Twitter Thread by Jan Giacomelli

Jan Giacomelli
@jangiacomelli

■ Python generators

What are they? How to use them?

#Python

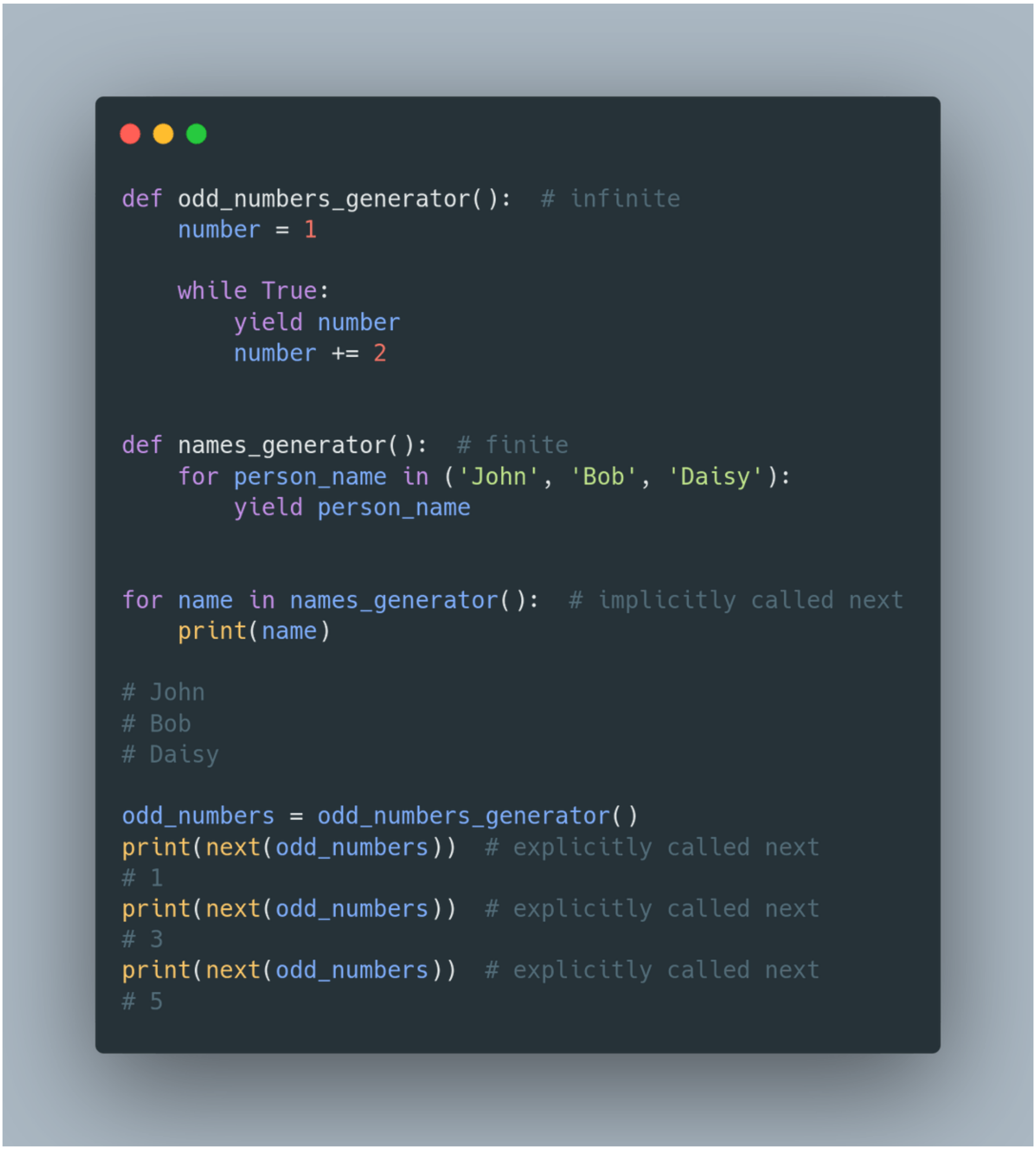
■Let's find out ■

1■■ Python generators are lazy iterators delivering the next value when their .next() is called.

They are created by using the yield keyword

next() can be called explicitly or implicitly inside for loop

They can be finite or infinite

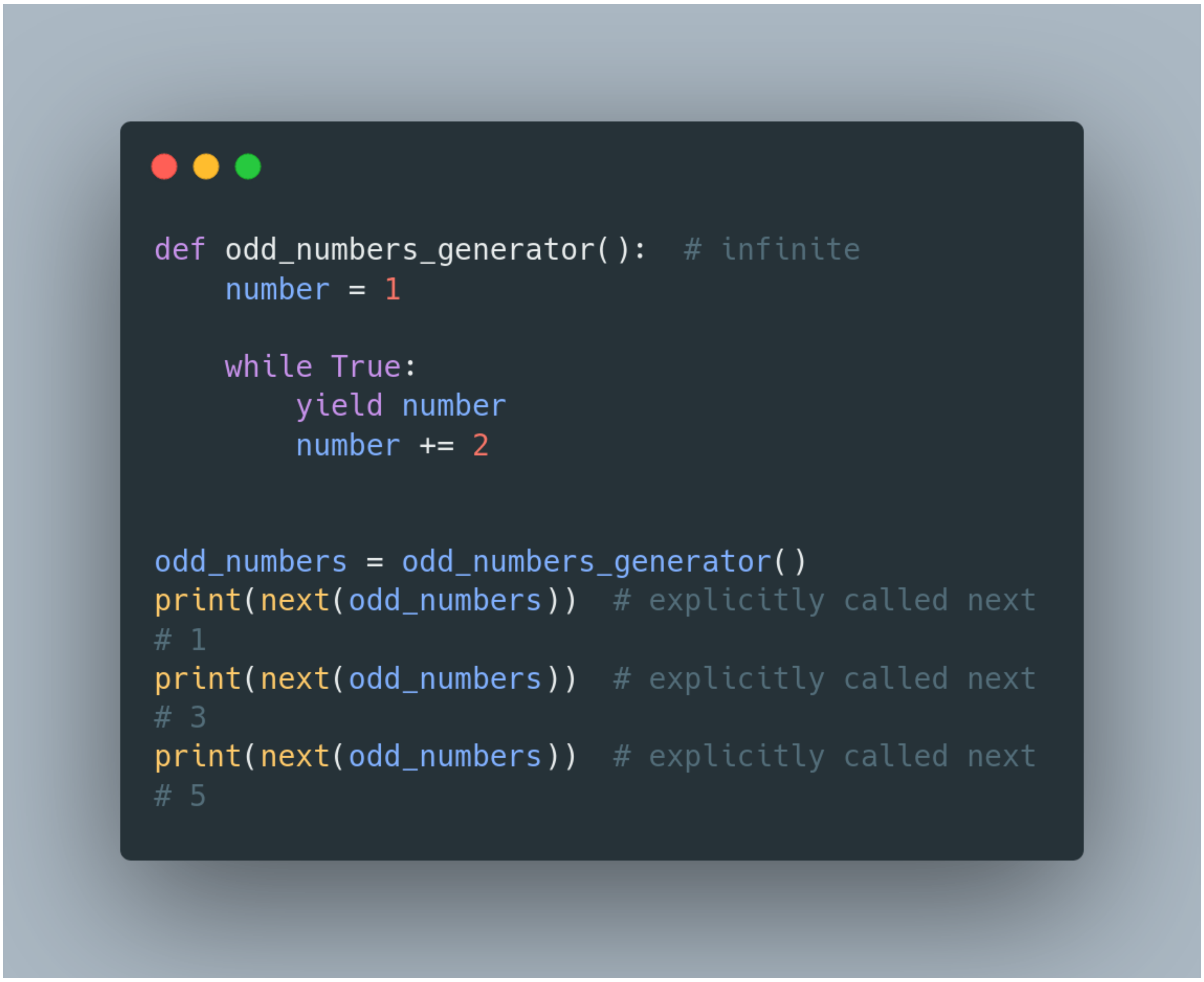


```python
def odd_numbers_generator():  # infinite
    number = 1

    while True:
        yield number
        number += 2


def names_generator():  # finite
    for person_name in ('John', 'Bob', 'Daisy'):
        yield person_name


for name in names_generator():  # implicitly called next
    print(name)

# John
# Bob
# Daisy

odd_numbers = odd_numbers_generator()
print(next(odd_numbers))  # explicitly called next
# 1
print(next(odd_numbers))  # explicitly called next
# 3
print(next(odd_numbers))  # explicitly called next
# 5
```

2■■ yield - where a value is sent back to the caller, but the function doesn’t exit afterward as with the return statement

The state of function is remembered.

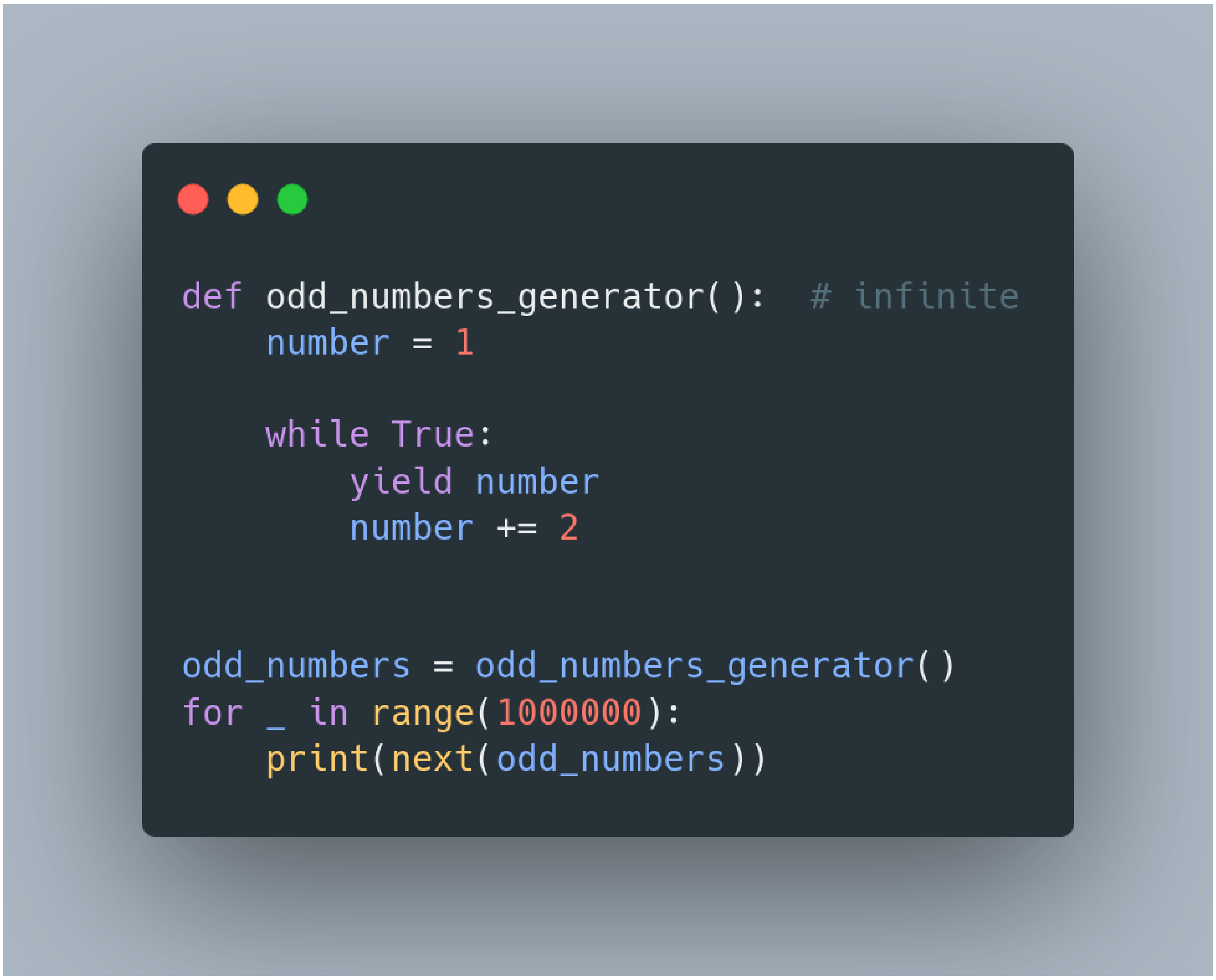
For example, the number is incremented and sent back from yield at the consecutive call next()



```python
def odd_numbers_generator():  # infinite
    number = 1

    while True:
        yield number
        number += 2


odd_numbers = odd_numbers_generator()
print(next(odd_numbers))  # explicitly called next
# 1
print(next(odd_numbers))  # explicitly called next
# 3
print(next(odd_numbers))  # explicitly called next
# 5
```

3■■ Generator stores only the current state of the function - it generates next element on next() call and forgets the previous one -> it saves memory

For example, we don't need to store 1mio elements in memory to do something with each element

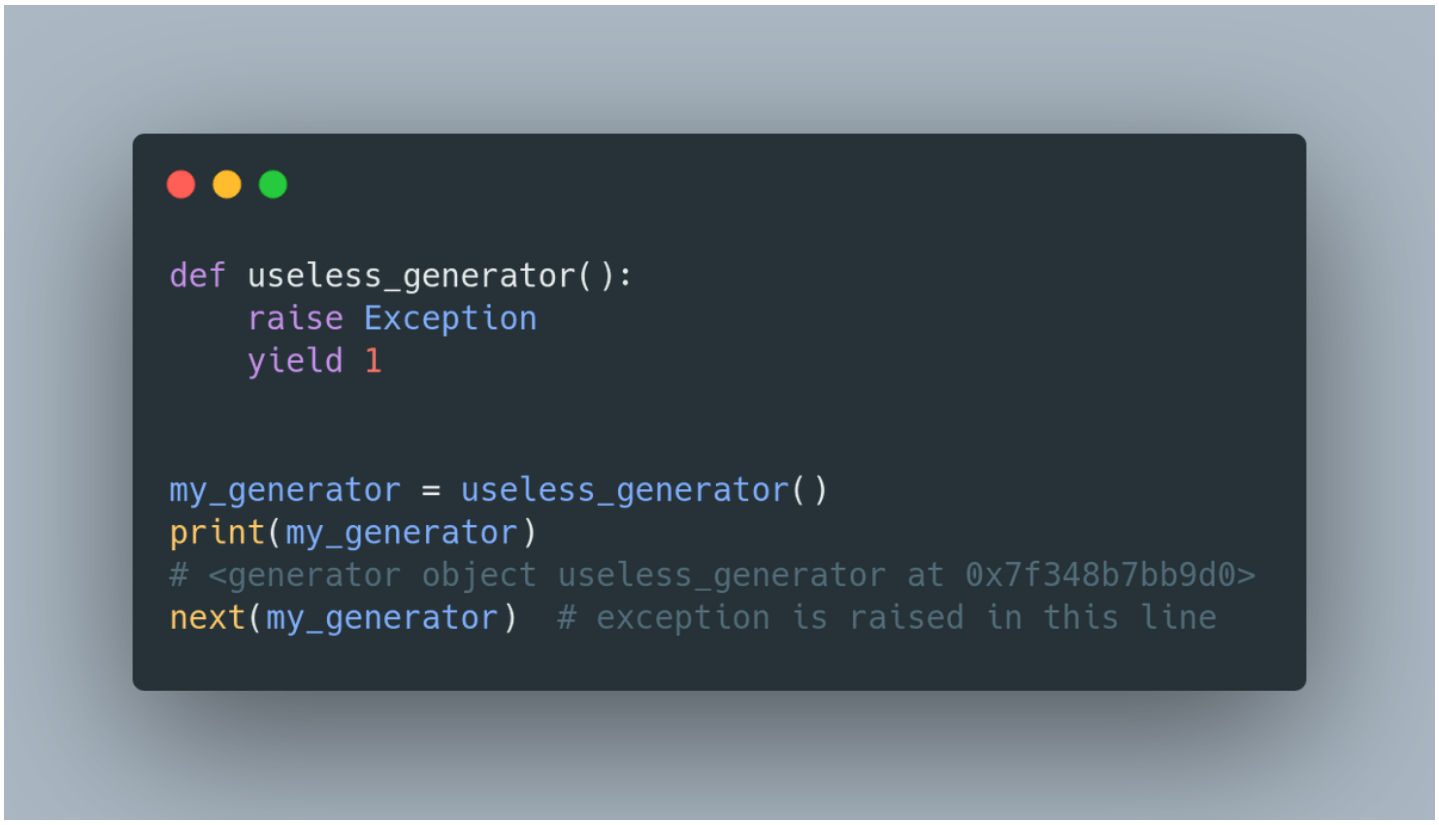


```python
def odd_numbers_generator():  # infinite
    number = 1

    while True:
        yield number
        number += 2


odd_numbers = odd_numbers_generator()
for _ in range(1000000):
    print(next(odd_numbers))
```

4■■ When you call a generator function generator object is returned - it's not executed yet

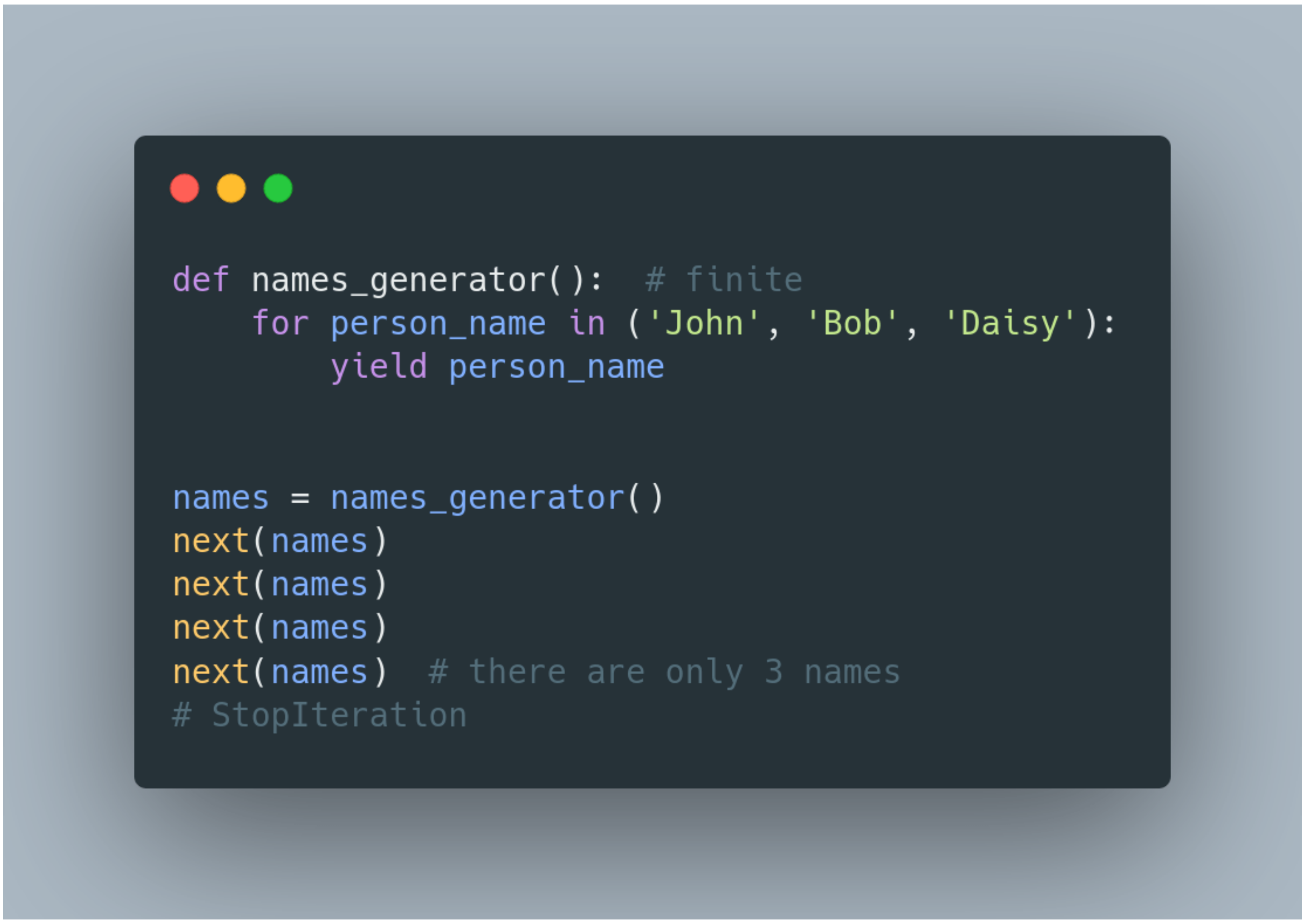
It executes only when next() is called

For example, that's why Exception is raised only on the next() call



```python
def useless_generator():
    raise Exception
    yield 1


my_generator = useless_generator()
print(my_generator)
# <generator object useless_generator at 0x7f348b7bb9d0>
next(my_generator)  # exception is raised in this line
```

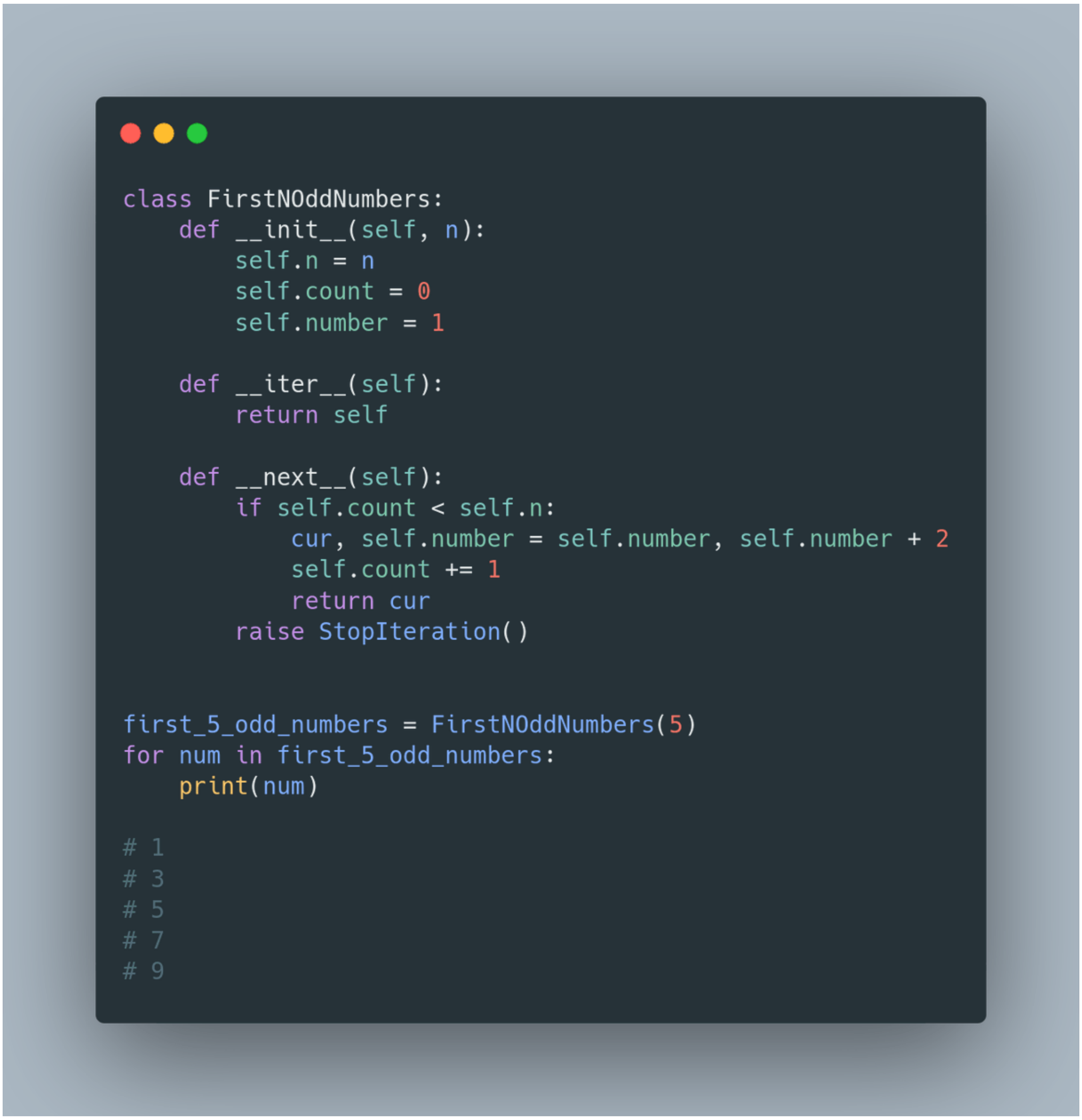
5■■ When the generator goes out of elements it raises StopIteration exception



```python
def names_generator():  # finite
    for person_name in ('John', 'Bob', 'Daisy'):
        yield person_name


names = names_generator()
next(names)
next(names)
next(names)
next(names)  # there are only 3 names
# StopIteration
```

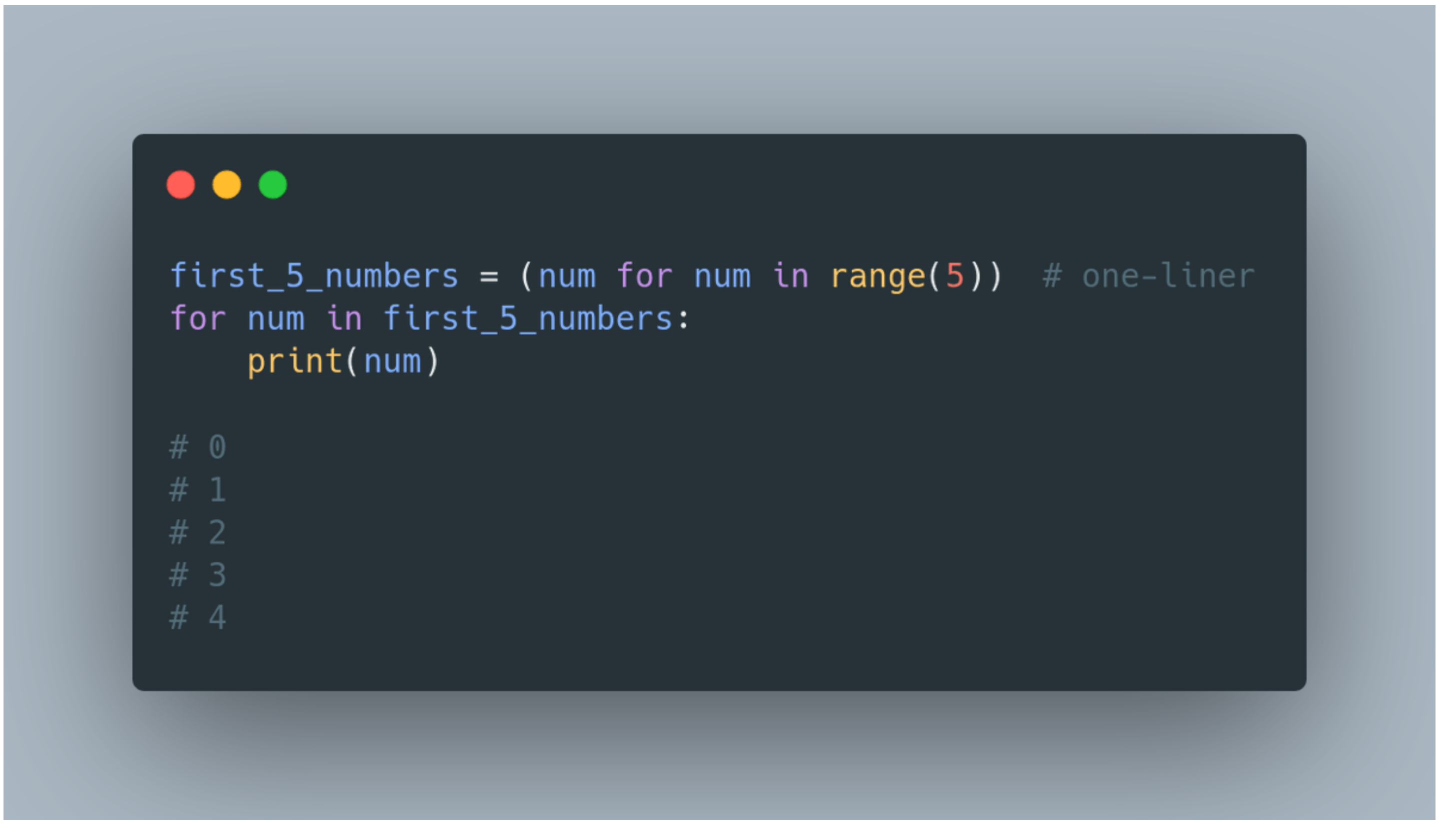
6■■ You can also create a class that behaves like a generator - it needs implemented methods:

- __iter__ -> to enable iteration
- __next__ -> to enable next element access



```python
class FirstNOddNumbers:
    def __init__(self, n):
        self.n = n
        self.count = 0
        self.number = 1

    def __iter__(self):
        return self

    def __next__(self):
        if self.count < self.n:
            cur, self.number = self.number, self.number + 2
            self.count += 1
            return cur
        raise StopIteration()


first_5_odd_numbers = FirstNOddNumbers(5)
for num in first_5_odd_numbers:
    print(num)

# 1
# 3
# 5
# 7
# 9
```

7■■ You can also create a generator with one-liner expression similar to list comprehension



```python
first_5_numbers = (num for num in range(5))  # one-liner
for num in first_5_numbers:
    print(num)

# 0
# 1
# 2
# 3
# 4
```

8■■ You can read more:

https://t.co/mXp7L8l2Ai