Introduction to Python

Workshop 3
Thomas Zilliox

Agenda

- Decorators and context managers
- Concurrency and parallelism (threading, multiprocessing, asyncio)
- Type hints, dataclasses, and Pydantic models
- Performance tuning and profiling

Decorators and context managers - *args and **kwargs

*args and **kwargs let any method accept any combination of arguments and pass them through untouched.

```
# *args collects all positional arguments into a tuple.

def f(*args):
    print(args)

f(1, 2, 3)   # → (1, 2, 3)

# **kwargs collects all keyword arguments into a dictionary.

def f(**kwargs):
   print(kwargs)

f(a=1, b=2)  # → {'a': 1, 'b': 2}
```

Decorators and context managers - Decorators

A decorator lets you wrap a function with extra behavior without changing its code. You pass a function into another function that returns a modified version of it.

```
import time

def timer(func):
    def wrapper(*args, **kwargs):
        start = time.time()
        result = func(*args, **kwargs)
        end = time.time()
        print(f"{func.__name__} took {end - start:.4f}s")
        return result
    return wrapper
```

```
@timer
def slow_add(a, b):
    time.sleep(0.5)
    return a + b

slow_add(3, 4)


# Expected output:
# slow_add took 0.5003s
# 7
```

Decorators and context managers - Context managers

A context manager sets something up before a block of code, and cleans it up afterward. They are commonly used for managing resources (files, connections, locks).

```
class MyContext:
    def __enter__(self):
        print("Entering")
        return self

    def __exit__(self, exc_type, exc, tb):
        print("Exiting")

with MyContext():
        print("Inside")
```

```
from contextlib import
contextmanager

@contextmanager
def my_context():
    print("Entering")
    yield
    print("Exiting")

with my_context():
    print("Inside")
```

Decorators and context managers

In summary :

- decorators are used to wrap a function
- context manager a block of code.

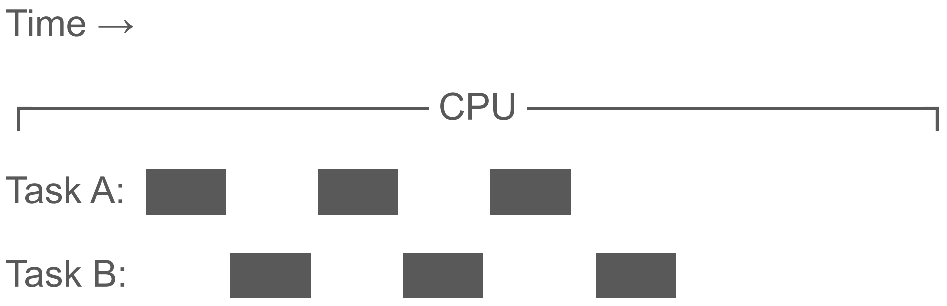
Agenda

- Decorators and context managers
- Concurrency and parallelism (threading, multiprocessing, asyncio)
- Type hints, dataclasses, and Pydantic models
- Performance tuning and profiling

Concurrency and parallelism - concurrency

Concurrency (one worker switching tasks)

One CPU core, but tasks *take turns*:



Concurrency and parallelism - concurrency

```python
import threading
import time

def task(name):
    print(f"{name} start")
    time.sleep(1)  # Simulates I/O
    print(f"{name} end")

t1 = threading.Thread(target=task, args=("A",))
t2 = threading.Thread(target=task, args=("B",))

t1.start() # Start thread
t2.start()
t1.join()  # Wait for completion
t2.join()
```

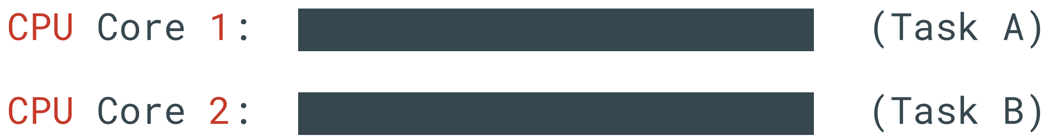
Concurrency and parallelism - concurrency

```python
import asyncio


async def task(name):
    print(f"{name} start")
    await asyncio.sleep(1)  # async wait -> you decide when to switch
    print(f"{name} end")


async def main():
    await asyncio.gather(task("A"), task("B"))

await main()
```

Concurrency and parallelism - Parallelism

Parallelism (multiple workers in true simultaneous execution)

Multiple CPU cores:



```
They run literally at the same time.
```

Concurrency and parallelism - Parallelism

```
from multiprocessing import Process
import time

def compute():
    start = time.time()
    s = sum(i * i for i in range(10_000_000))
    print("Done in:", time.time() - start)

p1 = Process(target=compute)
p2 = Process(target=compute)

p1.start()
p2.start()
p1.join()
p2.join()
```

Agenda

- Decorators and context managers
- Concurrency and parallelism (threading, multiprocessing, asyncio)
- Type hints, dataclasses, and Pydantic models
- Performance tuning and profiling

Type hints, dataclasses, and Pydantic models

Type hints describe what types your variables and functions expect. They do not enforce types at runtime, but help with: readability, IDE autocomplete, static checkers (mypy, pyright).

```
def add(a: int, b: int) -> int:
    return a + b


print(add(3,5)) # return 8
print(add("a","b")) # return ab
```

Type hints, dataclasses, and Pydantic models

A dataclass is a lightweight way to create data containers without boilerplate.
Python auto-generates:

__ init __

__ repr __

__ eq __

```python
from dataclasses import dataclass

@dataclass
class User:
    name: str
    age: int

    def is_adult(self) -> bool:
        return self.age >= 18
```

Type hints, dataclasses, and Pydantic models

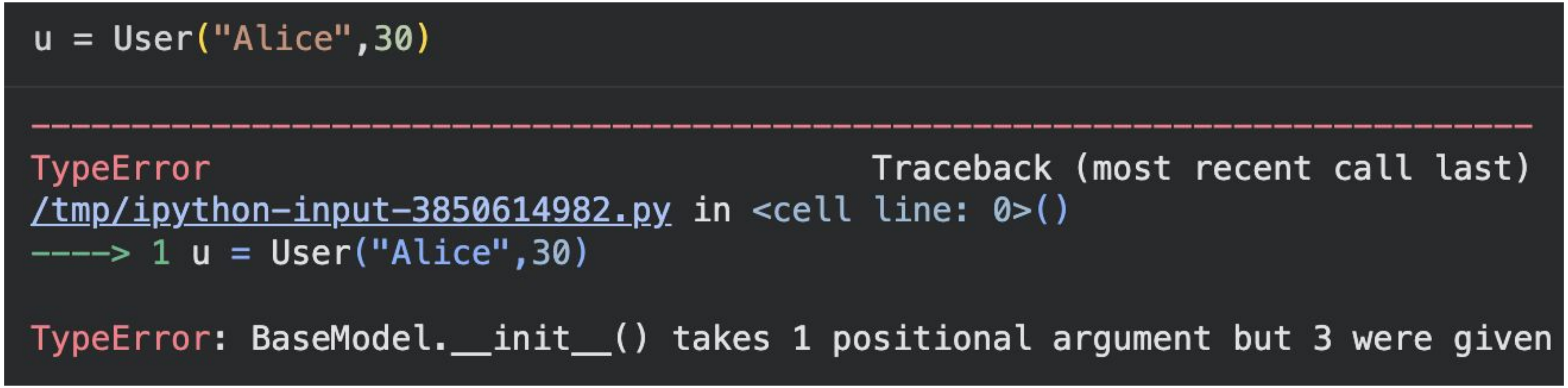
A Pydantic model is like a dataclass with validation and parsing.

It:

- checks types at runtime
- converts data when possible
- raises clear errors when invalid

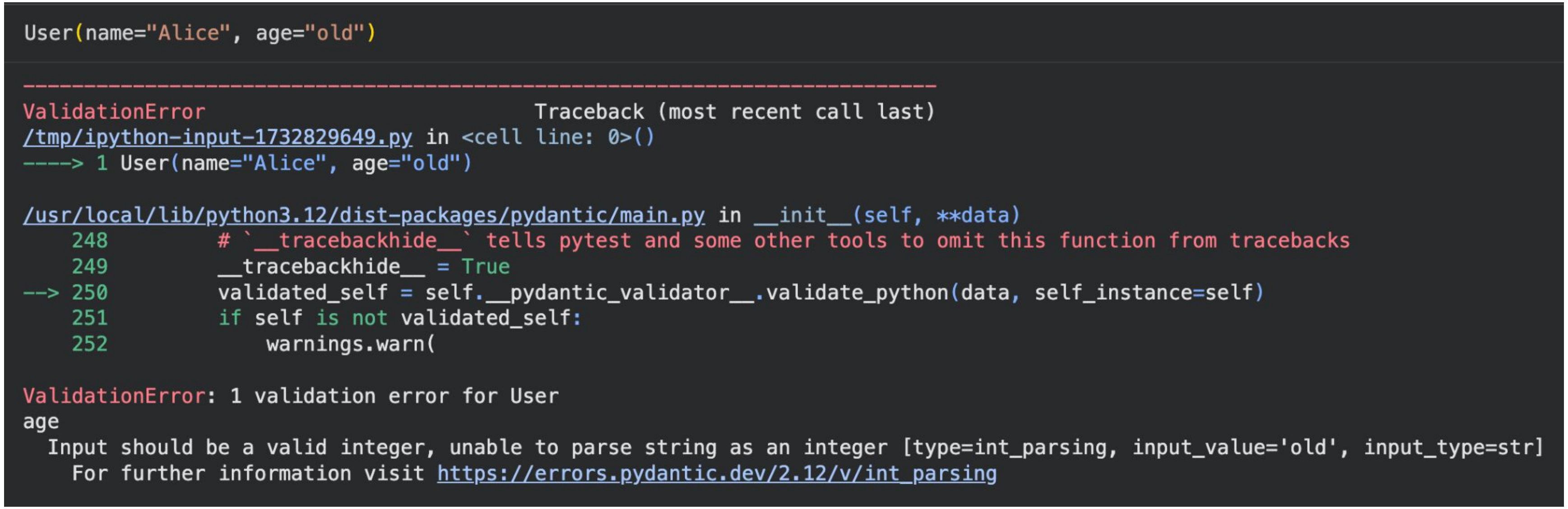
```python
from pydantic import BaseModel

class User(BaseModel):
    name: str
    age: int

u = User(name="Alice", age="30")
```

Type hints, dataclasses, and Pydantic models



```
u = User("Alice",30)

---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
/tmp/ipython-input-3850614982.py in <cell line: 0>()
----> 1 u = User("Alice",30)

TypeError: BaseModel.__init__() takes 1 positional argument but 3 were given
```

Type hints, dataclasses, and Pydantic models



```
User(name="Alice", age="old")
```

```
---------------------------------------------------------------------------
ValidationError                           Traceback (most recent call last)
/tmp/ipython-input-1732829649.py in <cell line: 0>()
----> 1 User(name="Alice", age="old")

/usr/local/lib/python3.12/dist-packages/pydantic/main.py in __init__(self, **data)
    248         # `__tracebackhide__` tells pytest and some other tools to omit this function from tracebacks
    249         __tracebackhide__ = True
--> 250         validated_self = self.__pydantic_validator__.validate_python(data, self_instance=self)
    251         if self is not validated_self:
    252             warnings.warn(

ValidationError: 1 validation error for User
age
  Input should be a valid integer, unable to parse string as an integer [type=int_parsing, input_value='old', input_type=str]
    For further information visit https://errors.pydantic.dev/2.12/v/int_parsing
```

Agenda

- Decorators and context managers
- Concurrency and parallelism (threading, multiprocessing, asyncio)
- Type hints, dataclasses, and Pydantic models
- Performance tuning and profiling

Performance tuning and profiling

It might be obvious but:

→ Don't optimize blindly. Measure first.

This should be taken with a grain of salt but Python is often fast enough. Optimize only when you know where time is spent.

Performance tuning and profiling - Timing

```python
# Basic example

import time

start = time.perf_counter()

time.sleep(1)

stop = time.perf_counter()

print(f"Time: {stop-start} s")
```

Performance tuning and profiling - Timing

```python
# Basic example

import time

for _ in range(3):

  start = time.perf_counter()

  sum(range(1_000))

  stop = time.perf_counter()

  print(f"Time: {stop-start} s")
```

As far as I know you can't have "Hard" real-time:

```
Time: 2.087199e-05 s
Time: 2.162400e-05 s
Time: 1.862499e-05 s
```

Performance tuning and profiling - Timing

```
# Basic example

import timeit

time = timeit.timeit(

    "sum(range(1_000))",

    number=10_000

)


print("Total time:", time)

print(f"Time per iteration {time/10_000}")
```

Previous implementation:

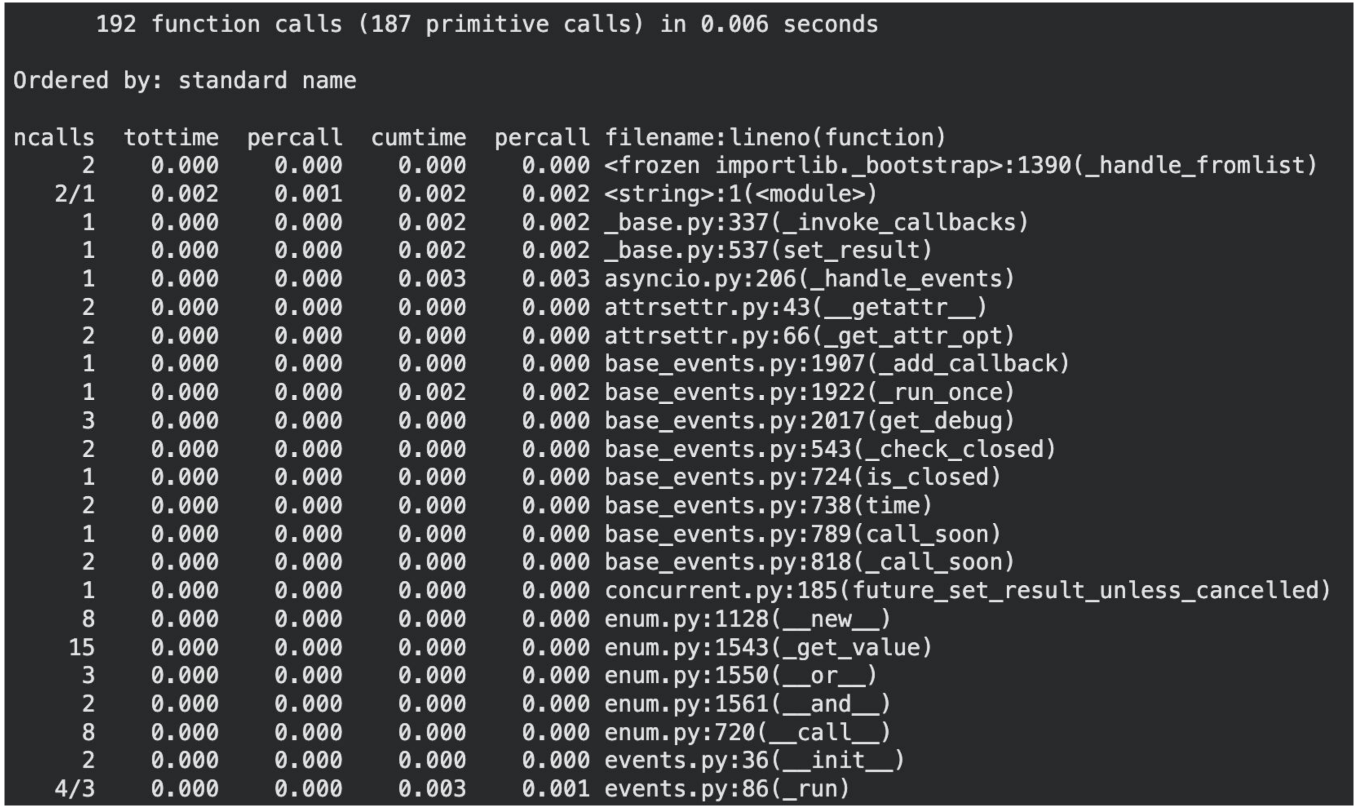
```
Time: 2.087199e-05 s
Time: 2.162400e-05 s
Time: 1.862499e-05 s


Total time: 0.193915
Time per iteration
1.939155e-05
```

Performance tuning and profiling - Profiling

cProfile doesn’t tell you what’s slow, it tells you *where* the time goes. Optimize the functions with the highest cumulative time.

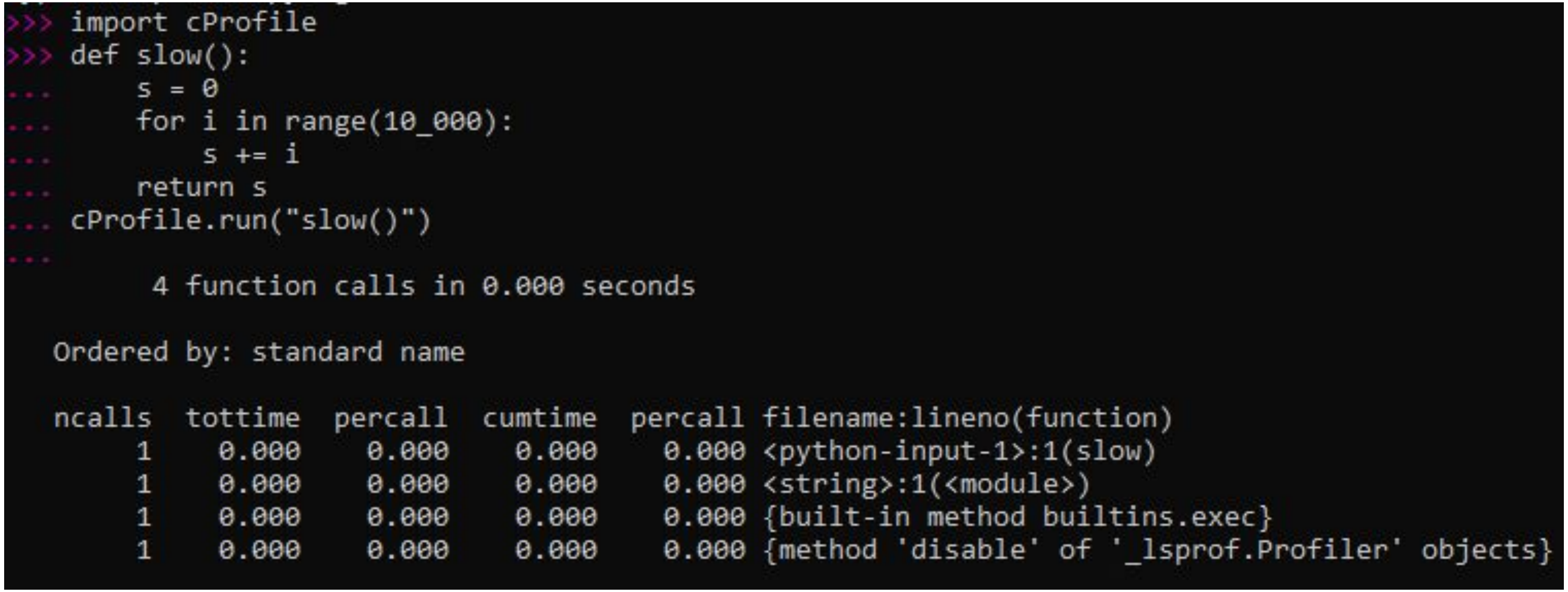
```
import cProfile

def slow():

    s = 0
    for i in range(10_000):
        s += i
    return s

cProfile.run("slow()")
```

Performance tuning and profiling - Profiling



```
      192 function calls (187 primitive calls) in 0.006 seconds

Ordered by: standard name

ncalls  tottime  percall  cumtime  percall filename:lineno(function)
     2    0.000    0.000    0.000    0.000 <frozen importlib._bootstrap>:1390(_handle_fromlist)
   2/1    0.002    0.001    0.002    0.002 <string>:1(<module>)
     1    0.000    0.000    0.002    0.002 _base.py:337(_invoke_callbacks)
     1    0.000    0.000    0.002    0.002 _base.py:537(set_result)
     1    0.000    0.000    0.003    0.003 asyncio.py:206(_handle_events)
     2    0.000    0.000    0.000    0.000 attrsettr.py:43(__getattr__)
     2    0.000    0.000    0.000    0.000 attrsettr.py:66(_get_attr_opt)
     1    0.000    0.000    0.000    0.000 base_events.py:1907(_add_callback)
     1    0.000    0.000    0.002    0.002 base_events.py:1922(_run_once)
     3    0.000    0.000    0.000    0.000 base_events.py:2017(get_debug)
     2    0.000    0.000    0.000    0.000 base_events.py:543(_check_closed)
     1    0.000    0.000    0.000    0.000 base_events.py:724(is_closed)
     2    0.000    0.000    0.000    0.000 base_events.py:738(time)
     1    0.000    0.000    0.000    0.000 base_events.py:789(call_soon)
     2    0.000    0.000    0.000    0.000 base_events.py:818(_call_soon)
     1    0.000    0.000    0.000    0.000 concurrent.py:185(future_set_result_unless_cancelled)
     8    0.000    0.000    0.000    0.000 enum.py:1128(__new__)
    15    0.000    0.000    0.000    0.000 enum.py:1543(_get_value)
     3    0.000    0.000    0.000    0.000 enum.py:1550(__or__)
     2    0.000    0.000    0.000    0.000 enum.py:1561(__and__)
     8    0.000    0.000    0.000    0.000 enum.py:720(__call__)
     2    0.000    0.000    0.000    0.000 events.py:36(__init__)
   4/3    0.000    0.000    0.003    0.001 events.py:86(_run)
```

Performance tuning and profiling - Profiling



```
>>> import cProfile
>>> def slow():
...     s = 0
...     for i in range(10_000):
...         s += i
...     return s
... cProfile.run("slow()")
...
         4 function calls in 0.000 seconds

   Ordered by: standard name

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000    0.000    0.000 <python-input-1>:1(slow)
        1    0.000    0.000    0.000    0.000 <string>:1(<module>)
        1    0.000    0.000    0.000    0.000 {built-in method builtins.exec}
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
```

Performance tuning and profiling - How to ? (Built-in)

When possible use built-in function:

```
def slow():    # 0.510 ms
  s = 0
  for i in range(10_000):
      s += i
  return s

def faster():  # 0.223 ms

  return sum(range(10_000))
```

Performance tuning and profiling - How to ? (RAM)

```python
import tracemalloc
tracemalloc.start()
t0 = time.perf_counter()
# do something
t1 = time.perf_counter()
ram_current, ram_peak = tracemalloc.get_traced_memory()
tracemalloc.stop()
print(f"Time: {t1 - t0:.6f} s")
print(f"Peak memory: {ram_peak / 1024 / 1024:.2f} MB")
```

Performance tuning and profiling - How to ? (RAM)

```
from functools import lru_cache

@lru_cache        # Without 0.113 ms
def fib(n):       # With    0.047 ms
    if n < 2:
        return n
    return fib(n - 1) + fib(n - 2)
```

Caching means trading memory for speed:
Reading from RAM (cached result): ~100 nanoseconds

Performance tuning and profiling - How to ? (RAM)

```python
from functools import lru_cache

@lru_cache
def square(n):
    print("Computing...")
    return n * n

square(4)
square(4)
```

Without
```
Computing...
Computing...
```

With
```
Computing...
```

Performance tuning and profiling - How to ? (RAM)

A **generator** is a function (or expression) that **produces values one at a time**, instead of creating them all at once.

```
gen = (i * i for i in range(5))
```

A common example in ML:

```
from pathlib import Path
from PIL import Image
def iter_images(directory):
    for path in Path(directory).iterdir():
        if path.suffix.lower() in {".jpg", ".png", ".jpeg"}:
            yield Image.open(path)
```

Performance tuning and profiling - How to ? (RAM)

```
Task: Generate an array 2’000’000 elements and sum the first
5.

List implementation
Time: 1.810434 s
Peak memory: 77.36 MB

Generator implementation
Time: 0.000270 s
Peak memory: 0.02 MB
```