a *GetInputInterrupt* exception is raised. This is primarily used to use tables as menus. Choosing exit or return in a menu is the same as returning no row.

Returns **True** if exited without an error, or **False** if a *GetInputInterrupt* exce[tion was raised

Table.**get_num_rows**()
Get the number of rows in the table.

Returns the number of rows in the table

Return type int

Table.**get_row**(*tag*)
Get the number of rows in the table.

Returns the number of rows in the table

Return type *TableItem*

Table.**get_action**(*tag*)
Return the action callback function for the first row matching the specified tag.

Parameters **tag** – the tag to search for

Returns the action for the first row containing the tag. Raises a **ValueError** exception if the tag is not found

Return type Callable

Table.**do_action**(*row*)
Call the action function for the specified row

Parameters **row** (TableItem) – the table row to call the action on

Returns returns the return value for the action. Returns the original row if no action is defined for the row.

The action function is called with the following parameters:

- **row** – The *TableItem* instance selected (i.e. this table item)

Table.**show_rows**(*start_row*)
Set the starting row for to display in the table. Last row shown is the start_row plus the number of rows per page (or the last row if start_row is within rows_per_page of the end of the table).

Parameters **start_row** (*int*) – the first row of the table to show

Returns None

Table.**page_up**()
Display the previous page of the table (if available)

Returns None

Table.**page_down**()
Display the next page of the table (if available)

Returns None

Table.**goto_home**()
Display the first page of the table

Returns None

Table.**goto_end**()
Display the last page of the table

Returns None

Table.**scroll_up_one_row**()
Display the one row earlier in the table (if available)

Returns None

Table.**scroll_down_one_row**()
Display the one row later in the table (if available)

Returns None

Table.**refresh_screen**()
Display the current page of the table (including any header or footer)

Returns None

Table.**refresh_items**(*rows=None*, *add_exit=False*, *item_filter=None*)
Refresh which rows of the table are enabled and shown. Used to update rows in the table. Adds tags if necessary. formatter is used so values can be substituted in format strings from action_dict using vformat. This is useful in case some TableItems have dynamic data. Can also be used by action to change table items. For instance a search action might filter for row entries using an item filter.

Parameters

- **rows** (*List*) – a list of rows to show. If **None**, will use all rows.

- **add_exit** (*bool*) – if **TABLE_ADD_EXIT** add an entry to exit, if **TABLE_ADD_RETURN** add an entry to return. Don't add an entry if **False** (default).
- **item_filter** (*Callable*) – an optional function used to filter rows. See *Table* for details regarding item filters.

Returns None

11.4 Table Action Functions:

The following pre-defined action functions can be used for the action for *Table* and *TableItem*:

11.4.1 return_table_item_action

cooked_input.**return_table_item_action**(*row*, *action_dict*)

Action function for Tables. This function returns the TableItem instance. Used by the **TABLE_RETURN_TABLE_ITEM** action.

Parameters

- **row** (*List*) – the data associated with the selected row
- **action_dict** (*Dict*) – the dictionary of values associated with the action - ignored in this function

Returns A list containing all of the data for the selected row of the table.

Return type List

11.4.2 return_row_action

cooked_input.**return_row_action**(*row*, *action_dict*)

Default action function for Tables. This function returns the whole row of data including the tag. Used by the **TABLE_RETURN_ROW** action.

Parameters

- **row** (*List*) – the data associated with the selected row
- **action_dict** (*Dict*) – the dictionary of values associated with the action - ignored in this function

Returns A list containing all of the data values for the selected row of the table.

Return type List

11.4.3 return_tag_action

cooked_input.**return_tag_action**(*row*, *action_dict*)

Default action function for tables. This function returns the tag for the row of data. Used by the **TABLE_RETURN_TAG** action.

Parameters

- **row** (*List*) – the data associated with the selected row

- **action_dict** (*Dict*) – the dictionary of values associated with the action - ignored in this function

Returns The tag for the selected row of the table.

11.4.4 return_first_col_action

cooked_input.**return_first_col_action**(*row*, *action_dict*)

Default action function for tables. This function returns the first data column value for the row of data. Used by the **TABLE_RETURN_FIRST_VAL** action.

Parameters

- **row** (*List*) – the data associated with the selected row
- **action_dict** (*Dict*) – the dictionary of values associated with the action - ignored in this function

Returns The first value from the list of data values for the selected row of the table.

CHAPTER 12

Exceptions

`cooked_input` defines a number of custom exceptions. These are mainly used for tables and commands.

`cooked_input` exceptions are generally only used for commands. See *GetInputCommand* for more information on using commands.

12.1 ConvertorError:

class cooked_input.**ConvertorError**
raised by a when a value does not pass conversion.

12.2 MaxRetriesError:

class cooked_input.**MaxRetriesError**
raised when the maximum number of retries is exceeded.

12.3 ValidationError:

class cooked_input.**ValidationError**
raised when a value does not pass validation.

12.4 GetInputInterrupt:

class cooked_input.**GetInputInterrupt**
A cancellation command (COMMAND_ACTION_CANCEL) occurred.

12.5 RefreshScreenInterrupt:

class cooked_input.**RefreshScreenInterrupt**
When raised, directs cooked_input to refresh the display. Used primarily to refresh table items.

12.6 PageUpRequest:

class cooked_input.**PageUpRequest**
When raised, directs cooked_input to go to the previous page in paginated tables

12.7 PageDownRequest:

class cooked_input.**PageDownRequest**
When raised, directs cooked_input to go to the next page in paginated tables

12.8 FirstPageRequest:

class cooked_input.**FirstPageRequest**
When raised, directs cooked_input to go to the first page in paginated tables

12.9 LastPageRequest:

class cooked_input.**LastPageRequest**
When raised, directs cooked_input to go to the last page in paginated tables

12.10 UpOneRowRequest:

class cooked_input.**UpOneRowRequest**
When raised, directs cooked_input to scroll up one row in paginated tables

12.11 DownOneRowRequest:

class cooked_input.**DownOneRowRequest**
When raised, directs cooked_input to scroll down one row in paginated tables

CHAPTER 13

error_callbacks

13.1 Creating Error functions:

Error functions are used by cooked_input to report errors from convertors and validators. Error functions take three parameters:

- `fmt_str`: a Python format string for the error. The format string can use the arguments **{value}** and **{error_content}**.
- `value`: the value that caused the error from the convertor or validator.
- `error_content`: the particulars of the error message from the convertor or validator.

The following example prints errors to sys.stdout:

```
def print_error(fmt_str, value, error_content):
    print(fmt_str.format(value=value, error_content=error_content))
```

An example of a convertor format string is as follows:

```
generic_convertor_fmt = '{value} cannot be converted to {error_content}'
```

and similarly for validation:

```
generic_validator_fmt = '{value} {error_content}'
```

13.2 error_callbacks:

13.2.1 log_error

cooked_input.**log_error**(*fmt_str*, *value*, *error_content*)
send errors to the log. See logging for details on using logs.

Parameters

- **fmt_str** (*str*) – a Python format string for the error. Can use arguments **{value}** and **{error_content}** in the format string
- **value** (*Any*) – the value the caused the error.
- **error_content** (*str*) – additional information for the error

Returns None

13.2.2 print_error

cooked_input.**print_error**(*fmt_str*, *value*, *error_content*)

send errors to stdout. This displays errors on the screen.

Parameters

- **fmt_str** (*int*) – a Python format string for the error. Can use arguments **{value}** and **{error_content}** in the format string
- **value** (*Any*) – the value the caused the error
- **error_content** (*str*) – additional information for the error

Returns None

13.2.3 silent_error

cooked_input.**silent_error**(*fmt_str*, *value*, *error_content*)

Ignores errors, causing them to be silent.

Parameters

- **fmt_str** (*str*) – a Python format string for the error. Can use arguments **{value}** and **{error_content}** in the format string
- **value** (*Any*) – the value the caused the error
- **error_content** (*str*) – additional information for the error

Returns None

CHAPTER 14

Commands

The *GetInputCommand* class is used to define commands that can be run while getting command line input in `cooked_input`.

14.1 GetInputCommand:

class cooked_input.**GetInputCommand**(*cmd_action*, *cmd_dict=None*)

GetInputCommand is used to create commands that can be used while getting input from *GetInput.get_input()*

param Callable[str, str, Dict[str, Any], Tuple[str, Any]] cmd_action callback function used to process the command

param Dict[Any, Any] cmd_dict (optional) a dictionary of data passed to the `cmd_action` callback function

Each command has a callback function (`cmd_action`) and optional data (`cmd_dict`).

`cmd_action` is a callback function used for the command. The callback is called as follows

cmd_action(*cmd_str*, *cmd_vars*, *cmd_dict*)

Parameters

- **cmd_str** (*str*) – the string used to call the command
- **cmd_vars** (*str*) – additional arguments for the command (i.e. the rest of string used for the command input)
- **Any] cmd_dict** (*Dict[str,*) – a dictionary of additional data for processing the command (often **None**)

Command callback functions return a a tuple containing (*COMMAND_ACTION_TYPE*, value), where the command action type is one of the following:

Action	Result
COMMAND_ACTION_USE_VALUE	use the second value of the tuple as the input
COMMAND_ACTION_CANCEL	cancel the current input (raises *GetInputInterrupt* exception)
COMMAND_ACTION_NOP	do nothing - continues to ask for the input

For convenience command action callbacks can return a `CommandResponse` namedtuple instance:

```
CommandResponse(action, value)
```

The `cmd_dict` dictionary can be used to pass data useful in processing the command. For instance, a database session and the name of the user can be passed with:

```
cmd_dict = {'session': db_session, 'user_name': user_name }
lookup_user_cmd = GetInputCommand(lookup_user_action, cmd_dict)
```

The following show examples of of each type of command:

```
def use_color_action(cmd_str, cmd_vars, cmd_dict):
    print('Use "red" as the input value')
    return (COMMAND_ACTION_USE_VALUE, 'red')

def cancel_action(cmd_str, cmd_vars, cmd_dict):
    return CommandResponse(COMMAND_ACTION_CANCEL, None)

def show_help_action(cmd_str, cmd_vars, cmd_dict):
    print('Commands:')
    print('---------')
    print('/?  - show this message')
    print('/cancel - cancel this operation')
    print('/red    - use red as a value')
    print('/reverse - return the user's name reversed')
    return CommandResponse(COMMAND_ACTION_NOP, None)

cmds = { '/?': GetInputCommand(show_help_action),
         '/cancel': GetInputCommand(cancel_action),
         '/red': GetInputCommand(use_color_action, {'color': 'red'}),
         '/reverse': GetInputCommand(user_color_action, {'user': 'fred'})␣
↪}

try:
    result = get_string(prompt=prompt_str, commands=cmds)
except GetInputInterrupt:
    print('Operation cancelled (received GetInputInterrupt)')
```

Note: Nothing stops you from using `cooked_input` to get additional input within a command action callback. For example, the cancel command could be extended to confirm the user wants to cancel the current input:

```
def cancel_action(cmd_str, cmd_vars, cmd_dict):
    response = get_yes_no(prompt="Are you sure you want to cancel?", default='no')

    if response == 'yes':
        print('operation cancelled!')
        return CommandResponse(COMMAND_ACTION_CANCEL, None)
```

(continues on next page)

(continued from previous page)

```
    else:
        return CommandResponse(COMMAND_ACTION_NOP, None)
```

14.2 Command Action Functions:

The following pre-defined action functions can be used for commands (see *GetInputCommand*).

14.2.1 first_page_cmd_action

cooked_input.**first_page_cmd_action**(*cmd_str*, *cmd_vars*, *cmd_dict*)

Command action to show the first (home) page in a paginated table. This command raises a *FirstPageRequest* exception.

Parameters

- **cmd_str** – ignored
- **cmd_vars** – ignored
- **cmd_dict** – ignored

Returns None

14.2.2 last_page_cmd_action

cooked_input.**last_page_cmd_action**(*cmd_str*, *cmd_vars*, *cmd_dict*)

Command action to show the last (end) page in a paginated table. This command raises a *LastPageRequest* exception.

Parameters

- **cmd_str** – ignored
- **cmd_vars** – ignored
- **cmd_dict** – ignored

Returns None

14.2.3 prev_page_cmd_action

cooked_input.**prev_page_cmd_action**(*cmd_str*, *cmd_vars*, *cmd_dict*)

Command action to show the previous page in a paginated table. This command raises a *PageUpRequest* exception.

Parameters

- **cmd_str** – ignored
- **cmd_vars** – ignored
- **cmd_dict** – ignored

Returns None

14.2.4 next_page_cmd_action

cooked_input.**next_page_cmd_action**(*cmd_str*, *cmd_vars*, *cmd_dict*)

Command action to show the next page in a paginated table. This command raises a *PageDownRequest* exception.

Parameters

- **cmd_str** – ignored
- **cmd_vars** – ignored
- **cmd_dict** – ignored

Returns None

14.2.5 scroll_up_one_row_cmd_action

cooked_input.**scroll_up_one_row_cmd_action**(*cmd_str*, *cmd_vars*, *cmd_dict*)

Command action to scroll up one row in a paginated table. This command raises a *UpOneRowRequest* exception.

Parameters

- **cmd_str** – ignored
- **cmd_vars** – ignored
- **cmd_dict** – ignored

Returns None

14.2.6 scroll_down_one_row_cmd_action

cooked_input.**scroll_down_one_row_cmd_action**(*cmd_str*, *cmd_vars*, *cmd_dict*)

Command action to scroll down one row in a paginated table. This command raises a *DownOneRowRequest* exception.

Parameters

- **cmd_str** – ignored
- **cmd_vars** – ignored
- **cmd_dict** – ignored

Returns None

CHAPTER 15

Indices and tables

- genindex
- modindex
- search

Python Module Index

C

cleaners, 37
convertors, 41

V

validators, 45

Index

A

AnyOfValidator (*class in cooked_input*), 46

B

BooleanConvertor (*class in cooked_input*), 42

C

CapitalizationCleaner (*class in cooked_input*), 37
ChoiceCleaner (*class in cooked_input*), 38
ChoiceConvertor (*class in cooked_input*), 44
ChoiceValidator (*class in cooked_input*), 46
cleaners (*module*), 37
ConvertorError (*class in cooked_input*), 61
convertors (*module*), 41
create_rows() (*in module cooked_input*), 32
create_table() (*in module cooked_input*), 34

D

DateConvertor (*class in cooked_input*), 42
do_action() (*cooked_input.Table method*), 57
DownOneRowRequest (*class in cooked_input*), 62

E

EqualToValidator (*class in cooked_input*), 47

F

first_page_cmd_action() (*in module cooked_input*), 67
FirstPageRequest (*class in cooked_input*), 62
FloatConvertor (*class in cooked_input*), 42

G

get_action() (*cooked_input.Table method*), 57
get_boolean() (*in module cooked_input*), 29
get_date() (*in module cooked_input*), 29
get_float() (*in module cooked_input*), 28
get_input() (*cooked_input.GetInput method*), 52
get_input() (*in module cooked_input*), 30
get_int() (*in module cooked_input*), 28
get_list() (*in module cooked_input*), 30
get_menu() (*in module cooked_input*), 32
get_num_rows() (*cooked_input.Table method*), 57
get_row() (*cooked_input.Table method*), 57
get_string() (*in module cooked_input*), 27
get_table_choice() (*cooked_input.Table method*), 57
get_table_input() (*in module cooked_input*), 35
get_yes_no() (*in module cooked_input*), 29
GetInput (*class in cooked_input*), 51
GetInputCommand (*class in cooked_input*), 65
GetInputCommand.cmd_action() (*in module cooked_input*), 65
GetInputInterrupt (*class in cooked_input*), 61
goto_end() (*cooked_input.Table method*), 58
goto_home() (*cooked_input.Table method*), 58

I

IntConvertor (*class in cooked_input*), 43
IsFileValidator (*class in cooked_input*), 47

L

last_page_cmd_action() (*in module cooked_input*), 67
LastPageRequest (*class in cooked_input*), 62
LengthValidator (*class in cooked_input*), 47
ListConvertor (*class in cooked_input*), 43
ListValidator (*class in cooked_input*), 47
log_error() (*in module cooked_input*), 63

M

MaxRetriesError (*class in cooked_input*), 61

N

next_page_cmd_action() (*in module cooked_input*), 68
NoneOfValidator (*class in cooked_input*), 48

P

page_down() (*cooked_input.Table method*), 58
page_up() (*cooked_input.Table method*), 58
PageDownRequest (*class in cooked_input*), 62
PageUpRequest (*class in cooked_input*), 62
PasswordValidator (*class in cooked_input*), 49
prev_page_cmd_action() (*in module cooked_input*), 67
print_error() (*in module cooked_input*), 64
process_value() (*cooked_input.GetInput method*), 52
process_value() (*in module cooked_input*), 31

R

RangeValidator (*class in cooked_input*), 49
refresh_items() (*cooked_input.Table method*), 58
refresh_screen() (*cooked_input.Table method*), 58
RefreshScreenInterrupt (*class in cooked_input*), 62
RegexCleaner (*class in cooked_input*), 38
RegexValidator (*class in cooked_input*), 50
RemoveCleaner (*class in cooked_input*), 39
ReplaceCleaner (*class in cooked_input*), 39
return_first_col_action() (*in module cooked_input*), 60
return_row_action() (*in module cooked_input*), 59
return_table_item_action() (*in module cooked_input*), 59
return_tag_action() (*in module cooked_input*), 59
run() (*cooked_input.Table method*), 57

S

scroll_down_one_row() (*cooked_input.Table method*), 58
scroll_down_one_row_cmd_action() (*in module cooked_input*), 68
scroll_up_one_row() (*cooked_input.Table method*), 58
scroll_up_one_row_cmd_action() (*in module cooked_input*), 68
show_rows() (*cooked_input.Table method*), 58
show_table() (*cooked_input.Table method*), 57
show_table() (*in module cooked_input*), 35
silent_error() (*in module cooked_input*), 64
SimpleValidator (*class in cooked_input*), 50
StripCleaner (*class in cooked_input*), 40

T

Table (*class in cooked_input*), 55
TableItem (*class in cooked_input*), 54
TableStyle (*class in cooked_input*), 53

U

UpOneRowRequest (*class in cooked_input*), 62

V

validate() (*in module cooked_input*), 31
ValidationError (*class in cooked_input*), 61
validators (*module*), 45

Y

YesNoConvertor (*class in cooked_input*), 43