Wextracto Documentation

Release 0.7.5

Giles Brown

July 04, 2015

Contents

Wextracto is a framework for data extraction from web resources.

CHAPTER 1

Tutorial

The tutorial gives step-by-step instructions to get started with Wextracto.

1.1 Tutorial

1.1.1 Introduction

This tutorial shows you how to extract data from an HTML web page using Wextracto.

To work through the tutorial you need to download and install Python.

You also need to install Wextracto. If you can, you should install it into a virtual environment because this makes things easier to manage. The recommended way to install Wextracto is using pip:

```
$ pip install Wextracto
```

This will install the `wex` command:

```
$ wex --help
```

You are now ready to begin the tutorial.

1.1.2 Writing A Minimal Extractor

An extractor is a function that takes an HTTP response and returns values extracted from it. Our extractor is going to return the URL of the response. Write or copy the following into a file called `tutorial.py`:

```
def extract(response):
    return response.geturl()
```

The `response` parameter here is file-like object of the type used by the standard library urllib2.

Now we need to tell the `wex` command about our new extractor. We do this by creating a file called `entry_points.txt` with the following contents:

```
[wex]
extract = tutorial:extract
```

Now run `wex` with the following URL:

```
$ wex http://gilessbrown.github.io/cheeses/cheddar.html
"http://gilessbrown.github.io/cheeses/cheddar.html"
```

Congratulations, you have just written an extractor!

1.1.3 Selecting Elements

Python has a great library for processing XML and HTML data called lxml. We can use this library in our extractor.

Let's use a simple XPath expression to get some text from our chosen web page.

Edit `tutorial.py` to look like this:

```
from lxml.html import parse

def extract(response):
    tree = parse(response)
    return tree.xpath('//h1/text()')
```

Now re-run `wex` with the same URL we used previously:

```
$ wex http://gilessbrown.github.io/cheeses/cheddar.html
["\n\t\t\tCheddar\n\t\t"]
```

You may be wondering about the square brackets around the text. That is because `wex` serializes values using JSON. Our XPath expression returns a Python list which gives us the square brackets in JSON.

You may also have noticed the leading and trailing whitespace. We'll look at how to get rid of that in the next section.

1.1.4 Extracting Text

We normally want text we extract HTML elements to be space-normalized. This means runs of whitespace are converted into a single space character and leading and trailing whitespace is trimmed.

Wextracto provides the *`text`* function to return the space-normalized text for each selected element.

Here is what our extractor now looks like:

```
from lxml.html import parse
from wex.etree import text


def extract(response):
    tree = parse(response)
    return text(tree.xpath('//h1/text()'))
```

Let's run `wex` with the usual URL again to check the result:

```
$ wex http://gilessbrown.github.io/cheeses/cheddar.html
"Cheddar"
```

That is much tidier.

You may be wondering why we don't just use the XPath normalize-space function. There actually several reasons why we do not want to do this, most of which are specific to extracting text from HTML as opposed to XML:

The *`text`* function:

- understands <*br*> tags
- uses a unicode definition of whitespace (e.g. non breaking spaces)
- can work with multiple nodes in an node-set

1.1.5 Multiple Values

Often we want to extract multiple values from our web page. This is done by *yield*-ing values instead *return*-ing a single value.

So that we know which value is which we also label the values by yielding a name for the value at the same time.

Modify `tutorial.py` to yield the names and values:

```
from lxml.html import parse
from wex.etree import text


def extract(response):
    tree = parse(response)
    yield "name", text(tree.xpath('//h1'))
    yield "country", text(tree.xpath('//dd[@id="country"]'))
    yield "region", text(tree.xpath('//dd[@id="region"]'))
```

Now re-run `wex`:

```
$ wex http://gilessbrown.github.io/cheeses/cheddar.html
"name"      "Cheddar"
"country"   "England"
"region"    "Somerset"
```

Wextracto uses the tab character to separate the label from the value.

1.1.6 Errors

Yielding multiple values from an extractor is ok if all the values extract successfully. Unfortunately, if they don't, we don't get the remaining values even if they would have extracted successfully.

Let's extend the extractor we wrote in the previous section and add a new attribute. This time let's deliberately make a mistake so we can see what happens:

```
from wex.etree import parse, text

def extract(response):
    tree = parse(response)
    yield "name", text(tree.xpath('//h1/text()'))
    yield "whoops", 1/0
    yield "country", text(tree.xpath('//dd[@id="country"]'))
    yield "region", text(tree.xpath('//dd[@id="region"]'))
```

Now re-run `wex`:

```
$ wex http://gilessbrown.github.io/cheeses/cheddar.html
"name"      "Cheddar"
#ZeroDivisionError('integer division or modulo by zero',)!
```

The `#` and `!` at the start and end of that final line is Wextracto's way of telling us that we ended up with a value that was not JSON encodable. In this case because there was a ZeroDivisionError exception.

Notice how we didn't see more values following the exception.

What we'd really like is for each attribute to be extracted in such a way that an exception while extracting one attribute doesn't mean the others don't get extracted.

To make that happen we'll need each attribute to be extracted in its own function. In the next section we'll see how Wextracto helps you do that.

1.1.7 Named

Wextracto provides a function specifically for extracting named values and it is called *wex.extractor.named()*. This lets you create a collection of extractors each of which has a name. The class instance is itself callable it it yields the results of each extractor in the collection together with its name.

Extractors can be added to the collection by decorating them with the collections **:method:'.Named.add'** method.

Copy the code from here:

```
from wex.extractor import named
from wex.etree import xpath, text


extract = named()


@extract.add
def name(response):
    return text(xpath('//h1')(response))


@extract.add
def whoops(response):
    return 1/0


@extract.add
def country(response):
    return text(xpath('//dd[@id="country"]')(response))


@extract.add
def region(response):
    return text(xpath('//dd[@id="region"]')(response))
```

You may notice that we have switched from calling `.xpath()` on the element tree to using the `wex.etree.xpath` function. The function produce by calling this function knows when to parse the response so we don't need to organize that.

Let's try running our extractor now and see what we get:

```
$ wex http://gilessbrown.github.io/cheeses/cheddar.html
"country"    "England"
"whoops"     #ZeroDivisionError('integer division or modulo by zero',)!
"region"     "Somerset"
"name"       "Cheddar"
```

Now we've got something for all the named values we wanted and in addition it tells which extractor isn't working.

1.1.8 Composing Extractors

If you need to write a lot of extractors then you may find that the using the decorator syntax for *wex.extractor.Named* leads to a lot of boilerplate code. Fortunately there is an alternative.

If you look at the examples in the *previous section*, you will see that the extractors (apart from `whoops`) all look something like:

```
def xyz(response):
    return text(xpath(...)(response))
```

It turns out this kind of pattern is very common in writing extractors. A technique called function composition lets us define these extractor functions very succinctly.

In Wextracto function composition is performed with the `|` operator (like Unix pipes).

So we can define the extractor above as:

```
xyz = xpath(...) | text
```

We can pass these composed functions directly into the constructor for *wex.extractor.Named* and get something that looks like:

```
from wex.extractor import named
from wex.etree import xpath, text


extract = named(name = xpath('//h1') | text,
                country = xpath('//dd[@id="country"]') | text,
                region = xpath('//dd[@id="region"]') | text)
```

As you can see, this is a very compact representation for simple extractors.

1.1.9 Labelling

So far we've only been extracting data from one web page, but eventually we'd like to move on to extracting from multiple pages. Let's see what happens:

```
$ wex http://gilessbrown.github.io/cheeses/cheddar.html http://gilessbrown.github.io/cheeses/brie.htr
"country"   "tEngland"
"region"    "Somerset"
"name"      "Cheddar"
"country"   "France"
"region"    "Seine-et-Marne"
"name"      "Brie"
```

Oh dear. It isn't very clear which value came from which web page.

We can fix this by using the `wex.extractor.label()` function:

```
from wex.extractor import label, named
from wex.url import url
from wex.etree import xpath, text


attrs = named(name = xpath('//h1') | text,
              country = xpath('//dd[@id="country"]') | text,
              region = xpath('//dd[@id="region"]') | text)

extract = label(url)(attrs)
```

The code here is going to label the output with the URL of the current response.

Let's try it:

```
$ wex http://gilessbrown.github.io/cheeses/cheddar.html http://gilessbrown.github.io/cheeses/brie.htm
"http://gilessbrown.github.io/cheeses/cheddar.html" "country"       "England"
"http://gilessbrown.github.io/cheeses/cheddar.html" "region"        "Somerset"
"http://gilessbrown.github.io/cheeses/cheddar.html" "name"  "Cheddar"
"http://gilessbrown.github.io/cheeses/brie.html"    "country"       "France"
"http://gilessbrown.github.io/cheeses/brie.html"    "region"        "Seine-et-Marne"
"http://gilessbrown.github.io/cheeses/brie.html"    "name"  "Brie"
```

As before, the labels are tab delimited.

1.1.10 Multiple Entities

In the *Labelling* section we saw how we can label values with the URL from which they came, but sometimes you get multiple entities on the same web page and they each have their own set of attributes.

Let's try our extractor on that kind of page:

```
$ wex http://gilessbrown.github.io/cheeses/gloucester.html
"http://gilessbrown.github.io/cheeses/gloucester.html"  "country"       #MultipleValuesError()!
"http://gilessbrown.github.io/cheeses/gloucester.html"  "region"        #MultipleValuesError()!
"http://gilessbrown.github.io/cheeses/gloucester.html"  "name"  #MultipleValuesError()!
```

Oh dear. What can we do? Well if we visit that web page in a browser and view the source we find that each `<h1>` helpfully has a International Cheese Identification Number (ICIN) as an attribute.

So what we can do is re-write the extractor to visit each `<h1>` and extract the data we want relative to that element.

Here is what the code looks like:

```
from wex.extractor import named, labelled
from wex.iterable import one
from wex.etree import xpath, text

cheeses = xpath('//h1[@data-icin]')
icin_attr = xpath('@data-icin') | one

attrs = named(name = text,
              country = xpath('following::dd[@id="country"][1]') | text,
              region = xpath('following::dd[@id="region"][1]') | text)

extract_cheese = labelled(icin_attr, attrs)


def extract(response):
    for cheese in cheeses(response):
        for item in extract_cheese(cheese):
            yield item
```

And then we run `wex`:

```
$ wex http://gilessbrown.github.io/cheeses/gloucester.html
"SNGGLCD7DDFD41"    "country"       "England"
"SNGGLCD7DDFD41"    "region"        "Gloucestershire"
"SNGGLCD7DDFD41"    "name"  "Single Gloucester"
"DBLGLCCECAA22C"    "country"       "England"
"DBLGLCCECAA22C"    "region"        "Gloucestershire"
"DBLGLCCECAA22C"    "name"  "Double Gloucester"
```

1.1.11 What Next?

- Read the *User Guide*.
- Read the source code.

CHAPTER 2

User Guide

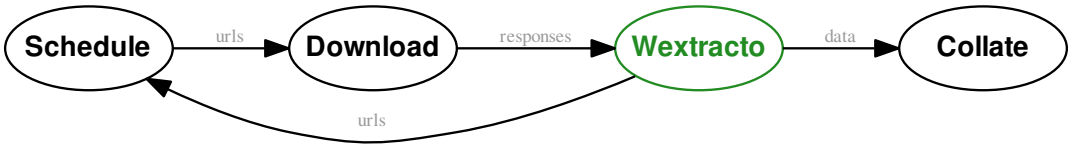
The user guide gives broad general guidance about how to use Wextracto.

2.1 User Guide

2.1.1 What is Wextracto?

Wextracto is a Python package designed to be the core of a web crawling/scraping system.

To understand how it fits in, let us look at the general architecture of a web crawling/scraping based on Wextracto.



This architecture has these components:

Schedule This component manages the URLs to be downloaded. The goal is to keep track which URLs you have downloaded and which URLs you have yet to download.

Download This component requests web pages and stores the responses for use by the *Wextracto* component.

Wextracto This component reads the stored responses and extracts URLs and data. URLs are routed to the *Schedule* component.

Collate This component receives data from the *Wextracto* component and organizes it ready for use. Organizing the data might involve storing it in a database.

Each of the other three components (*Schedule*, *Download* and *Collate*) can be implemented in multiple ways depending on the requirements of the crawl system. Keeping just the core in Wextracto gives better seperation of concerns

2.1.2 Interfaces

In the *architecture* diagram you can see Wextracto has three data flows. One incoming (*responses*) and two outgoing (*urls* and *data*).

Responses

Although Wextracto can download and extract in one go, it is designed to be used in system where the downloading is done separately from the extraction.

Having the download separate from extraction is generally helpful because:

- it allows us to repeat the extraction process exactly for problem finding
- it gives us easy access to large sample data sets
- it can make the management of I/O in the system clearer

Wextracto can process responses that look like HTTP responses (headers then content). For example:

```
$ curl -D- http://httpbin.org/ip
HTTP/1.1 200 OK
Connection: keep-alive
Server: gunicorn/18.0
Date: Tue, 30 Dec 2014 19:32:18 GMT
Content-Type: application/json
Content-Length: 32
Access-Control-Allow-Origin: *
Access-Control-Allow-Credentials: true
Via: 1.1 vegur

{
  "origin": "67.180.76.235"
}
```

Although most *extractors* will require the presence of a custom HTTP header, `X-wex-request-url`, that contains the requested URL. Any component preparing responses for processing using Wextracto should add this header.

A request can lead to multiple responses, each with their own URL. In the case where the response URL is not the same as the request URL an additional header, `X-wex-url`, that contains the responses URL.

Wextracto looks for responses such as these in files that have a `.wexin` extension. It can also read a .tar file containing files with the same extension.

URLs & Data

The handling of *urls* and *data* is described in the *reference*.

CHAPTER 3

Reference

The reference gives detailed information on the modules and classes provided by the Wextracto package.

3.1 Reference

This is the reference for the Wextracto web data extraction package.

3.1.1 Command

The `wex` command extracts data from HTTP-like responses. These responses can come from files, directories or URLs specified on the command line. The command calls any *`extractors`* that have been *`registered`* and writes any data extracted as *`output`*.

The output and input can be saved, using the `--save` or `--save-dir` command line arguments. This is useful for regression testing. existing extractor functions. The test are run using *`py.test`*.

For the complete list of command line arguments run:

```
$ wex --help
```

3.1.2 Registering Extractors

The simplest way to register *`extractors`* is to have a file named `entry_points.txt` in the current directory. This file should look something like this:

```
[wex]
.example.net = mymodule:extract_from_example_net
```

The `[wex]` section heading tells Wextracto that the following lines register extractors.

Extractors are registered using `name = value` pairs. If the name starts with `.` then the extractor is only applied to responses from domain names that match that name. Our example would match responses from `www.example.net` or `example.net`.

If the name does not start with `.` it will be applied responses whatever their domain.

You can register the same extractor against multiple domain names by having multiple lines with the same value but different names.

This is exactly the same format and content that you would use in the `entry_points` parameter for a setup function, if and when you want to package and your extractor functions.

3.1.3 Extractor

An extractor is a callable that returns or yields data. For example:

```
def extract(response):
    return "something"
```

The `response` parameter here is an instance of *wex.response.Response*.

Extractors can be combined in various ways.

class `wex.extractor.`**`Named`**(***kw*)

A extractor that is a collection of named extractors.

Extractors can be added to the collection on construction using keyword arguments for the names or they can be added using *add()*.

The names are labels in the output produced. For example, an extractor function `extract` defined as follows:

```
extract = Named(
    name1 = (lambda response: "one"),
    name2 = (lambda response: "two"),
)
```

Would produce the extraction output something like this:

```
$ wex http://example.net/
"name1"    "one"
"name2"    "two"
```

The ordering of sub-extractor output is arbitrary.

`add`(*extractor*, *label=None*)

Add an attribute extractor.

Parameters

- **`extractor`** (*callable*) – The extractor to be added.
- **`label`** (*str*) – The label for the extractor. This may be `None` in which case the extractors `__name__` attribute will be used.

This method returns the extractor added. This means it can also be used as a decorator. For example:

```
attrs = Named()

@attrs.add
def attr1(response):
    return "one"
```

`wex.extractor.`**`chained`**(**extractors*)

Returns an extractor that chains the output of other extractors.

The output is the output from each extractor in sequence.

Parameters **`extractors`** – an iterable of extractor callables to chain

For example an extractor function `extract` defined as follows:

```
def extract1(response):
    yield "one"

def extract2(response):
    yield "two"
```

```
extract = chained(extract1, extract2)
```

Would produce the following extraction output:

```
$ wex http://example.net/
"one"
"two"
```

wex.extractor.**labelled**(**args*)
Returns an extractor decorator that will label the output an extractor.

Parameters **literals_or_callables** – An iterable of labels or callables.

Each item in `literals_or_callables` may be a literal or a callable. Any callable will called with the same parameters as the extractor and whatever is returned will by used as a label.

For example an extractor function `extract` defined as follows:

```
def extract1(response):
    yield "one"


def label2(response):
    return "label2"


extract = label("label1", label2)(extract1)
```

Would produce the following extraction output:

```
$ wex http://example.net/
"label1"    "label2"    "one"
```

Note that if any of the labels are false then no output will be generated from that extractor.

wex.extractor.**named**(***kw*)
Returns a *Named* collection of extractors.

3.1.4 Element Tree

Composable functions for extracting data using lxml.

wex.etree.**base_url_pair_getter**(*get_url*)
Returns a function for gettting a tuple of *(base_url, url)* when called with an etree *Element* or *ElementTree*.

In the returned pair *base_url* is the value returned from *:func:get_base_url* on the etree *Element* or *ElementTree*. There second value is the value returned by calling the *get_url* on the same the same etree *Element* or *ElementTree*, joined to the *base_url* using *urljoin*. This allows *get_url* to return a relative URL.

wex.etree.**css**(*expression*)
Returns a *composable* callable that will select elements defined by a CSS selector expression.

Parameters **expression** – The CSS selector expression.

The callable returned accepts a *wex.response.Response*, a list of elements or an individual element as an argument.

wex.etree.**drop_tree**(**selectors*)
Return a function that will remove trees selected by *selectors*.

wex.etree.**href_any_url**
A wex.composed.ComposedFunction that returns the absolute URL from an href attribute.

wex.etree.**href_url**
A wex.composed.ComposedFunction that returns the absolute URL from an href attribute as long as it is from the same domain as the base URl of the response.

wex.etree.**href_url_same_suffix**
A wex.composed.ComposedFunction that returns the absolute URL from an href attribute as long as it is from the same public suffix as the base URl of the response.

wex.etree.**itertext**(**tags*, ***kw*)
Return a function that will return an iterator for text.

wex.etree.**same_domain**(*url_pair*)
Return second url of pair if both are from same domain.

wex.etree.**same_suffix**(*url_pair*)
Return second url of pair if both have the same public suffix.

wex.etree.**src_url**
A wex.composed.ComposedFunction that returns the absolute URL from an src attribute.

wex.etree.**text**
Alias for *normalize-space | list2set*

wex.etree.**text_content**
Return text content from an object (typically node-set) excluding from content from within <*script*> or <*style*> elements.

wex.etree.**xpath**(*expression*, *namespaces={u're': u'http://exslt.org/regular-expressions'}*)
Returns *composable* callable that will select elements defined by an XPath expression.

Parameters

- **expression** – The XPath expression.
- **namespaces** – The namespaces.

The callable returned accepts a *wex.response.Response*, a list of elements or an individual element as an argument.

For example:

```
>>> from lxml.html import fromstring
>>> tree = fromstring('<h1>Hello</h1>')
>>> selector = xpath('//h1')
```

3.1.5 Regular Expressions

wex.regex.**re_group**(*pattern*, *group=1*, *flags=0*)
Returns a *composable* callable that extract the specified group using a regular expression.

Parameters

- **pattern** – The regular expression.
- **group** – The group from the MatchObject.
- **flags** – Flags to use when compiling the pattern.

wex.regex.**re_groupdict**(*pattern*, *flags=0*)
Returns a *composable* callable that extract the a group dictionary using a regular expression.

Parameters

- **pattern** – The regular expression.
- **flags** – Flags to use when compiling the pattern.

3.1.6 String Functions

wex.string.**partition**(*separator*, ***kw*)
Returns a function that yields tuples created by partitioning text using *separator*.

3.1.7 Iterables

Helper functions for things that are iterable

exception wex.iterable.**MultipleValuesError**
More than one value was found when one or none were expected.

exception wex.iterable.**ZeroValuesError**
Zero values were found when at least one was expected.

wex.iterable.**islice**(**islice_args*)
Returns a function that will perform itertools.islice on its input.

wex.iterable.**one**(*iterable*)
Returns an item from an iterable of exactly one element.

If the iterable comprises zero elements then *ZeroValuesError* is raised. If the iterable has more than one element then *MultipleValuesError* is raised.

wex.iterable.**one_or_none**(*iterable*)
Returns one item or None from an iterable of length one or zero.

If the iterable is empty then None is returned.

If the iterable has more than one element then *MultipleValuesError* is raised.

wex.iterable.**first**(*iterable*)
Returns first item from an iterable.

Parameters **iterable** – The iterable.

If the iterable is empty then None is returned.

wex.iterable.**flatten**(*iterable*, *yield_types*)
Yield objects from all sub-iterables from obj.

3.1.8 URLs

class wex.url.**Method**(*scheme*, *name*, *args=None*)
Method objects 'get' responses from a url.

The Method object looks-up the correct implementation based on its name and the scheme of the url.

The default method name is 'get'. Other method names can be specified in the fragment of the url.

get(*url*, ***kw*)
Get responses for 'url'.

class wex.url.**URL**
URL objects.

fragment_dict
Client side data dict represented as JSON in the fragment.

get(**kw)
Get *url* using the appropriate *Method*.

method
The *Method* for this URL.

3.1.9 Other Methods

class wex.form.**ParserReadable**(*readable*)
Readable that feeds a parser as it is reads.

wex.form.**form_values**(*self*)
Return a list of tuples of the field values for the form. This is suitable to be passed to `urllib.urlencode()`.

class wex.ftp.**RETRReadable**(*ftp*, *basename*)
Just like ftplib.FTP.retrbinary, but implements read and readline.

wex.ftp.**close_on_empty**(*unbound*)
Calls 'close' on first argument when *method* return something falsey.

The first argument is presumed to the *self*.

wex.ftp.**format_header**()
S.format(*args, **kwargs) -> unicode

Return a formatted version of S, using substitutions from args and kwargs. The substitutions are identified by braces ('{' and '}').

wex.ftp.**format_status_line**()
S.format(*args, **kwargs) -> unicode

Return a formatted version of S, using substitutions from args and kwargs. The substitutions are identified by braces ('{' and '}').

wex.ftp.**get**(*url*, *recipe*, **kw)
Recipe for an FTP get.

Functions for getting responses for HTTP urls.

wex.http.**format_header**()
S.format(*args, **kwargs) -> unicode

Return a formatted version of S, using substitutions from args and kwargs. The substitutions are identified by braces ('{' and '}').

wex.http.**format_status_line**()
S.format(*args, **kwargs) -> unicode

Return a formatted version of S, using substitutions from args and kwargs. The substitutions are identified by braces ('{' and '}').

wex.http.**readable_from_response**(*response*, *url*, *decode_content*, *context*)
Make an object that is readable by *Response*.from_file.

wex.http.**request**(*url*, *method*, *session=None*, **kw)
Makes an HTTP request following redirects.

3.1.10 Sitemaps

Extractors for URLs from /robots.txt and sitemaps.

`wex.sitemaps.`**`urls_from_robots_txt`**(*response*)
Yields sitemap URLs from "/robots.txt"

`wex.sitemaps.`**`urls_from_sitemaps = Chained([<function urls_from_robots_txt at 0x7f401c0c6398>, <function urls_fro`**
Extractor that combines *`urls_from_robots_txt()`* and *`urls_from_urlset_or_sitemapindex()`*.

`wex.sitemaps.`**`urls_from_urlset_or_sitemapindex`**(*response*)
Yields URLs from `<urlset>` or `<sitemapindex>` elements as per sitemaps.org.

3.1.11 Response

class `wex.response.`**`Response`**(*content*, *headers*, *url*, *code=None*, ***kw*)
A urllib2 style Response with some extras.

Parameters

- **`content`** – A file-like object containing the response content.
- **`headers`** – An HTTPMessage containing the response headers.
- **`url`** – The URL for which this is the response.
- **`code`** – The status code recieved with this response.
- **`protocol`** – The protocol received with this response.
- **`version`** – The protocol version received with this response.
- **`reason`** – The reason received with this response.
- **`request_url`** – The URL requested that led to this response.

`seek`(*offset=0*, *whence=0*)
Seek the content file position.

Parameters

- **`offset`** (*int*) – The offset from whence.
- **`whence`** (*int*) – 0=from start,1=from current position,2=from end

3.1.12 Composed

Wextracto uses Function composition as an easy way to build new functions from existing ones:

```
>>> from wex.composed import compose
>>> def add1(x):
...     return x + 1
...
>>> def mult2(x):
...     return x * 2
...
>>> f = compose(add1, mult2)
>>> f(2)
6
```

Wextracto uses the pipe operator, `|`, as a shorthand for function composition.

This shorthand can be a powerful technique for reducing boilerplate code when used in combination with *named()* extractors:

```
from wex.etree import css, text
from wex.extractor import named

attrs = named(title = css('h1') | text
              description = css('#description') | text)
```

class wex.composed.**ComposedCallable**(**functions*)

A callable, taking one argument, composed from other callables.

```
    def mult2(x):
        return x * 2

    def add1(x):
        return x + 1

    composed = ComposedCallable(add1, mult2)

    for x in (1, 2, 3):
        assert composed(x) == mult2(add1(x))
```

ComposedCallable objects are *composable*. It can be composed of other ComposedCallable objects.

wex.composed.**composable**(*func*)

Decorates a callable to support function composition using `|`.

For example:

```
    @Composable.decorate
    def add1(x):
        return x + 1

    def mult2(x):
        return x * 2

    composed = add1 | mult2
```

wex.composed.**compose**(**functions*)

Create a *ComposedCallable* from zero more functions.

3.1.13 Output

Extracted data values are represented with tab-separated fields. The right-most field on each line is the value, all preceding fields are labels that describe the value. The labels and the value are all JSON encoded.

So for example, a value 9.99 with a labels `product` and `price` would look like:

```
"product"   "price" 9.99
```

And we could decode this line with the following Python snippet:

```
>>> import json
>>> line = '"product"\t"price"\t9.99\n'
>>> [json.loads(s) for s in line.split('\t')]
[u'product', u'price', 9.99]
```

Using tab-delimiters is convenient for downstream processing using Unix command line tools such as **cut** and **grep**.

URL Labelling

The convention for Wextracto is that any URL that should be downloaded is has the left-most label `url`. For example:

```
"url"    "http://example.net/some/url"
```

Data Labelling

If you are extracting multiple types of data (for example people and addresses) then a good labelling scheme is important.

It is a good idea to label the extracted values so that you can sort them easily using the Unix **sort** command.

An example of a labelling scheme that allows this would be:

```
{type}      {identifier}     {attribute}      {value}
```

So we might end up with output that look like this:

```
"person"    "http://example.net/person/1"   "name"  "Tom Bombadil"
"person"    "http://example.net/person/1"   "email" "tom1@example.net"
"address"   "http://example.net/address/2"  "city"  "New York"
"address"   "http://example.net/address/2"  "postal code"   "10001"
"person"    "http://example.net/person/3"   "name"  "Jack Sprat"
"person"    "http://example.net/person/3"   "email" "jack3@example.net"
"address"   "http://example.net/address/4"  "city"  "London"
"address"   "http://example.net/address/4"  "postal code"   "E14 5AB"
```

With output like this we can easily sort and group it.

3.1.14 Regression Tests

When maintaining extractors it can be helpful to have some sample input and output so that regression testing can be performed when we need to change the extractors.

Wextracto supports this by using the `--save` or `--save-dir` options to the *wex* command. This option saves both the input and output to a local directory.

This input and output can then be used for comparison with the current extractor output.

To check compare current output against saved output run py.test like so:

```
$ py.test saved/
```

W

U

W

X

Z