Advanced Python Flow control

Python Summit, Berlin

Oz Tiram, 14 October 2019

Agenda

- iterables, iterators and generators
- Co-routines, Futures and **asyncio**
- Parallel tasks processing?

/'wəːkʃɒp/

noun: workshop; plural noun: workshops

> a meeting at which a group of people engage in intensive discussion and activity on a particular subject or project.

iterables - what's behind a for loop

```
for item in container:
    do_something_with_item(item)

# or

[process(item) for item in container]
```

iterables - what's behind a for loop

```
for item in container:
    do_something_with_item(item)

# or

[process(item) for item in container]
```

iterables - what's behind a for loop

```
for item in container:
    do_something_with_item(item)

# or

[process(item) for item in container]
```

An **Iterable** has an **__iter__** method

see class **collections.abc.Iterable**

iterables - a naive iterable

```
class March0:
    """Walk 1024 steps"""
    def __iter__(self):
        for i in ['Left', 'Right']*512:
            return i

for step in March():
    print(step)
```

iterables - a naive iterable

```
class March0:
    """Walk 1024 steps"""
    def __iter__(self):
        for i in ['Left', 'Right']*512:
            return i

for step in March():
    print(step)
```

a working iterable

```
class March1:
    def __iter__(self):
        return iter("Left" if i%2 else "Right" for i in range(1,1
```

iterables - exercise

compare the output of **dis.dis** with

```
class March2:

    def __iter__(self):
        return ("Left" if i%2 else "Right" for i in range(1,1025)


dis.dis("""for item in March1():
    print(item)""")

dis.dis("""for item in March2():
    print(item)""")
```

iter built-in behind the scences

1. object has **__iter__**? call it to get an iterator.
2. object has **__getitem__**? create an iterator on items in order, starting from index 0 (zero).
3. raises TypeError("C object is not iterable")

iterables with __getitem__

exercise: implement **March3** with a **__getitem__**

iterables with __getitem__

exercise: implement **March3** with a **__getitem__**

```
class March3:

    def __init__(self):
        self.steps = ["Left" if i%2 else "Right" for i in range(1,
    def __getitem__(self, index):
        return self.steps[index]
```

iterators vs. iterables

iterables

> Any object from which the iter built-in function can obtain an iterator. Objects implementing an **__iter__** method returning an iterator are iterable Sequences are always iterable; as are objects implementing a **__getitem__** method that takes 0-based indexes.

iterators

Excercise - implementing an iterator

- Read https://docs.python.org/3/library/
- Implement **March4**

```
class March4(collections.abc.Iterator):
    """march N steps and stop"""
    def __init__(self, steps):
        self.steps = steps
    ...
```

```
class March4(Iterator):
    def __init__(self, steps):
        self.steps = steps
    def __next__(self):
        while self.steps:
            self.steps -= 1
            return "Left" if self.steps % 2 else "Right"
        raise StopIteration("No more steps")

>>> m = March4(10)
>>> next(m)
'Left'
>>> next(m)
'Right'
>>> next(m)
...
>>> next(m)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 8, in __next__
StopIteration: No more steps
```

A Generator

> A function which returns a **generator iterator**. It looks like a normal function except that it contains **yield** expressions for producing a series of

A Generator

> A function which returns a **generator iterator**. It looks like a normal function except that it contains **yield** expressions for producing a series of

```
def march():
    step = 0
    while True:
        if step % 2:
            yield "Left"
        else:
            yield "Right"
        step += 1
```

Generators - attributes

- Generators are lazy

```
>>> m = march()
>>> m
<generator object march at
>>> next(m)
'Right'
>>># do_something_else()
>>># go back to march()
... next(m)
'Left'
```

Generator expressions

- syntactic sugar

```
>>> m = (item for item in {1
>>> m
<generator object <genexp
>>> next(m)
1
```

Generators yield from

- Nested **for** loops are needed[1] to iterate over multiple generators.

```
s = 'abc'
l = [1,2,3]
def chain(*iters):
    for it in iters:
        for i in it:
            yield i

list(chain(s, l))
['a', 'b', 'c', 1, 2, 3]
```

[1] check **itertools.chain**

Generators yield from

- Since Python 3.3 we have **yield from**

```
s = 'abc'
l = [1,2,3]
def chain(*iters):
    for it in iters:
        yield from i

list(chain(s, l))
['a', 'b', 'c', 1, 2, 3]
```

Exercise - Range

- create **class Range**,
  - the built-in **range**, it can go infinitely

```
class Range:
    def __init__(self, begin, step, end=None):
        self.begin = begin
        self.step = step
        self.end = end # None -> "infinite" series
    ...
```

Solution

```
class Range:
    def __init__(self, begin, step, end=None):
        self.begin = begin
        self.step = step
        self.end = end # None -> "infinite" series
    def __iter__(self):
        result = type(self.begin + self.step)(self.begin)
        forever = self.end is None
        index = 0
        while forever or result < self.end:
            yield result
            index += 1
            result = self.begin + self.step * index

>>> Range(1,1.0,5)
<__main__.Range object at 0x7f3fde4acd30>
>>> list(Range(1,1.0,5))
[1.0, 2.0, 3.0, 4.0]
```

Solution with a generator function

```
>>> def range_gen(begin, step, end=None):
...     result = type(begin + step)(begin)
...     forever = end is None
...     index = 0
...     while forever or result < end:
...         yield result
...         index += 1
...         result = begin + step * index
...
>>> range_gen(1, 0.5, 10)
<generator object range_gen at 0x7f3fde4aaf68>
>>> list(range_gen(1, 0.5, 10))
```

Part 2 - Coroutines, Futures, asyncio

> If Python books are any guide, [coroutines are] the most poorly documented, obscure, and apparently useless feature of Python.

David Beazley, Python author

PEP 342 — Coroutines via Enhanced Generators

- **.send()** and **yield** in an expression
- **.trow()** - raise exception inside a generator
- **.close()** - terminate a generator

PEP 388 - Syntax for delegating to a subgenerator

- This PEP allowed to **return** from a generator
- Allows **yield from** (seen earlier)

A basic coroutine

```
def basic_coro():
    print("started and waiting for input ...")
    x = yield
    print("I got %s" % s, )
    print("I am going to finish now ...")

>>> b = basic_coro()
>>> b
<generator object basic_coro at 0x7fca059fcdb0>
>>> next(b)  # priming
>>> b.send(2)
got 2, exiting now ...
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

States of a generator - exercise

use **inspect.getgeneratorstatus** to find the different states of **basic_coro**

Basic coroutine with multiple yields

```
def basic_coro2(a):
    print(" *** started a: ", a)
    b = yield a
    print(" *** got b: ", s)
    c = yield a + b
    print(" *** received c: ", c ,)
    print(" will exit now ... ")
```

Running average - infinite generator example

```
>>> def averager():
...     total = 0.0
...     count = 0
...     average = None
...     while True:
...         try:
...             term = yield averag
...         except GeneratorExi
...             print("done")
...             raise
...         else:
...             total += term
...             count += 1
...             average = total/cou
...
```

Running average - infinite generator example

```
>>> def averager():
...     total = 0.0
...     count = 0
...     average = None
...     while True:
...         try:
...             term = yield averag
...         except GeneratorExi
...             print("done")
...             raise
...         else:
...             total += term
...             count += 1
...             average = total/cou
...
```

```
>>> avg = averager()
>>> next(avg) # start corout
>>> avg.send(1.0)
1.0
>>> avg.send(2.0)
1.5
>>> avg.close()
done
```

https://bit.ly/2xNk3th

Priming co-routines

```
from functools import wra

def coroutine(func):
    "primes `func` by advanc
    @wraps(func)
    def primer(*args,**kwarg
        gen = func(*args,**kwar
        next(gen)
        return gen
    return primer

@coroutine
def averager():
    ...
```

```
# now the usage of averager

>>> avg = averager()
>>> avg.send(1.0)
1.0
>>> avg.send(2.0)
1.5
>>> avg.close()
done
```

Terminating coroutines

- generator.throw(exc_type[, exc_value[, traceback]])
- generator.close()

Exercise

Handling a custom execption in a generator

Python docs https://bit.ly/2Iqlv8R

Exercise - merge CSVs and implement a qeuery interface

Build an interface for a CSV file which accepts a city name, and returns the row. This should be similar to this:

```
def get_key(data):
    val = None
    while True:
      get_val = yield
      yield data[get_val]

g = get_key({'a':1, 'b':2})
g.send('a')
1
```

Concurrecy with Futures

> To handle network I/O efficiently, you need concurrency, as it involves high latency, so instead of wasting CPU cycles waiting, it’s better to do something else until a response comes back from the network.

Luciano Ramalho, Fluent Python

Commonly used in the past ...

http://code.activestate.com/recipes/577187-python-thread-pool/

Let's examine the code together ...

Shiny concurrent.futures in Python 3.2

```
def get_gdp(country, year=2017):
...:   url = ('http://api.worldbank.org/v2/countries/{}'
...:          '/indicators/NY.GDP.MKTP.CD?format=json&date={}
...:          ''.format(country, year))
...:   resp = requests.get(url)
...:   return {country: resp.json()[-1][0]["value"]}


>>> with ThreadPoolExecutor(5) as executor:
 res = executor.map(get_gdp, ['us', 'br', 'de', 'ir', 'il'])
>>> res
<generator object Executor.map.<locals>.result_iterator

>>> list(res)
[{'us': 19390604000000},
 {'br': 2055505502224.73},
 {'de': 3677439129776.6},
```

ThreadPoolExecutor.map - what happens under the hood?

- Despite Python's GIL multiple threads run really quickly.
- Every Blocking I\O in the STD releases the GIL
- Hence, while a thread is waiting for response it gives control to another

ThreadPoolExecutor with explicit submit

```
with ThreadPoolExecutor(max_workers=5) as executor:
    tasks = []
    for country in ['us', 'br', 'de', 'ir', 'il']:
        future = executor.submit(get_gdp, country)
        tasks.append(future)
        print("Scheduled task at ", future)
    for task in futures.as_completed(tasks):
        print(task.result())
```

```
Scheduled task at  <Future at 0x7f2273ad72b0 state=runnin
Scheduled task at  <Future at 0x7f2268037b70 state=runnin
Scheduled task at  <Future at 0x7f2268e9a240 state=runnin
Scheduled task at  <Future at 0x7f2268e9ab38 state=runnin
Scheduled task at  <Future at 0x7f22447e2128 state=runnin
{'ir': 439513511620.591}
{'us': 19390604000000}
```

ProcessPoolExecutor

concurrent.futures.ProcessPoolExecutor
for heavy CPU processes.

Threads aren't perfect

- in fact they are dumb … and hard to manage
- and they consume a lot of memory …

Concurrency with asyncio

```python
import asyncio

loop = asyncio.get_event_loop()
for country in ['us', 'br', 'de', 'ir', 'il']:
    tasks.append(loop.create_task(get_gdp(country)))

loop.run_until_complete(asyncio.gather(*tasks))
```

Diving into Python's coroutines

bit.ly/coroutines

https://www.youtube.com/watch?v=7sCu4gEjH5I&list=WL&index=17&t=0s

Credits

- A lot of ideas and material are taken from Fluent Python, by Luciano Ramalho
- A. Jesse Jiryu Davis who's blogs and talk have inspired this workshop.