Generators

Generators are very interesting and very powerful feature of python. Will start with basics. As we have noticed before, there is always a top-level syntax or function and some underscore method that implements it. If we have parentheses after smth. this is what is called *call* protocol and there is some method __call__ that implements it.

In [1]:

```
def add1(x,y):
    return x + y

class Adder:
    def __call__(self, x, y):
        return x + y
add2 = Adder()
```

What is the difference between this two methods? The answer is unless we start digging into it we will not be able to tell the difference.

In [4]:

```
print(type(add1))

print(type(add2))

print(add1(3,5))
print(add2(3,5))
```

```
<class 'function'>
<class '__main__.Adder'>
8
8
```

But functionally they are identical. From the perspective of writing the code we will always want to do the shorter add1 syntax. But there is a fundamental difference. If we want to add some stateful behaviour like this:

In [5]:

```
def add1(x,y):
    return x + y

class Adder:
    def __init__(self):
        self.z = 0
    def __call__(self, x, y):
        self.z += 1
        return x + y + self.z
add2 = Adder()
```

The core idea here is that fundamentally there is some very nice syntax and then some object model that everything kind of sits in. And this is closer to what the object model looks like.

But lets think of it in a different way. Let's think about the function that takes up a lot of time to do something.

Maybe a function that goes off and performs a request and it needs to load data from the database.

In [6]:

```
def load_data():
    rows = []
    while db.read():
        row.append()
```

This will be a bit tricky for us to actually demo so we will do just a couple simplifications for this code to kind of mimic what long running computation looks like.

In [10]:

```
from time import sleep
def compute():
    rv = []
    for i in range(10):
        sleep(.5)
        rv.append(i)
    return rv

print(compute())
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Even in this very simple case we are sitting for 5 seconds and waiting for our list of values to even start processing the very first entry of this values. And maybe we don't even need to have this entries all at the same time. If we had range(1000000000000000) in the function we would spend gygabytes of memory. And if we need to look at it one by one it is both wasteful in terms on time and memory it takes. Let's think of a better way to do this. And we can start with our object mode.

In [ ]:

```
class Compute:
    def __call(self):
        rv = []
        for i in range(10):
            sleep(.5)
            rv.append(i)
        return rv

compute = Compute()
```

So we have made a class and this does not help us much. But we kind of seen that pattern before: we want one element at a time and we gonna get each element, process it and maybe throw it away. It is a basic looping construct! And one thing we might now is that in python we see top-level syntax or some top-level functions have some corresponding underscore method that implements it.

In [ ]:

```
# for x in xs:
#     pass

# this syntx looks like this  (more or less) under the cover:

# xi = iter(xs)
# while True:
#     x = next(xi)
```

And we can probably guess what the underscore functions do we have to implement would be. They would be __iter__ and __next__:

In [11]:

```
# xi = iter(xs)          --> __iter__
# while True:
#     x = next(xi)       --> __next__
```

So what that means is that we can take a class in python, we can add an __iter__ and the __next__ method and suddenly that class will be iterable. And in the process of iteration we can do just anything, including performing long computations.

In [13]:

```
from time import sleep
def compute():
    rv = []
    for i in range(10):
        sleep(.5)
        rv.append(i)
    return rv

class Compute:
    def __iter__(self):
        self.last = 0
        return self
    def __next__(self):
        # computes what was the last value we looked at
        rv = self.last
        # increments the last value we looked at
        self.last +=1
        # raises stop iteration if there is too many
        if self.last > 10:
            raise StopIteration()
        # sleeps corresponding to our long running computation
        sleep(.5)
        return rv


compute = Compute()
```

And now compute methods are totally different. The first compute returns a list of values, and the second method returns a single value. The way we use it is like this:

In [14]:

```
for val in Compute():
    print(val)
```

```
0
1
2
3
4
5
6
7
8
9
```

It still takes 5 seconds to prind out the whole thing but we only have to pay for one unit of computation before we start using that value. And there is no storage, so we are storage-efficient. But this whole *class Compute* is really ugly to read. There is a much simpler way to write a function that operates in this fashion. And that is what the generator syntax is:

In [15]:

```
def compute():
    for i in range(10):
        sleep(.5)
        yield i
```

The state of this function is maintained internally and we returning just one element of computation at once. This is the core concept behind generators. Instead of eagerly computing values we give them to the user as the user asks for them.

But there is another similarly important and critical core concept to how generators work beyond just this.

We have all seen APIs that look like this:

In [17]:

```
class Api:
    def run_this_first(self):
        first()
    def run_this_second(self):
        second()
    def run_this_last(self):
        last()
```

Which have methods that indicate the order in which we have to run the functions. If we don't run run_this_first() first, everything blows up. In the documentation they tell to run the functions in a prescribed order. But this API is lying to us because nothing stops us to doing this:

In [ ]:

```
# Api().run_this_last()
# Api().run_this_second()
# Api().run_this_first()
```

One interesting thing about the generator formulation is that it performes some computation and not only does it yeild the result back but it also yields the control back to the caller. We perform one long-running computation modeled by the sleep and give value back to the user to do smth. with and then ask for the next computation.

In the eager formulation (which returns a list) the library code run to completion and gave the whole result to the user and then user was able to do whatever he wanted to do with it. In the generator formulation we have little bit of library code run and little bit of user code run, and so on. This is the actual core conceptualization behind what the generators are built upon, **the idea of co-routines**.

Subroutines we can think of as any piece of executable code that runs from substarting point to subending point to completion. They have one single entry point and one single exit point, and that is it: they run, they are done. If we look at how user code and library code interact, if subroutine is in library code it runs to completion and the user code has to pick up.

For the generator, or a co-routine, we enter the generator and as we ask for values the generator runs, but we can have this nice interliving where some user code runs and asks the generator for the next value, the generator runs its code, yilds back to the user code, the user code runs whatever he runs and asks for the next value and so on.

The interesting point about our API is that if all three methods were designed to run sequentially, it would look like this:

In [17]:

```
class Api:
    def doit():
        first()
        second()
        last
```

And we will not be able to screw things up. But the reason the API gives us three pieces is because it specifically indends us to interlive code. They just want to make sure we interlive code in a fixed sequencing, in a fixed order.

Generators are mechanisms by which we can create code that can interlive with other code and also enforce sequencing.

In [20]:

```
# What happens when this generator runs is that it will run
def api():
    first()
    # up to this point and yield NO VALUE BUT CONTROL back to the caller
    yield
    # then the caller can resume
    second()
    yield
    # and the caller can resume
    last()
```

In this particular formulation first, second and last will always run in that order (guaranteed). We can't guarantee the last will run at all but we can never guarantee it.

Here the generator forces that sequencing on you.