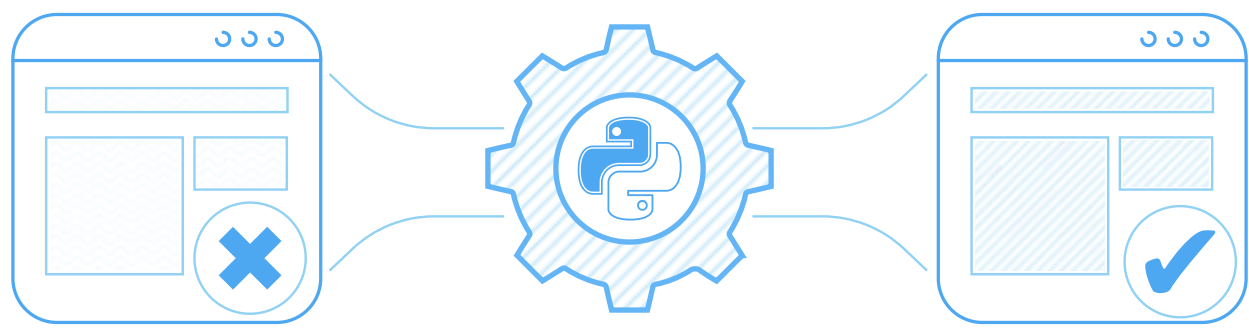


# How to Ensure Web Scrapped Data Quality

by Bernardas Ališkauskas   Aug 15, 2024   #Python   #Data Parsing      



Data on the public web often comes in unpredictable formats, types and structures. To ensure data quality when web scraping we cannot simply trust that our code will understand any scraped web page without issues. So, various validation tools can be used to test and validate scraper parsing logic.

In this tutorial, we'll take a look at how we can use Python tools such as [Cerberus](#) to validate and test scrapped data and tools like [Pydantic](#) to enforce data types and even normalize data values.

To illustrate common data validation challenges we'll be scraping [Glasdoor.com](#) for company overview data.

## Guide for scraping Glasdoor

If you're interested in scraping other Glasdoor details, see our complete scraping guide for this target



### Data Quality Challenges

- Setup
  - Soft Validation with Cerberus
    - Defining Schema
    - Creating Validation Rules
    - Real Life Example
  - Typed Validation with Pydantic
    - Transforming and Interpreting
  - Cerberus or Pydantic?
  - Scraped Data Validation Summary

JOIN THE NEWSLETTER

## Data Quality Challenges

When web scraping we're working with a public resource that we have no control over, so we cannot trust the format and quality of the collected data. In other words, we can never be sure when a website would change something like a date format - from `2022-11-26` to `2022/11/26` - or use different number formatting - from `10 000` to `10,000` .

To approach data quality in web scraping we can take advantage of several tools to test, validate and even transform scrapped data.

For example, we can write a validation function to check that all `date` fields follow `<4-digit-number>-<2-digit-number>-<2-digit-number>` pattern or even automatically correct it to a standard value if possible.

## Setup

In this article, we'll be using a few Python packages:

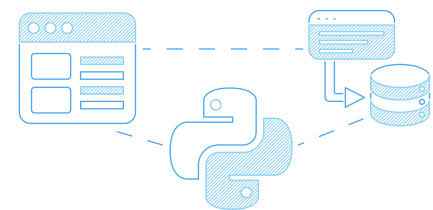
- [Cerberus](#) for schema-based data validations
- [Pydantic](#) for strict type-based data validation.

We'll also write a short example scraper using popular web scraping packages:

- [httpx](#) as a HTTP client to collect public HTML pages.
- [parsel](#) as HTML parsing library to extract details from HTML pages.

## Hands on Python Web Scraping Tutorial and Example Project

If you're completely new to web scraping in Python see our complete introduction article which covers these two packages we'll be using in this tutorial.



We can install all of these packages using `pip` console command:

```
$ pip install cerberus pydantic httpx parsel
```

## Soft Validation with Cerberus

To start let's take a look at [Cerberus](#) - a popular data validation library in Python - and how can we use it with web scrapped data.

To start using Cerberus all we have to do is define some rules and apply them to our data.

### Defining Schema

To validate data Cerberus needs a data schema which contains all of the validation rules and structure expectations. For example:

```
from cerberus import Validator

schema = {
    "name": {
        # name should be a string
        "type": "string",
        # between 2 and 20 characters
        "minlength": 3,
        "maxlength": 20,
        "required": True,
    },
}

v = Validator(schema)
# this will pass
v.validate({"name": "Homer"})
print(v.errors)

# This will not
v.validate({"name": "H"})
print(v.errors)
# {'name': ['min length is 3']}
```

Cerberus comes with a lot of predefined rules like `minlength`, `maxlength` and many others which can be found in the [validation rules docs](#). Though, here are some commonly used ones:

- [allowed](#) - validates to a list of allowed values.
- [contains](#) - validates that value contains some other value.
- [required](#) - ensures that value is present. For example, fields like `id` are often required in scraping.
- [depends](#) - ensures that related fields are present. For example, if product discount price is present, the full price should be present as well.
- [regex](#) - check value to match specified regular expressions pattern. This is very useful for fields like phone numbers or emails.

However, to truly take advantage of this we can define our own rules!

## Creating Validation Rules

Web scrapped data can be quite complex. For example, name field alone can come in many shapes and forms.

Using Cerberus we can provide our own validation function in Python that can validate data values for complex issues.

```
from cerberus import Validator

def validate_name(field, value, error):
    if "." in value:
        error(field, f"contains a dot character: {value}")
    if value.lower() in ["classified", "redacted", "missing"]:
        error(field, f"redacted value: {value}")
    if "<" in value.lower() and ">" in value.lower():
        error(field, f"contains html nodes: {value}")

schema = {
    "name": {
        # name should be a string
        "type": "string",
        # between 2 and 20 characters
        "minlength": 2,
        "maxlength": 20,
        # extra validation
        "check_with": validate_name,
    },
}

v = Validator(schema)

v.validate({"name": "H."})
print(v.errors)
# {'name': ['contains a dot character: H.']}
```

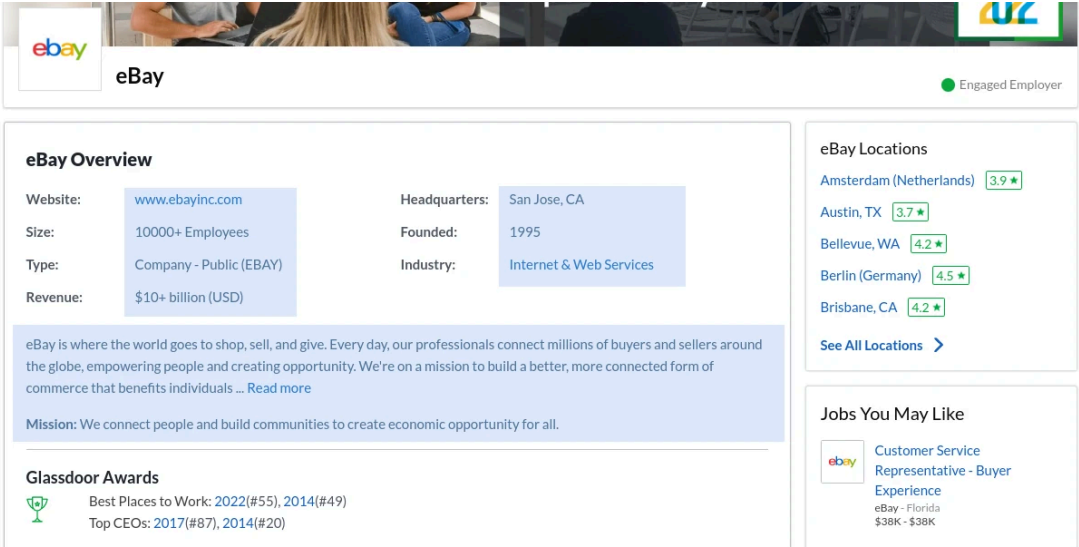
```
v.validate({"name": "Classified"})
print(v.errors)
# {'name': ['redacted value: Classified']}
```

```
v.validate({"name": "<a>Homer</a>"})
print(v.errors)
# {'name': ['contains html nodes: <a>Homer</a>']}
```

In the example above, we added our own validation function for the name field which does more advanced string validation like checking for invalid values and potential html parsing errors.

## Real Life Example

Let's put together a quick example web scraper that we'll validate using Cerberus. We'll be scraping company overview data on [Glassdoor.com](#):



In this example, we'll focus on scraping basic company details.

As an example, let's scrape [Ebay's](#) glassdoor profile:

```
import httpx
from parsel import Selector

response = httpx.get("https://www.glassdoor.com/Overview/Working-at-eBay-
EI_IE7853.11,15.htm")
selector = Selector(response.text)
overview_rows = selector.css('[data-test="employerOverviewModule"]>ul>li>div')
data = {}
for row in overview_rows:
    label = row.xpath("@data-test").get().split('-', 1)[-1]
    value = row.xpath("text()").get()
    data[label] = value
print(data)
```

This short scraper will scrape the basic company overview details:

```
{
  "headquarters": "San Jose, CA",
  "size": "10000+ Employees",
  "founded": "1995",
  "type": "Company - Public (EBAY)",
  "revenue": "$10+ billion (USD)"
}
```

We can already see how vague the raw fields on this page are. Bigger companies will have `revenue` in billions and different regions will have different currencies etc. The data values are human-readable but not normalized for machine interpretation.

Let's parse this data out to something more general and validate it using Cerberus:

```

from cerberus import Validator

def parse_and_validate(data):
    schema = {}
    parsed = {}

    # company size is integer between 1 employee and several thousand
    parsed["size"] = int(data["size"].split()[0].strip("+"))
    schema["size"] = {"type": "integer", "min": 1, "max": 20_000}

    # founded date is a realistic year number
    parsed["founded"] = int(data["founded"])
    schema["founded"] = {"type": "integer", "min": 1900, "max": 2022}

    # headquarter details consist of city and state/province/country
    hq_details = data["headquarters"].split(", ")
    parsed["hq_city"] = hq_details[0] if hq_details else None
    parsed["hq_state"] = hq_details[1] if len(hq_details) > 1 else None
    schema["hq_city"] = {"type": "string", "minlength": 2, "maxlength": 20}
    schema["hq_state"] = {"type": "string", "minlength": 2, "maxlength": 2}

    # let's presume we only want to ensure we're scraping US and GB companies:
    parsed["revenue_currency"] = data["revenue"].split("(")[-1].strip("(")")
    schema["revenue_currency"] = {"type": "string", "allowed": ["USD", "GBP"]}

    validator = Validator(schema)
    if not validator.validate(parsed):
        print("failed to validate parsed data:")
        for key, error in validator.errors.items():
            print(f"{key}={parsed[key]} got error: {error}")
    return parsed

ebay_data = {
    "headquarters": "San Jose, CA",
    "size": "10000+ Employees",
    "founded": "1995",
    "type": "Company - Public (EBAY)",
    "revenue": "$10+ billion (USD)",
}
print(parse_and_validate(ebay_data))
# will print:
{
    "size": 10000,
    "founded": 1995,
    "hq_city": "San Jose",
    "hq_state": "CA",
    "revenue_currency": "USD"
}

```

Above, we wrote our parser that converts raw company overview data to something more standard and concrete. As we can see it works great with our Ebay profile example, though we wrote validation to ensure that our parser delivers a consistent quality of data. With a sample size of one we can't say that our validator is doing it's job well.

Let's confirm our validation schema by expanding our test sample with another company page - [Tesco](#)

```
tesco_data = {
    "headquarters": "Welwyn Garden City, United Kingdom",
    "size": "10000+ Employees",
    "founded": "1919",
    "type": "Company - Private",
    "revenue": "Unknown / Non-Applicable"
}
print(parse_and_validate(tesco_data))
# will print:
# failed to validate parsed data:
# hq_state=United Kingdom got error: ['max length is 2']
# revenue_currency=Unknown / Non-Applicable got error: ['unallowed value Unknown /
Non-Applicable']
{
    "size": 10000,
    "founded": 1919,
    "hq_city": "Welwyn Garden City",
    "hq_state": "United Kingdom",
    "revenue_currency": "Unknown / Non-Applicable"
}
```

As we can see our validator indicates that our parser is not doing such a good job with Tesco's page as it did with Ebay's.

Using Cerberus we can quickly define how our desired output should look which helps to develop and maintain complex data parsing operations which in turn ensures consistent data quality.

Cerberus offers more features like [value defaults](#), [normalization rules](#) and powerful [extension support](#).

Next, let's take a look at a different validation technique - strict data types.

## Typed Validation with Pydantic

Our Cerberus validation let us know about data parsing inconsistencies but if we're building a reliable data API we might need something more strict.

[Pydantic](#) allows defining strict types and validation rules using Python's type hint system. Unlike cerberus Pydantic is much more strict and raises errors when encountering invalid data values.

Let's take a look at our Glassdoor example through the lens of Pydantic. Validation in Pydantic is done through explicit `BaseModel` objects and python type hints.

For example, our Glassdoor company overview validator would look something like this:



```

from typing import Optional
from pydantic import BaseModel, validator

# to validate data we must create a Pydantic Model:
class Company(BaseModel):
    # define allowed field names and types:
    size: int
    founded: int
    revenue_currency: str
    hq_city: str
    # some fields can be optional (i.e. have value of None)
    hq_state: Optional[str]

    # then we can define any extra validation functions:
    @validator("size")
    def must_be_reasonable_size(cls, v):
        if not (0 < v < 20_000):
            raise ValueError(f"unreasonable company size: {v}")
        return v

    @validator("founded")
    def must_be_reasonable_year(cls, v):
        if not (1900 < v < 2022):
            raise ValueError(f"unreasonable found date: {v}")
        return v

    @validator("hq_state")
    def looks_like_state(cls, v):
        if len(v) != 2:
            raise ValueError(f'state should be 2 character long, got "{v}"')
        return v

def parse_and_validate(data):
    parsed = {}

    parsed["size"] = data["size"].split()[0].strip("+")
    parsed["founded"] = data["founded"]
    hq_details = data["headquarters"].split(", ")
    parsed["hq_city"] = hq_details[0] if hq_details else None
    parsed["hq_state"] = hq_details[1] if len(hq_details) > 1 else None
    parsed["revenue_currency"] = data["revenue"].split("(")[-1].strip("(")")
    return Company(**parsed)

ebay_data = {
    "headquarters": "San Jose, CA",
    "size": "10000+ Employees",
    "founded": "1995",
    "type": "Company - Public (EBAY)",
    "revenue": "$10+ billion (USD)",
}
print(parse_and_validate(ebay_data))

tesco_data = {
    "headquarters": "Welwyn Garden City, United Kingdom",
    "size": "10000+ Employees",
    "founded": "1919",
    "type": "Company - Private",
    "revenue": "Unknown / Non-Applicable",
}
print(parse_and_validate(tesco_data))

```

Here, we've converted our parser and validator to use pydantics models to validate parsed data. Our Ebay data will be validated successfully, while Tesco data will raise a validation error:



```
pydantic.error_wrappers.ValidationError: 1 validation error for Company
hq_state
  state should be 2 character long, got "United Kingdom" (type=value_error)
```

Pydantic is very strict and will raise exceptions whenever it fails to validate or interpret data. This is a great way to ensure web scraped data quality though it comes with a higher setup overhead than our Cerberus example.

## Transforming and Interpreting

Another advantage of Pydantic is the ability to transform and cast data types from strings. For example, in our `hq_state` field we can ensure that all incoming values are lowercase:

```
class Company(BaseModel):
    @validator("hq_state")
    def looks_like_state(cls, v):
        if len(v) != 2:
            raise ValueError(f'state should be 2 characters long, got "{v}"')
        return v.lower()
```

Above, our validator will check whether `hq_state` field is 2 characters long and convert it to lower case standardizing our scraped dataset.

Pydantic also automatically casts common data types from string values, making our parsing process much more streamlined and easier to follow:

```
from datetime import date
from pydantic import BaseModel

class Company(BaseModel):
    founded: date

print(Company(founded="1994-11-24"))
# will print:
# founded=datetime.date(1994, 11, 24)
```

As you can see, Pydantic automatically interpreted string date as a python `date` object.

## Cerberus or Pydantic?

We've explored both of these popular data validation packages and even though they are quite similar there are a few key differences.

Primarily, Pydantic integrates with Python's type hint ecosystem whereas Cerberus uses a more generic schema approach. So, while Pydantic can be a bit more complicated it can provide major benefits like code completion in IDE's and automatic data conversion. On the other hand, Cerberus is a bit easier to understand and less strict making it ideal for smaller web scraping projects.

## Scraped Data Validation Summary

In this article, we've taken a look at two popular ways to ensure web-scraped data quality:

- [Cerberus](#) - schema-based validator that is easy to setup and configure.

- [Pydantic](#) - type-based validator that not only validates data but can easily normalize it to standard python data types like date and time objects.

No matter which approach you choose to go with both are very powerful tools when it comes to data quality in web scraping.

Check out ScrapFly Python SDK

Try ScrapFly for FREE!

Related Questions

- What Python libraries support HTTP2?
- How to scrape HTML table to Excel Spreadsheet (.xlsx)?
- How to use proxies with Python httpx?
- How to select dictionary key recursively in Python?
- What are some ways to parse JSON datasets in Python?
- Selenium: chromedriver executable needs to be in PATH?
- How to fix python requests ConnectTimeout error?
- How to fix Python requests MissingSchema error?
- Python httpx vs requests vs aiohttp - key differences
- How to handle popup dialogs in Playwright?
- How to scrape images from a website?
- How to check if element exists in Playwright?
- How to use cURL in Python?
- Selenium: geckodriver executable needs to be in PATH?
- How to fix Python requests ReadTimeout error?

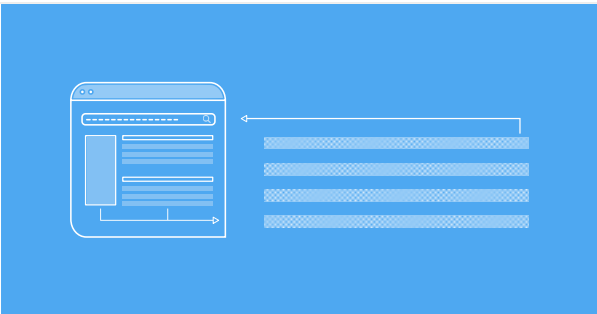
More >

PYTHON

DATA PARSING



Related Posts



Mar 10, 2025

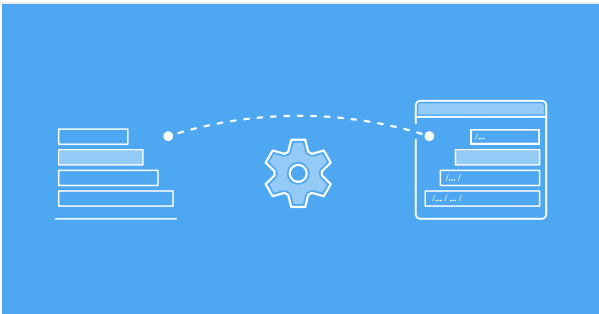
**Guide to List Crawling: Everything You Need to Know**

In-depth look at list crawling - how to extract valuable data from list-formatted content like tables, listicles and paginated pages.

WEB CRAWLING

BEAUTIFULSOUP

PYTHON



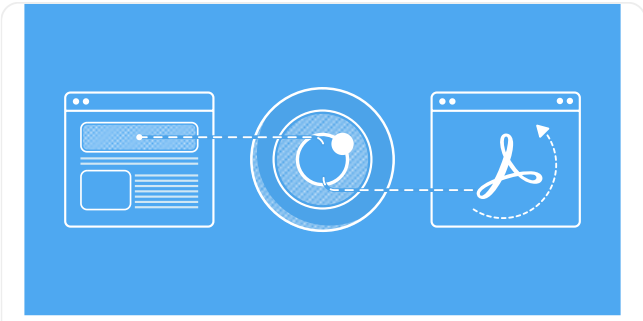
Jan 29, 2025

**How to Find All URLs on a Domain**

Learn how to efficiently find all URLs on a domain using Python and web crawling. Guide on how to crawl entire domain to collect all website data

WEB CRAWLING

PYTHON



Jan 22, 2025

## How to Capture and Convert a Screenshot to PDF

Quick guide on how to effectively capture web screenshots as PDF documents

SCREENSHOTS

PYTHON

NODEJS

## Company

- Careers
- Terms of service
- Privacy Policy
- Data Processing Agreement
- KYC Compliance
- Status

## Integrations

- Zapier
- Make
- N8n
- LlamaIndex
- LangChain

## Social



## Tools

- Convert cURL commands to Python code
- JA3/TLS Fingerprint
- HTTP2 Fingerprint
- Xpath/CSS Selector Tester

## Resources

- API Documentation
- Web Scraping Academy
- Is Web Scraping Legal?
- Web Scraping Tools
- FAQ

## Learn Web Scraping

- Web Scraping with Python
- Web Scraping with PHP
- Web Scraping with Ruby
- Web Scraping with R
- Web Scraping with NodeJS
- Web Scraping with Python Scrapy
- How to Scrape without getting blocked tutorial
- Web Scraping with Python and BeautifulSoup
- Web Scraping with Nodejs and Puppeteer
- How To Scrape GraphQL
- Best Proxies for Web Scraping
- Top 5 Best Residential Proxies

Usage

- What is Web Scraping used for?
- Web Scraping for AI Training
- Web Scraping for Compliance
- Web Scraping for eCommerce
- Web Scraping for Finance
- Web Scraping for Fraud Detection
- Web Scraping for Jobs
- Web Scraping for Lead Generation
- Web Scraping for News & Media
- Web Scraping for Real Estate
- Web Scraping for SERP & SEO
- Web Scraping for Social Media
- Web Scraping for Travel

© 2025 Scrapfly - The Best Web Scraping API For Developers