

# Python unplugged

Mining for hidden batteries

---



[loki.dev/ep24](https://loki.dev/ep24)



# \$ whoami

---

**name:** Torsten

**occupation:** Polylang Software  
Architect/Backend Developer Mara  
Solutions

**dev since:** +12y

**first code:** ~25y if you count creating  
crappy Amiga Quickbasic textadventure

**other stuff:** Being a proud dad and  
husband. Reading, tabletennis, ...the usual  
stuff



# What problem am I trying to solve here?

---

- learn the possibilities of the built in standard library
- for simple cases external libraries are not always necessary<sup>1</sup>
  - *<sup>1</sup>for simple cases!*
- better portable one-off scripts!
- sometimes even easier!

# What am I NOT trying to solve?

---

- Performance issues
- If performance is really **crucial**:  
Pandas/Polars/Numpy/etc. are perfect for you

# Outline: 4 Chapters

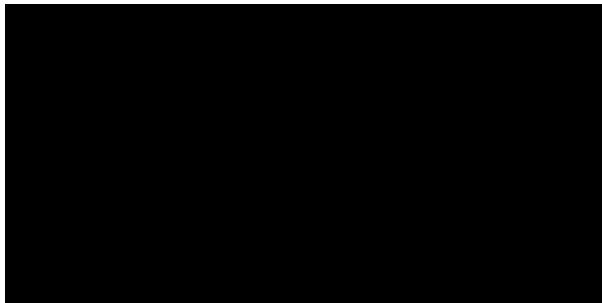
---

1. Fetching data
2. Cleaning data
3. Processing data
4. Miscellaneous / Additional stuff (Might be dropped because of time)
5. (Maybe some vanilla python server implementation)

# What's the story?

- Who is familiar with *Terry Pratchett*?
- We are working on a project in *Ankh Morpork*
- The "Unseen Library" and its librarian finally want to implement some digitalisation magic
- The first step is, that we already have some API \*cough\* magic funnel sending us the book data. One by one.
- It is our job to clean the data, work on the data and also save it in our excel (because why not)





*\*I actually created a full (vanilla python) server and generator to generate those books - And we most likely won't even see nor use it -.'*

# Example book

Puzzling Eating of the Golems

by Senior Guildmaster Danielle Warren

```
title: Puzzling Eating of the Golems
author: Senior Guildmaster Danielle Warren
lent_by: Jessica Silvermist
lent_since: year 0065 in the 1st month
lent_times: 42
year: The 10th year after Silver Horde
catalogued: year 0300 in the 7th month
location: Restricted Section: middle shelve. 4m from the end
excerpt: Amidst the ruins of a lost civilization trembled
         as a jester sang a melody. Hidden truths come to
         light.
```



# Chapter 1: Fetching data

# Fetching data

```
"""
An example response for our project
"""

def fetch_book(url): ...

fetch_book(URL_BOOK) == dict(
    title="Remarkable Saga of the Clacks",
    author="Alexandra Scott",
    lent_by=null,
    lent_since=null,
    lent_times=8,
    year="The 7th year after Turtle Moves",
    catalogued="year -395 in the 2nd month",
    location="Great Hall: bottom shelve. 5m from the end",
    excerpt=(
        "Near the bubbling cauldron had never seen such a"
        " sight: some dwarf miner raised the dead. The "
        "annual magical cooking competition begins."
    ),
)
```

# Working with data

```
"""
With TypedDict from typing module we help the IDE
to give us proper type hints!
Positive Sideeffect: This helps static code checkers help us
-> e.g. ruff,mypy,pyright,pyre,...
"""
```

```
from typing import TypedDict
```

```
class Book(TypedDict):
    title: str
    author: str
    lent_by: str | None
    lent_since: str | None
    lent_times: int
    year: str
    catalogued: str
    location: str
    excerpt: str
```

```
def fetch_book() -> Book: ...
```

# This is what it looks like

```
2     catalogued: str
1
0         lo
9             ex π 'year'
8                 π 'title'
7                 π 'author'
6             def fe π 'lent_by'           Response:
5                 re π 'excerpt'          rlopen(URL_BOOK).read()
4                 re π 'location'        sponse)
3                 π 'lent_since'
2                 π 'lent_times'
1             def ma π 'catalogued'
0
7     book['']      ■ Could not access item in TypedDict    "" is not a de
```



## Let's continue

---

- There are a LOT of books. And we only have limited ressources
- This means that we need to work on the books in batches
- Plenty of solutions!
- Let's first look on how we might have done it before python3.12
- And then how python3.12 makes our life easier :)

# Batching

```
from csv import DictWriter
from itertools import batched

LIBRARY_DB = Path("library_raw.csv")

# fetches the keys from our TypedDict :)
BOOK_COLS = Book.__annotations__.keys()

def work_on_the_library() -> None:
    lib = fetch_library()

    with LIBRARY_DB.open("w") as file:
        writer = DictWriter(file, fieldnames=BOOK_COLS)
        writer.writeheader()
        for batch in batched(lib, 10):
            writer.writerows(batch)
```

## Summary

---

We started very small, but which modules did we learn about?

## Trivial

---

Let's start with the stuff you likely already knew...

- `csv ("duh!")`
- `json ("Yeah I knew that!")`
- `TypedDict` (Maybe new for a few?)
- All of those are in more detail in the `/code` folder inside the repository

## urllib.requests

---

- fetch and send data via HTTP(S)
- restricted to `GET / POST` (`POST` by setting the `data` parameter)
- not as bad to use as one might think, considering the vast amount of modules to replace it

## itertools.islice

---

- basically works like `mylist[from:to:steps]`
- but also works on Generators with unknown size :)
- cannot go backwards or use negative indices:
  - works: `mylist[::-1]` (reversing a list)
  - doesn't work: `islice(mygenerator, start=-9, step=-3)`

## itertools.batched

---

- Finally arrived in `python3.12`
- Works on all kind of iterable stuff
- delivers `tuple[T]` of size `n`
  - `list(batched(range(5), 3)) == [(0, 1, 2), (3, 4)]`



## Chapter 2: Cleaning data

---

*(wrong meme universe again - I know)*

# Duplicate books

- We want to fetch data from the library...
- But: Magical library. Means: the books will duplicate!
- The orang utan librarian of the Unseen University wants the list of unique books as CSV (obviously!)
- This is **NOT** a leetcode talk ;)
  - no fancy but obvious solution if P in NP
  - also: no super clever text similarity or duplication algorithms



```
from csv import DictWriter
from itertools import chain, pairwise

LIBRARY_DB = Path("library_raw.csv")
COLUMNS = Book.__annotations__.keys()

def only_save_non_duplicates() -> None:
    """
    chain to the rescue!
    It lazily chains multiple iterators without effort
    """
    book_gen = fetch_library()

    with LIBRARY_DB.open("w") as file:
        writer = DictWriter(file, fieldnames=COLUMNS)
        writer.writeheader()

        for book, book2 in pairwise(chain([None], lib)):
            if book != book2:
                writer.writerow(book2)
```

What new modules did we learn about?

## pairwise

---

- Signature: `def pairwise(iterable: Iterable[T]) -> Iterable[tuple[T, T]]`
- Uses an iterator like `range(10)` ...
- ... and creates a new iterator like `((0, 1), (1, 2), (2, 3), ..., (8, 9))`
- other example: `list(pairwise("Hello")) == [("H", "e"), ("e", "l"), ("l", "l"), ("l", "o")]`
- is lazy -> Without `list(...)` or other ways of consuming pairwise does (almost) nothing

# chain

---

- Chains together multiple iterables
- From this `list(chain('Hello', 'World'))` ...
  - We get this `['H', 'e', 'l', 'l', 'o', 'W', 'o', 'r', 'l', 'd']`
- Advantage: it's lazy and doesn't create a large<sup>1</sup> object, like:
  - `combined_list = [*my_list, *my_other_list]`
  - <sup>1</sup>(not large means: it only holds references)
- It can flatten lists with a classmethod: `chain.from_iterable(...)`

# It can flatten lists

*(also a common thing people do themselves)*

---

```
my_list = [[1,2,3], [3,4,5]]  
flattened = chain.from_iterable(my_list)  
  
assert list(flattened) == [1,2,3,3,4,5]
```

# ChainMap

---

*For the brevity of completeness...*

- There is also `itertools.ChainMap`
- Works similar to `chain`:
- `cm = ChainMap({ "a" : 1}, {"b" : 2}, {"a" : 3})` gives us:
  - `cm[ "a" ] == 1` (Not `3` !)
  - `cm[ "b" ] == 2`
  - `cm[ "c" ] --> KeyError`
- Attention: Different behaviour than `**dict`
  - `{**adict, **bdict} => dict(ChainMap(bdict, adict))`

# Saving the deduplicated data

Basically just an excuse to show grouping of contextmanagers

```
OUTPUT_FILE = 'library_clean.csv'

def remove_duplicates(iterable: Iterable) -> Iterator: ...

def clean_data(input_csv_file: str) -> str:
    out_name = input_csv_file.replace(".csv", "-clean.csv")

    with (
        Path(input_csv_file).open("r") as read_file,
        Path(out_name).open("w") as write_file,
    ):
        reader = csv.DictReader(read_file)
        writer = csv.DictWriter(write_file)

        for row in remove_duplicates(reader):
            writer.writerow(row)
```

## Remove the books we cannot get back

```
import re

def extract_year(morporkyear: str) -> int:
    """
        But how do we know this really does what we want?
        We could try it, but this is boring and doesn't scale!
    """
    return int(re.search(r"(\d)", morporkyear).group(0))

def hide_lost_books(iterable: Iterable[Book]) -> Iterator[Book]:
    ...
```

# Doctest - to the rescue

```
"""Let's start with the same function, but add some  
minimal documentation"""
```

```
import re
```

```
def extract_year(morporkyear: str) -> int:  
    """  
    This extracts the year from a yearstring  
  
    >>> extract_year("year 450 in the 3rd month")  
    450  
    """  
  
    return int(  
        re.search(r"(\d)", morporkyear).group(0)  
)
```



```
python -m doctest code/chapter2/doctest_example.py

*****
File ".../code/chapter2/doctest_example.py",
  line 8, in doctest_example.extract_year
Failed example: extract_year("year 450 in the 3rd
 month")
Expected: 450
Got: 4
*****
1 items had failures:
  1 of 1 in doctest_example.extract_year
***Test Failed*** 1 failures.
```



Long story short: lets fix it!

```
"""
Adding a few more "tests" & fixing the issues
"""

import re

def extract_year(morporkyear: str) -> int:
    """
    This extracts the actual year from a yearstring

    >>> extract_year("year 450 in the 3rd month")
    450

    >>> extract_year("year -450 in the 3rd month")
    -450

    >>> extract_year("year 0 in the 3rd month")
    0

    """

    return int(
        re.search(r"(-?\d+)", morporkyear).group(0)
    )
```



 Let's continue hiding the lost books

```
"""
Usage
"""

def hide_lost_books(iterable: Iterable[Book]) -> \
    Generator[Book, None, list[Book]]: ...

def get_return_value() -> tuple[list[Book], list[Book]]:
    non_hidden_generator = hide_lost_books(mylibrary)

    # consume the whole generator (just as example)
    non_lost_books = list(non_hidden_generator)

    dropped_books = []
    try:
        next(non_hidden_generator)
    except StopIteration as e:
        dropped_books = e.value

    return non_lost_books, dropped_books
```

## Summary

---

- Generators `yield` values
- Generators always also return a value inside `StopIteration` (default `None`)
- Is it a good idea?
  - No. Most likely not, as it is kind of surprising behaviour
- But why did we just see it?
  - Because sometimes in one-off scripts this is faster and easier than a sophisticated object containing the data solution
- What about the middle argument in `Generation[A, B, C]`?
  - Out of scope, but basically we can SEND stuff to generators after their creation
  - `gen = create_generator(); gen.send(123)`
  - Fetch with `received = yield anothervalue; received == 123`

## 🤓 But how would I do it?

- Perfect time to learn a new thing
- We use an object to track the data.
  - `NamedTuple` to avoid ugly `result[0]` and `result[1]`
  - Better: `result.lost` and `result.current`
  - relatively cheap: We only transport references
  - just yield this container instead of complex surprising methods

```
from typing import NamedTuple

class BookMeta(NamedTuple):
    current: Book
    lost: list[Book]

def hide_lost_books(
    iterable: Iterable[Book]
) -> Iterator[BookMeta]:
    lost_books = []

    for book in iterable:
        if extract_year(book["lent_since"]) >= -300:
            yield BookMeta(book, lost_books)
        else:
            lost_books.append(book)
```

# Chapter 3: Summarize and key data

---

*Now the wonderful librarian wants us to get some data to get to know in which dimensions we're working here?*

## Creating an Index

We want to have list of words in all the titles.

---

Very simple example Example

- The: 123
- a: 42
- magician: 99

## Count Words

```
from collections import defaultdict

def most_common_words_in_title(books: Iterable[Book]) -> dict:
    """
    Much more concise :)
    """

    word_counter = defaultdict(int)

    for book in books:
        for word in book['title'].split():
            word_counter[word] += 1
    return word_counter
```

*but now the librarian updated the assignment...*

*So glad customers always know what they want in the first iteration :)*

```
from collections import defaultdict, Counter
from itertools import chain

def most_common_words_in_data(books: Iterable[Book]):
    """
    This finally gives us the words in all the books
    as one long word generator
    """
    word_generator = chain.from_iterable(
        chain(
            book['title'].split(),
            book['author'].split(),
            book['excerpt'].split(),
        )
        for book in books
    )

    return Counter(word_generator)
```

Let's add some more data:

---

- min lent year (We all know this `min(x for x in thing)` - trivial)
- max lent year (`max(x for x in thing)` - trivial)
- family which lent the most (little bit more interesting)
- largest book shelve in the library (let's see)

```
from itertools import groupby

def family_name(book: Book) -> str:
    return book['lent_by'].split()[1]

class BiggestLender(NamedTuple):
    family: str
    lent_books: int

def family_lent_the_most(books: Iterable[Book]) -> BiggestLender:
    only_lent_books = sorted((b for b in books if b["lent_by"]), key=family_name)

    books_by_family_name = groupby(only_lent_books, key=family_name)

    family, lent_books = max(
        (
            (family, list(group))
            for family, group in books_by_family_name
        ),
        key=lambda item: len(item[1]),
    )

    return BiggestLender(family, len(lent_books))
```

But the librarian changed requirements again...

---

:/

*"ookh! ookh! (I want you the total lenders, unique family names and amount of unlent\_books)*

```
def total_unique_lenders(books: Iterable[Book]) -> int: ...
def unique_families(books: Iterable[Book]) -> set[str]: ...
def unlent_books_count(books: Iterable[Book]) -> int: ...

class Statistics(NamedTuple): ...

def gather_statistics(books: Iterable[Book]) -> Statistics:
    """This would solve that, right?"""
    all_books = list(books)

    return Statistics(
        unique_lenders=total_unique_lenders(all_books),
        families=unique_families(all_books),
        unlent_books=unlent_books_count(all_books),
    )
```

Nope, it doesn't :(

# Reduced Tee

Two tools which might help us:

1. `itertools.reduce`
2. `itertools.tee`

**Let's start with reduce!**



```
from itertools import reduce

@dataclasses.dataclass
class StatisticsAccumulator:
    lenders: set[str] = set()
    families: set[str] = set()
    unlent: int = 0

    def statistics_reducer(
        acc: StatisticsAccumulator,
        book: Book,
    ) -> StatisticsAccumulator:
        if book['lent_by']:
            acc.lenders.add(book['lent_by'])
            acc.families.add(family_name(book))
        else:
            acc.unlent += 1

        return acc

    def gather_statistics(books: Iterable[Book]) -> Statistics:
        accumulated_stats = reduce(statistics_reducer, books, StatisticsAccumulator())

        # We could just return the Accumulator, but I like to keep the
        # amount of mutable objects as confined as possible
        return Statistics(
            unique_lenders = len(accumulated_stats.lenders),
            families = len(accumulated_stats.families),
            unlent = accumulated_stats.unlent)
```

How about tee?

---

*Maybe our computation is not CPU bound, but IO bound?*



```
from itertools import tee

def gather_statistics(books: Iterable[Book]) -> Statistics:
    """
    Now we create 3 "references" of the generator
    Caveat:
        - This is only a pseudo solution!
    """
    books_1, books_2, books_3 = tee(books, n=3)

    return Statistics(
        """And use those here"""
        unique_lenders=unique_lenders(books_1),
        families=unique_families(books_2),
        unlent_books=unlent_books_count(books_3),
    )
```



## Tee caveats

1. You don't want to consume the source generator
  - It will move forward without the `tees` to catch up!
2. `tee` is not thread safe: it might cause `RuntimeError` if used in a threading/async environment!
  - Not standard vanilla python, but for easy `async` and threading usage:  
`asyncstdlib`
  - I have a vanilla python implementation with `Queue` and



# Miscellaneous

---

*(this might not make it, due to time restrictions)*

# Mixin for ordering objects

---

- Imagine we have our own object containing data we need
- It should be sortable (e.g. the books by year!)
- But taking care of all the magic dunder methods: `__lt__( )`, `__le__( )`, `__gt__( )`, `__ge__( )`

# Order objects

```
from functools import total_ordering

@total_ordering
class Book:
    def __init__(self, name: str, year: str):
        self.name = name
        self.year = year

    def __eq__(self, other: 'Book') -> bool:
        own_year = extract_year(self.year)
        other_year = extract_year(other.year)
        return own_year == other_year

    def __lt__(self, other: 'Book') -> bool:
        own_year = extract_year(self.year)
        other_year = extract_year(other.year)
        return own_year < other.year
```

- Add `__eq__` to check if the object is considered equal
- Add `__lt__` to check if it is true lower (not equal)
- Add `total_ordering` to just fill the other dunder\_methods:
  - `__le__`
  - `__ge__`
  - `__gt__`
- Obviously we should cache the calculation ;)

# with suppress

```
import contextlib

def ignore_failing_call():
    with contextlib.suppress(RuntimeError):
        """We still don't care about failing"""
        might_fail_but_I_dont_care()
```

# The "server"

---

- I actually wrote a whole server to generate random books, just to drop the usage due to time limitations
- But still: We can look into some of the last "batteries included"
  1. A pure python server (means: very simple and unmighty Flask/FastAPI)
  2. Argument parsing (not as beautiful as typer, but easy to work with)
  3. Caching
  4. suppressing errors the right way!
  5. partials of functions/methods (basically lambdas in pythonic)



Questions?

---

# Konec a poděkování

---

*(The end of the talk and thank you)*

