Pylan

. . . a Python library for time series simulation

by Timo Kats

March 13, 2025

Introduction

Context

- A common use-case is to plan ahead based on expected patterns. E.g.:
  - "*Given a salary (which increases each year), monthly costs, and inflation, how much money will I have in 10 years?*"
  - "*Given the half-life of caffeine, and my daily consumption of coffee, how much caffeine do I have in my system at 22:00?*"

Problem

- Typically, this is planned through an application like **Excel** or in **Python**.
  - **Excel** is not as flexible/powerful as programming.
  - **Python** is better, but libraries like **pandas** or **numpy** don't support creating new data based on different time series schedules out of the box.

Solution

My Python library for simulating and analyzing the effect of these patterns over time, called **“pylan”**.

In summary, it combines 3 core components:

- **Schedules**: cron, static interval, varying intervals, specific dates, ...
- **Impacts**: Multiply by x, subtract y, ...
- **Analysis**: Show a plot for every month, YTD, YOY, max value, ...

Usage - Patterns

Pylan uses **patterns** to capture the effect of scheduled events. You can apply these patterns to **items** (e.g. a savings account) or other **patterns**.

```
from pylan import Add, Granularity, Item, Multiply

savings = Item(start_value=10)

salary_payments = Add("1m", 250, offset="24d")
salary_increase = Multiply("1y", 1.2)

salary_payments.add_pattern(salary_increase) # Salary increases
savings.add_pattern(salary_payments) # Add salary payments to item
```

Usage - Timed simulations

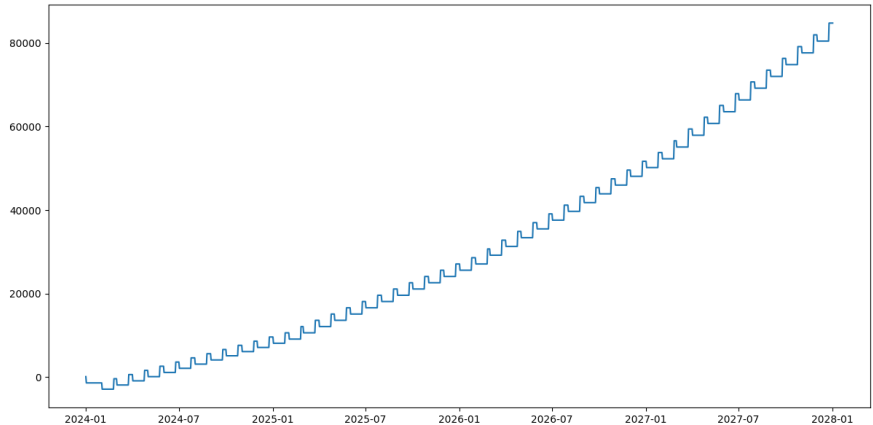
After adding patterns, you can run the simulation. This can be done in different ways. First, we can run the patterns between two datetimes. This will create a **result** object.

```
result = savings.run("2024-1-1", "2028-1-1")

print(result["2024-5-6"]) # result on a specific day
print(result.final)  # last result

result.to_csv("something.csv") # export data
x, y = result.plot_axes()  # to matplotlib
```

Usage - Timed simulations (plot)



Usage - Targeted simulations

Next, we can run the patterns **until** a target value is reached. This will result in a **timedelta** object.

```
>>> savings.until(500)
>>> relativedelta(years=+3, months=+1)
>>> savings.until(500, max_iterations=50000)
>>> relativedelta(years=+3, months=+1)
```

Note

The default max_iterations is 1000.

Usage - For loops

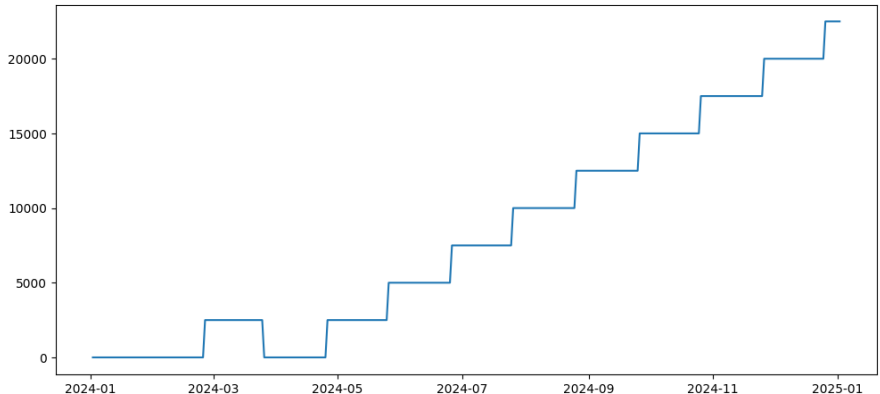
Finally, we can iterate through the simulation using **Granularity**.

```
from pylan import Granularity, Result

buy_car = Subtract("", 5000)
x = Item(start_value=10)
car_bought = False
result = Result()

for date, saved in x.iterate("2024-1-1", "2025-1-1", Granularity.day):
    if x.value > 5000 and not car_bought:
        buy_car.apply(x)
        car_bought = True
    result.add_result(date=date, value=x.value)
```

Usage - For loops (plot)



Usage - Granularity

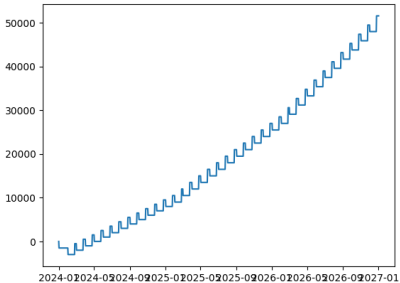
Note, we can also change the granularity when running "normal" simulations.

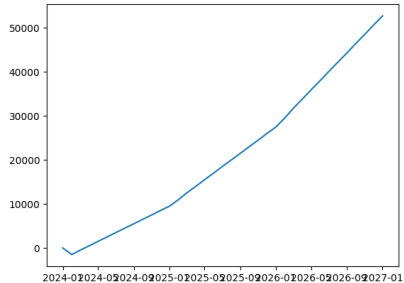
```
from pylan import Granularity

result = savings.run("2024-1-1", "2027-1-1", Granularity.day)
result = savings.run("2024-1-1", "2027-1-1", Granularity.week)
result = savings.run("2024-1-1", "2027-1-1", Granularity.month)
```

Usage - Granularity (plot)

The figure on the left shows the daily granularity. The figure on the right shows the same patterns, but with a monthly granularity.





Next steps

- Support more patterns, such as bonds that pay out over time and at the end.
- More options for analysis.
- Get feedback :)

Links

- GitHub
- PyPi (Also contains docs)