Filters Documentation

Phoenix Zerin

Feb 16, 2023

Contents

1 Getting Started 1

2 Chaining Filters 3

3 Much Ado About None 5

4 Next Steps 7

5 Writing Your Own Filters 9

6 Extending the Filters Namespace 13

7 Trade-Offs 15

8 Prerequisites 17

9 Registering Your Filters 19

10 Simple Filters 21

11 Complex Filters 51

12 Filterception 57

13 Official Extensions 59

14 Filters 63

CHAPTER 1

Getting Started

The fastest way to get started with filters is to use the `FilterRunner` class. This class provides an interface very similar to a Django form.

```
import datetime
import filters as f

# Incoming data.
data = u'1879-03-14'

# Initialize the FilterRunner.
runner = f.FilterRunner(f.Date, data)

if runner.is_valid():
  # Input is valid; do something with the filtered data.
  cleaned_data = runner.cleaned_data
  assert cleaned_data == datetime.date(1879, 3, 14)

else:
  # Input is not valid; display error message(s) for each incoming value.
  for key, errors in runner.errors.items():
    print('{key}:'.format(key=key))
    for error in errors:
      print('  - ({error[code]}) {error[message]}'.format(error=error))
```

`FilterRunner` provides a few key attributes to make it easy to apply filters:

- `is_valid()`: Returns whether the value is valid.
- `cleaned_data`: If the value is valid, this property holds the filtered value(s).
- `errors`: If the value is not valid, this property holds the validation errors.

1.1 Reusing FilterRunners

If you want to run the same set of filters on different values, you can create a single `FilterRunner` instance and call its `apply()` method:

```
import filters as f

runner = f.FilterRunner(f.Choice({'foo', 'bar', 'baz', 'luhrmann'}))

input_1 = 'foo'
input_2 = 'foobie'

runner.apply(input_1)
assert runner.is_valid() is True
assert runner.cleaned_data == 'foo'

runner.apply(input_2)
assert runner.is_valid() is False
assert runner.cleaned_data is None
```

CHAPTER 2

Chaining Filters

The filters library conforms to the unix philosophy of, "Do One Thing, and Do It Well".

Each filter provides a specific transformation and/or validation feature. This alone can be useful, but the real power of the filters library lies in its ability to "chain" filters together.

By using the `|` operator, you can "pipe" the output of one filter directly into the input of another. This allows you to quickly and easily create complex data pipelines.

Here's an example:

```
import filters as f

# Convert to unicode, strip leading and trailing whitespace, reject empty
# string, fold case and split into words.
filter_ = f.Unicode | f.Strip | f.NotEmpty | f.CaseFold | f.Split(r'\W+')

runner = f.FilterRunner(filter_, '       ')
assert runner.is_valid() is True
assert runner.cleaned_data == ['', '']

runner = f.FilterRunner(filter_, '\r\n')
assert runner.is_valid() is False
```

CHAPTER 3

Much Ado About None

`None` is a special value to the Filters library. By default, it passes every filter, no matter how strictly configured.

For example:

```
import filters as f

# Convert to unicode, strip leading and trailing whitespace, reject empty
# string, fold case and split into words.
filter_ = f.Unicode | f.Strip | f.NotEmpty | f.CaseFold | f.Split(r'\W+')

runner = f.FilterRunner(filter_, None)
assert runner.is_valid() is True
assert runner.cleaned_data is None
```

If you want to reject `None`, add the `Required` filter to your chain:

```
import filters as f

# Note that we replace ``NotEmpty`` with ``Required``.
filter_ = f.Unicode | f.Strip | f.Required | f.CaseFold | f.Split(r'\W+')

runner = f.FilterRunner(filter_, None)

assert runner.is_valid() is False
```

CHAPTER 4

Next Steps

See *Simple Filters* for a list of all the filters that come bundled with the Filters library.

Be sure to pay special attention to *Complex Filters*, which lists filters designed exclusively to work with other filters, allowing you to construct powerful data schemas and transformation pipelines.

There are also several *Official Extensions* that you can install, to add even more filters to work with.

Once you've gotten the hang of working with filters, you'll want to *write your own filters and macros*, so that you can reduce code duplication and inject your own functionality into filter pipelines.

CHAPTER 5

Writing Your Own Filters

Although the Filters library comes with *lots of built-in filters*, oftentimes it is useful to be able to write your own.

There are three ways that you can create new filters:

- Macros
- Partials
- Custom Filters

5.1 Macros

If you find yourself using a particular filter chain over and over, you can create a macro to save yourself some typing.

To create a macro, define a function that returns a filter chain, then decorate it with the `filters.filter_macro` decorator:

```
import filters as f

@f.filter_macro
def String(allowed_types=None):
  return f.Type(allowed_types or str) | f.Unicode | f.Strip
```

You can now use your filter macro just like any other filter:

```
runner = f.FilterRunner(String | f.Required, '   Hello, world!    ')
assert runner.is_valid() is True
assert runner.cleaned_data == 'Hello, world!'
```

5.2 Partials

A partial is a special kind of macro. Instead of returning a filter chain, it returns a single filter, but with different configuration values.

Here's an example of a partial that can be used to validate datetimes from New Zealand, convert to UTC, and strip `tzinfo` from the result:

```
import filters as f

# Create a partial for ``f.Datetime(timezone=13, naive=True)``.
NZ_Datetime = f.filter_macro(f.Datetime, timezone=13, naive=True)
```

Just like with macros, you can use a partial anywhere you can use a regular filter:

```
from datetime import datetime

runner = f.FilterRunner(NZ_Datetime | f.Required, '2016-12-11 15:00:00')
assert runner.is_valid() is True
assert runner.cleaned_data == datetime(2016, 12, 11, 2, 0, 0, tzinfo=None)
```

Additionally, partials act just like `functools.partial()` objects; you can invoke them with different parameters if you want:

```
from pytz import utc

# Override the ``naive`` parameter for the ``NZ_Datetime`` partial.
filter_ = NZ_Datetime(naive=False) | f.Required

runner = f.FilterRunner(filter_, '2016-12-11 15:00:00')
assert runner.is_valid() is True
assert runner.cleaned_data == datetime(2016, 12, 11, 2, 0, 0, tzinfo=utc)
```

5.3 Custom Filters

Sometimes you just can't get what you want by assembling existing filters, and you need to write your own.

To create a new filter, write a class that extends `filters.BaseFilter` and implement the `_apply` method:

```
import filters as f

class Pkcs7Pad(f.BaseFilter):
  block_size = 16

  def _apply(self, value):
     extra_bytes = self.block_size - (len(value) % self.block_size)
     return value + bytes([extra_bytes] * extra_bytes)
```

5.3.1 Validation

To implement validation in your filter, add the following:

- Define a unique code for each validation error.
- Define an error message template for each validation error.

- Add the logic to the filter's `_apply` method.

Here's the `Pkcs7Pad` filter with a little bit of validation logic:

```
import filters as f

class Pkcs7Pad(f.BaseFilter):
  CODE_INVALID_TYPE = 'invalid_type'

  templates = {
    CODE_INVALID_TYPE = 'Binary string required.',
  }

  block_size = 16

  def _apply(self, value):
     if not isinstance(value, bytes):
       return self._invalid_value(value, self.CODE_INVALID_TYPE)

     extra_bytes = self.block_size - (len(value) % self.block_size)
     return value + bytes([extra_bytes] * extra_bytes)
```

5.3.2 Invoking Other Filters

You can also invoke other filters in your custom filters by calling the `self._filter` method.

For example, we can simplify the implementation of `Pkcs7Pad` by incorporating the `filters.ByteString` filter:

```
import filters as f

class Pkcs7Pad(f.BaseFilter):
  block_size = 16

  def _apply(self, value):
     # The incoming value must be a byte string.
     value = self._filter(value, f.Type(bytes))
     if self._has_errors:
         return None

     extra_bytes = self.block_size - (len(value) % self.block_size)
     return value + bytes([extra_bytes] * extra_bytes)
```

Important: `self._filter` will not raise an exception if the value is invalid; your filter *must* check `self._has_errors` after calling `self._filter(...)`!

5.3.3 Unit Tests

To help you unit test your custom filters, the Filters library provides a helper class called `filters.test.BaseFilterTestCase`.

This class defines two methods that you can use to test your filter:

- `assertFilterPasses`: Given an input value, asserts that the filter returns an expected value when applied.

- `assertFilterErrors`: Given an input value, asserts that the filter generates the expected filter error messages when applied.

Here's a starter test case for `Pkcs7Pad`:

```
import filters as f
from filters.test import BaseFilterTestCase

class Pkcs7PadTestCase(BaseFilterTestCase):
    # Specify your filter as ``filter_type``.
    filter_type = Pkcs7Pad

    def test_pass_none(self):
        """``None`` always passes this filter."""
        self.assertFilterPasses(None)

    def test_pass_padding(self):
        """Padding a value to the correct length."""
        # Use ``self.assertFilterPasses`` to check the result of filtering a
        # valid value.
        self.assertFilterPasses(
            # If this is the input...
            b'Hello, world!',
            # ... this is the expected result.
            b'Hello, world!\x03\x03\x03'
        )

    def test_fail_wrong_type(self):
        """The incoming value is not a byte string."""
        # Use ``self.assertFilterErrors`` to check the errors from filtering
        # an invalid value.
        self.assertFilterErrors(
            # If this is the input...
            'Hello, world!',
            # ... these are the expected filter errors.
            [f.Type.CODE_WRONG_TYPE],
        )
```

5.3.4 Registering Your Filters (Optional)

Once you've packaged up your filters, you can register them with the Extensions framework to add them to the (nearly) top-level `filters.ext` namespace.

This is an optional step; it may make your filters easier to use, though there are some trade-offs.

See *Extending the Filters Namespace* for more information.

CHAPTER 6

Extending the Filters Namespace

Once you've *written your own filters*, you can start using them right away!

```
In [1]: from filters_iso import Currency

In [2]: Currency().apply('pen')
Out[2]: PEN

In [3]: Currency().apply('foo')
FilterError: This is not a valid ISO 4217 currency code.
```

Depending on your situation (and preferences), you might not mind importing your custom filters explicitly.

However, sometimes all those imports start to get unwieldy, especially if you have to use namespaces in order to keep them all straight:

```
import filters as f
import filters_iso as iso_filters
import api.filters as api_filters

request_filter = f.FilterMapper({
    'locale': f.Unicode | f.Strip | iso_filters.Locale,
    'username': f.Unicode | f.Strip | api_filters.User | f.Required,
})
```

And so on.

Some developers don't mind this; others can't stand it.

For those of you who fall into the latter group, the Filters library provides an extensions framework that allows you to add your filters to the (nearly) top-level `filters.ext` namespace:

```
import filters as f

request_filter = f.FilterMapper({
    'locale': f.Unicode | f.Strip | f.ext.Locale,
```

(continues on next page)

(continued from previous page)

```
    'username': f.Unicode | f.Strip | f.ext.User | f.Required,
})
```

Note in the above example that the `Locale` and `User` filters do not need to be imported explicitly, and they are added automatically to the `f.ext` namespace.

CHAPTER 7

Trade-Offs

There is one major downside to using the Extensions framework: IDE autocompletion won't work (or at least, I haven't figured out how to make it work yet).

Extension filters are registered at runtime, so your IDE's static analysis has no way to know what's available in `filters.ext`.

Depending on your IDE, however, there may be ways to work around this. For example, PyCharm's debugger can be configured to collect type information at runtime.

CHAPTER 8

Prerequisites

In order to register your filters with the Extensions framework, your project must use setuptools and have a valid `pyproject.toml` or `setup.py` file.

CHAPTER 9

Registering Your Filters

To add custom filters to the `filters.ext` namespace, register them as entry points using the `filters.extensions` key.

Here's an example using `pyproject.toml`:

```
[project.entry-points."filters.extensions"]
Country = "filters_iso:Country"
Currency = "filters_iso:Currency"
Locale = "filters_iso:Locale"
```

If your project is using `setup.py`, it looks like this instead:

```
from setuptools import setup

setup(
  ...
  entry_points = {
    'filters.extensions': [
      'Country = filters_iso:Country',
      'Currency = filters_iso:Currency',
      'Locale = filters_iso:Locale',
    ],
  },
)
```

Note in the examples above that you can register as many filters as you want.

Tip: The name that you assign to each entry point is used as the attribute name when the corresponding filter is registered.

To use an absurd example, if you register a filter like this:

```
[project.entry-points."filters.extensions"]
HelloWorld = "filters_iso:Currency"
```

Then it will be registered like this:

```
In [1]: import filters as f

In [1]: f.ext.HelloWorld().apply('NZD')
Out[1]: NZD
```

This feature may be useful to resolve conflicts, in the event that two filter classes have the same name (see below).

9.1 Conflicts

In the event that two filters are registered with the same name, one of them will replace the other. The order that entry points are processed is not defined, so it is not predictable which filter will "win".

9.2 Troubleshooting

Remember to `pip install -e .` each time you modify your entry points; this is required in order to install the new entry points into your project's `egg-info` directory.

If your filter is still not showing up in `f.ext`, try turning on debug logging. You will see log messages as the Filters library searches for extension filters to load:

```
In [1]: import logging, sys

In [2]: logging.basicConfig(level=logging.DEBUG, stream=sys.stderr)

In [3]: import filters as f

In [4]: dir(f.ext)
DEBUG:filters.extensions:Registering extension filter filters_iso.Country as Country.
DEBUG:filters.extensions:Registering extension filter filters_iso.Currency as␣
↪Currency.
DEBUG:filters.extensions:Registering extension filter filters_iso.Locale as Locale.
Out[4]: ['Country', 'Currency', 'Locale']
```

CHAPTER 10

Simple Filters

Simple filters perform validations and transformations on individual values.

Tip: The filters library has extensive unit tests that are thoroughly documented and designed to be easy for humans to read. If you have any questions about how individual filters are meant to be used that aren't answered in the documentation, there's a good chance that you can find the answers in the unit tests

10.1 Array

Checks that a value is a `Sequence` type, but not a string.

For example, `list` or any class that extends `typing.Sequence` will pass, but any string type (or subclass thereof) will fail.

```
import filters as f
from typing import Sequence

runner = f.FilterRunner(f.Array, ['foo', 'bar', 'baz'])
assert runner.is_valid() is True
assert runner.cleaned_data == ['foo', 'bar', 'baz']

runner = f.FilterRunner(f.Array, 'foo, bar, baz')
assert runner.is_valid() is False

# Don't use ``Type(Sequence)`` unless you also want to allow strings!
runner = f.FilterRunner(f.Type(Sequence), 'foo, bar, baz')
assert runner.is_valid() is True
```

10.2 Base64Decode

Decodes a byte string (`bytes` type) that is encoded using Base 64.

Automatically handles URL-safe variant and incorrect/missing padding.

```
import filters as f

runner = f.FilterRunner(f.Base64Decode, b'SGVsbG8sIHdvcmxkIQ==')
assert runner.is_valid() is True
assert runner.cleaned_data == b'Hello, world!'
```

Note: This filter operates on (and returns) byte strings, not unicode strings!

If the incoming value could be a unicode string, chain a *ByteString* in front of `filters.Base64Decode`:

```
import filters as f

runner = f.FilterRunner(
    f.ByteString | f.Base64Decode,
    'SGVsbG8sIHdvcmxkIQ==',
)
assert runner.is_valid() is True
assert runner.cleaned_data == b'Hello, world!'
```

If you want the resulting value to be a unicode string as well, add *Unicode* to the end of the chain:

```
import filters as f

runner = f.FilterRunner(
    f.ByteString | f.Base64Decode | f.Unicode,
    'SGVsbG8sIHdvcmxkIQ==',
)
assert runner.is_valid() is True
assert runner.cleaned_data == 'Hello, world!'
```

10.3 ByteArray

Attempts to convert a value into a `bytearray`.

```
import filters as f

runner = f.FilterRunner(
    f.ByteArray,
    b'|\xa8\xc1.8\xbd4\xd5s\x1e\xa6%+\xea!6',
)
# Note that "numeric" characters like "8" and "6" are NOT interpreted
# literally (e.g., "8" is ASCII code point 58, so it gets converted to
# ``58`` in the resulting ``bytearray``, not ``8``).  This matches the
# behaviour of Python's built-in ``bytearray`` type.
assert runner.is_valid() is True
assert runner.cleaned_data == bytearray([
    124, 168, 193, 46, 56, 189, 52, 213,
```

(continues on next page)

(continued from previous page)

```
    115, 30, 166, 37, 43, 234, 33, 54,
])
```

If the incoming value is a unicode string, it is first converted into `bytes` using the UTF-8 encoding by default. If you want it to use a different encoding, you can provide it to the filter's initialiser:

```
import filters as f

# Unicode string is encoded using UTF-8 by default.
runner = f.FilterRunner(f.ByteArray, 'Iñtërnâtiônàlizætiøn')
assert runner.is_valid() is True
assert runner.cleaned_data == bytearray([
    73, 195, 177, 116, 195, 171, 114, 110, 195, 162, 116, 105, 195,
    180, 110, 195, 160, 108, 105, 122, 195, 166, 116, 105, 195, 184, 110,
])

# You can specify a different encoding.
runner = f.FilterRunner(f.ByteArray('iso-8859-1'), 'Iñtërnâtiônàlizætiøn')
assert runner.is_valid() is True
assert runner.cleaned_data == bytearray([
    73, 241, 116, 235, 114, 110, 226, 116, 105, 244,
    110, 224, 108, 105, 122, 230, 116, 105, 248, 110,
])
```

10.4 ByteString

Converts a value into a byte string (`bytes` type).

By default, this filter encodes the result using UTF-8, but you can change this via the `encoding` parameter in the filter initialiser.

```
import filters as f

runner = f.FilterRunner(f.ByteString, 'Iñtërnâtiônàlizætiøn')
assert runner.is_valid() is True
# 'Iñtërnâtiônàlizætiøn' encoded as bytes using utf-8:
assert runner.cleaned_data ==\
    b'I\xc3\xb1t\xc3\xabrn\xc3\xa2ti\xc3' \
    b'\xb4n\xc3\xa0liz\xc3\xa6ti\xc3\xb8n'
```

10.5 Call

Calls an arbitrary function on the incoming value.

Note: This filter is almost always inferior to *Writing Your Own Filters*, but it can be useful for quickly injecting a function into a filter chain, just to see if it will work.

```
import filters as f

def div_two(value):
```

(continues on next page)

(continued from previous page)

```
    if value % 2:
        raise f.FilterError('value is not even!')
    return value / 2

runner = f.FilterRunner(f.Call(div_two), 42)
assert runner.is_valid() is True
assert runner.cleaned_data == 21

runner = f.FilterRunner(f.Call(div_two), 43)
assert runner.is_valid() is False
```

Important: The function must raise a `filters.FilterError` to indicate that the incoming value is not valid.

If the function returns any value (including *False*, *None*, etc.) then the incoming value will be considered valid.

```
def div_two(value):
    return False if value % 2 else value / 2

runner = f.FilterRunner(f.Call(div_two), 43)
assert runner.is_valid() is True
assert runner.cleaned_data is False
```

10.6 CaseFold

Applies case folding to a string value.

```
import filters as f

runner = f.FilterRunner(f.CaseFold, 'Weißkopfseeadler')
assert runner.is_valid() is True
assert runner.cleaned_data == 'weisskopfseeadler'

# Note that case-folded does not necessarily mean ASCII-compatible!
runner = f.FilterRunner(f.CaseFold, 'İstanbul')
assert runner.cleaned_data == 'i\u0307stanbul'
```

10.7 Choice

Requires the incoming value to match one of the values specified in the filter's initialiser.

```
import filters as f

runner = f.FilterRunner(f.Choice(choices=('Moe', 'Larry', 'Curly')))

runner.apply('Curly')
assert runner.is_valid() is True
assert runner.cleaned_data == 'Curly'

runner.apply('Shemp')
assert runner.is_valid() is False
```

The comparison is case-sensitive by default; you can override this by passing `case_sensitive=False` to the filter initialiser.

The `choices` passed to the filter initialiser are the 'canonical' ones; when a match is found, the filter will always return the matching choice, rather than the raw input.

```
import filters as f

runner = f.FilterRunner(
    f.Choice(
        choices=['Weiße Taube', 'Wellensittich', 'Spatz'],
        case_sensitive=False
    )
)

runner.apply('weisse taube')
assert runner.is_valid() is True
assert runner.cleaned_data == 'Weiße Taube'
```

10.8 Date

Interprets a string as a date. The result is a `datetime.date` instance.

```
import filters as f
from datetime import date

runner = f.FilterRunner(f.Date, '2015-05-11')
assert runner.is_valid() is True
assert runner.cleaned_data == date(2015, 5, 11)
```

Note: If the incoming value appears to be a datetime with tzinfo, it is first converted to UTC. In some cases, this can make the resulting date appear to be off by 1 day.

```
import filters as f
from datetime import date

runner = f.FilterRunner(f.Date, '2015-05-11T19:56:58-05:00')
assert runner.is_valid() is True
# The resulting date appears to occur 1 day later that the original
# value because it gets converted to UTC.
assert runner.cleaned_data == date(2015, 5, 12)
```

By default, the filter assumes that naive timestamps are UTC; if you need to change this, you can pass an optional `timezone` argument to the filter's initialiser:

```
import filters as f
from datetime import date
from dateutil.tz import tzoffset

# The filter is configured to interpret naive timestamps as if they are
# UTC+8.
filter_ = f.Date(timezone=tzoffset('UTC+8', 8 * 3600))

runner = f.FilterRunner(filter_, '2015-05-12 03:20:03')
```

(continues on next page)

(continued from previous page)

```
assert runner.is_valid() is True
# The resulting date appears to occur 1 day earlier because the filter
# subtracted 8 hours to convert the value to UTC.
assert runner.cleaned_data == date(2015, 5, 11)

# Note that non-native timestamps are NOT coerced!
runner = f.FilterRunner(filter_, '2015-05-12T03:20:03+01:00')
assert runner.is_valid() is True
assert runner.cleaned_data == date(2015, 5, 12)
```

10.9 Datetime

Interprets a string as a datetime. The result is a `datetime.datetime` instance with `tzinfo=utc`.

If the incoming value includes a timezone indicator, it is automatically converted to UTC. Otherwise, it is assumed to already be UTC (this can be configured via the filter initialiser).

```
import filters as f
from datetime import datetime
from pytz import utc

runner = f.FilterRunner(f.Datetime, '2015-05-11 14:56:58')
assert runner.is_valid() is True
assert runner.cleaned_data == datetime(2015, 5, 11, 14, 56, 58, tzinfo=utc)
```

Important: The resulting datetime **always** has `tzinfo=utc`.

Like *Date*, `filters.Datetime` assumes that incoming naive timestamps are UTC; you can change this by providing a `timezone` argument to the filter initializer. The filter will use this value when converting naive timestamps to UTC.

This is really important (and potentially confusing): the filter **always** returns a UTC `datetime`! The `timezone` argument tells the filter how to interpret naive timestamps, **not** which timezone to use for the resulting `datetime` values!

Example:

```
import filters as f
from datetime import datetime
from dateutil.tz import tzoffset
from pytz import utc

# Interpret naive timestamps as UTC+8.
filter_ = f.Datetime(timezone=tzoffset('UTC+8', 8 * 3600))

# Naive timestamps are assumed to be UTC+8 and converted to UTC.
runner = f.FilterRunner(filter_, '2015-05-12 09:20:03')
assert runner.is_valid() is True
assert runner.cleaned_data == datetime(2015, 5, 12, 1, 20, 3, tzinfo=utc)

# Non-naive timestamp tzinfo is respected by the filter, and the result is
# still converted to UTC for consistency.
runner = f.FilterRunner(filter_, '2015-05-11T21:14:38+04:00')
```

(continues on next page)

(continued from previous page)

```
assert runner.is_valid() is True
assert runner.cleaned_data ==\
    datetime(2015, 5, 11, 17, 14, 38, tzinfo=utc)
```

10.10 Decimal

Interprets the incoming value as a `decimal.Decimal`.

Virtually any value that can be passed to `decimal.Decimal.__init__` is accepted (including scientific notation), with a few exceptions:

- Non-finite values (e.g., `NaN`, `+Inf`, etc.) are not allowed.
- Tuple/list values (e.g., `(0, (4, 2), -1)`) are allowed by default, but you can disallow these values in the filter initialiser.

```
import filters as f
from decimal import Decimal

runner = f.FilterRunner(f.Decimal, '3.1415926')
assert runner.is_valid() is True
assert isinstance(runner.cleaned_data, Decimal)
assert runner.cleaned_data == Decimal('3.1415926')
```

The filter initialiser also accepts a parameter to set max precision. If specified, the resulting values will be *rounded* to the specified number of decimal places.

```
import filters as f
from decimal import Decimal

runner = f.FilterRunner(f.Decimal(3), '3.1415926')
assert runner.is_valid() is True
assert runner.cleaned_data == Decimal('3.142')
```

Tip: If you want to control how the rounding is applied (e.g., always round down), chain this filter with *Round*:

```
import filters as f
from decimal import Decimal, ROUND_FLOOR

runner = f.FilterRunner(
    f.Decimal | f.Round('0.001', ROUND_FLOOR),
    '3.1415926',
)
assert runner.is_valid() is True
# Value will always be rounded down.
assert runner.cleaned_data == Decimal('3.141')
```

10.11 Empty

Requires that a value have a length of zero.

Values that are not `Sized` (i.e., do not have `__len__`) are considered to be not empty. In particular, this means that `0` and `False` are *not* considered empty in this context.

```
import filters as f

runner = f.FilterRunner(f.Empty, [])
assert runner.is_valid() is True
assert runner.cleaned_data == []

runner = f.FilterRunner(f.Empty, ['foo', 'bar', 'baz', 'luhrmann'])
assert runner.is_valid() is False
```

This filter also works on strings, as well as anything else that has a length (i.e., whose type implements `typing.Sized`):

```
import filters as f

runner = f.FilterRunner(f.Empty, '')
assert runner.is_valid() is True
assert runner.cleaned_data == ''

runner = f.FilterRunner(f.Empty, 'Hello, world!')
assert runner.is_valid() is False
```

10.12 Int

Interprets the incoming value as an int.

Strings and other compatible types will be converted transparently:

```
import filters as f

runner = f.FilterRunner(f.Int, '42')
assert runner.is_valid() is True
assert runner.cleaned_data == 42
```

Floats are valid only if they have an empty fpart:

```
import filters as f

runner = f.FilterRunner(f.Int, '42.000000000000000000')
assert runner.is_valid() is True
assert runner.cleaned_data == 42

runner = f.FilterRunner(f.Int, '42.000000000000000001')
assert runner.is_valid() is False
```

10.13 IpAddress

Validates the incoming value as an IP address.

```
import filters as f
```

(continues on next page)

(continued from previous page)

```
runner = f.FilterRunner(f.IpAddress, '127.0.0.1')
assert runner.is_valid() is True
assert runner.cleaned_data == '127.0.0.1'

runner = f.FilterRunner(f.IpAddress, 'localhost')
assert runner.is_valid() is False
```

By default, this filter only accepts IPv4 addresses, but you can configure the filter to also/only accept IPv6 addresses via its initialiser.

For IPv6 addresses, the result is always converted to its short form.

```
import filters as f

# Accept IPv6 addresses only.
filter_ = f.IpAddress(ipv4=False, ipv6=True)

runner = f.FilterRunner(filter_, '0:0:0:0:0:0:0:1')
assert runner.is_valid() is True
assert runner.cleaned_data == '::1'

runner = f.FilterRunner(filter_, '1027.0.0.1')
assert runner.is_valid() is False
```

10.14 Item

Extracts a single item from a mapping (e.g., `dict`) or sequence (e.g., `list`).

By default, the filter extracts the first item from the incoming value:

```
import filters as f

# Extract the value of the first item in a mapping:
runner = f.FilterRunner(f.Item, {'name': 'Indy', 'job': 'archaeologist'})
assert runner.is_valid() is True
assert runner.cleaned_data == 'Indy'

# Extract the item at the 0th index in a sequence:
runner = f.FilterRunner(f.Item, ['Indiana', 'Marcus', 'Marion'])
assert runner.cleaned_data == 'Indiana'
```

You can also provide the key/index that you want extracted to the filter initialiser:

```
import filters as f

# Extract the 'job' value from a mapping:
runner = f.FilterRunner(
    f.Item('job'),
    {'name': 'Indy', 'job': 'archaeologist'},
)
assert runner.is_valid() is True
assert runner.cleaned_data == 'archaeologist'

# Extract the item at the 2nd index in a sequence:
runner = f.FilterRunner(f.Item(2), ['Indiana', 'Marcus', 'Marion'])
```

(continues on next page)

(continued from previous page)

```
assert runner.is_valid() is True
assert runner.cleaned_data == 'Marion'
```

If the incoming value is empty, or if it does not contain the required key, it is invalid:

```
import filters as f

runner = f.FilterRunner(f.Item, {})
assert runner.is_valid() is False

runner = f.FilterRunner(
    f.Item('profession'),
    {'name': 'Indy', 'job': 'archaeologist'},
)
assert runner.is_valid() is False

runner = f.FilterRunner(f.Item, [])
assert runner.is_valid() is False

runner = f.FilterRunner(f.Item(42), ['Indiana', 'Marcus', 'Marion'])
assert runner.is_valid() is False
```

10.15 JsonDecode

Decodes a string that is JSON-encoded.

```
import filters as f

runner = f.FilterRunner(f.JsonDecode, '{"foo": "bar", "baz": "luhrmann"}')
assert runner.is_valid() is True
assert runner.cleaned_data == {'foo': 'bar', 'baz': 'luhrmann'}
```

Note that this filter can be chained with other filters. For example, you can use `f.JsonDecode | f.FilterMapper(...)` to apply filters to a JSON-encoded dict:

```
import filters as f
from datetime import date

runner = f.FilterRunner(
    f.JsonDecode |
    f.FilterMapper({
        'birthday':  f.Date,
        'gender':    f.CaseFold | f.Choice(choices={'m', 'f', 'x'}),
    }),
    '{"birthday":"1879-03-14", "gender":"M"}'
)
assert runner.is_valid() is True
assert runner.cleaned_data == {
    'birthday': date(1879, 3, 14),
    'gender': 'm',
}
```

Check out *Filterception* for more fun examples

10.16 Length

Requires that a value's length matches the value specified in the filter initialiser.

Values that are not `Sized` (i.e., do not have `__len__`) automatically fail.

```
import filters as f

runner = f.FilterRunner(f.Length(3), ['foo', 'bar', 'baz'])
assert runner.is_valid() is True
assert runner.cleaned_data == ['foo', 'bar', 'baz']

runner = f.FilterRunner(f.Length(3), ['foo', 'bar', 'baz', 'luhrmann'])
assert runner.is_valid() is False
```

This filter also works on strings, as well as anything else that has a length (i.e., whose type implements `typing.Sized`):

```
import filters as f

runner = f.FilterRunner(f.Length(23), 'Kia ora e te ao whānui!')
assert runner.is_valid() is True
assert runner.cleaned_data == 'Kia ora e te ao whānui!'

runner = f.FilterRunner(f.Length(23), '¡Hola, mundo!')
assert runner.is_valid() is False
```

Note: `filters.Length` requires the incoming value to have *exactly* the specified length; if you want to check that the incoming value has a minimum or maximum length, use *MinLength* or *MaxLength*, respectively.

10.17 Max

Requires that the value be less than [or equal to] the value specified in the filter initialiser.

```
import filters as f

# Incoming value is less than max.
runner = f.FilterRunner(f.Max(5), 4)
assert runner.is_valid() is True
assert runner.cleaned_data == 4

# Incoming value is equal to max.
runner = f.FilterRunner(f.Max(5), 5)
assert runner.is_valid() is True
assert runner.cleaned_data == 5

# Incoming value is greater than max.
runner = f.FilterRunner(f.Max(5), 6)
assert runner.is_valid() is False
```

If you only want to allow incoming values that are less than (not equal to) the max value, set `exclusive=True` in the filter's initialiser:

```
import filters as f

runner = f.FilterRunner(f.Max(5, exclusive=True), 5)
assert runner.is_valid() is False
```

10.18 MaxBytes

Checks that a string will fit into a max number of bytes when encoded (using UTF-8 by default).

Important: The resulting value will be a byte string (`bytes` type), not a unicode string!

- If you want to check the length of a unicode string (`str` type), use *MaxChars*.
- If you want to check the length of a list or other arbitrary sequence, use *MaxLength*.

```
import filters as f

runner = f.FilterRunner(f.MaxBytes(25), 'Γεισου Κσμε')
assert runner.is_valid() is True
assert runner.cleaned_data ==\
  b'\xce\x93\xce\xb5\xce\xb9\xce\xac\xcf\x83\xce\xbf' \
  b'\xcf\x85 \xce\x9a\xcf\x8c\xcf\x83\xce\xbc\xce\xb5'

runner = f.FilterRunner(f.MaxBytes(24), 'Γεισου Κσμε')
assert runner.is_valid() is False
assert runner.cleaned_data is None
```

Instead of treating too-long values as invalid, you can configure the filter to truncate them instead:

```
import filters as f

runner = f.FilterRunner(f.MaxBytes(22, truncate=True), ' ')
# Truncated values are considered valid.
assert runner.is_valid() is True
assert runner.cleaned_data ==\
    b'\xe0\xa4\xb9\xe0\xa5\x88\xe0\xa4\xb2\xe0' \
    b'\xa5\x8b \xe0\xa4\xb5\xe0\xa4\xb0\xe0\xa5\x8d'
```

Note: When truncating with a multibyte encoding (e.g., UTF-8), the filter may remove additional bytes as needed to avoid orphaned sequences:

```
import filters as f

runner = f.FilterRunner(f.MaxBytes(21, truncate=True), ' ')
assert runner.is_valid() is True
# Result is truncated to 19 bytes instead of 21, so as not to orphan a
# multibyte sequence.
assert len(runner.cleaned_data) == 19
assert runner.cleaned_data.decode('utf-8') == ' '
```

You can also configure the filter to apply a prefix and/or suffix to the value when truncating:

```
import filters as f

# Apply a prefix to truncated values:
runner = f.FilterRunner(
    f.MaxBytes(12, truncate=True, prefix='(more) '),
    'Hello, world!'
)
assert runner.is_valid() is True
# The length of the prefix is taken into account, so that the result is still
# 12 bytes long.
assert runner.cleaned_data == b'(more) Hello'

# Apply a suffix to truncated values:
runner = f.FilterRunner(
    f.MaxBytes(12, truncate=True, suffix='...'),
    'Hello, world!',
)
assert runner.is_valid() is True
assert runner.cleaned_data == b'Hello, wo...'

# Apply both, why not..
runner = f.FilterRunner(
    f.MaxBytes(12, truncate=True, prefix='->', suffix='<-'),
    'Hello, world!',
)
assert runner.is_valid() is True
assert runner.cleaned_data == b'->Hello, w<-'
```

By default, the filter uses UTF-8; if you need to use a different encoding, you can specify it when initialising the filter:

```
import filters as f

runner = f.FilterRunner(
    f.MaxBytes(32, truncate=True, encoding='utf-16'),
    'kia ora e te ao whānui',
)
assert runner.is_valid() is True
assert runner.cleaned_data ==\
    b'\xff\xfek\x00i\x00a\x00 \x00o\x00r\x00a\x00' \
    b' \x00e\x00 \x00t\x00e\x00 \x00a\x00o\x00'

# Prefix and suffix also work with alternate encodings.
runner = f.FilterRunner(
    f.MaxBytes(40, truncate=True, prefix='[] ', suffix=' ()', encoding='utf-16'),
    '       ?',
)
assert runner.is_valid() is True
assert runner.cleaned_data == (
    b"\xff\xfe"                              # BOM
    b"[\x00\x05\t'\t?\t\x15\t]\x00 \x00"  # Prefix
    b'.\tH\t\x02\t \x00\x05\t'             # Truncated string
    b" \x00(\x00\x05\t'\t?\t\x15\t)\x00"  # Suffix
)
assert len(runner.cleaned_data) == 40
```

10.19 MaxChars

Requires that a string's length is less than or equal to the value specified in the filter initialiser.

Note: This filter only works on string values.

- If you want to check the length of a byte string, use *MaxBytes*.
- If you want to check the length of a list or other arbitrary sequence, use *MaxLength*.

```
import filters as f

runner = f.FilterRunner(f.MaxChars(12))

runner.apply('Hello, world')
assert runner.is_valid() is True
assert runner.cleaned_data == 'Hello, world'

runner.apply('Hello, world!')
assert runner.is_valid() is False
```

Instead of treating too-long values as invalid, you can configure the filter to truncate them instead:

```
import filters as f

runner = f.FilterRunner(f.MaxChars(4, truncate=True), 'Chào th gii!')
assert runner.is_valid() is True
assert runner.cleaned_data == 'Chào'
```

You can also configure the filter to apply a prefix and/or suffix to the value when truncating:

```
import filters as f

# Apply a prefix to truncated values:
runner = f.FilterRunner(
    f.MaxChars(12, truncate=True, prefix='(more) '),
    'Hello, world!'
)
assert runner.is_valid() is True
# The length of the prefix is taken into account, so that the result is still
# 12 characters long.
assert runner.cleaned_data == '(more) Hello'

# Apply a suffix to truncated values:
runner = f.FilterRunner(
    f.MaxChars(12, truncate=True, suffix='...'),
    'Hello, world!',
)
assert runner.is_valid() is True
assert runner.cleaned_data == 'Hello, wo...'

# Apply both, why not..
runner = f.FilterRunner(
    f.MaxChars(12, truncate=True, prefix='->', suffix='<-'),
    'Hello, world!',
)
```

(continues on next page)

(continued from previous page)

```
assert runner.is_valid() is True
assert runner.cleaned_data == '->Hello, w<-'
```

10.20 MaxLength

Requires that a value's length is less than or equal to the value specified in the filter initialiser.

Values that are not `Sized` (i.e., do not have `__len__`) automatically fail.

Note: If you are working with a unicode string (`str` type) or byte string (`bytes` type), you might want to use *MaxChars* or *MaxBytes*, respectively.

`filters.MaxLength` will still work on strings, but it doesn't have advanced capabilities like applying prefix and/or suffix to truncated values, avoiding orphaned multibyte sequences, etc.

```
import filters as f

runner = f.FilterRunner(f.MaxLength(3))

runner.apply(['foo', 'bar', 'baz'])
assert runner.is_valid() is True
assert runner.cleaned_data == ['foo', 'bar', 'baz']

runner.apply(['foo', 'bar', 'baz', 'luhrmann'])
assert runner.is_valid() is False
```

This filter also works on strings, as well as anything else that has a length (i.e., whose type implements `typing.Sized`):

```
import filters as f

runner = f.FilterRunner(f.MaxLength(20))

runner.apply('¡Hola, mundo!')
assert runner.is_valid() is True
assert runner.cleaned_data == '¡Hola, mundo!'

runner.apply('Kia ora e te ao whānui!')
assert runner.is_valid() is False
```

Instead of treating too-long values as invalid, you can configure the filter to truncate them instead:

```
import filters as f

runner = f.FilterRunner(f.MaxLength(3, truncate=True))

runner.apply(['foo', 'bar', 'baz', 'luhrmann'])
assert runner.is_valid() is True
assert runner.cleaned_data == ['foo', 'bar', 'baz']
```

Caution: When truncating a byte string (`bytes` type), you can end up with invalid multibyte sequences, resulting in a value that cannot be decoded back into a unicode string!

If you want to truncate a byte string (`bytes` type) use *MaxBytes* instead, which knows how to avoid this problem.

```
import filters as f

value = ' '.encode('utf-8')

runner = f.FilterRunner(f.MaxLength(21, truncate=True), value)
assert runner.is_valid() is True
# The resulting sequence is exactly 21 bytes long, but the last 2 bytes
# cannot be decoded because the last code point in the truncated value
# ```` requires 3 bytes to encode in UTF-8.
assert len(runner.cleaned_data) == 21
try:
    runner.cleaned_data.decode('utf-8')
except UnicodeDecodeError:
    pass

# MaxBytes knows how to avoid this problem.
runner = f.FilterRunner(f.MaxBytes(21, truncate=True), value)
assert runner.is_valid() is True
# Result is truncated to 19 bytes instead of 21, so as not to orphan a
# multibyte sequence.
assert len(runner.cleaned_data) == 19
assert runner.cleaned_data.decode('utf-8') == ' '
```

10.21 Min

Requires that the value be greater than [or equal to] the value specified in the filter initialiser.

```
import filters as f

# Incoming value is greater than min.
runner = f.FilterRunner(f.Min(5), 6)
assert runner.is_valid() is True
assert runner.cleaned_data == 6

# Incoming value is equal to min.
runner = f.FilterRunner(f.Min(5), 5)
assert runner.is_valid() is True
assert runner.cleaned_data == 5

# Incoming value is less than min.
runner = f.FilterRunner(f.Min(5), 4)
assert runner.is_valid() is False
```

If you only want to allow incoming values that are greater than (not equal to) the min value, set `exclusive=True` in the filter's initialiser:

```
import filters as f

runner = f.FilterRunner(f.Min(5, exclusive=True), 5)
assert runner.is_valid() is False
```

10.22 MinLength

Requires that a value's length is greater than or equal to the value specified in the filter initialiser.

Values that are not `Sized` (i.e., do not have `__len__`) automatically fail.

```
import filters as f

runner = f.FilterRunner(f.MinLength(3), ['foo', 'bar', 'baz'])
assert runner.is_valid() is True
assert runner.cleaned_data == ['foo', 'bar', 'baz']

runner = f.FilterRunner(f.MinLength(3), ['foo', 'bar'])
assert runner.is_valid() is False
```

This filter also works on strings, as well as anything else that has a length (i.e., whose type implements `typing.Sized`):

```
import filters as f

runner = f.FilterRunner(f.MinLength(20), 'Kia ora e te ao whānui!')
assert runner.is_valid() is True
assert runner.cleaned_data == 'Kia ora e te ao whānui!'

runner = f.FilterRunner(f.MinLength(20), '¡Hola, mundo!')
assert runner.is_valid() is False
```

10.23 NamedTuple

Converts the incoming value into a named tuple

Initialize this filter with the type of named tuple that you want to use for conversions.

```
import filters as f
from collections import namedtuple

Colour = namedtuple('Colour', ('r', 'g', 'b', 'a'))

runner = f.FilterRunner(f.NamedTuple(Colour), [65, 105, 225, 1])
assert runner.is_valid() is True
assert isinstance(runner.cleaned_data, Colour)
assert runner.cleaned_data == Colour(65, 105, 225, 1)
```

Tip: You can also provide an optional filter map, which will be applied to the values in the resulting named tuple.

```
import filters as f
from collections import namedtuple
from decimal import Decimal

Colour = namedtuple('Colour', ('r', 'g', 'b', 'a'))

runner = f.FilterRunner(
    f.NamedTuple(Colour, {
        'r': f.Required | f.Int | f.Min(0) | f.Max(255),
```

(continues on next page)

(continued from previous page)

```
        'g': f.Required | f.Int | f.Min(0) | f.Max(255),
        'b': f.Required | f.Int | f.Min(0) | f.Max(255),
        'a': f.Optional(default=1) | f.Decimal | f.Min(0) | f.Max(1),
    }),
    ["65", "105", "225", "0.75"],
)
assert runner.is_valid() is True
assert isinstance(runner.cleaned_data, Colour)
assert runner.cleaned_data == Colour(65, 105, 225, Decimal('0.75'))
```

10.24 NoOp

This filter returns the incoming value unmodified.

It can be useful in cases where you need a function to return a filter instance, even in cases where no filtering is needed.

```
import filters as f

runner = f.FilterRunner(f.NoOp, 'literally anything')
assert runner.is_valid() is True
assert runner.cleaned_data == 'literally anything'
```

Tip: In many contexts, you can safely substitute `None` for `filters.NoOp`:

```
import filters as f

runner = f.FilterRunner(
   f.Unicode | None | f.NotEmpty,
   'literally anything',
)
assert runner.is_valid() is True
assert runner.cleaned_data == 'literally anything'
```

An example of a case where you might need to use `NoOp` is if you want to make the first filter in a chain dynamic, e.g.:

```
import filters as f
from decimal import Decimal

@f.filter_macro
def Number(strip_sign: bool = False):
    # Can't return ``None`` here, or else an error will occur when we
    # try to chain it with ``f.Min`` below, so we have to use ``f.NoOp``
    # instead.
    return f.Strip(r'-') if strip_sign else f.NoOp | f.Decimal

runner = f.FilterRunner(Number | f.Min(42), '-100')
assert runner.is_valid() is False
```

10.25 NotEmpty

Requires that a value have a length greater than zero.

Values that are not `Sized` (i.e., do *not* have `__len__`) are considered to be **not empty**. In particular, this means that `0` and `False` are *not* considered empty in this context.

```
import filters as f

runner = f.FilterRunner(f.NotEmpty, ['foo', 'bar', 'baz', 'luhrmann'])
assert runner.is_valid() is True
assert runner.cleaned_data == ['foo', 'bar', 'baz', 'luhrmann']

runner = f.FilterRunner(f.NotEmpty, [])
assert runner.is_valid() is False
```

This filter also works on strings, as well as anything else that has a length (i.e., whose type implements `typing.Sized`):

```
import filters as f

runner = f.FilterRunner(f.NotEmpty, 'Hello, world!')
assert runner.is_valid() is True
assert runner.cleaned_data == 'Hello, world!'

runner = f.FilterRunner(f.NotEmpty, '')
assert runner.is_valid() is False
```

Important: `None` always passes this filter (see *Much Ado About None* for more information). Use *Required* to reject `None`:

```
import filters as f

runner = f.FilterRunner(f.NotEmpty, None)
assert runner.is_valid() is True

runner = f.FilterRunner(f.Required, None)
assert runner.is_valid() is False
```

10.26 Omit

Filters an incoming mapping (e.g., `dict`) or sequence (e.g., `list`), omitting the keys specified when the filter is initialised.

```
import filters as f

# Omit 'alpha' and 'hex' from a mapping
runner = f.FilterRunner(
    f.Omit({'alpha', 'hex'}),
    {'red': 65, 'green': 105, 'blue': 225, 'alpha': 1, 'hex': '#4169E1'}
)
assert runner.is_valid() is True
```

(continues on next page)

(continued from previous page)

```
assert runner.cleaned_data == {'red': 65, 'green': 105, 'blue': 225}

# Remove the first 2 items from a sequence:
runner = f.FilterRunner(f.Omit({0, 1}), [42, 86, 99])
assert runner.is_valid() is True
assert runner.cleaned_data == [99]
```

Note: The incoming value is considered valid regardless of whether any values were actually filtered out. For example, if an incoming mapping doesn't contain any of the keys to be omitted, then it is passed through unmodified:

```
import filters as f

# Filter omits 'age' and 'profession', but incoming value doesn't have
# either of those keys.
runner = f.FilterRunner(
    f.Omit({'age', 'profession'}),
    {'name': 'Indy', 'job': 'archaeologist', 'actor': 'Harrison'},
)
assert runner.is_valid() is True
assert runner.cleaned_data ==\
    {'name': 'Indy', 'job': 'archaeologist', 'actor': 'Harrison'}
```

If you want to validate the shape of an incoming value, then you may prefer:

- For mappings: *FilterMapper*.
- For sequences: *Length*, *MaxLength*, *MinLength*.

10.27 Optional

Provides a default value that will be returned if the incoming value is empty (has a length of zero) or is `None`.

Values that are not `Sized` (i.e., do not have `__len__`) are considered to be *not* empty. In particular, this means that `0` and `False` are *not* considered empty in this context.

```
import filters as f

runner = f.FilterRunner(f.Optional('t') | f.Choice({'t', 'f'}))

runner.apply('f')
assert runner.is_valid() is True
assert runner.cleaned_data == 'f'

runner.apply('')
assert runner.is_valid() is True
assert runner.cleaned_data == 't'

# Also returns the default when the incoming value is ``None``:
runner.apply(None)
assert runner.is_valid() is True
assert runner.cleaned_data == 't'
```

If the default value is callable, then the filter will call it instead:

```
import filters as f

runner = f.FilterRunner(f.Optional(list), None)

assert runner.is_valid() is True
assert runner.cleaned_data == []
```

To pass arguments to the default callable, use a partial or a lambda:

```
import filters as f

def power_of_two(power):
    return pow(2, power)

# Using a partial:
from functools import partial
runner = f.FilterRunner(f.Optional(partial(power_of_two, power=8)), None)
assert runner.is_valid() is True
assert runner.cleaned_data == 256

# Using a lambda:
runner = f.FilterRunner(f.Optional(lambda: power_of_two(power=8)), None)
assert runner.is_valid() is True
assert runner.cleaned_data == 256
```

Important: This filter only substitutes a default for **empty** values, not **invalid** ones.

A filter chain stops processing as soon as any filter in the chain flags an invalid value, so putting this filter at the end of a chain very likely will not do what you expect.

```
import filters as f

runner = f.FilterRunner(f.Choice({'t', 'f'}) | f.Optional('t'), '')
# Incoming value ``''`` does not match any valid choices, so the filter
# chain stops before it gets to the ``Optional`` filter!
assert runner.is_valid() is False
assert runner.cleaned_data is None
```

This is how the above example could be rewritten:

```
import filters as f

runner = f.FilterRunner(
    # ``f.Optional`` comes after ``f.Strip``, so that if the incoming
    # string is empty or only contains whitespace, the default value is
    # substituted instead.
    f.Unicode | f.Strip | f.Optional('t') | f.Choice({'t', 'f'})
)

runner.apply('      ')
assert runner.is_valid() is True
assert runner.cleaned_data == 't'

# ``f.Optional`` does not do anything for invalid values; only empty ones!
runner.apply('n')
assert runner.is_valid() is False
```

10.28 Pick

Filters an incoming mapping (e.g., `dict`) or sequence (e.g., `list`), collecting only the keys specified when the filter is initialised and omitting the rest:

```
import filters as f

# Pick 'red', 'green', and 'blue' items from a mapping:
runner = f.FilterRunner(
    f.Pick(['red', 'green', 'blue']),
    {'red': 65, 'green': 105, 'blue': 225, 'alpha': 1, 'hex': '#4169E1'}
)
assert runner.is_valid() is True
assert runner.cleaned_data == {'red': 65, 'green': 105, 'blue': 225}

# Pick the first 2 items from a sequence:
runner = f.FilterRunner(f.Pick([0, 1]), [42, 86, 99])
assert runner.is_valid() is True
assert runner.cleaned_data == [42, 86]
```

Important: The order of the keys you provide will determine the order that they appear in the resulting value. This is particularly important for sequences:

```
import filters as f

runner = f.FilterRunner(
    f.Pick([1, 0, 2]),
    ['Indiana', 'Marion', 'Marcus'],
)
assert runner.is_valid() is True
assert runner.cleaned_data == ['Marion', 'Indiana', 'Marcus']
```

In particular, note that sets are unordered collections, so you probably want to avoid using them to specify keys to pick:

```
#  Order of items is not guaranteed, because sets are unordered.
f.Pick({1, 2, 3})

#  Order of items is guaranteed, because lists are ordered.
f.Pick([1, 2, 3])
```

By default, any picked keys that aren't present in the incoming value are set to `None`:

```
import filters as f

# Incoming mapping is missing ``age`` key, so ``None`` is substituted:
runner = f.FilterRunner(
    f.Pick(['name', 'age']),
    {'name': 'Indiana', 'job': 'Archaeologist'},
)
assert runner.is_valid() is True
assert runner.cleaned_data == {'name': 'Indiana', 'age': None}

# Incoming sequence doesn't have a 4th item, so ``None`` is substituted:
runner = f.FilterRunner(f.Pick([0, 2, 4]), ['Indiana', 'Marion', 'Marcus'])
```

(continues on next page)

(continued from previous page)

```
assert runner.is_valid() is True
assert runner.cleaned_data == ['Indiana', 'Marcus', None]
```

If you want the filter to treat values with missing keys as invalid, pass an optional `allow_missing_keys` argument to the filter initialiser:

```
import filters as f

# All keys are required:
runner = f.FilterRunner(
    f.Pick(['name', 'age'], allow_missing_keys=False),
    {'name': 'Indiana', 'job': 'Archaeologist'},
)
assert runner.is_valid() is False

# Or, only specified keys are required:
runner = f.FilterRunner(
    f.Pick(['name', 'age'], allow_missing_keys={'age'}),
    {'name': 'Indiana', 'job': 'Archaeologist'},
)
assert runner.is_valid() is True
assert runner.cleaned_data == {'name': 'Indiana', 'age': None}

# Also works for sequences:
runner = f.FilterRunner(
    f.Pick([0, 2, 4], allow_missing_keys=False),
    ['Indiana', 'Marion', 'Marcus'],
)
assert runner.is_valid() is False

runner = f.FilterRunner(
    f.Pick([0, 2, 4], allow_missing_keys={4}),
    ['Indiana', 'Marion', 'Marcus'],
)
assert runner.is_valid() is True
assert runner.cleaned_data == ['Indiana', 'Marcus', None]
```

10.29 Regex

Executes a regular expression against a string value. The regex must match in order for the string to be considered valid.

This filter returns a list of matches.

Important: The result is **always** a list, even if there is only a single match.

Groups are not included in the result.

```
import filters as f

runner = f.FilterRunner(f.Regex(r'\d+'), '42-86-99')
assert runner.is_valid() is True
assert runner.cleaned_data == ['42', '86', '99']
```

Tip: You can chain `filters.Regex` with *FilterRepeater* to apply filters to the matched values:

```
import filters as f

runner = f.FilterRunner(
    f.Regex(r'\d+') | f.FilterRepeater(f.Int),
    '42-86-99',
)
assert runner.is_valid() is True
assert runner.cleaned_data == [42, 86, 99]
```

If you know there will only be a single match from the regular expression, you can use *Item* instead:

```
import filters as f

# Adapted from https://stackoverflow.com/a/6640851
uuid_regex =\
    r'^[\da-f]{8}-[\da-f]{4}-[\da-f]{4}-[\da-f]{4}-[\da-f]{12}$'

# ``f.Regex`` returns an array, so we have to use ``f.Item`` to extract
# the UUID value before we can pass it along to ``f.Uuid``.
runner = f.FilterRunner(f.Regex(uuid_regex) | f.Item | f.Uuid)

runner.apply('3466c56a-2ebc-449d-97d2-9b119721ff0f')
assert runner.is_valid() is True
assert runner.cleaned_data ==\
    UUID('3466c56a-2ebc-449d-97d2-9b119721ff0f')
```

10.30 Required

Basically the same as `NotEmpty`, except it also rejects `None`.

This filter is the only exception to the "`None` always passes" rule (see *Much Ado About None* for more information).

```
import filters as f

runner = f.FilterRunner(f.Required, ['foo', 'bar', 'baz', 'luhrmann'])
assert runner.is_valid() is True
assert runner.cleaned_data == ['foo', 'bar', 'baz', 'luhrmann']

runner = f.FilterRunner(f.Required, [])
assert runner.is_valid() is False

runner = f.FilterRunner(f.Required, None)
assert runner.is_valid() is False

# Note that every other filter allows ``None``!
runner = f.FilterRunner(f.NotEmpty, None)
assert runner.is_valid() is True
assert runner.cleaned_data is None
```

10.31 Round

Rounds the incoming value to the nearest integer or fraction specified in the filter initialiser.

The result is always a `decimal.Decimal` instance, to avoid issues with floating-point precision.

```
import filters as f
from decimal import Decimal

runner = f.FilterRunner(f.Round('5'), 42)
assert runner.is_valid() is True
assert isinstance(runner.cleaned_data, Decimal)
assert runner.cleaned_data == Decimal('40')

runner = f.FilterRunner(f.Round('5'), 43)
assert runner.is_valid() is True
assert isinstance(runner.cleaned_data, Decimal)
assert runner.cleaned_data == Decimal('45')
```

Important: When specifying a decimal value to round to, use a string value, in order to prevent aforementioned issues with floating-point precision.

```
import filters as f

#  Potentially unsafe; don't do this!
runner = f.FilterRunner(f.Round(0.001), '3.1415926')

#  Do this instead:
runner = f.FilterRunner(f.Round('0.001'), '3.1415926')
```

You can also control the rounding behaviour by specifying a rounding mode:

```
import filters as f
from decimal import ROUND_CEILING, ROUND_FLOOR

# Always round up:
runner = f.FilterRunner(f.Round('0.25', ROUND_CEILING), '0.26')
assert runner.is_valid() is True
assert runner.cleaned_data == Decimal('0.5')

# Always round down:
runner = f.FilterRunner(f.Round('0.25', ROUND_FLOOR), '0.49')
assert runner.is_valid() is True
assert runner.cleaned_data == Decimal('0.25')
```

10.32 Split

Uses a regular expression to split a string value into chunks.

The result is always a list. If the regular expression doesn't match anything in an incoming value, then that value is returned as a single-item list (see example below).

```
import filters as f

filter_ = f.Split(r':+')

runner = f.FilterRunner(filter_, 'foo:bar::baz:::')
assert runner.is_valid() is True
assert runner.cleaned_data == ['foo', 'bar', 'baz', '']

runner = f.FilterRunner(filter_, 'foo bar baz')
assert runner.is_valid() is True
assert runner.cleaned_data == ['foo bar baz']
```

10.33 Strip

Removes whitespace from the start and end of a string.

```
import filters as f

runner = f.FilterRunner(f.Strip, '\r  \t \x00 Hello, world! \x00 \t  \n')
assert runner.is_valid() is True
assert runner.cleaned_data == 'Hello, world!'
```

Alternatively, you can use regular expressions to control what the filter strips from incoming values:

```
import filters as f

runner = f.FilterRunner(
    f.Strip(leading=r'\d', trailing=r"['a-z ]+"),
    "54321 A long time ago... in a galaxy far far away ",
)
assert runner.is_valid() is True
assert runner.cleaned_data == '4321 A long time ago...'
```

10.34 Type

Requires that the incoming value have the type(s) specified in the filter initialiser.

```
import filters as f

runner = f.FilterRunner(f.Type(str), 'Hello, world!')
assert runner.is_valid() is True
assert runner.cleaned_data == 'Hello, world!'

runner = f.FilterRunner(f.Type(str), 42)
assert runner.is_valid() is False
```

You can specify a tuple of types, the same as you would for `isinstance`:

```
import filters as f

runner = f.FilterRunner(f.Type((str, int)), 'Hello, world!')
assert runner.is_valid() is True
```

(continues on next page)

(continued from previous page)

```
assert runner.cleaned_data == 'Hello, world!'

runner = f.FilterRunner(f.Type((str, int)), 42)
assert runner.is_valid() is True
assert runner.cleaned_data == 42

runner = f.FilterRunner(f.Type((str, int)), ['Hello, world!', 42])
assert runner.is_valid() is False
```

By default, the filter permits subclasses, but you can configure it via the initialiser to require an exact type match:

```
import filters as f

runner = f.FilterRunner(f.Type(int, allow_subclass=False), 1)
assert runner.is_valid() is True
assert runner.cleaned_data == 1

runner = f.FilterRunner(f.Type(int, allow_subclass=False), True)
assert runner.is_valid() is False

# Default behaviour is to allow subclasses.
runner = f.FilterRunner(f.Type(int), True)
assert runner.is_valid() is True
assert runner.cleaned_data is True
```

Tip: If you want to check that an incoming value is a list or other sequence, use *Array* instead of `Type(Sequence)`:

```
import filters as f
from typing import Sequence

# Works as expected for lists...
runner = f.FilterRunner(f.Type(Sequence), ['foo', 'bar', 'baz'])
assert runner.is_valid() is True

# ... but strings are also sequences!
runner = f.FilterRunner(f.Type(Sequence), 'foo, bar, baz')
assert runner.is_valid() is True

# To avoid this issue, use ``f.Array`` instead.
runner = f.FilterRunner(f.Array, ['foo', 'bar', 'baz'])
assert runner.is_valid() is True

runner = f.FilterRunner(f.Array, 'foo, bar, baz')
assert runner.is_valid() is False
```

10.35 Unicode

Converts a value to a unicode string (`str` type).

By default the filter also applies the following transformations:

- Convert to NFC form.
- Remove non-printable characters.

- Convert line endings to unix style (e.g., `\r\n` => `\n`).

If desired, you can disable these extra transformations by passing `normalize=False` (note American spelling) to the filter initialiser.

```
import filters as f

runner = f.FilterRunner(
    f.Unicode,

    # You get used to it.  I don't even see the code; all I see is,
    # "blond"... "brunette"... "redhead"...
    # Hey, you uh... want a drink?
    b'\xe2\x99\xaa '
    b'\xe2\x94\x8f(\xc2\xb0.\xc2\xb0)\xe2\x94\x9b '
    b'\xe2\x94\x97(\xc2\xb0.\xc2\xb0)\xe2\x94\x93 '
    b'\xe2\x99\xaa',
)
assert runner.is_valid() is True
assert runner.cleaned_data == '♪ (°.°) (°.°) ♪'
```

The filter expects the incoming value to be encoded using UTF-8. If you need to use a different encoding, provide it to the filter's initialiser:

```
import filters as f

# Incoming value is not valid UTF-8.
runner = f.FilterRunner(f.Unicode, b'\xc4pple')
assert runner.is_valid() is False

# Tell the filter to decode using Latin-1 instead.
runner = f.FilterRunner(f.Unicode('iso-8859-1'), b'\xc4pple')
assert runner.is_valid() is True
assert runner.cleaned_data == 'Äpple'
```

10.36 Uuid

Converts a string value into a `uuid.UUID` object.

```
import filters as f
from uuid import UUID

runner = f.FilterRunner(f.Uuid, '3466c56a-2ebc-449d-97d2-9b119721ff0f')
assert runner.is_valid() is True
assert isinstance(runner.cleaned_data, UUID)
assert runner.cleaned_data.hex == '3466c56a2ebc449d97d29b119721ff0f'
assert runner.cleaned_data.version == 4
```

By default, any UUID version is allowed, but you can specify the required version in the filter initialiser:

```
import filters as f

filter_ = f.Uuid(version=4)

runner = f.FilterRunner(filter_, '3466c56a-2ebc-449d-97d2-9b119721ff0f')
assert runner.is_valid() is True
```

(continues on next page)

(continued from previous page)

```
runner = f.FilterRunner(filter_, '2830f705596911e59628e0f8470933c8')
# Incoming value is a v1 UUID, but we're expecting a v4.
assert runner.is_valid() is False
```

Note: UUIDs can be provided in several different formats; the following values are all considered to be correct representations of the same UUID:

- `3466c56a-2ebc-449d-97d2-9b119721ff0f`
- `3466c56a2ebc449d97d29b119721ff0f`
- `{3466c56a2ebc449d97d29b119721ff0f}`
- `urn:uuid:3466c56a-2ebc-449d-97d2-9b119721ff0f`

This flexibility is baked into Python's UUID class; if for some reason you do not want to allow alternative formats, chain the filter with *Regex*:

```
import filters as f
from uuid import UUID

# Adapted from https://stackoverflow.com/a/6640851
uuid_regex =\
    r'^[\da-f]{8}-[\da-f]{4}-[\da-f]{4}-[\da-f]{4}-[\da-f]{12}$'

# ``f.Regex`` returns an array, so we have to use ``f.Item`` to extract
# the UUID value before we can pass it along to ``f.Uuid``.
runner = f.FilterRunner(f.Regex(uuid_regex) | f.Item | f.Uuid)

runner.apply('3466c56a-2ebc-449d-97d2-9b119721ff0f')
assert runner.is_valid() is True
assert runner.cleaned_data ==\
    UUID('3466c56a-2ebc-449d-97d2-9b119721ff0f')

runner.apply('urn:uuid:3466c56a-2ebc-449d-97d2-9b119721ff0f')
assert runner.is_valid() is False
```

CHAPTER 11

Complex Filters

Complex filters are filters that work in tandem with other filters, allowing you to create complex data schemas and transformation pipelines.

11.1 FilterMapper

Applies filters to an incoming mapping (e.g., `dict`).

When initialising the FilterMapper, provide a dict that assigns a filter chain to apply to each item.

When the FilterMapper gets applied to a mapping, the filter chain for each key is applied to the corresponding value in the mapping. A new mapping is returned containing the filtered values.

Invalid values in the result will be replaced with `None` (with a few exceptions, such as `filters.MaxBytes` which can be configured to return a truncated version of the incoming string instead of `None`).

```
import filters as f

filter_ = f.FilterMapper({
    'id':      f.Int,
    'subject': f.Unicode | f.NotEmpty | f.MaxLength(16),
})

# Incoming value is 100% valid.
runner = f.FilterRunner(filter_, {
    'id':      '42',
    'subject': 'Hello, world!',
})
assert runner.is_valid() is True
assert runner.cleaned_data == {
    'id':      42,
    'subject': 'Hello, world!',
}
```

(continues on next page)

(continued from previous page)

```
# Incoming value contains invalid items.
runner = f.FilterRunner(filter_, {
    'id':      '42',
    'subject': 'Did you know that Albert Einstein was born on Pi Day?',
})
assert runner.is_valid() is False
assert runner.cleaned_data == {
    'id':      42,
    'subject': None,
}
```

By default, the FilterMapper will ignore missing/unexpected keys, but you can configure this via the filter initialiser as well.

```
import filters as f

filter_ = f.FilterMapper(
    {
        'id':      f.Int,
        'subject': f.Unicode | f.NotEmpty | f.MaxLength(16),
    },

    # Only allow keys that we are expecting.
    allow_extra_keys = False,

    # All keys are required.
    allow_missing_keys = False,
)

# Incoming value is valid.
runner = f.FilterRunner(filter_, {
    'id':      '42',
    'subject': 'Hello, world!',
})
assert runner.is_valid() is True
assert runner.cleaned_data == {
    'id':      42,
    'subject': 'Hello, world!',
}

# Incoming value is missing required key and contains unexpected extra key.
runner = f.FilterRunner(filter_, {
    'id':          -1,
    'attachment':  'virus.exe',
})
assert runner.is_valid() is False
assert runner.cleaned_data == {
    'id':      -1,
    'subject': None,
}
```

You can also provide explicit key names for allowed extra/missing parameters:

```
import filters as f

filter_ = f.FilterMapper(
```

(continues on next page)

(continued from previous page)

```
    {
        'id':      f.Int,
        'subject': f.Unicode | f.NotEmpty | f.MaxLength(16),
    },

    # Ignore `attachment` if present; any other extra keys are invalid.
    allow_extra_keys = {'attachment'},

    # Only `subject` is optional.
    allow_missing_keys = {'subject'},
)

# Incoming value is valid.
runner = f.FilterRunner(filter_, {
    'id': 42,
    'attachment': 'signature.asc',
})
assert runner.is_valid() is True
assert runner.cleaned_data == {
    'id': 42,
    'subject': None,
    'attachment': 'signature.asc',
}

# Incoming value is missing required key and contains unexpected extra key.
runner = f.FilterRunner(filter_, {
    'from':        'admin@facebook.com',
    'attachment':  'virus.exe',
})
assert runner.is_valid() is False
assert runner.cleaned_data == {
    'id':         None,
    'subject':    None,
    'attachment': 'virus.exe'
}
```

Tip: This filter is often chained with `filters.JsonDecode`, when parsing a JSON object into a `dict`.

11.2 FilterRepeater

Applies a filter chain to every value in an incoming iterable (e.g., `list`) or mapping (e.g., `dict`).

When initialising the FilterRepeater, provide a filter chain to apply to each item.

When the FilterRepeater gets applied to an iterable or mapping, the filter chain gets applied to each value, and a new iterable or mapping of the same type is returned which contains the filtered values.

Invalid values in the result will be replaced with `None` (with a few exceptions, such as `filters.MaxBytes` which can be configured to return a truncated version of the incoming string instead of `None`).

```
import filters as f

filter_ = f.FilterRepeater(f.Int | f.Required)
```

(continues on next page)

(continued from previous page)

```
# Incoming value is 100% valid.
runner = f.FilterRunner(filter_, ['42', 86.0, 99])
assert runner.is_valid() is True
assert runner.cleaned_data == [42, 86, 99]

# Incoming value contains invalid values.
runner = f.FilterRunner(
    filter_,
    ['42', 98.6, 'not even close', 99, {12, 34}, None],
)
assert runner.is_valid() is False
assert runner.cleaned_data == [42, None, None, 99, None, None]
```

`FilterRepeater` can also process mappings (e.g., `dict`); it will apply the filters to every value in the mapping, preserving the keys.

```
import filters as f

filter_ = f.FilterRepeater(f.Int | f.Required)

# Incoming value is 100% valid.
runner = f.FilterRunner(filter_, {
    'alpha':   '42',
    'bravo':   86.0,
    'charlie': 99,
})
assert runner.is_valid() is True
assert runner.cleaned_data == {
    'alpha':   42,
    'bravo':   86,
    'charlie': 99,
}

# Incoming value contains invalid values.
runner = f.FilterRunner(filter_, {
    'alpha':   None,
    'bravo':   86.1,
    'charlie': 99
})
assert runner.is_valid() is False
assert runner.cleaned_data == {
    'alpha':   None,
    'bravo':   None,
    'charlie': 99,
}
```

Note: Note how this differs from `filters.FilterMapper` — `FilterRepeater` will apply the **same** filter chain to each item in the mapping, whereas `FilterMapper` allows you to specify a **different** filter chain to apply to each item based on its key.

11.3 FilterSwitch

Conditionally invokes a filter based on the output of a function.

`FilterSwitch` takes 2-3 parameters:

- `getter: Callable[[Any], Hashable]` - a function that extracts the comparison value from the incoming value. Whatever this function returns will be matched against the keys in `cases`.
- `cases: Mapping[Hashable, FilterCompatible]` - a mapping of possible return values from `getter` and the corresponding filter chains.
- `default: Optional[FilterCompatible]` - Filter chain that will be used if the return value from `getter` doesn't match any keys in `cases`.

When a `FilterSwitch` is applied to an incoming `value`:

1. The `getter` will be called and `value` will be passed in.
2. The return value from `getter` will be compared against the keys in `cases`:
    - If a match is found, the corresponding filter chain will be applied to `value`.

      Important: Note that the actual `value` gets passed to the filter chain, **not** the result from calling `getter`; the latter is **only** used to figure out which filter chain to use!

    - If no match is found, the `FilterSwitch` will check to see if it has a `default` filter chain:
        - If there is a `default` filter chain, that chain gets applied to the `value`.
        - If not, then the incoming value is invalid.

Example of a `FilterSwitch` that selects the correct filter to use based upon the incoming value's `name` item:

```
import filters as f
from operator import itemgetter

filter_ = f.FilterSwitch(
    # This function will extract the comparison value.
    getter=itemgetter('name'),

    # These are the cases that the comparison value might match.
    cases={
        # If ``value.name == 'price'`` use this filter:
        'price': f.FilterMapper({'value': f.Int | f.Min(0)}),

        # If ``value.name == 'colour'`` use this filter instead:
        'colour': f.FilterMapper({'value': f.Choice({'r', 'g', 'b'})}),
    },

    # (optional) If none of the above cases match, use this filter instead.
    default=f.FilterMapper({'value': f.Unicode}),
)

# Applies the 'price' filter:
runner = f.FilterRunner(filter_, {'name': 'price', 'value': '995'})
assert runner.is_valid() is True
assert runner.cleaned_data == {'name': 'price', 'value': 995}
```

(continues on next page)

(continued from previous page)

```
# Applies the 'colour' filter:
runner = f.FilterRunner(filter_, {'name': 'colour', 'value': 'b'})
assert runner.is_valid() is True
assert runner.cleaned_data == {'name': 'colour', 'value': 'b'}

# Applies the default filter:
runner = f.FilterRunner(filter_, {'name': 'size', 'value': 42})
assert runner.is_valid() is True
assert runner.cleaned_data == {'name': 'size', 'value': '42'}
```

Important: Note in the above example that the entire incoming dict gets passed to the corresponding filter chain, **not** the result of calling `itemgetter('name')`!

CHAPTER 12

Filterception

Just like any other filter, complex filters can be chained with other filters.

For example, to decode a JSON string that describes an address book card, the filter chain might look like this:

```
import filters as f

filter_ =\
   f.Unicode | f.Required | f.JsonDecode | f.Type(dict) | f.FilterMapper(
       {
           'name': f.Unicode | f.Strip | f.Required,
           'type': f.Unicode | f.Strip | f.Optional('person') |
               f.Choice({'business', 'person'}),

           # Each person may have multiple phone numbers, which must be
           # structured a particular way.
           'phone_numbers': f.Array | f.FilterRepeater(
               f.FilterMapper(
                   {
                       'label': f.Unicode | f.Required,
                       'country_code': f.Int,
                       'number': f.Unicode | f.Required,
                   },
                   allow_extra_keys=False,
                   allow_missing_keys=('country_code',),
               ),
           ),
       },
       allow_extra_keys=False,
       allow_missing_keys=False,
   )

runner = f.FilterRunner(
    filter_,
    '{"name": "Ghostbusters", "type": "business", "phone_numbers": '
    '[{"label": "office", "number": "555-2368"}]}'
```

(continues on next page)

(continued from previous page)

```
)
assert runner.is_valid() is True
assert runner.cleaned_data == {
    'name': 'Ghostbusters',
    'type': 'business',
    'phone_numbers': [
        {'label': 'office', 'country_code': None, 'number': '555-2368'},
    ],
}
```

CHAPTER 13

Official Extensions

The following filters are provided by the *Extensions framework* as official add-ons to the Filters library.

Note that extension filters are located in a different namespace; use `filters.ext` to access them instead of `filters`. For example:

```
import filters as f

# Standard filter
f.Unicode().apply('foo')

# Extension filter - note `f.ext`.
f.ext.Country().apply('pe')
```

13.1 Django Filters

Adds filters for Django-specific features. To install this extension:

```
pip install filters[django]
```

13.1.1 Model

Attempts to find a database record that matches the incoming value.

The filter initialiser accepts a few arguments:

- `model` (required) The Django model that will be queried.
- `field` (optional) The name of the field that will be matched against. If not provided, the default is `pk`.

You may also provide "predicates" to the initialiser that will allow you to further filter/customise the query as desired.

Here's an example:

```
import filters as f

filter_ = f.ext.Model(
  # Find a Post record with a ``slug`` that matches the input.
  model = Post,
  field = 'slug',

  # Add predicates to the query.
  filter={'published': True},
  exclude={'comments__isnull': True'},
  select_related=('author', 'comments'),
)

runner = f.FilterRunner(filter_, 'introducing-filters-library')
```

Any method in `QuerySet` can be used as a predicate so long as that method returns a `QuerySet` object (e.g., `filter` and `select_related` are valid predicates, but `count` and `update` are not).

Refer to the QuerySet API for more information.

13.2 ISO Filters

Adds filters for interpreting standard codes and identifiers. To install this extension:

```
pip install filters[iso]
```

13.2.1 Country

Interprets the incoming value as an ISO 3166-1 alpha-2 or alpha-3 country code.

The resulting value is a `iso3166.Country` object (provided by the iso3166 library).

```
import filters as f
from iso3166 import Country

runner = f.FilterRunner(f.ext.Country, 'nz')
assert runner.is_valid() is True
assert isinstance(runner.cleaned_data, Country) is True
assert runner.cleaned_data.name == 'New Zealand'
assert runner.cleaned_data.alpha2 == 'NZ'
assert runner.cleaned_data.alpha3 == 'NZL'
assert runner.cleaned_data.numeric == '554'
assert runner.cleaned_data.apolitical_name == 'New Zealand'

runner = f.FilterRunner(f.ext.Country, 'nzl')
assert runner.is_valid() is True
assert runner.cleaned_data.name == 'New Zealand'

runner = f.FilterRunner(f.ext.Country, 'xxxx')
assert runner.is_valid() is False

# Only ISO codes are accepted.
runner = f.FilterRunner(f.ext.Country, 'New Zealand')
assert runner.is_valid() is False
```

13.2.2 Currency

Interprets the incoming value as an ISO 4217 currency code.

The resulting value is a `moneyed.Currency` object (provided by the py-moneyed library).

```
import filters as f
from moneyed import Currency

runner = f.FilterRunner(f.ext.Currency, 'nzd')
assert runner.is_valid() is True
assert isinstance(runner.cleaned_data, Currency) is True
assert runner.cleaned_data.name == 'New Zealand Dollar'

runner = f.FilterRunner(f.ext.Currency, 'xxxx')
assert runner.is_valid() is False

# Only ISO codes are accepted.
runner = f.FilterRunner(f.ext.Currency, 'New Zealand Dollar')
assert runner.is_valid() is False
```

13.2.3 Locale

Interprets the incoming value as an IETF Language Tag (also known as BCP 47).

The resulting value is a `language_tags.Tag.Tag` object (provided by the language_tags library).

```
import filters as f
from language_tags.Tag import Tag

runner = f.FilterRunner(f.ext.Locale, 'en-nz')
assert runner.is_valid() is True
assert isinstance(runner.cleaned_data, Tag) is True
assert runner.cleaned_data.format == 'en-NZ'

runner = f.FilterRunner(f.ext.Locale, 'xx-XX')
assert runner.is_valid() is False

# Only ISO codes are accepted.
runner = f.FilterRunner(f.ext.Locale, 'English')
assert runner.is_valid() is False
```

CHAPTER 14

Filters

The Filters library provides an easy and readable way to create complex data validation and processing pipelines, including:

- Validating complex JSON structures in API requests or config files.
- Parsing timestamps and converting to UTC.
- Converting Unicode strings to NFC, normalizing line endings and removing unprintable characters.
- Decoding Base64, including URL-safe variants.

And much more!

The output from one filter can be "piped" into the input of another, enabling you to "chain" filters together to quickly and easily create complex data pipelines.

14.1 Examples

Validate a latitude position and round to manageable precision:

```
(
    f.Required |
    f.Decimal |
    f.Min(Decimal(-90)) |
    f.Max(Decimal(90)) |
    f.Round(to_nearest='0.000001')
).apply('-12.0431842')
```

Parse an incoming value as a datetime, convert to UTC and strip tzinfo:

```
f.Datetime(naive=True).apply('2015-04-08T15:11:22-05:00')
```

Convert every value in an iterable (e.g., list) to unicode and strip leading/trailing whitespace. This also applies Unicode normalization, strips unprintable characters and normalizes line endings automatically.

```
f.FilterRepeater(f.Unicode | f.Strip).apply([
    b'\xe2\x99\xaa ',
    b'\xe2\x94\x8f(\xc2\xb0.\xc2\xb0)\xe2\x94\x9b ',
    b'\xe2\x94\x97(\xc2\xb0.\xc2\xb0)\xe2\x94\x93 ',
    b'\xe2\x99\xaa ',
])
```

Parse a JSON string and check that it has correct structure:

```
(
    f.JsonDecode |
    f.FilterMapper(
        {
            'birthday':  f.Date,
            'gender':    f.CaseFold | f.Choice(choices={'m', 'f', 'x'}),

            'utcOffset':
                f.Decimal |
                f.Min(Decimal('-15')) |
                f.Max(Decimal('+15')) |
                f.Round(to_nearest='0.25'),
        },

        allow_extra_keys   = False,
        allow_missing_keys = False,
    )
).apply('{"birthday":"1879-03-14", "gender":"M", "utcOffset":"1"}')
```

14.2 Requirements

Filters is known to be compatible with the following Python versions:

- 3.11
- 3.10
- 3.9

Note: I'm only one person, so to keep from getting overwhelmed, I'm only committing to supporting the 3 most recent versions of Python. Filters may work in versions not listed here — there just won't be any test coverage to prove it

14.3 Installation

Install the latest stable version via pip:

```
pip install phx-filters
```

Important: Make sure to install *phx-filters*, **not** *filters*. I created the latter at a previous job years ago, and after I left they never touched that project again and stopped responding to my emails — so in the end I had to fork it

14.3.1 Extensions

The following extensions are available:

- Django Filters: Adds filters designed to work with Django applications. To install:

```
pip install phx-filters[django]
```

- ISO Filters: Adds filters for interpreting standard codes and identifiers. To install:

```
pip install phx-filters[iso]
```

Tip: To install multiple extensions, separate them with commas, e.g.:

```
pip install phx-filters[django,iso]
```

Happy filtering!