Doc-tracing

Fouiller une base de code fossile par tracage d'execution

Numeric Park

La documentation disparue

Luc Sorel-Giffo — jeudi 29 juin 2023 - 12h45 amphi D — BreizhCamp

@lucsorelgiffo@floss.social

Synopsis

Le site de fouilles : le code

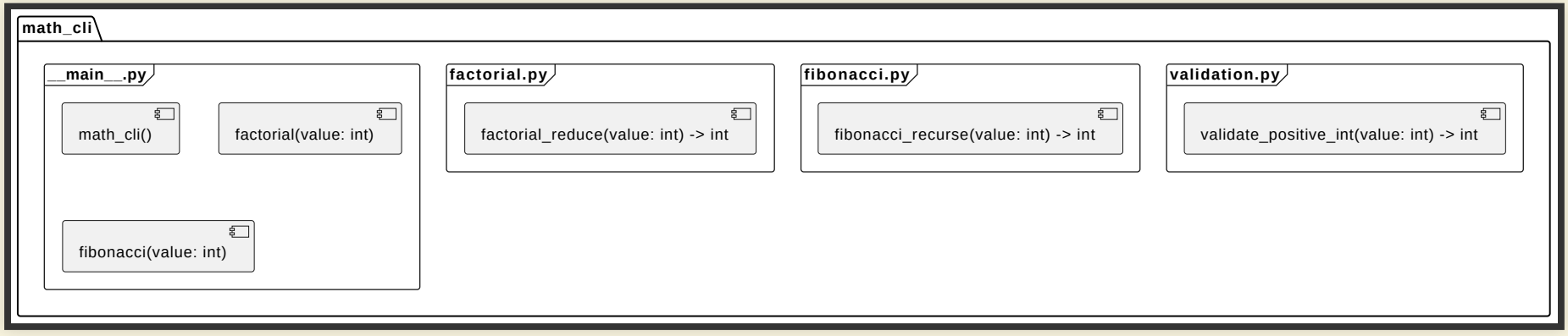
```
math_cli
 ├── __main__.py
 ├── factorial.py
 ├── fibonacci.py
 └── validation.py
```

💡 Tous les exemples de code se trouvent dans github.com/lucsorel/doc-tracing/tree/main/examples.

La doc

Doc-as-code

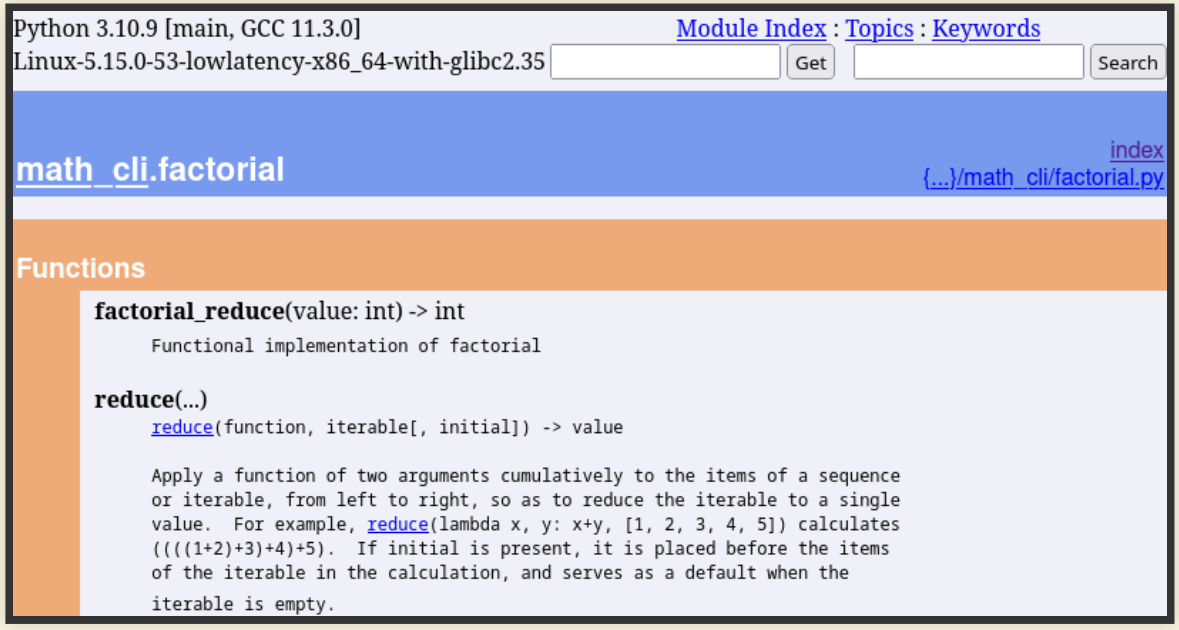
```
@startuml math_cli
package math_cli {
  frame __main__.py {
    component "math_cli()"
    component "factorial(value: int)"
    component "fibonacci(value: int)"
  }

  frame factorial.py { component "factorial_reduce(value: int) -> int" }

  frame fibonacci.py { component "fibonacci_recurse(value: int) -> int" }

  frame validation.py { component "validate_positive_int(value: int) -> int" }
}
@enduml
```



Analyse statique de code - "Code-as-doc" 😉

```python
# math_cli/factorial.py
from functools import reduce

def factorial_reduce(value: int) -> int:
    '''Functional implementation of factorial'''
    if value == 1:
        return 1

    return reduce(lambda agg, index: agg * index, range(value, 1, -1), 1)
```

```
python -m pydoc -p 8080 -b # -> http://localhost:8080
```



Analyse statique de code – apports et limites

🙂

- génération / mise à jour de la doc à partir d'une source de vérité
- valorisation : docstring & annotations de typage

😕

- comment **s'articulent** les fonctions ?
- comment sont gérées **les erreurs** ?

Tracage d'execution avec sys.settrace

sys.settrace permet de déclarer un hook qui va être appelé à chaque étape d'exécution du code :

```
from sys import settrace

def trace_func(tracer: Callable, func: Callable, *args, **kwargs) -> Any:
    '''Applies the tracer hook during the execution of the given func'''

    settrace(tracer)
    try:
        return func(*args, **kwargs)
    finally:
        settrace(None)
```

Global tracer : appel d'un bloc de code

Rôle : l'exécution du bloc de code (script, corps de fonction) doit-elle être tracée ?

```python
def global_tracer(frame, event: str, arg: Any) -> Callable:
    # ici : event vaut toujours 'call', arg est toujours None
    if should_trace_call(frame):
        return local_tracer # appel tracé

    return None             # appel non tracé
```

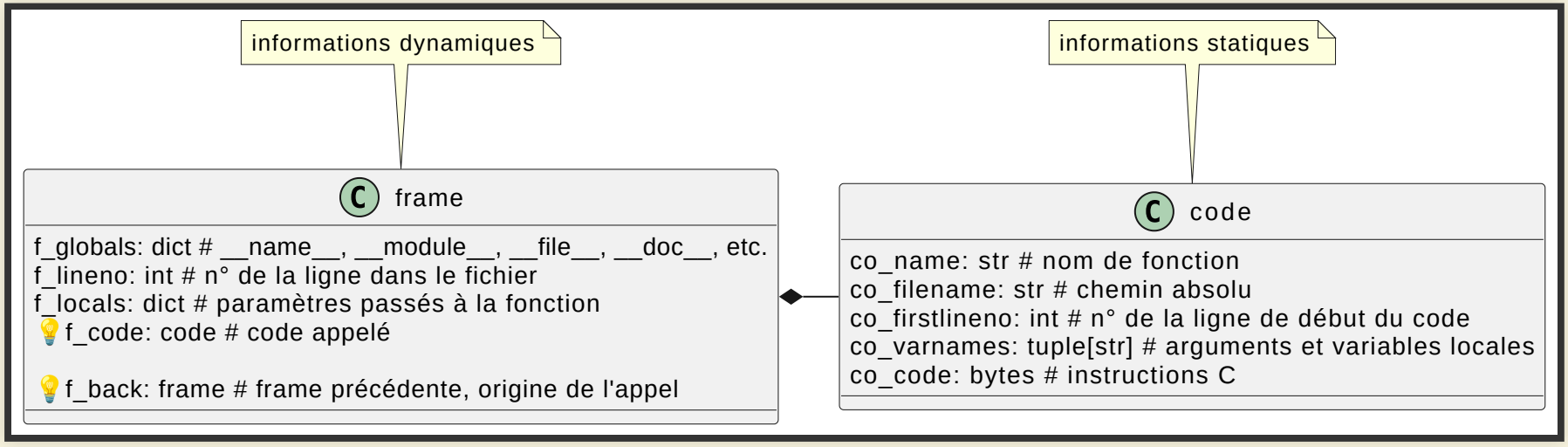


Figure 1. Doc officielle : frame, code

Local tracer : execution interne d'un bloc de code

Rôle : poursuivre le traçage après chaque expression ou lors d'une levée d'erreur ?

```python
def local_tracer(frame, event: str, arg: Any) -> Callable:
    # ici : event peut valoir 'line', 'return' ou 'exception'
    if event == 'line':
        ... # arg : toujours None
    if event == 'return':
        ... # arg : la valeur renvoyée ; local_tracer ne sera plus appelée
    if event == 'exception':
        error_class, error, traceback = arg
        # 💡 stocker l'erreur et sa ligne d'émission (frame.f_lineno)

        # poursuite du traçage avec un traceur spécifique
        return error_tracer

    # pour continuer à tracer l'exécution du bloc
    return local_tracer
```

💡 créer `error_tracer` avec un pattern *factory* pour lui associer le contexte de l'erreur.

Error tracer : gestion de l'erreur dans le bloc de code

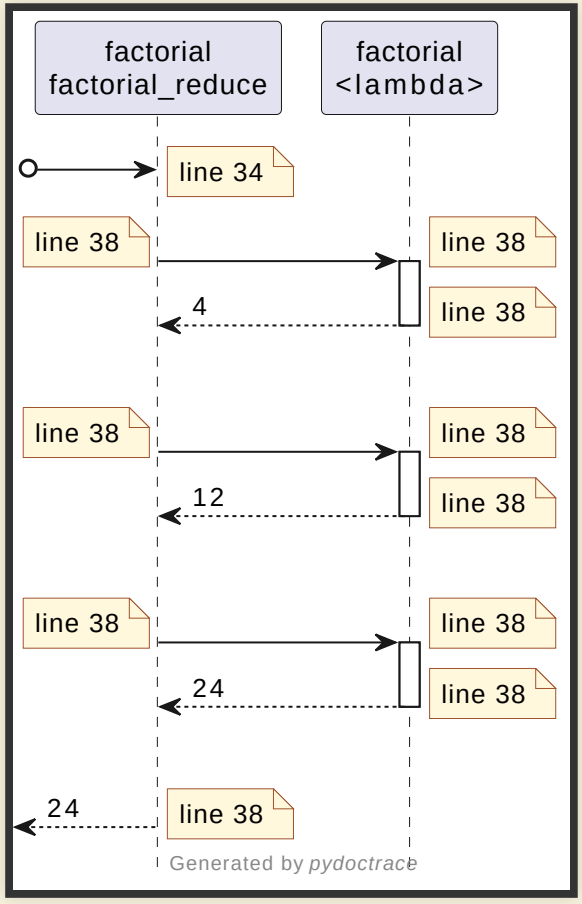
Rôle : tracer la gestion de l'erreur par le code.

```python
def error_tracer(self, frame, event: str, arg: Any):
    # ici : event peut valoir 'line', 'return' ou 'exception'

    if event == 'exception':
        # levée d'une erreur (la même, ou une autre qui la remplace ou l'enrobe)
        # 💡 stocker l'erreur et sa ligne d'émission (frame.f_lineno)
        ...

    elif event == 'return':
        # si il y a une erreur stockée :
        # - si n° ligne "return" == n° ligne "levée d'erreur" : propagation de l'erreur
        # - sinon : l'erreur a été traitée dans un except, sortie du bloc avec `arg`
        # 💡 déréférencer l'erreur stockée

        # s'il n'y a plus d'erreur stockée : sortie du bloc de code

    return error_tracer
```

PYDOCTRACE 🎉

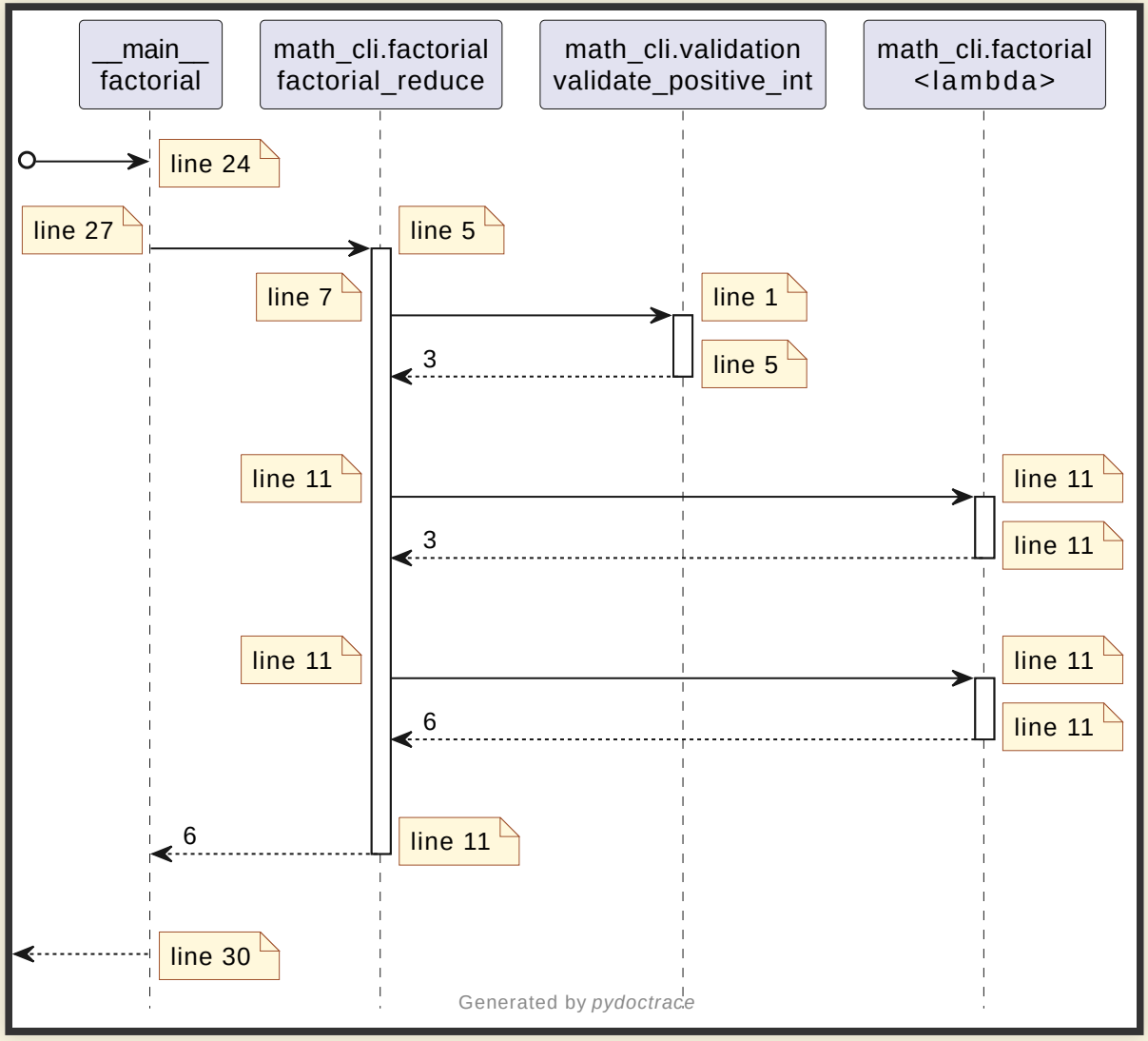
- décorateur (peu intrusif) → diagramme de séquence PlantUML
- github.com/lucsorel/pydoctrace

```python
from pydoctrace.doctrace import trace_to_puml

@trace_to_puml
def factorial_reduce(value: int) -> int:
    '''Functional implementation of factorial'''
    if value == 1:
        return 1

    return reduce(
      lambda agg, index: agg * index,
      range(value, 1, -1),
      1
    )
```



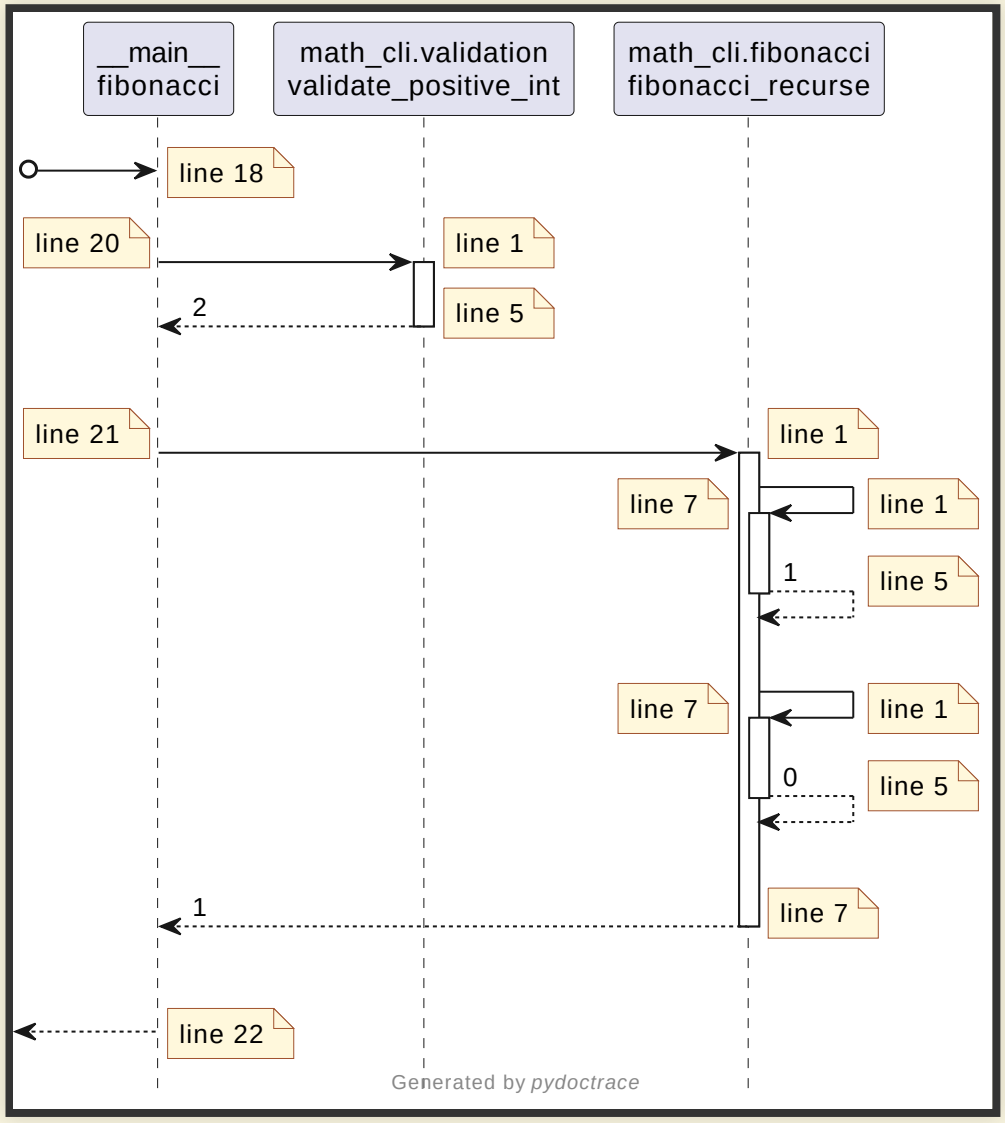
PYDOCTRACE : FACTORIAL REDUCE

```
python -m math_cli --factorial 3
```



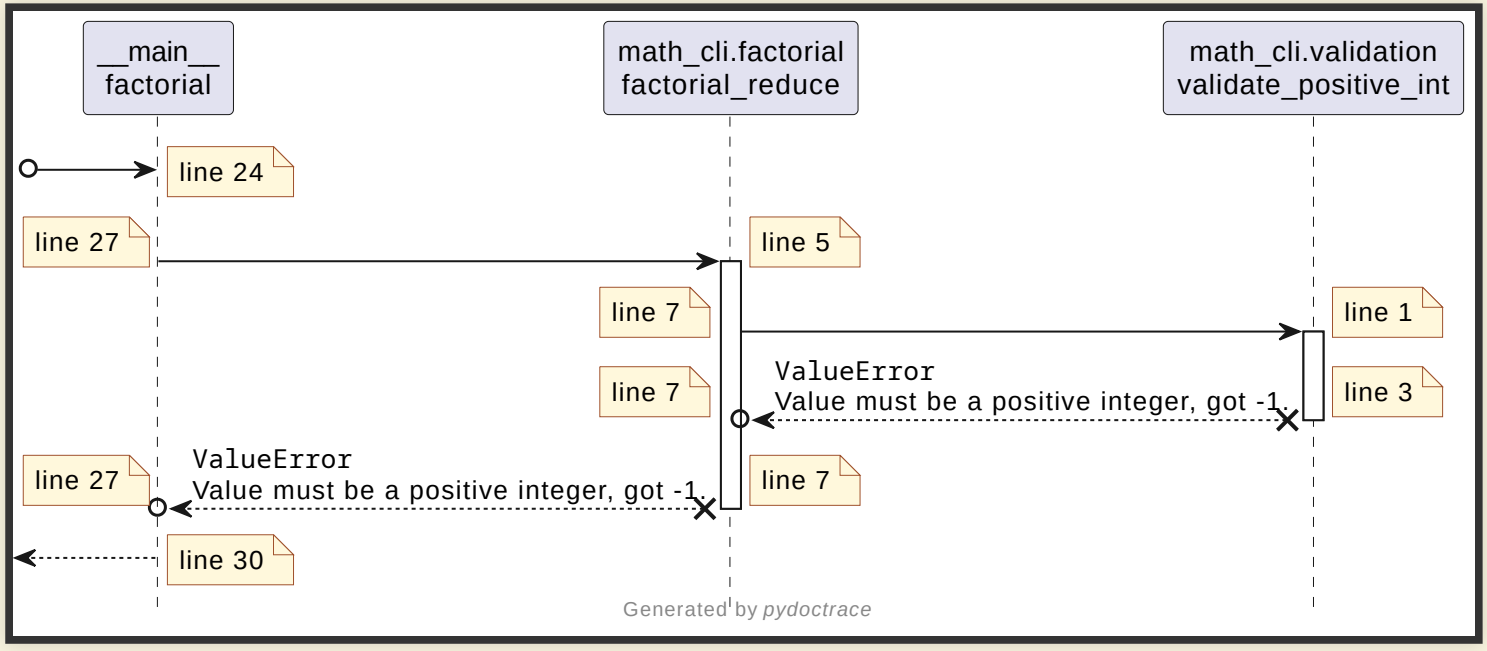
PYDOCTRACE : FIBONACCI

```
python -m math_cli --fibonacci 2
```



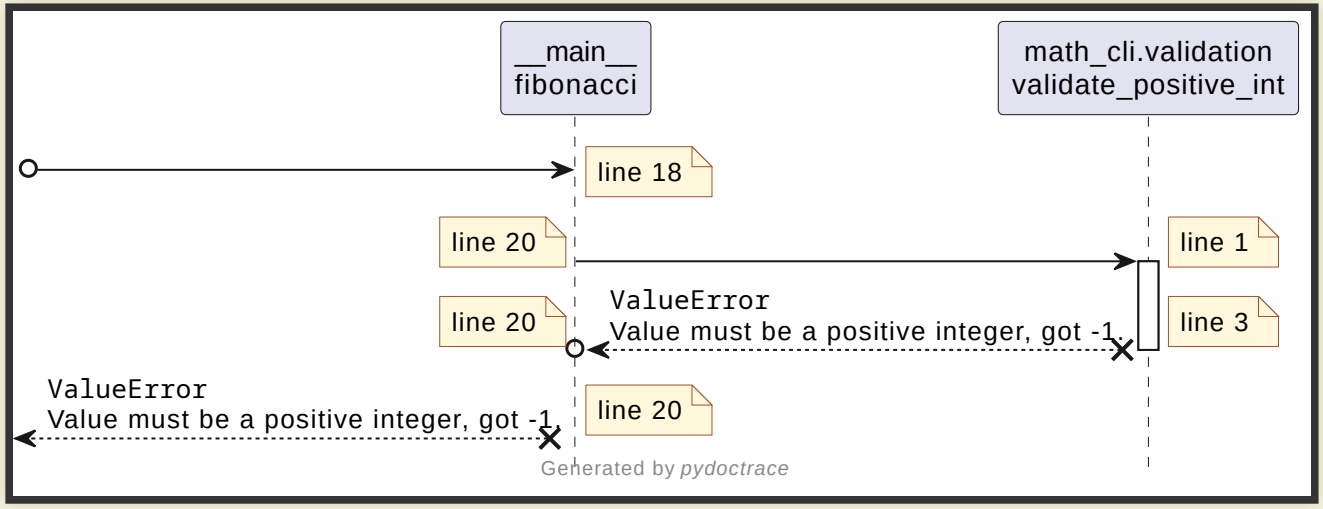
PYDOCTRACE : FACTORIAL REDUCE, ERREUR INTERCEPTEE

```
python -m math_cli --factorial -1
```



PYDOCTRACE : FIBONACCI, ERREUR NON GEREE

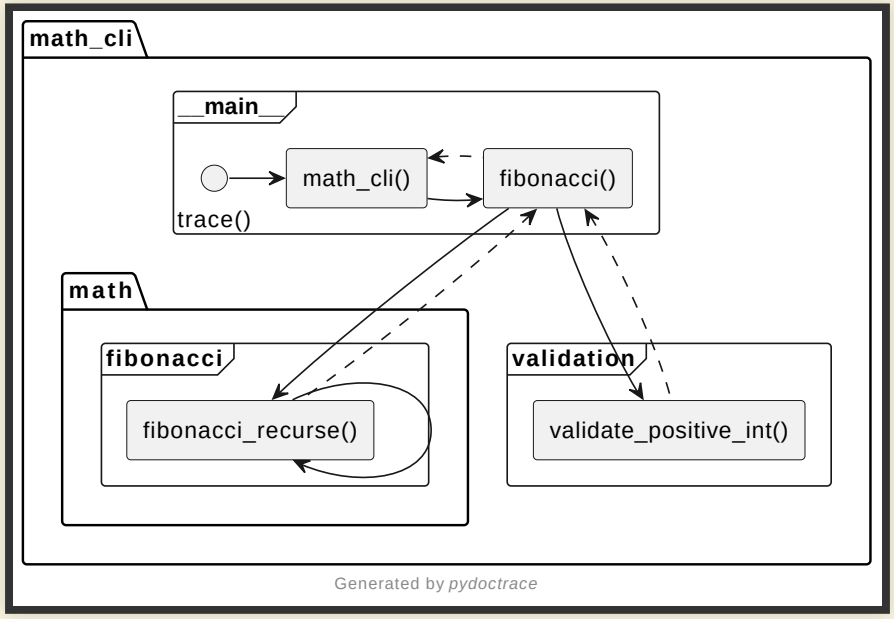
```
python -m math_cli --fibonacci -1
```

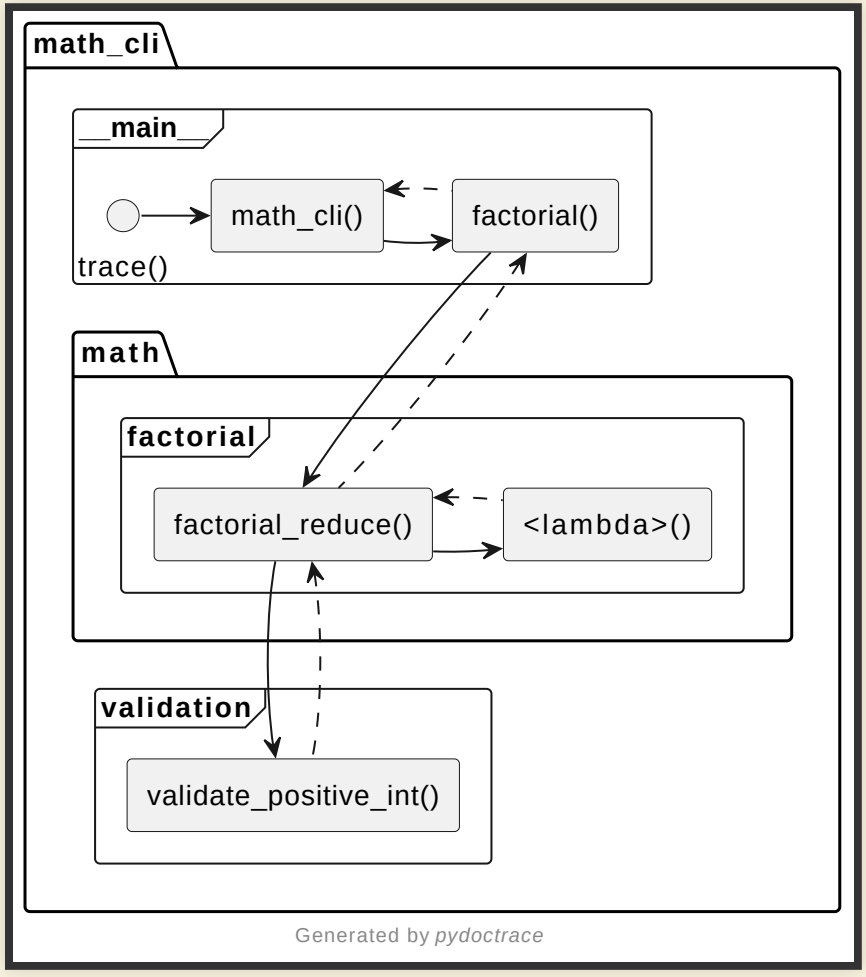


Architecture & diagramme de sequence : le bon niveau ?

Architecture & diagramme de composants 💡 !

- `@trace_to_sequence_puml`
- 👷 `@trace_to_components_puml`





Conclusions concernant le doc-tracing

Mise en œuvre :

- API du débuggueur Python
- programmation orientée aspect, agents

Documenter la vraie vie du code :

- les tests fonctionnels associés à une *user story*
- le code d'une base qu'on découvre
- observabilité applicative ? (OpenTelemetry)

Issues et ⭐ : github.com/lucsorel/pydoctrace 👍

Merci !

Des questions ?

Présentation à retrouver sur github.com/lucsorel/doc-tracing 📑

Rejoignez Python Rennes !

Figure 2. Pour rejoindre le slack : https://join.slack.com/t/pythonrennes/shared_invite/zt-1yd4yioap-lBAngm3Q0jxAKLP6fYJR8w