Deriving Asyncio 

Python Mauritius UserGroup (pymug)

More info: mscc.mu/python-mauritius-usergroup-pymug/

github: github.com/pymug
twitter: twitter.com/pymugdotcom
linkedin: linkedin.com/company/pymug
website: pymug.com
fb group: facebook.com/groups/318161658897893
fb page: facebook.com/pymug

Abdur-Rahmaan Janhangeer
(Just a Python programmer)

twitter: @osdotsystem
github: github.com/abdur-rahmaanj

compileralchemy.github.com

Driving Asyncio 

What are generators really?

The story starts with generators. Why were generators introduced?

> When a producer function has a hard enough job that it requires maintaining state between values produced, [1]

more explicitly

> provide a kind of function that can return an intermediate result ("the next value") to its caller, but maintaining the function's local state so that the function can be resumed again right where it left off. [1]

A kind of function that

- i. returns intermediate value
- ii. save state of fuctions

compared to normal functions, once you return from a function, you can go back to return more values

Normal function:

```
def x():
    print('abc')
    return
    print('def') # not reached
x()
```

generator function:

```
def x():
  a = 0
  while 1:
    yield a

    a += 1

z = x()

print(z)
print(next(z))
print(next(z))
```

```
<generator object x at 0x01ACB760>
0
1
```

To understand it better, here are some more names that were proposed instead of yield [1]

- return 3 and continue
- return and continue 3
- return generating 3
- continue return 3

Guido says [1]:

> In practice (how you think about them), generators are functions, but with the twist that they're resumable.

Execution Flow

```
def x():
  print('started')
  while 1:
    print('before yield')
    yield
    print('after yield')


z = x()

next(z)
print('-- 2nd call')
next(z)
```

```
started
before yield
-- 2nd call
after yield
before yield
```

Yield as stop points

```
def x():
  print('start')
  yield
  print('after 1st yield')
  yield
  print('after 2nd yield')

z = x()
next(z)
next(z)
next(z)
```

```
start
after 1st yield
after 2nd yield
Traceback (most recent call last):
  File "lab.py", line 11, in <module>
    next(z)
StopIteration
```

Auto handling of StopIteration

```
def x():
  print('start')
  yield
  print('after 1st yield')
  yield
  print('after 2nd yield')

z = x()
for _ in z:
  pass
```

```
def x():
  print('start')
  yield 1
  print('after 1st yield')
  yield 2
  print('after 2nd yield')

z = x()
for _ in z:
  print(_)
```

```
start
1
after 1st yield
2
after 2nd yield
```

Immediate usefulness

Since we saw that we can use yield with an infinite loop, this is extremely powerful. We can break infinity in steps. Consider this:

```
def odd_till(number):
  n = 1
  while n < number:
    yield n
    n += 2

for odd_num in odd_till(10):
  print(odd_num)
```

We yield one number and the function exists, the for loop calls it again. It yields one number and exits. And so on. It goes about it in micro steps.

`odd_till(10)` or `odd_till(1000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000)` don't not cause memory errors

Generators introduced for memory saving

Consider

```
sum([x*x for x in range(10)])
```

More efficient, generator expressions [2]:

```
sum(x*x for x in range(10))
```

Generators for tasks

Lets modify our two functions with print

```
def odd_till(number):
  n = 1
  while n < number:
    print('{} odd_till - {}'.format(number, n))
    yield n
    n += 2

def even_till(number):
  n = 2
  while n < number:
    print('{} even_till - {}'.format(number, n))
    yield n
    n += 2
```

Lets have a class to run functions

```
from collections import deque

class RunFunc:
  def __init__(self):
    self._queue = deque()

  def add_func(self, func):
    self._queue.append(func)

  def run(self):
    while self._queue:
      func = self._queue.popleft()
      try:
        next(func)
        self._queue.append(func)
      except StopIteration:
        pass
```

usage

```
func_runner = RunFunc()

func_runner.add_func(odd_till(5))
func_runner.add_func(even_till(4))
func_runner.add_func(odd_till(6))
func_runner.run()
```

output

```
5 odd_till - 1
4 even_till - 2
6 odd_till - 1
5 odd_till - 3
6 odd_till - 3
6 odd_till - 5
```

If we rename the same thing we get a mini task scheduler [4]

```
from collections import deque

class TaskScheduler:
  def __init__(self):
    self._queue = deque()

  def add_task(self, task):
    self._queue.append(task)

  def run(self):
    while self._queue:
      task = self._queue.popleft()
      try:
        next(task)
        self._queue.append(task)
      except StopIteration:
        pass
```

usage

```
scheduler = TaskScheduler()

scheduler.add_task(odd_till(5))
scheduler.add_task(even_till(4))
scheduler.add_task(odd_till(6))
scheduler.run()
```

Q: Why remove and readd?

```
        task = self._queue.popleft()
        try:
          next(task)
          self._queue.append(task)
        except StopIteration:
          pass
```

A: If it was finished (exception raised) well and good, it will go straight to the except block. Else the .append will get executed.

In other words if task terminated, dont add it back else add it back.

send

```
def times2():
    while True:
        val = yield
        yield val * 2

z = times2()


next(z)
print(z.send(1))
next(z)
print(z.send(2))
next(z)
print(z.send(3))
```

```
2
4
6
```

Deriving send

Send derivation [4]

```
from collections import deque

class ActorScheduler:
  def __init__(self):
    self._actors = { } # Mapping of names to actors
    self._msg_queue = deque() # Message queue

  def new_actor(self, name, actor):
    '''
    Admit a newly started actor to the scheduler and give it a name
    '''
    self._msg_queue.append((actor,None))
    self._actors[name] = actor

  def send(self, name, msg):
    '''
    Send a message to a named actor
    '''
    actor = self._actors.get(name)
    if actor:
      self._msg_queue.append((actor,msg))

  def run(self):
    '''
    Run as long as there are pending messages.
    '''
    while self._msg_queue:
      actor, msg = self._msg_queue.popleft()
      try:
        actor.send(msg)
      except StopIteration:
        pass
```

```
# Example use
if __name__ == '__main__':
  def printer():
    while True:
      msg = yield
      print('Got:', msg)

  def counter(sched):
    while True:
    # Receive the current count
      n = yield
      if n == 0:
        break
      # Send to the printer task
      sched.send('printer', n)
      # Send the next count to the counter task (recursive)
      sched.send('counter', n-1)


  sched = ActorScheduler()
  # Create the initial actors
  sched.new_actor('printer', printer())
  sched.new_actor('counter', counter(sched))
  # Send an initial message to the counter to initiate
  sched.send('counter', 100)
  sched.run()
```

The above can be expanded with more areas like ready, ready to read, ready to write and writing the appropriate code to switch between the areas and ... you have a concurrent app. This is the basics of an operating system [4]. Using `sched.send` allows to have a loop beyond the recursion limit of python. The recursion limit is `import sys; sys.getrecursionlimit()` usually 1000. try `sched.send('counter', 1001)`.

AsyncIo as explained [5]

```
>>> import asyncio

>>> async def main():
...     print('hello')
...     await asyncio.sleep(1)
...     print('world')

>>> asyncio.run(main())
hello
world
```

Blocking

```
import asyncio
import time

async def say_after(delay, what):
    await asyncio.sleep(delay)
    print(what)

async def main():
    print(f"started at {time.strftime('%X')}")

    await say_after(1, 'hello')
    await say_after(2, 'world')

    print(f"finished at {time.strftime('%X')}")

asyncio.run(main())
```

```
started at 17:13:52
hello
world
finished at 17:13:55
```

Concurrent

```
async def main():
    task1 = asyncio.create_task(
        say_after(1, 'hello'))

    task2 = asyncio.create_task(
        say_after(2, 'world'))

    print(f"started at {time.strftime('%X')}")

    # Wait until both tasks are completed (should take
    # around 2 seconds.)
    await task1
    await task2

    print(f"finished at {time.strftime('%X')}")
```

```
started at 17:14:32
hello
world
finished at 17:14:34
```

```python
import asyncio

async def nested(): # coroutine
    return 42

async def main():
    # Nothing happens if we just call "nested()".
    # A coroutine object is created but not awaited,
    # so it *won't run at all*.
    nested()

    # Let's do it differently now and await it:
    print(await nested())  # will print "42".

asyncio.run(main())
```

tasks

```
import asyncio

async def nested():
    return 42

async def main():
    # Schedule nested() to run soon concurrently
    # with "main()".
    task = asyncio.create_task(nested())

    # "task" can now be used to cancel "nested()", or
    # can simply be awaited to wait until it is complete:
    await task

asyncio.run(main())
```

Futures

A Future is a special low-level awaitable object that represents an eventual result of an asynchronous operation.

When a Future object is awaited it means that the coroutine will wait until the Future is resolved in some other place.

Future objects in asyncio are needed to allow callback-based code to be used with async/await.

Normally there is no need to create Future objects at the application level code.

Future objects, sometimes exposed by libraries and some asyncio APIs, can be awaited:

```
async def main():
    await function_that_returns_a_future_object()

    # this is also valid:
    await asyncio.gather(
        function_that_returns_a_future_object(),
        some_python_coroutine()
    )
```

[1] https://www.python.org/dev/peps/pep-0255/
[2] https://www.python.org/dev/peps/pep-0289/
[3] https://dev.to/abdurrahmaanj/add-superpowers-to-your-python-lists-using-this-feature-24nf
[4] David Beazley
[5] https://docs.python.org/3/library/asyncio-task.html