Functions and generators

June 19, 2021

Review HW

Sum of multiples

Sum of all #s below 1000 that are divisible by 3 and 5.

```
c = 0
for i in range(1000):
    if i %3 ==0 or i %5 ==0:
        c += i
print(c)
```

Sum of multiples

Sum of all #s below 1000 that are divisible by 3 and 5.

```
c = 0
for i in range(1000):
    if i %3 ==0 or i %5 ==0:
        c += i
print(c)
```

Sum of multiples

Sum of all #s below 1000 that are divisible by 3 and 5.

initializing a counter variable set to 0

```
c = 0
for i in range(1000):
    if i %3 ==0 or i %5 ==0:
        c += i
print(c)
```

Sum of multiples

Sum of all #s below 1000 that are divisible by 3 and 5.

?

initializing a counter variable set to 0

```
c = 0
for i in range(1000):
    if i %3 ==0 or i %5 ==0:
        c += i
print(c)
```

Sum of multiples

Sum of all #s below 1000 that are divisible by 3 and 5.

assignment statement

initializing a counter variable set to 0

```
c = 0
for i in range(1000):
    if i %3 ==0 or i %5 ==0:
        c += i
print(c)
```

Sum of multiples

Sum of all #s below 1000 that are divisible by 3 and 5.

```
c = 0
for i in range(1000):
    if i %3 ==0 or i %5 ==0:
        c += i
print(c)
```

Sum of multiples

Sum of all #s below 1000 that are divisible by 3 and 5.

```
c = 0
for i in range(1000):
    if i %3 ==0 or i %5 ==0:
        c += i
print(c)
```

Looping through a range (sequence type, object) of 1000

Sum of multiples

Sum of all #s below 1000 that are divisible by 3 and 5.

```
c = 0

for i in range(1000):

    if i %3 ==0 or i %5 ==0:

        c += i

print(c)
```

Sum of multiples

Sum of all #s below 1000 that are divisible by 3 and 5.

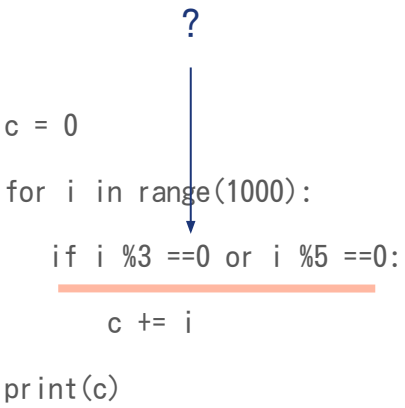
```
c = 0

for i in range(1000):

    if i %3 ==0 or i %5 ==0:

        c += i

print(c)
```

if statement used for conditional execution

conditional expression: if the remainder of i divided by 3 is 0 or i divided by 5 is 0

Sum of multiples

Sum of all #s below 1000 that are divisible by 3 and 5.

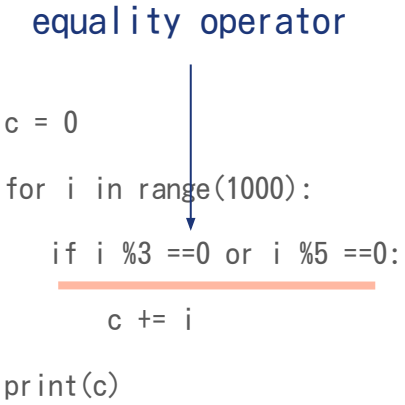


if statement used for conditional execution

conditional expression: if the remainder of i divided by 3 is 0 or i divided by 5 is 0

Sum of multiples

Sum of all #s below 1000 that are divisible by 3 and 5.

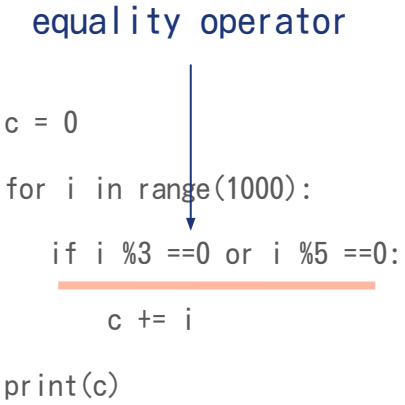


if statement used for conditional execution

conditional expression: if the remainder of i divided by 3 is 0 or i divided by 5 is 0

Sum of multiples

Sum of all #s below 1000 that are divisible by 3 and 5.



if statement used for conditional execution

conditional expression: if the remainder of i divided by 3 is 0 or i divided by 5 is 0

Sum of multiples

Sum of all #s below 1000 that are divisible by 3 and 5.

```
c = 0

for i in range(1000):

    if i %3 ==0 or i %5 ==0:

        c += i

print(c)
```

print function to print-out output

Sum of multiples

Sum of all #s below 1000 that are divisible by 3 and 5.

```
c = 0

for i in range(1000):

    if i %3 ==0 or i %5 ==0:

        c += i

print(c)
```

print function to print-out output

print() is a f(x), *not* a method .. difference is b/c method belongs to a *class* object whereas print is a block of code

We build up a silent understanding first, then vocalize it.


fineart
america


STUBBORN
& proud of it


0:48 / 3:55

What if I wanted to make ‘Sum of multiples’ a function that takes in a given stopping integer?

Where to start?

What do we want? Inputs & Outputs

Where to start?

What do we want? Inputs & Outputs

```
def name( __inputs__ ):

    ...

    outputs!
```

Now you!

What’s our input? Output?

```
def name( __inputs__ ):

    ...

    outputs!
```

Now you!

What’s our input? Output?

```
def name( n ):

    ...

    counter!
```

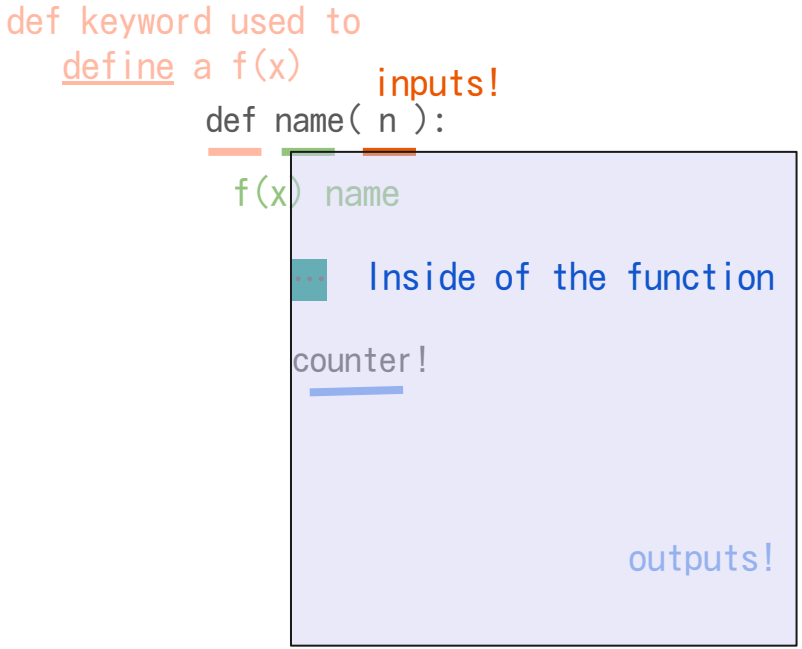
Question!!

```
def name( n ):
    ...
```

counter!

What do we do to get to our output form our input?

Question!!


def keyword used to
define a f(x)
inputs!
def name(n):
f(x) name
...
Inside of the function
counter!
outputs!

Same code as b4!

We've been doing the "..." this whole time. Now we're just putting it into a function to get from a dynamic input to an output

```
def name( n ):

    counter = 0

    for i in range(1000):

        if i %3 ==0 or i %5 ==0:

            counter += i


    counter
```

There’s a problem.

Can you spot it?

```
def name( n ):

    counter = 0

    for i in range(1000):

        if i %3 ==0 or i %5 ==0:

            counter += i


    counter
```

Outputs can’t just be variables

They must be **returned** or **yielded**

```
def name( n ):

    counter = 0

    for i in range(1000):

        if i %3 ==0 or i %5 ==0:

            counter += i


    return counter

    yield  counter
```

What’s the big deal btwn return and yield?

First, a silent understanding.
Let’s just see for ourselves.

———

```python
def sum_multiples(n):
    '''
    Returns the sum of all multiples of 3 and 5 below input n
    '''

    #Initialize a counter
    counter =0

    #Cycle through all numbers up to n
    for i in range(n):
        #Get multiples
        if i %3 ==0 or i %5 ==0:
            counter +=i

    return counter

print(sum_multiples(1000))
```

gives/outputs:

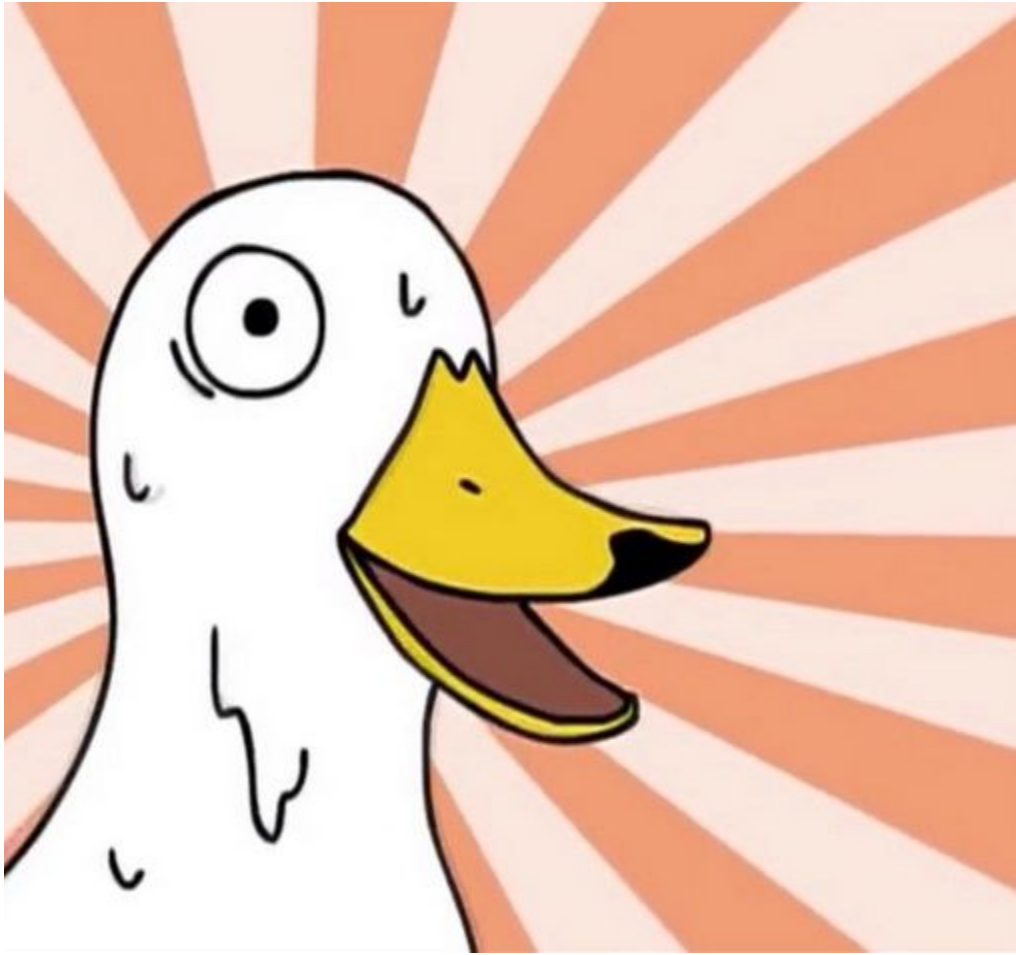
233168

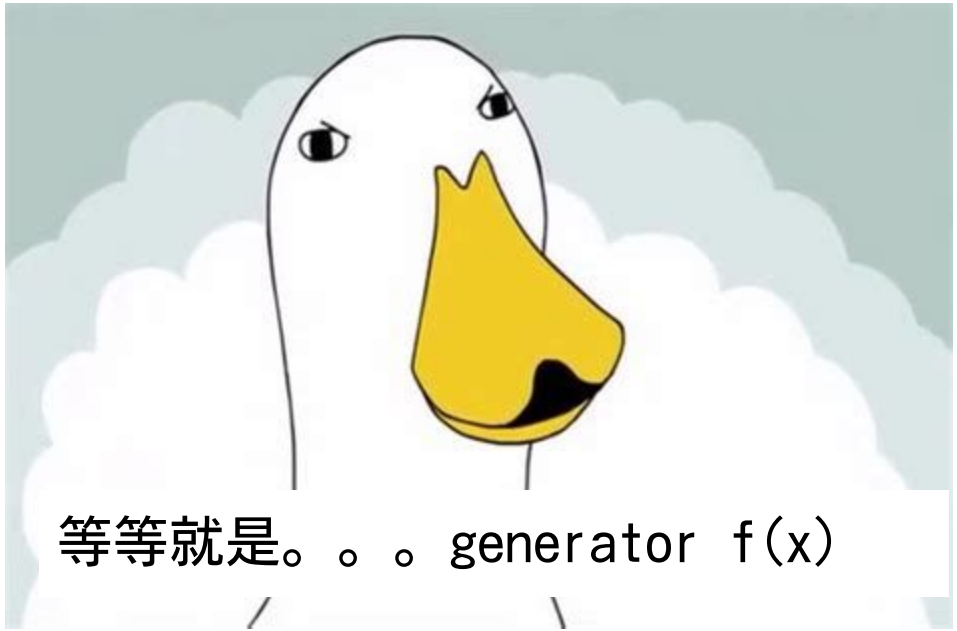
```
def sum_multiples(n):
    '''
    Returns the sum of all multiples of 3 and 5 below input n
    '''

    #Initialize a counter
    counter =0

    #Cycle through all numbers up to n
    for i in range(n):
        #Get multiples
        if i %3 ==0 or i %5 ==0:
            counter +=i

    yield counter

print(sum_multiples(1000))
```

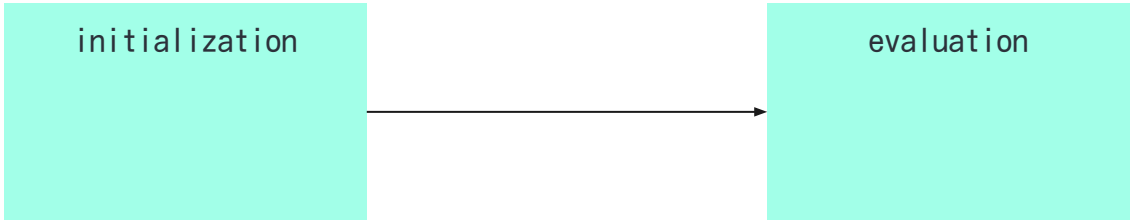
gives/outputs:

```
<generator object sum_multiples at 0x7f8aa8071ac0>
```

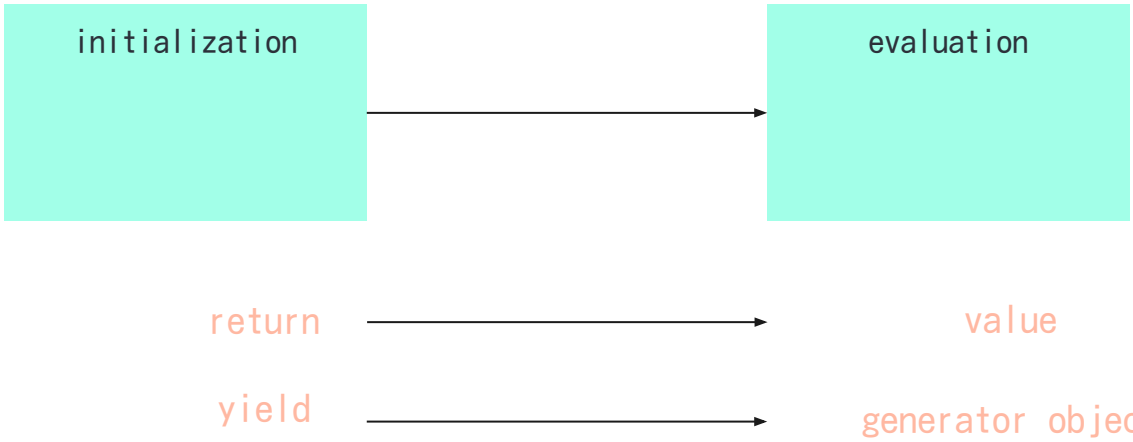
完全看不懂啊救助啊



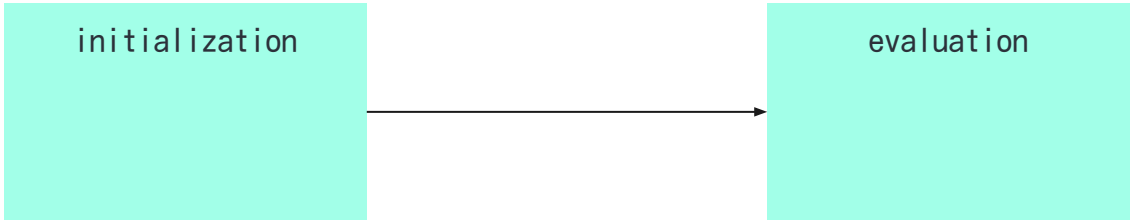

等等就是。。。generator f(x)



Functions have two steps: (1) initialization and (2) evaluation



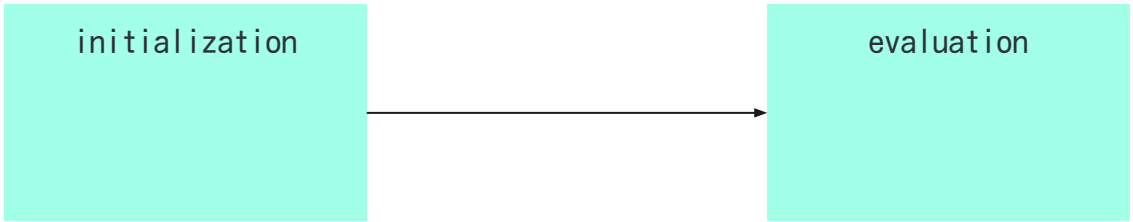
Return keyword: f(x); yield keyword: converts f(x) to **generator**



return → hands back a value to its caller

yield → generator object

Return keyword: f(x); yield keyword: converts f(x) to **generator**



return → hands back a value to its caller

yield → Starts from where it was paused

Return has to start from the start; yield starts from the pause

So when it is useful to use generators?

So when it is useful to use generators?

Sequences!!

0, 1, 1, 2, 3, 5, 8, 13, 21, 34

The Fibonacci sequence

a, b

0, 1, 1, 2, 3, 5, 8, 13, 21, 34

The Fibonacci sequence

a, b

0, 1, 1, 2

The Fibonacci sequence

a, b

0, 1, 1, 2

a->b, b->a+b

The Fibonacci sequence

a, b

1, 2, 3, 5

a->b, b->a+b

If only I could just remember the previous 2 values!

a, b

0, 1, 1, 2, 3, 5

a->b, b->a+b

If only I could just remember the previous 2 values!

Fibonacci sequence

Want a f(x) that yields a and assigns a, b

- Only remember the last 2 entries in sequence while yielding current value & assigning next 2 entries

```
def fib(limit):
    ‘initialize’  current_val, next_val
    so long as < limit:
        yield current_val
        assign next two values
```

Keep this in mind.

First, we have to cover “so long as”

So long as

Want a condition that is just “so long as this is True”

```
bean = 0
beans = [ ‘beans’ ,  ‘beans’ ,  ‘beans’ ]
so long as bean < all_beans:
    bean += 1
print(bean)
```

So long as

Want a condition that is just “so long as this is True”

```
bean = 0

beans = [ ‘beans’ ,  ‘beans’ ,  ‘beans’ ]

so long as bean < all_beans:

    bean += 1

print(bean)
```

Gonna start calling this “while”

While

Want a condition that is just “so long as this is True”

```
bean = 0

beans = [ ‘beans’ ,  ‘beans’ ,  ‘beans’ ]

while bean < all_beans:

    bean += 1

print(bean)
```

What is all_beans?

While

Want a condition that is just “so long as this is True”

```
bean = 0
beans = [ ‘beans’ ,  ‘beans’ ]
while bean < all_beans:
    bean += 1
print(bean)
```

What is all_beans?

While

Want a condition that is just “so long as this is True”

```
bean = 0
beans = [ ‘beans’ ,  ‘beans’ ]
while bean < all_beans:
    bean += 1
print(bean)
```

The length of beans!

While

Want a condition that is just “so long as this is True”

```
bean = 0
beans = [ ‘beans’ ,  ‘beans’ ]
while bean < len(beans):
    bean += 1
print(bean)
```

The length of beans!

While

Want a condition that is just “so long as this is True”

```
bean = 0

beans = [ ‘beans’ ,  ‘beans’ ]

while bean < len(beans):

    bean += 1

print(bean)
```

built-in f(x) that returns number of items in an object

While

Want a condition that is just “so long as this is True”

```
bean = 0
beans = [ ‘beans’ ,  ‘beans’ ]
while bean < len(beans):
    bean += 1
print(bean)
```

While

Want a condition that is just “so long as this is True”

```
bean = 0

beans = [ ‘beans’ ,  ‘beans’ ]

while bean < len(beans):

    bean += 1

print(bean)
```

Iteratively incrementing a counter variable initialized @ 0

While LOOP!

Want a condition that is just “so long as this is True”

```
bean = 0

beans = [ ‘beans’ ,  ‘beans’ ]

while bean < len(beans):

    bean += 1

print(bean)
```

Iteratively incrementing a counter variable initialized @ 0

We are looping!

While LOOP!

Want a condition that is just “so long as this is True”

```
bean = 0

beans = [ ‘beans’ ,  ‘beans’ ]

while bean < len(beans):

    bean += 1

print(bean)
```

Iteratively incrementing a counter variable initialized @ 0

We are looping!

Check for understanding:

Sum all even numbers below 10000 using a while loop iterator.

Fibonacci sequence

Want a f(x) that yields a and assigns a, b

- Only remember the last 2 entries in sequence while yielding current value & assigning next 2 entries

```
def fib(limit):
    ‘initialize’  current_val, next_val
    so long as < limit:
        yield current_val
        assign next two values
```

Fibonacci sequence

Want a f(x) that yields a and assigns a, b

- Only remember the last 2 entries in sequence while yielding current value & assigning next 2 entries

```
def fib(limit):
    ‘initialize’  current_val, next_val
    while current_val < limit:
        yield current_val
        assign next two values
```

Fibonacci sequence

Want a f(x) that yields a and assigns a, b

- Only remember the last 2 entries in sequence while yielding current value & assigning next 2 entries

```
def fib(limit):
    current_val, next_val = 0, 1
    while current_val < limit:
        yield current_val
        assign next two values
```

Fibonacci sequence

Want a f(x) that yields a and assigns a, b

- Only remember the last 2 entries in sequence while yielding current value & assigning next 2 entries

```
def fib(limit):
    current_val = 0
    next_val = 1
    while current_val < limit:
        yield current_val
        assign next two values
```

a, b

0, 1, 1, 2

a->b, b->a+b

The Fibonacci sequence

Fibonacci sequence

Crap! Why doesnt this work!

```
def fib(limit):
    current_val = 0
    next_val = 1
    while current_val < limit:
        yield current_val
        a = b
        b = a+b
```

Fibonacci sequence

Crap! Why doesnt this work!

```
def fib(limit):
    current_val = 0
    next_val = 1
    while current_val < limit:
        yield current_val
        a = b
        b = a+b
```

Fibonacci sequence

Crap! Why doesnt this work!

```
def fib(limit):
    current_val = 0
    next_val = 1
    while current_val < limit:
        yield current_val
        a = b
        b = a+b
```

Fibonacci sequence

This happens one after the other. We need it to happen at the same time.

```
def fib(limit):
    current_val = 0
    next_val = 1
    while current_val < limit:
        yield current_val
        a = b
        b = a+b
```

Fibonacci sequence

This happens one after the other (**synchronously**). We need it to happen at the same time (**asynchronously**).

```
def fib(limit):
    current_val = 0
    next_val = 1
    while current_val < limit:
        yield current_val
        a = b
        b = a+b
```

1 ↓ 2

```
def fib(n):
    a, b = 0, 1
    while a < n:
        yield a
        a = b
        b = a+b
```

1 ↓ 2

code diagram of fib(10):

a=0, b=1

a=0 < n=10:

yield a=0

a=1

b=1+1

a is being set to 1, then referenced in the next line

a, b

0, 1, 1, 2

a->b, b->a+b

This happens @ the same time.

The Fibonacci sequence

Fibonacci sequence

This happens one after the other (**synchronously**). We need it to happen at the same time (**asynchronously**).

```
def fib(limit):
    current_val = 0
    next_val = 1
    while current_val < limit:
        yield current_val
        temp = b
        a = temp
        b = a+b
```

Fibonacci sequence

This happens one after the other (**synchronously**). We need it to happen at the same time (**asynchronously**).

```
def fib(limit):
    current_val = 0
    next_val = 1
    while current_val < limit:
        yield current_val
        a, b = b, a+b
```

```
def fib(limit):
    a, b = 0, 1
    while a < limit:
        yield a
        a, b = b, a+b

print(fib(4e6))
```

gives/outputs:

```
<generator object fib at 0x7fd2e80cfb30>
```

```
def fib(limit):
    a, b = 0, 1
    while a < limit:
        yield a
        a, b = b, a+b

print(sum(fib(4e6)))
```

gives/outputs:

```
9227464
```

4613732 higher than the right answer..

```
def fib(limit):
    a, b = 0, 1
    while a < limit:
        yield a
        a, b = b, a+b

print(sum(fib(4e6)))
```

gives/outputs:

```
9227464
```

4613732 higher than the right answer..

Sums everything, not just the even numbers!!!

Use a for loop to iterate over the generator object!

Fibonacci sequence

Iterating over a generator object

```
def fib(limit):
    current_val = 0
    next_val = 1
    while current_val < limit:
        yield current_val
        a = b
        b = a+b
sum = 0
for i in fib(4e6):
    if i %2 ==0:
        sum += i
print(sum)
```

A faster way

```
print(sum(a for a in fib(4e6) if not (a % 2)))
```

Check for understanding:

Try to use this same pattern to sum all even numbers below 1000.

```
print(sum(a for a in fib(4e6) if not (a % 2)))
```

This is called a: List comprehension!

```
print(sum(a for a in fib(4e6) if not (a % 2)))
```

First pass

1, 10, 9, 4

1, 10, 9, 4 — 10 > 9

1, 9, 10, 4 — swap places

1, 9, 10, 4 — 10 > 4

1, 9, 4, 10 — swap places, 10 is done

Second pass

1, 9, 4, 10

1, 9, 4, 10 — 9 > 4

1, 4, 9, 10 — swap places

1, 4, 9, 10 — 9, 10 done

Third pass

1, 4, 9, 10

1, 4, 9, 10

1, 4, 9, 10 — 4, 9, 10 done

Fourth pass

1, 4, 9, 10

1, 4, 9, 10

1, 4, 9, 10 — 1, 4, 9, 10 done