Iterators, generators and asyncio in Python 3

SERGII TISHCHENKO

GITHUB LINK: HTTPS://TISHYK@BITBUCKET.ORG/TISHYK/DEMOS.GIT

What is iterator?

How and where we can use it?

What is generator?

Yield statement

Asynchronous programming with generators

Python concurrent programming without asyncio library

Concurrent programming with asyncio

Coroutine, tasks and event-loops

What is iterator?

Iterators are everywhere in Python. They are elegantly implemented within for loops, comprehensions, generators etc. but hidden in plain sight.

Iterator in Python is simply an object that can be iterated upon. An object which will return data, one element at a time.

Python iterator object must implement two special methods, __iter__() and __next__(), collectively called the iterator protocol.

An object is called iterable if we can get an iterator from it. Most of built-in containers in Python like: list, tuple, string etc. are iterables.

The iter() function (which in turn calls the __iter__() method) returns an iterator from them.

What is iterator? Iterator protocol

There are only a couple of functions specifically for working with iterators.

```
int PyIter_Check( PyObject *o)
   Return true if the object o supports the iterator protocol.

PyObject* PyIter_Next( PyObject *o)
   Return value: New reference.
   Return the next value from the iteration o.
```

If the object is an iterator, this retrieves the next value from the iteration, and returns NULL with no exception set if there are no remaining items. If the object is not an iterator, TypeError is raised, or if there is an error in retrieving the item, returns NULL and passes along the exception.

What is iterator? Iterator protocol

```
>>> # define a list
>>> my_list = [4, 7, 0, 3]
>>> # get an iterator using iter()
>>> my_iter = iter(my_list)
>>> ## iterate through it using next()
>>> print(next(my_iter))
4
>>> print(next(my_iter))
7
>>> print(my_iter.__next__())
0
>>> print(my_iter.__next__())
3
>>> next(my_iter)
Traceback (most recent call last):
  File "<pyshell#13>", line 1, in <module>
    next(my_iter)
StopIteration
```

What is iterator? Iterator examples

```
for match in re.finditer(pattern, string):
    # once for each regex match ...

for root, dirs, files in os.walk('./some/dir'):
    # once for each sub-directory...

for num in itertools.count():
    # once for each integer... Infinite!

from itertools import chain, repeat, cycle
seq = chain(repeate(17,3), cycle(range(3)))
for num in seq:
    # 17, 17, 17, 0, 1, 2, 3, 0, 1, 2, 3, 0, 1, 2, 3
```

What is generator?

- Generator functions are commonly used to feed values to for-loops (iteration).

A generator is a function that returns an object (iterator) which we can iterate over (one value at a time).

```
def countdown(n):
    while n > 0:
        yield n
        n -= 1

for x in countdown(10):
    print(x)
```

- Under the covers, the countdown function executes on successive next() calls

```
>>> c = countdown(10)
>>> c
<generator object countdown at 0x000001AAFBE32518>
>>> next(c)
10
>>> next(c)
9
```

What is generator?

Whenever a generator function hits the yield statement, it suspends execution

```
def countdown(n):
    while n > 0:
        yield n
        n -= 1
```

Here's the idea : Instead of yielding a value, a generator can yield control

We can write a little scheduler that cycles between generators, running each one until it explicitly yields

What is generator?

1. Set up a set of "tasks“
2. Each task is a generator function
3. Run a task scheduler

```
def countdown_task(n):
    while n > 0:
        print(n)
        yield n
        n -= 1

# A list of tasks to run
from collections import deque
tasks = deque([countdown_task(5),
               countdown_task(7),
               countdown_task(9)])

def scheduler(tasks):
    while tasks:
        task = tasks.popleft()
        try:
            next(task)
            tasks.append(task)
        except StopIteration:
            pass

scheduler(tasks)
```

This while loop is what drives the application between 3 tasks

Generator is a way to control flow between different tasks and run it concurrent

Asynchronous programming with generators. What for?

- I/O is high latency
- Sequential programs waste resources waiting on I/O
- Multithreading/multiprocessing carry
  - Large resource overheads
  - Large cognitive overheads
- Python interpreter is shared mutable state protected by Global Interpreter Lock
- Make a single Python process run as fast as possible without any thread/process overhead

Asynchronous programming with generators. Fibonacci sequence

```
def fibonacci():
    yield 2
    a = 2
    b = 1
    while True:
        yield b
        a, b = b, a + b
```

```
>>> from main import fibonacci
>>> from itertools import islice
>>>
>>> list(islice(fibonacci(), 10))
[2, 1, 3, 4, 7, 11, 18, 29, 47, 76]
```

Asynchronous programming with generators. Linear search

```
def search(iterable, predicate):
    """ Return the first item satisfying a predicate """
    for item in iterable:
        if predicate(item):
            return item
    raise ValueError('Not found')


>>> from main import search
>>> search(fibonacci(), lambda x: len(str(x)) >= 6)
103682
```

Asynchronous programming with generators. Cooperative search 1

```
def async_search(iterable, predicate):
    for item in iterable:
        if predicate(item):
            return item
        yield
    raise ValueError('Not found')
```

Asynchronous programming with generators. Cooperative search 2

```
>>> from main import async_search
>>> g = async_search(fibonacci(), lambda x: x >= 10)
>>> g
<generator object async_search at 0x000002CA101B1518>
>>> next(g)
>>> next(g)
>>> next(g)
>>> print("Do something useful here")
Do something useful here
>>> next(g)
>>> next(g)
>>> next(g)
Traceback (most recent call last):
  File "<pyshell#17>", line 1, in <module>
    next(g)
StopIteration: 11
```

Asynchronous programming with generators. Cooperative search 2

```
>>> from main import async_search
>>> g = async_search(fibonacci(), lambda x: x >= 10)
>>> g
<generator object async_search at 0x000002CA101B1518>
>>> next(g)
>>> next(g)
>>> next(g)
>>> print("Do something useful here")
Do something useful here
>>> next(g)
>>> next(g)
>>> next(g)
Traceback (most recent call last):
  File "<pyshell#17>", line 1, in <module>
    next(g)
StopIteration: 11
```

Asynchronous programming with generators. Task

```
class Task:
    """ Aggregates a coroutine and integer id """
    next_id = 0

    def __init__(self, routine):
        self.id = Task.next_id
        Task.next_id += 1
        self.routine = routine
```

Asynchronous programming with generators. Scheduler 1

```
class Scheduler:
    def __init__(self):
        self.executable_tasks = deque()
        self.completed_task_results = {}
        self.failed_task_error = {}

    def add(self, routine):
        task = Task(routine)
        self.executable_tasks.append(task)
        return task.id
```

Asynchronous programming with generators. Scheduler 2

```
def run_until_complete(self):
    while len(self.executable_tasks) !=0:
        task = self.executable_tasks.popleft()
        print("Running task {} ... ".format(task.id), end='')
        try:
            yielded = next(task.routine)
        except StopIteration as stopped:
            print("completed with result: {!r}".format(stopped.value))
            self.completed_task_results[task.id] = stopped.value
        except Exception as exec:
            print("failed with exception: {}".format(exec))
        else:
            assert yielded is None
            print('now yielded')
            self.executable_tasks.append(task)
```

Asynchronous programming with generators. Task execution

```
>>> from main import Scheduler
>>> scheduler = Scheduler()
>>> scheduler.add(async_search(fibonacci(), lambda x: len(str(x))>=6))
0
>>> scheduler.run_until_complete()
Running task 0 ... now yielded
Running task 0 ... now yielded
Running task 0 ... now yielded
 .
 .
 .
Running task 0 ... now yielded
Running task 0 ... completed with result: 103682
```

Asynchronous programming with generators. Blocking task

```
from math import sqrt

def is_prime(x):
    if x < 2:
        return False
    for i in range(2, int(sqrt(x)+1)):
        if x % i == 0:
            return False
    return True
```

```
>>> is_prime(12)
False
>>> is_prime(13)
True
>>> is_prime(2**31 - 1)
True
>>> is_prime(2**61 - 1)
True
```

Simple but inefficient with long numbers

Asynchronous programming with generators. Blocking task

```
>>> scheduler = Scheduler()
>>> scheduler.add(async_print_matches(fibonacci(), is_prime))
0
>>> scheduler.run_until_complete()
Running task 0 ... Found:  2, now yielded
Running task 0 ... now yielded
Running task 0 ... Found:  3, now yielded
Running task 0 ... now yielded
Running task 0 ... Found:  7, now yielded
Running task 0 ... Found:  11, now yielded
Running task 0 ... now yielded
Running task 0 ... Found:  29, now yielded
Running task 0 ... Found:  47, now yielded
Running task 0 ... now yielded
Running task 0 ... now yielded
Running task 0 ... Found:  199, now yielded
Running task 0 ... now yielded
Running task 0 ... Found:  521, now yielded
Running task 0 ... now yielded
```

Asynchronous programming with generators. Blocking task

```
def async_print_matches(iterable, predicate):
    "Similar to async_search, but prints all matches"

    for item in itareble:
        if predicate(item):
            print("Found: ", item, end=', ')
        yield

def async_repetitive_msg(msg, interval_seconds):
    while True:
        print(msg)
        start = time.time()
        expiry = start + interval_seconds
        while True:
            yield  # Ensure coroutine always yield at least once
            now = time.time()
            if now >= expiry:
                break
```

Asynchronous programming with generators. Blocking task

```
>>> scheduler = Scheduler()
>>> scheduler.add(async_print_matches(fibonacci(), is_prime))
0
>>> scheduler.add(async_repetitive_msg('Interval message', 2))
1
>>> scheduler.run_until_complete()
Running task 0 ... Found:  2, now yielded
Running task 1 ... Interval message
now yielded
Running task 0 ... now yielded
Running task 1 ... now yielded
Running task 0 ... Found:  3, now yielded
 .
 .
 .
Running task 1 ... Interval message
now yielded
Running task 0 ... now yielded
Running task 1 ... now yielded
```

Asynchronous programming with generators. Blocking task

Everything you call -transitively- from a coroutine should be non-blocking.

Coroutines are contagious to callees.

Asynchronous programming with generators. Non-blocking task

```
from math import sqrt

def is_prime(x):
    if x < 2:
        return False
    for i in range(2, int(sqrt(x)+1)):
        if x % i == 0:
            return False
    return True
```

```
def async_is_prime(x):
    if x < 2:
        return False
    for i in range(2, int(sqrt(x)+1)):
        if x % i == 0:
            return False
        yield
    return True
```

Return a bool

Return a generator object

Asynchronous programming with generators. Non-blocking task

```
def async_print_matches(iterable, async_predicate):
    "Similar to async_search, but prints all matches"

    for item in iterable:
        ''' Allow the predicate to make progress and
            yield control by invoking with yield from.
            Nested generator'''
        matches = yield from async_predicate(item)
        if async_predicate(item):
            print("Found: ", item, end=', ')
        yield
```

generator object

no longer needed

Asynchronous programming with generators. Non-blocking task

Everything that call – transitively - to a coroutine must iterate the generator.

Coroutines are contagious to callers.

Asynchronous programming with generators. Refactoring

```
def async_repetitive_msg(msg, interval_seconds):
    while True:
        print(msg)
        start = time.time()
        expiry = start + interval_seconds
        while True:
            yield  # Ensure coroutine always yield at least once
            now = time.time()
            if now >= expiry:
                break
```

```
def async_repetitive_msg(msg, interval):
    while True:
        print(msg)
        yield from async_sleep(interval)
```

```
def async_sleep(interval):
    start = time.time()
    expiry = start + interval
    while True:
        yield
        now = time.time()
        if now >= expiry:
            break
```

Asynchronous programming with generators. Refactoring

```
def async_sleep(interval)
    start = time.time()
    expiry = start + interval
    while True:
        yield
        now = time.time()
        if now >= expiry:
            break
```

```
def async_is_prime(x):
    if x < 2:
        return False
    for i in range(2, int(sqrt(x)+1)):
        if x % i == 0:
            return False
        yield from async_sleep(0)
    return True

def async_search(iterable, predicate):
    for item in iterable:
        if predicate(item):
            return item
        yield from async_sleep(0)
    raise ValueError('Not found')

def async_print_matches(iterable, async_predicate):
    for item in iterable:
        matches = yield from async_predicate(item)
        if async_predicate(item):
            print("Found: ", item, end=', ')
```

Asynchronous programming with generators. Refactoring

```
>>> scheduler = Scheduler()
>>> scheduler.add(async_print_matches(fibonacci(), is_prime))
0
>>> scheduler.add(async_repetitive_msg('Interval message', 2))
1
>>> scheduler.run_until_complete()
Running task 0 ... Found:  2, now yielded
Running task 1 ... Interval message
now yielded
Running task 0 ... now yielded
Running task 1 ... now yielded
Running task 0 ... Found:  3, now yielded
Running task 1 ... now yielded
```

Concurrent programming with asyncio. Definitions 1

Concurrency

Tasks start, run, and complete in

overlapping time periods

asyncio

Parallelism

Tasks run simultaneously

Threads/Processes

Concurrent programming with asyncio. Definitions 2

Synchronous(Sequential)

Must complete before proceeding

Overall duration longer

Asynchronous

No need to wait before proceeding

Overall duration shorter

Concurrent programming with asyncio. Definitions 3

Asynchronous

Returns immediately with a promise for the complete result

Callbacks / Futures / Promises

Non-blocking

Returns immediately, with no result, partial result, or complete result

Polling

Concurrent programming with asyncio. Definitions 4

Preemptive multitasking

Scheduler interrupts tasks

Inconvenient context switches

Cooperative multitasking

Tasks yield to scheduler

Uncooperative tasks hang system

Concurrent programming with asyncio

asyncio

Write asynchronous, concurrent, cooperative tasks …

… in a sequential style.

https://upload.wikimedia.org/wikipedia/commons/9/97/12_Armige_Statue.JPG

Concurrent programming with asyncio. Refactor to asyncio code 1

```
import time
from math import sqrt

def fibonacci():
    yield 2
    a = 2
    b = 1
    while True:
        yield b
        a, b = b, a + b


def async_is_prime(x):
    if x < 2:
        return False
    for i in range(2, int(sqrt(x)+1)):
        if x % i == 0:
            return False
        yield from async_sleep(0)
    return True

def async_search(iterable, predicate):
    for item in iterable:
        if predicate(item):
            return item
        yield from async_sleep(0)
    raise ValueError('Not found')
```

```
def async_print_matches(iterable, async_predicate):
    for item in iterable:
        matches = yield from async_predicate(item)
        if async_predicate(item):
            print("Found: ", item, end=', ')

def async_repetitive_msg(msg, interval):
    while True:
        print(msg)
        yield from async_sleep(interval)

def async_sleep(interval):
    start = time.time()
    expiry = start + interval
    while True:
        yield
        now = time.time()
        if now >= expiry:
            break
```

Concurrent programming with asyncio. Refactor to asyncio code 2

```
import asyncio
import time
from math import sqrt

def fibonacci():
    yield 2
    a = 2
    b = 1
    while True:
        yield b
        a, b = b, a + b

async def is_prime(x):
    if x < 2:
        return False
    for i in range(2, int(sqrt(x)+1)):
        if x % i == 0:
            return False
        await asyncio.sleep(0)
    return True

async def search(iterable, predicate):
    for item in iterable:
        if predicate(item):
            return item
        await asyncio.sleep(0)
    raise ValueError('Not found')
```

```
async def print_matches(iterable, async_predicate):
    for item in iterable:
        matches = await async_predicate(item)
        if async_predicate(item):
            print("Found: ", item, end=', ')

async def repetitive_msg(msg, interval):
    while True:
        print(msg)
        await asyncio.sleep(interval)
```

```
def async_func ---> async def func
```

```
import asyncio
```

```
async_sleep ---> asyncio.sleep
```

```
yield from ---> await
```

Coroutine, tasks and event-loops

```
>>> scheduler = asyncio.get_event_loop()
>>> scheduler.create_task(repetitive_msg("Interval message", 2))
<Task pending coro=<repetitive_msg() running at C:\Users\tishyk\Desktop\main_async.py:38>>
>>> scheduler.create_task(print_matches(fibonacci(), is_prime))
<Task pending coro=<print_matches() running at C:\Users\tishyk\Desktop\main_async.py:32>>
>>> scheduler.run_forever()
Interval message
Found:  2
Found:  3
Found:  7
Found:  11
Found:  29
Found:  47
Found:  199
Found:  521
Found:  2207
Found:  3571
Found:  9349
Found:  3010349
Found:  54018521
Found:  370248451
Found:  6643838879
Interval message
Interval message
```

Coroutine, tasks and event-loops

- Coroutines implement tasks
- Coroutines await other coroutines
- Event-loop schedules concurrent tasks
- Tasks must not block
- Awaiting facilitates context switches
- To yield control without needing a result `await asyncio.sleep(0)`

Coroutine, tasks and event-loops. Coroutine

Coroutine

```
async def search(iterable, predicate):
    for item in iterable:
        if predicate(item):
            return item
        await asyncio.sleep(0)
    raise ValueError('Not found')
```

code / callable

Coroutine object

```
>>> coro = search(fibonacci(),is_prime)
>>> coro
<coroutine object search at 0x000002851B909620>
>>> coro2 = is_prime(11)
>>> coro2
<coroutine object is_prime at 0x000002851B9097D8>
>>> search
<function search at 0x000002851B91D598>
```

code + execution state / awaitable

Coroutine, tasks and event-loops. Feature

```
async def twenty_digit_prime(x):
    return (await is_prime(x))and len(str(x)) == 12

async def monitored_search(iterable, predicate, future):
    try:
        found_item = await search(iterable, predicate)
    except ValueError as not_found:
        future.set_exception(not_found)
    else: # no exception
        future.set_result(found_item)

async def monitor_feature(future, interval):
    while not future.done():
        print("Waiting ..")
        await asyncio.sleep(interval)
    print("Done!")
```

Encapsulates a potential result or error

```
loop = asyncio.get_event_loop()
future = loop.create_future()
coro_obj = monitored_search(fibonacci(), twenty_digit_prime, future)
loop.create_task(coro_obj)
loop.create_task(monitor_feature(future, 1))
loop.run_until_complete(future)
print(future.result())
loop.close()
```

```
Waiting ..
Waiting ..
Waiting ..
Waiting ..
Waiting ..
Waiting ..
Waiting ..
Waiting ..
119218851371
>>>
```

Coroutine, tasks and event-loops. Task

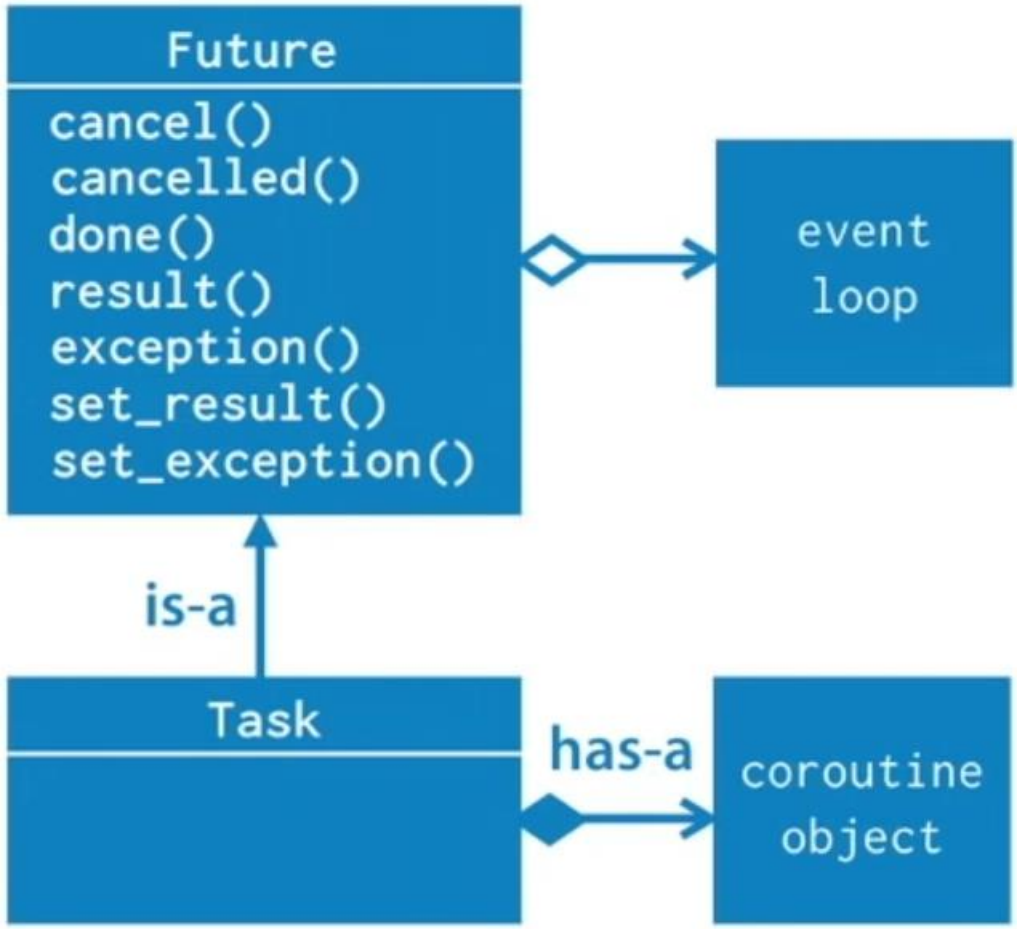
A subclass of a Future which wraps a coroutine

```
async def twenty_digit_prime(x):
    return (await is_prime(x))and len(str(x)) == 12

async def monitor_feature(future, interval):
    while not future.done():
        print("Waiting ..")
        await asyncio.sleep(interval)
    print("Done!")


loop = asyncio.get_event_loop()
future = loop.create_future()
coro_obj = search(fibonacci(), twenty_digit_prime)
search_task = loop.create_task(coro_obj)

loop.create_task(monitor_feature(search_task, 1))
loop.run_until_complete(search_task)
print(search_task.result())
loop.close()
```



Coroutine, tasks and event-loops. Event loop

AbstractEventLoop interfaces:

- `run_forever()`
- `run_until_complete(future)`
- `stop()`
- `is_closed()`
- `shutdown_asyncgens()`
- `create_future()`
- `create_task(coroutine_object)`

Дякую! Спасибо! Köszönjük! Дзякуй! Děkuji! Благодаря ти! Mulțumesc! Dziękuję Ci!

Thank You!