GENERATORS EXPLAINED

Yielding from iteration to concurrency

Follow along with the slides at : https://bit.ly/2wTv1Ni (https://bit.ly/2wTv1Ni)

Rajat Vadiraj Dwaraknath

ITERATION

Iteration is a type of control flow in which a set of statements are repeated.

ITERATION

Iteration is a type of control flow in which a set of statements are repeated.

We implement iteration using *loops*.

ITERATION

Iteration is a type of control flow in which a set of statements are repeated.

We implement iteration using *loops*.

Let's say we want the squares of natural numbers. We could do something like:

ITERATION

Iteration is a type of control flow in which a set of statements are repeated.

We implement iteration using *loops*.

Let's say we want the squares of natural numbers. We could do something like:

In [67]:
```
def squares(n):
    i = 1
    out = []
    while i<=n:
        out.append(i**2)
        i+=1
    return out
```

ITERATION

Iteration is a type of control flow in which a set of statements are repeated.

We implement iteration using *loops*.

Let's say we want the squares of natural numbers. We could do something like:

In [67]:
```
def squares(n):
    i = 1
    out = []
    while i<=n:
        out.append(i**2)
        i+=1
    return out
```

In [68]:
```
squares(10)
```

Out[68]:
```
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

This method has some problems:

This method has some problems:

- If we don't know an upper limit on how many squares we want, we can't use this loop.

This method has some problems:

- If we don't know an upper limit on how many squares we want, we can't use this loop.

- If we know an upper limit on how many squares we may use, we can use this to give us all the squares until the limit, and use only what we require. However, this is highly inefficient in terms of memory consumption.

This method has some problems:

- If we don't know an upper limit on how many squares we want, we can't use this loop.
- If we know an upper limit on how many squares we may use, we can use this to give us all the squares until the limit, and use only what we require. However, this is highly inefficient in terms of memory consumption.

We can solve these issues using a **generator**.

GENERATORS

Any function with a `yield` statement in it becomes a special type of function called a generator function. Here's an example:

GENERATORS

Any function with a `yield` statement in it becomes a special type of function called a generator function. Here's an example:

In [3]:
```
def squares():
    i = 1
    while True:
        yield i**2 # where the magic happens
        i+=1
```

Let's see what happens when we call this function:

Let's see what happens when we call this function:

In [4]:
```
squares()
```

Out[4]:
```
<generator object squares at 0x00000278040A0AF0>
```

Let's see what happens when we call this function:

In [4]:
```
squares()
```

Out[4]:
```
<generator object squares at 0x00000278040A0AF0>
```

To get the stuff which generators yield, we need to use the `next` built-in function:

Let's see what happens when we call this function:

In [4]:
```
squares()
```

Out[4]:
```
<generator object squares at 0x00000278040A0AF0>
```

To get the stuff which generators yield, we need to use the `next` built-in function:

In [5]:
```
g = squares() # lets store the generator in a variable
next(g)
```

Out[5]:
```
1
```

Let's see what happens when we call this function:

In [4]:
```
squares()
```

Out[4]:
```
<generator object squares at 0x00000278040A0AF0>
```

To get the stuff which generators yield, we need to use the `next` built-in function:

In [5]:
```
g = squares() # lets store the generator in a variable
next(g)
```

Out[5]:
```
1
```

In [6]:
```
next(g)
```

Out[6]:
```
4
```

Let's see what happens when we call this function:

In [4]:
```
squares()
```

Out[4]:
```
<generator object squares at 0x00000278040A0AF0>
```

To get the stuff which generators yield, we need to use the `next` built-in function:

In [5]:
```
g = squares() # lets store the generator in a variable
next(g)
```

Out[5]:
```
1
```

In [6]:
```
next(g)
```

Out[6]:
```
4
```

In [7]:
```
next(g)
```

Out[7]:
```
9
```

Let's walk through what just happened. The first time `next(g)` was called, control started executing from the beginning of the function. It continues to execute normally until it hits a `yield`. So, it executed the following statements:

Let's walk through what just happened. The first time `next(g)` was called, control started executing from the beginning of the function. It continues to execute normally until it hits a `yield`. So, it executed the following statements:

```
i=1
while True:
    yield i**2
```

Let's walk through what just happened. The first time `next(g)` was called, control started executing from the beginning of the function. It continues to execute normally until it hits a `yield`. So, it executed the following statements:

```
i=1
while True:
    yield i**2
```

Once it hits a `yield`, control stops executing the generator function body. It returns control to where ever `next` was called, and it also returns the value which was yielded. In this case, the value yielded is 1**2 = 1.

Now, the second time `next` is called, control continues where it left off last time. It also **retains local variables**. This means that `i` is still equal to `1`. In order, the statements it executes are:

Now, the second time `next` is called, control continues where it left off last time. It also **retains local variables**. This means that `i` is still equal to `1`. In order, the statements it executes are:

- `i+=1` `i` is now 2. Now, control is at the end of the while loop body, so it loops back to the beginning.

Now, the second time `next` is called, control continues where it left off last time. It also **retains local variables**. This means that `i` is still equal to `1`. In order, the statements it executes are:

- `i+=1` `i` is now 2. Now, control is at the end of the while loop body, so it loops back to the beginning.

- `yield i**2` Encounters a `yield`, so it stops executing. Value yielded is 2**2 = 4.

Now, the second time `next` is called, control continues where it left off last time. It also **retains local variables**. This means that `i` is still equal to `1`. In order, the statements it executes are:

- `i+=1` `i` is now 2. Now, control is at the end of the while loop body, so it loops back to the beginning.
- `yield i**2` Encounters a `yield`, so it stops executing. Value yielded is 2**2 = 4.

We see that the generator is somewhat 'stateful' in that it remembers it's local namespace. The next time you call `next`, `i` would be equal to 2, which would resulting in yielding a 9. This is exactly what we saw earlier.

Note that this generator will always encounter a yield statement as it is embedded in an infinite loop. What happens if we write a generator which doesn't have infinite yields?

Note that this generator will always encounter a yield statement as it is embedded in an infinite loop. What happens if we write a generator which doesn't have infinite yields?

In [8]:
```
def countdown(n):
    while n>0:
        yield n
        n-=1
```

Note that this generator will always encounter a yield statement as it is embedded in an infinite loop. What happens if we write a generator which doesn't have infinite yields?

In [8]:
```
def countdown(n):
    while n>0:
        yield n
        n-=1
```

This generator basically counts down from some `n` until it hits 0.

In [9]:
```
countdown_3 = countdown(3)
```

In [9]:
```
countdown_3 = countdown(3)
```

In [10]:
```
next(countdown_3)
```

Out[10]:
```
3
```

In [9]:
```
countdown_3 = countdown(3)
```

In [10]:
```
next(countdown_3)
```

Out[10]:
```
3
```

In [11]:
```
next(countdown_3)
```

Out[11]:
```
2
```

In [9]:
```
countdown_3 = countdown(3)
```

In [10]:
```
next(countdown_3)
```

Out[10]: 3

In [11]:
```
next(countdown_3)
```

Out[11]: 2

In [12]:
```
next(countdown_3)
```

Out[12]: 1

In [9]:
```
countdown_3 = countdown(3)
```

In [10]:
```
next(countdown_3)
```

Out[10]:
```
3
```

In [11]:
```
next(countdown_3)
```

Out[11]:
```
2
```

In [12]:
```
next(countdown_3)
```

Out[12]:
```
1
```

In [13]:
```
next(countdown_3)
```

```
---------------------------------------------------------------------------
StopIteration                             Traceback (most recent call last)
<ipython-input-13-7239c8dfeb4a> in <module>()
----> 1 next(countdown_3)

StopIteration:
```

Now we see that if there are no yield statements when next is called, a `StopIteration` exception is raised. This means the generator is out of things to generate.

Now we see that if there are no yield statements when next is called, a `StopIteration` exception is raised. This means the generator is out of things to generate.

This business of calling next everytime is a bit of a hassle. But, python's for loops can actually loop over a generator and automatically call the next for you:

Now we see that if there are no yield statements when next is called, a `StopIteration` exception is raised. This means the generator is out of things to generate.

This business of calling next everytime is a bit of a hassle. But, python's for loops can actually loop over a generator and automatically call the next for you:

In [14]:
```
for i in countdown(10):
    print(i)
```

```
10
9
8
7
6
5
4
3
2
1
```

Now we see that if there are no yield statements when next is called, a `StopIteration` exception is raised. This means the generator is out of things to generate.

This business of calling next everytime is a bit of a hassle. But, python's for loops can actually loop over a generator and automatically call the next for you:

In [14]:
```
for i in countdown(10):
    print(i)
```

```
10
9
8
7
6
5
4
3
2
1
```

Notice that no errors are thrown, and we didn't need to explicitly call next.

What we have done is to reformulate iteration using generators. When the above loop was executed, all 10 numbers were not stored. Only the single `n` was stored in the generator, which will save a lot of memory at scale.

EXAMPLE - DATA LOADING IN MACHINE LEARNING

EXAMPLE - DATA LOADING IN MACHINE LEARNING

Almost all machine learning and deep learning methods require data.

EXAMPLE - DATA LOADING IN MACHINE LEARNING

Almost all machine learning and deep learning methods require data.

A LOT of data.

EXAMPLE - DATA LOADING IN MACHINE LEARNING

Almost all machine learning and deep learning methods require data.

A LOT of data.

But, most of the processing is done only on small batches at a time. This means we don't need to load the whole dataset into our RAM and consume huge amounts of memory.

EXAMPLE - DATA LOADING IN MACHINE LEARNING

Almost all machine learning and deep learning methods require data.

A LOT of data.

But, most of the processing is done only on small batches at a time. This means we don't need to load the whole dataset into our RAM and consume huge amounts of memory.

A perfect place to use generators!

An example using pytorch. The `DataLoader` class allows us to make `iterators`, which for all practical purposes is basically the same as a generator.

An example using pytorch. The `DataLoader` class allows us to make `iterators`, which for all practical purposes is basically the same as a generator.

In [60]:
```
mnist_dataset = MNIST('Z:/MNIST', transform=ToTensor()) # this downloads the dataset
```

An example using pytorch. The `DataLoader` class allows us to make `iterators`, which for all practical purposes is basically the same as a generator.

In [60]:
```
mnist_dataset = MNIST('Z:/MNIST', transform=ToTensor()) # this downloads the dataset
```

In [61]:
```
mnist_loader = data.DataLoader(mnist_dataset) # this creates a data loader object
```

An example using pytorch. The `DataLoader` class allows us to make `iterators`, which for all practical purposes is basically the same as a generator.

In [60]:
```
mnist_dataset = MNIST('Z:/MNIST', transform=ToTensor()) # this downloads the dataset
```

In [61]:
```
mnist_loader = data.DataLoader(mnist_dataset) # this creates a data loader object
```

To get an iterator from the DataLoader object, we use the `iter` keyword.

An example using pytorch. The `DataLoader` class allows us to make `iterators`, which for all practical purposes is basically the same as a generator.

In [60]:
```
mnist_dataset = MNIST('Z:/MNIST', transform=ToTensor()) # this downloads the dataset
```

In [61]:
```
mnist_loader = data.DataLoader(mnist_dataset) # this creates a data loader object
```

To get an iterator from the DataLoader object, we use the `iter` keyword.

In [62]:
```
mnist_iterator = iter(mnist_loader)
mnist_iterator
```

Out[62]:
```
<torch.utils.data.dataloader._DataLoaderIter at 0x2781808ee48>
```

This iterator yields batches of images from the dataset. In this case, I'm using a batch size of 1. Let's get a batch and look at the first image in it.

This iterator yields batches of images from the dataset. In this case, I'm using a batch size of 1. Let's get a batch and look at the first image in it.

In [63]:
```
images, labels = next(mnist_iterator)
images.shape
```

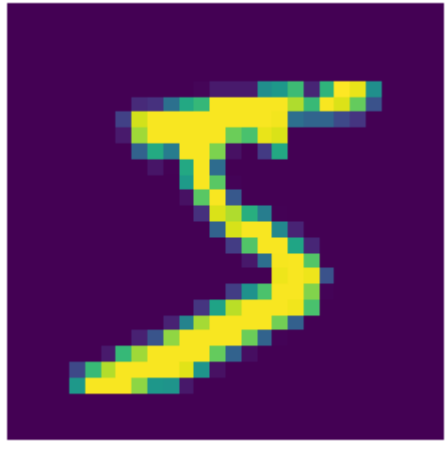
Out[63]:
```
torch.Size([1, 1, 28, 28])
```

This iterator yields batches of images from the dataset. In this case, I'm using a batch size of 1. Let's get a batch and look at the first image in it.

In [63]:
```
images, labels = next(mnist_iterator)
images.shape
```

Out[63]:
```
torch.Size([1, 1, 28, 28])
```

In [64]:
```
axis('off')
imshow(images[0].squeeze())
show()
print(f"Label: {labels[0]}")
```



```
Label: 5
```

Now, if we wish to loop over all the batches in our dataset, we can simply do the following:

Now, if we wish to loop over all the batches in our dataset, we can simply do the following:

In [65]:
```
batch_count = 0
for images, labels in mnist_loader:

    # do some training logic

    # counting batches to demonstrate that the iterator works
    batch_count += 1
print(f"Total of {batch_count} batches with a batch size of {images.shape[0]}")
```

```
Total of 60000 batches with a batch size of 1
```

SENDING TO GENERATORS

The yield statement can do more than just return a value when a next is called. It can also be used in expressions as the right hand side of an assignment, to receive values as inputs.

SENDING TO GENERATORS

The yield statement can do more than just return a value when a next is called. It can also be used in expressions as the right hand side of an assignment, to receive values as inputs.

In [69]:
```
def recv():
    a = yield # acts like an input to the generator
    print(f"I got: {a}")
```

In [19]:
```
receiver = recv()
```

In [19]:
```
receiver = recv()
```

The first `next` moves control over to the yield statement.

In [19]:
```
receiver = recv()
```

The first `next` moves control over to the yield statement.

In [20]:
```
next(receiver)
```

In [19]:
```
receiver = recv()
```

The first `next` moves control over to the yield statement.

In [20]:
```
next(receiver)
```

Then, we can send in a value into the yield statement using the `generator.send` function:

In [19]:
```
receiver = recv()
```

The first `next` moves control over to the yield statement.

In [20]:
```
next(receiver)
```

Then, we can send in a value into the yield statement using the `generator.send` function:

In [21]:
```
receiver.send(24)
```

```
I got: 24

---------------------------------------------------------------------------
StopIteration                             Traceback (most recent call last)
<ipython-input-21-4f5b32c5862a> in <module>()
----> 1 receiver.send(24)

StopIteration:
```

In [19]:
```
receiver = recv()
```

The first `next` moves control over to the yield statement.

In [20]:
```
next(receiver)
```

Then, we can send in a value into the yield statement using the `generator.send` function:

In [21]:
```
receiver.send(24)
```

```
I got: 24

---------------------------------------------------------------------------
StopIteration                             Traceback (most recent call last)
<ipython-input-21-4f5b32c5862a> in <module>()
----> 1 receiver.send(24)

StopIteration:
```

Since there were no more yield statements, a stop iteration was also raised.

Note that the first ever call to a generator must be a next call, as there is no yield statement in the beginning to send a value to:

Note that the first ever call to a generator must be a next call, as there is no yield statement in the beginning to send a value to:

In [22]:
```
receiver = recv()
receiver.send(5)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-22-a8ba8ce5fdc5> in <module>()
      1 receiver = recv()
----> 2 receiver.send(5)

TypeError: can't send non-None value to a just-started generator
```

Note that the first ever call to a generator must be a next call, as there is no yield statement in the beginning to send a value to:

In [22]:
```
receiver = recv()
receiver.send(5)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-22-a8ba8ce5fdc5> in <module>()
      1 receiver = recv()
----> 2 receiver.send(5)

TypeError: can't send non-None value to a just-started generator
```

Also, this reveals that calling the `next` function is equivalent to `generator.send(None)`

The `yield` statement can be used both to recieve and return values. Lets look at this example which returns whatever was sent to it:

The `yield` statement can be used both to recieve and return values. Lets look at this example which returns whatever was sent to it:

In [23]:
```
def echo():
    a = yield # first next call moves to this yield statement
    while True:
        a = yield a
        # when control stops at this yield, return a
        # when control begins from this yield, get the sent value and put it into a
```

The `yield` statement can be used both to recieve and return values. Lets look at this example which returns whatever was sent to it:

In [23]:
```
def echo():
    a = yield # first next call moves to this yield statement
    while True:
        a = yield a
        # when control stops at this yield, return a
        # when control begins from this yield, get the sent value and put it into a
```

In [24]:
```
e = echo()
next(e)
```

The `yield` statement can be used both to recieve and return values. Lets look at this example which returns whatever was sent to it:

In [23]:
```
def echo():
    a = yield # first next call moves to this yield statement
    while True:
        a = yield a
        # when control stops at this yield, return a
        # when control begins from this yield, get the sent value and put it into a
```

In [24]:
```
e = echo()
next(e)
```

In [25]:
```
e.send('hi')
```

Out[25]:
```
'hi'
```

The `yield` statement can be used both to recieve and return values. Lets look at this example which returns whatever was sent to it:

In [23]:
```
def echo():
    a = yield # first next call moves to this yield statement
    while True:
        a = yield a
        # when control stops at this yield, return a
        # when control begins from this yield, get the sent value and put it into a
```

In [24]:
```
e = echo()
next(e)
```

In [25]:
```
e.send('hi')
```

Out[25]:
```
'hi'
```

In [26]:
```
e.send(42)
```

Out[26]:
```
42
```

We can better understand the `a = yield a` statement as follows:

```
variable = yield expr
```

translates to

```
yield expr
variable = what was sent
```

We can better understand the `a = yield a` statement as follows:

```
variable = yield expr
```

translates to

```
yield expr
variable = what was sent
```

So, what you yield doesn't affect what is assigned to the left hand side

EXAMPLE - A RUNNING AVERAGER

EXAMPLE - A RUNNING AVERAGER

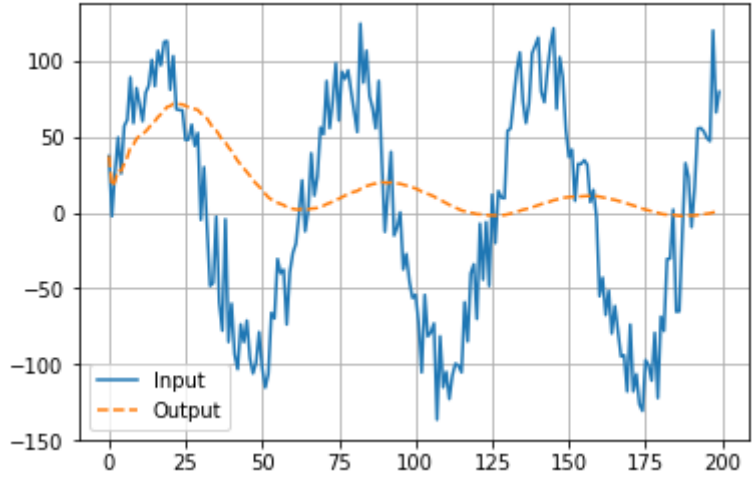
First, let's build one without using generators:

EXAMPLE - A RUNNING AVERAGER

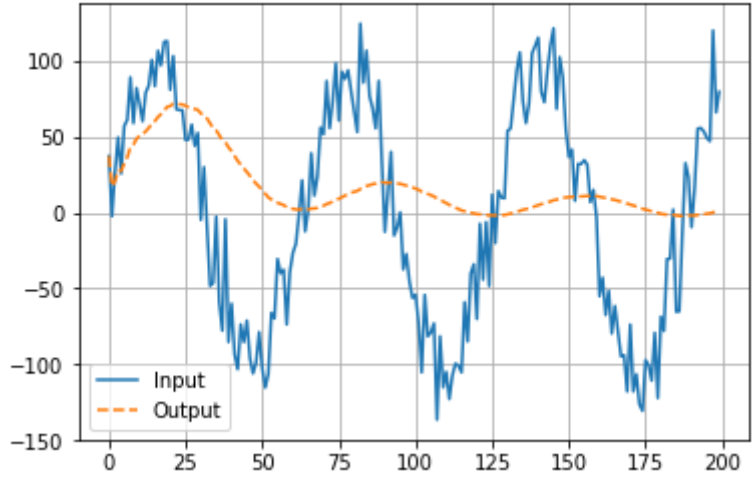
First, let's build one without using generators:

In [30]:
```
class RunningAvg():
    def __init__(self):
        self.running_sum=0
        self.count=0

    def reset(self):
        self.running_sum=0
        self.count=0

    def send(self, x):
        self.running_sum+=x
        self.count+=1

        return self.running_sum/self.count
```

In [31]:
```
test_updater(RunningAvg())
```



In [31]:
```
test_updater(RunningAvg())
```

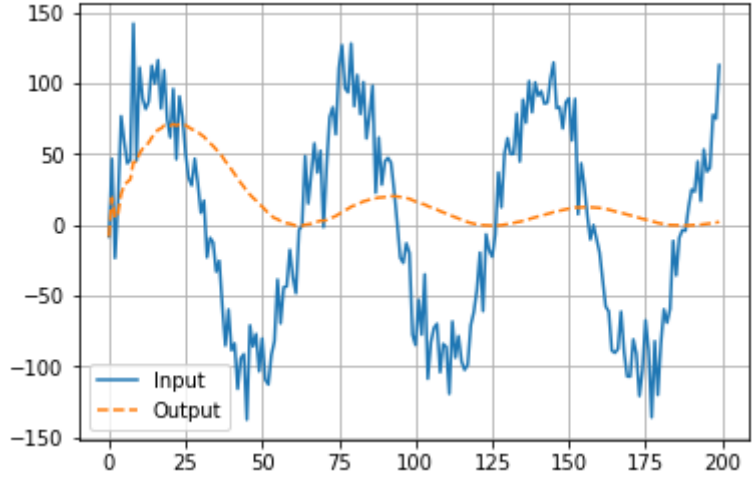


It works fine, but the code is quite verbose, and we had to write a whole class just to build quite a simple thing. Let's write a version using the `yield` statement:

In [32]:
```
def averager():
    running_sum = yield
    count = 1
    while True:
        running_sum += yield running_sum/count
        count += 1
```

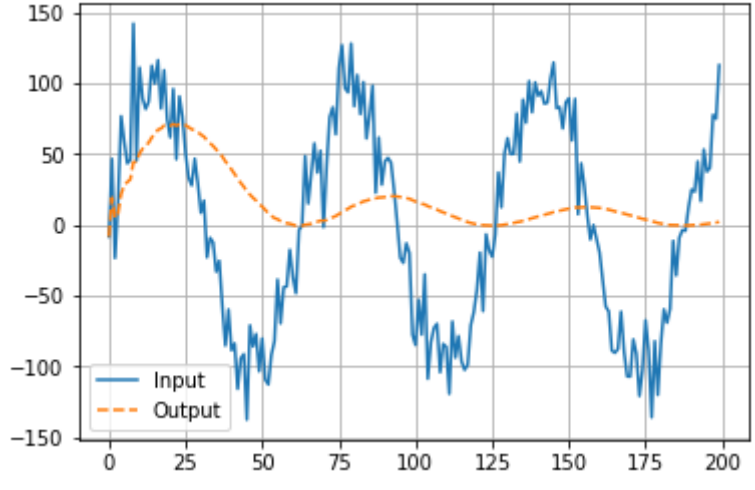
In [32]:
```
def averager():
    running_sum = yield
    count = 1
    while True:
        running_sum += yield running_sum/count
        count += 1
```

In [33]:
```
avg = averager()
next(avg)
test_updater(avg)
```



In [32]:
```
def averager():
    running_sum = yield
    count = 1
    while True:
        running_sum += yield running_sum/count
        count += 1
```

In [33]:
```
avg = averager()
next(avg)
test_updater(avg)
```



Not only does that code look much cleaner and it is just one function, but it is also slightly faster as it avoids the overheads of a whole class:

In [35]:
```
%%timeit
u = averager()
next(u)
test(u)
```

```
632 ms ± 18.4 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

In [35]:
```
%%timeit
u = averager()
next(u)
test(u)
```

```
632 ms ± 18.4 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

In [36]:
```
%%timeit
u = RunningAvg()
test(u)
```

```
873 ms ± 12.2 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

STACKING GENERATORS

We can have generators which yield elements from another generator.

STACKING GENERATORS

We can have generators which yield elements from another generator.

In [69]:
```
def gen1():
    yield 'gen1 a'
    yield 'gen1 b'
    yield 'gen1 c'
```

STACKING GENERATORS

We can have generators which yield elements from another generator.

In [69]:
```
def gen1():
    yield 'gen1 a'
    yield 'gen1 b'
    yield 'gen1 c'
```

In [70]:
```
def gen2():
    for i in gen1():
        yield i
    yield 'd'
    yield 'e'
```

The gen2 object yields elements from both generators sequentially

The gen2 object yields elements from both generators sequentially

In [71]:
```
for i in gen2():
    print(i)
```

```
gen1 a
gen1 b
gen1 c
d
e
```

A shorthand for the for loop over `gen1` is `yield from`.

A shorthand for the for loop over `gen1` is `yield from`.

In [72]:
```
def gen3():
    yield from gen1()
    yield 'd'
    yield 'e'
```

A shorthand for the for loop over `gen1` is `yield from`.

In [72]:
```
def gen3():
    yield from gen1()
    yield 'd'
    yield 'e'
```

In [73]:
```
for i in gen3():
    print(i)
```

```
gen1 a
gen1 b
gen1 c
d
e
```

The `yield from gen` expression can be roughly translated to:

```
for i in gen:
    yield i
```

However, it has some more subtleties which we will get into later.

EXAMPLE OF STACKING GENERATORS

One neat example of stacked generators is binary tree traversal. Recall that you can traverse a binary tree in three ways:

- In-order
- Pre-order
- Post-order

Before we get into generators, let's write some simple code to setup a binary tree, and then write an inorder traversal function without using generators.

EXAMPLE OF STACKING GENERATORS

One neat example of stacked generators is binary tree traversal. Recall that you can traverse a binary tree in three ways:

- In-order
- Pre-order
- Post-order

Before we get into generators, let's write some simple code to setup a binary tree, and then write an inorder traversal function without using generators.

In [70]:
```
class BinaryTree():
    def __init__(self, data, left=None, right=None):
        self.data = data
        self.left = left
        self.right = right
```

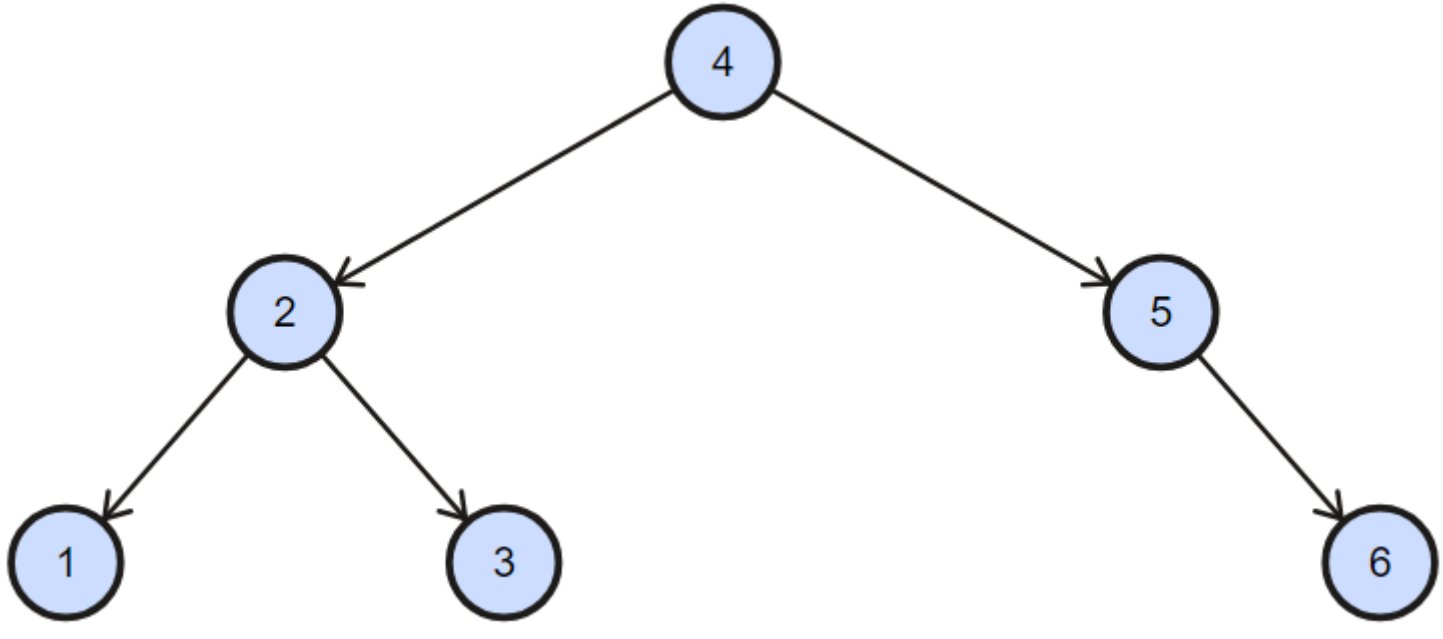
Let's make a simple binary tree with 6 elements:

Let's make a simple binary tree with 6 elements:

In [71]:
```
one, three, six = BinaryTree(1),BinaryTree(3),BinaryTree(6)
two = BinaryTree(2,one,three)
five = BinaryTree(5,left=None,right=six)
four = BinaryTree(4,two,five)
root = four
```

Let's make a simple binary tree with 6 elements:

In [71]:
```
one, three, six = BinaryTree(1),BinaryTree(3),BinaryTree(6)
two = BinaryTree(2,one,three)
five = BinaryTree(5,left=None,right=six)
four = BinaryTree(4,two,five)
root = four
```



Inorder traversal without using generators:

Inorder traversal without using generators:

In [72]:
```
def inorder_list(btree):
    if btree==None:
        return []
    left = inorder_list(btree.left)
    right = inorder_list(btree.right)
    return left + [btree.data] + right
```

Inorder traversal without using generators:

In [72]:
```
def inorder_list(btree):
    if btree==None:
        return []
    left = inorder_list(btree.left)
    right = inorder_list(btree.right)
    return left + [btree.data] + right
```

In [73]:
```
inorder_list(root)
```

Out[73]:
```
[1, 2, 3, 4, 5, 6]
```

Preorder traversal

Preorder traversal

In [74]:
```
def preorder_list(btree):
    if btree==None:
        return []
    left = preorder_list(btree.left)
    right = preorder_list(btree.right)
    return [btree.data] + left  + right
```

Preorder traversal

In [74]:
```
def preorder_list(btree):
    if btree==None:
        return []
    left = preorder_list(btree.left)
    right = preorder_list(btree.right)
    return [btree.data] + left  + right
```

In [75]:
```
preorder_list(root)
```

Out[75]:
```
[4, 2, 1, 3, 5, 6]
```

It works well, and the code is quite concise and readable. However, we face the issue of memory if the tree is very large, as the traversal returns the whole list at once. Let's write a version using stacked generators:

It works well, and the code is quite concise and readable. However, we face the issue of memory if the tree is very large, as the traversal returns the whole list at once. Let's write a version using stacked generators:

In [76]:
```
def inorder_gen(btree):
    if btree != None:
        yield from inorder_gen(btree.left)
        yield btree.data
        yield from inorder_gen(btree.right)
```

It works well, and the code is quite concise and readable. However, we face the issue of memory if the tree is very large, as the traversal returns the whole list at once. Let's write a version using stacked generators:

In [76]:
```
def inorder_gen(btree):
    if btree != None:
        yield from inorder_gen(btree.left)
        yield btree.data
        yield from inorder_gen(btree.right)
```

In [77]:
```
g = inorder_gen(root)
for i in g:
    print(i, end=' ')
```

```
1 2 3 4 5 6
```

It's simple to change it to preorder:

It's simple to change it to preorder:

In [78]:
```
def preorder_gen(btree):
    if btree != None:
        yield btree.data
        yield from preorder_gen(btree.left)
        yield from preorder_gen(btree.right)
```

It's simple to change it to preorder:

In [78]:
```
def preorder_gen(btree):
    if btree != None:
        yield btree.data
        yield from preorder_gen(btree.left)
        yield from preorder_gen(btree.right)
```

In [79]:
```
g = preorder_gen(root)
for i in g:
    print(i, end=' ')
```

```
4 2 1 3 5 6
```

It's simple to change it to preorder:

In [78]:
```
def preorder_gen(btree):
    if btree != None:
        yield btree.data
        yield from preorder_gen(btree.left)
        yield from preorder_gen(btree.right)
```

In [79]:
```
g = preorder_gen(root)
for i in g:
    print(i, end=' ')
```

```
4 2 1 3 5 6
```

This version is much more sleek, and more memory efficient. Try to see where you can use generators in your code base to improve its memory consumption!

DIGGING DEEPER INTO YIELD FROM

DIGGING DEEPER INTO YIELD FROM

The above example showed that `yield from` expands generators within other generators and allows us to stack them. However, the `yield from` statement is much more that just a for loop.

DIGGING DEEPER INTO YIELD FROM

The above example showed that `yield from` expands generators within other generators and allows us to stack them. However, the `yield from` statement is much more that just a for loop.

- The for loop expansion doesn't work if we send values into the nested generator, but `yield from` takes care of that

DIGGING DEEPER INTO YIELD FROM

The above example showed that `yield from` expands generators within other generators and allows us to stack them. However, the `yield from` statement is much more that just a for loop.

- The for loop expansion doesn't work if we send values into the nested generator, but `yield from` takes care of that

- Putting things on the left hand side of `yield from` works differently from just a `yield`. This functionality is what lies at the heart of *asynchronous concurrency* in python.

COROUTINES

COROUTINES

Broadly, coroutines are functions that transfer control between each other, while maintaining their local namespace and state.

COROUTINES

Broadly, coroutines are functions that transfer control between each other, while maintaining their local namespace and state.

We noted earlier that generators are like 'stateful' functions, whose execution can be 'paused' at `yield` statements.

COROUTINES

Broadly, coroutines are functions that transfer control between each other, while maintaining their local namespace and state.

We noted earlier that generators are like 'stateful' functions, whose execution can be 'paused' at `yield` statements.

These characteristics are exactly those of a *coroutine* - the basic unit in async control flow.

COROUTINES

Broadly, coroutines are functions that transfer control between each other, while maintaining their local namespace and state.

We noted earlier that generators are like 'stateful' functions, whose execution can be 'paused' at `yield` statements.

These characteristics are exactly those of a *coroutine* - the basic unit in async control flow.

The benefit of having such a block is that a coroutine for reading or writing (I/O) can send control to some other coroutine while waiting for results from the I/O operation. This minimizes idle time of the CPU.

AN EXAMPLE

AN EXAMPLE

In [63]:
```
def sum_n(n, name):
    s = 0
    while n>0:
        s += n
        n -= 1
        print(f"{name} s = {s}")
        yield

    return s
```

AN EXAMPLE

In [63]:
```
def sum_n(n, name):
    s = 0
    while n>0:
        s += n
        n -= 1
        print(f"{name} s = {s}")
        yield

    return s
```

This 'coroutine' finds the sum of natural numbers until the given number, in reverse order. It has a `yield` after each step in the loop, so that it can be paused. Notice that it also **returns** the sum at the end.

In [67]:
```
def print_result(coroutine):
    result = yield from coroutine
    print("Result = {result}")
```

In [67]:
```
def print_result(coroutine):
    result = yield from coroutine
    print("Result = {result}")
```

This one just prints the result from another coroutine. Notice how `yield from` is used on the right hand side of an assignment. The value of `result` equals whatever the `coroutine` *returns*.

Now, we can execute multiple coroutines concurrently using an *event loop*. I'm going to show a basic version of an event loop which just executes tasks using a round robin schedule.

Now, we can execute multiple coroutines concurrently using an *event loop*. I'm going to show a basic version of an event loop which just executes tasks using a round robin schedule.

In [68]:
```
def run(tasks):
    """Run the list of tasks until completion"""
    while tasks:
        to_run = tasks[0] # get the first task in the list

        try: # try to run it
            next(to_run)
        except StopIteration as e:
            # if stop iteration is raised, that means that the task is complete
            print(f'Completed task')
            del tasks[0] # remove it from the task queue
        else:
            # if it isn't raised, move it to the end of the queue
            del tasks[0]
            tasks.append(to_run)
```

Let's use this basic event loop to run two of the sum coroutines concurrently. We'll wrap the sum coroutines using the `print_result` coroutine as well.

Let's use this basic event loop to run two of the sum coroutines concurrently. We'll wrap the sum coroutines using the `print_result` coroutine as well.

In [66]:
```
run([print_result(sum_n(3, 'three')),
     print_result(sum_n(5, 'five'))])
```

```
three s = 3
five s = 5
three s = 5
five s = 9
three s = 6
five s = 12
6
Completed task
five s = 14
five s = 15
15
Completed task
```

Let's use this basic event loop to run two of the sum coroutines concurrently. We'll wrap the sum coroutines using the `print_result` coroutine as well.

In [66]:
```
run([print_result(sum_n(3, 'three')),
     print_result(sum_n(5, 'five'))])
```

```
three s = 3
five s = 5
three s = 5
five s = 9
three s = 6
five s = 12
6
Completed task
five s = 14
five s = 15
15
Completed task
```

We can see that the two coroutines are executed alternatingly, and not one after the other. This is the essence of async concurrency.

SUMMARY

We started off by reformulating iteration using `yield` - it was a pretty cool way to solve some memory issues, but the `yield` keyword seemed like just a synctactic trick to make our life easier.

SUMMARY

We started off by reformulating iteration using `yield` - it was a pretty cool way to solve some memory issues, but the `yield` keyword seemed like just a synctactic trick to make our life easier.

By digging deeper into stacking generators and using `yield from`, we saw that the basic ability to pause function execution and maintain state was the main power of the `yield` keyword, and that that formed the heart of asynchronous concurrency.

SUMMARY

We started off by reformulating iteration using `yield` - it was a pretty cool way to solve some memory issues, but the `yield` keyword seemed like just a synctactic trick to make our life easier.

By digging deeper into stacking generators and using `yield from`, we saw that the basic ability to pause function execution and maintain state was the main power of the `yield` keyword, and that that formed the heart of asynchronous concurrency.

In python 3.5+ the keywords `async` and `await` can be used to explicitly define coroutines. They work using the same principles as the `yield from` statement.

SUMMARY

We started off by reformulating iteration using `yield` - it was a pretty cool way to solve some memory issues, but the `yield` keyword seemed like just a synctactic trick to make our life easier.

By digging deeper into stacking generators and using `yield from`, we saw that the basic ability to pause function execution and maintain state was the main power of the `yield` keyword, and that that formed the heart of asynchronous concurrency.

In python 3.5+ the keywords `async` and `await` can be used to explicitly define coroutines. They work using the same principles as the `yield from` statement.

If you're interested, look up stuff on asyncio and the event loop to get a deeper understanding of this part of python! I recommend starting with this video (https://www.youtube.com/watch?v=MCs5OvhV9S4).

THANKS FOR LISTENING

You can find the code for this talk on my github - rajatvd (https://github.com/rajatvd). I've included an example using `async` and `await` in the code as well, which can be appreciated only when it is run live.

I've hosted the slides for this talk on my blog (https://rajatvd.github.io/) here (https://rajatvd.github.io/PySangamam/).

I've put up a bunch of projects on there as well, mainly related to deep learning and math in general.

Feel free to contact me at rajatvd@gmail.com!