Notes & Homework: June 19, 2021

How to think about problems

Let's say you are solving a problem "sum all the numbers below 1000 that are divisible by 4 or 7". Can you try writing that?

Active solve: Can I sum all the numbers below 100000 that are divisible by 414 or 4543?

```
#Desired output: ???
```

I'll give you a spoiler alert: this isn't about your answer. It's about how you got to your answer.

Let's take me as a pianist.

me

Ever since I was a wee lad, I understand the piano symbolically. I would hear the D-flat major arpeggio as a hat flying off in the wind, or a song as a cat walking on a fence. A "boat on the waves" is a symbol. Describing music as a "boat on the waves" is rendering it as an image, as a symbol, that is some form of encoding – maybe it represents cyclical bravery in the heart of the unknown, or maybe it means nothing at all.

The point is, I would just have an "imagistic rendering" going on in my head when I played the piano. Strangely I took this to mean that I was already above the *un-necessary mechanics* of the piano (that is, that it's an instrument where each key is a lever) and certainly had no need to study how a piano worked. I never really cared to know *about* the piano, as I already had my own ideas about music; I just wanted ot use it. What the hell did music theory have to do with me? Maybe this isn't strange. A lot of painters don't get paint chemistry before painting a canvas – they may even intentionally want to say "I hate paint chemistry". Maybe it is strange. Who designs a spaceship without knowing how it works?

It wasn't until I was 17 that I even started to read music, even though at that point I'd played the piano for 10 years. That's like saying, I spoke Mandarin for 10 years before knowing how to read it. Even then, when I would read others' music, I would play the same notes but in a totally-me way – I would mostly change the rhythm and *totally* change the dynamic, such that it did not even sound like the same song. I would make up my own dynamics. If you asked me to play Viva la Vida, I would play it so different, it would sound like … Bob Marley?

It wasn't until I was 18 that I met Dr. Braun, my professor for piano from 2018-2020. Dr. Braun is a tall, lengthy person, with glasses and a sharp wit. It was when I first met Dr. Braun that I would say I started studying the piano. Dr. Braun never explicitly taught me any philosophy, but through years of his patient mantras – "exhale the air out of your shoulders, keep your fingers on the very edge of the key, the weight of your body not the speed of your stoke" – I came to know each one's meaning more deeply. I was being taught patience. I was also being taught tempered self-acceptance: how to mix a mechanical understanding of the piano with my emotional, symbolic one. I finally began to compliment "a hat flying off in the wind" with tilting my body to the right as I ascended to a higher register of the piano. It's this mixture that made me even be able to conceptualize what being a pianist truly meant.

We want the <u>same content</u>. You may instantly, when seeing a problem, *think*

```
sum=0

for i in range(1000):

    if i %4 ==0 or i %7 ==0:

        sum += i

print(sum)
```

There's nothing wrong with that: things like this are sometimes just habits, I think, at least if you learn in the same way as I do. I am not telling you to exclude this. What we want to do is combine our mechanical understanding of Python with the code we are writing. There is no perfect way to do this, but we can try.

This is the use of comments – `#comment or ```comment``` ` – in code, they are to reflect your thought process in a way that indicates understanding. Take this as an example:

~~~
```
To sum multiples cycle through all #s, test multiplicity, and increment to counter initialized @
0 if True
```

#Initialize counter @ 0
sum=0

#Increment over range object – what does range() really do?
for i in range(1000):

    #Multiple -> divisible by. Divisible by -> no remainder upon division.
    #Remainder -> %. No remainder -> x % y equals 0. Equals -> equality operator -> ==.
    if i %4 ==0 or i %7 ==0:

       #Add to itself each time -> increment a counter
       #Counter -> variable initialized @ 0
       #Increment -> counter = counter + i
       #shorten syntax -> counter += i
        sum += i

#Use print function or is it print method?? To print out in console?? What is console?
#To print out result
print(sum)
~~~

Active solve: What do you think of this way of thinking? Do you normally approach problems this way, or are you, like me, inclined towards more *gestural* ways of thinking (silent understanding)? How do you think about combining the *mechanics* of Python into your code?

```
#Desired output: ERROR DOES NOT COMPUTE
```

If I’m telling you this much about my psychology, you should also know that I haven’t moved past listening to 2000s-like alternative music. The Killers remain in my mind, taking up form in slightly more poly-rhythmic tunes.

Functions: inputs, and outputs

Let’s say I wanted to write a f(x) , a function, say it with me *function* ; yes, it’s like Funk Tion. Let’s say I wanted to write a f(x) to … (read this next part very fast in your head):

sum all the multiples of 3 and 5 below 1000

AHHH, god! Liam has given us this problem again. Liam can’t *possibly* be wanting to know this again. Surely, Liam has memorized that this is equal to `233168` . Fear not, dear students, I *do* have memorized that the sum of all multiples of 3 and 5 below 1000 is equal to `233168` . However, *this* time we are studying how to write functions.

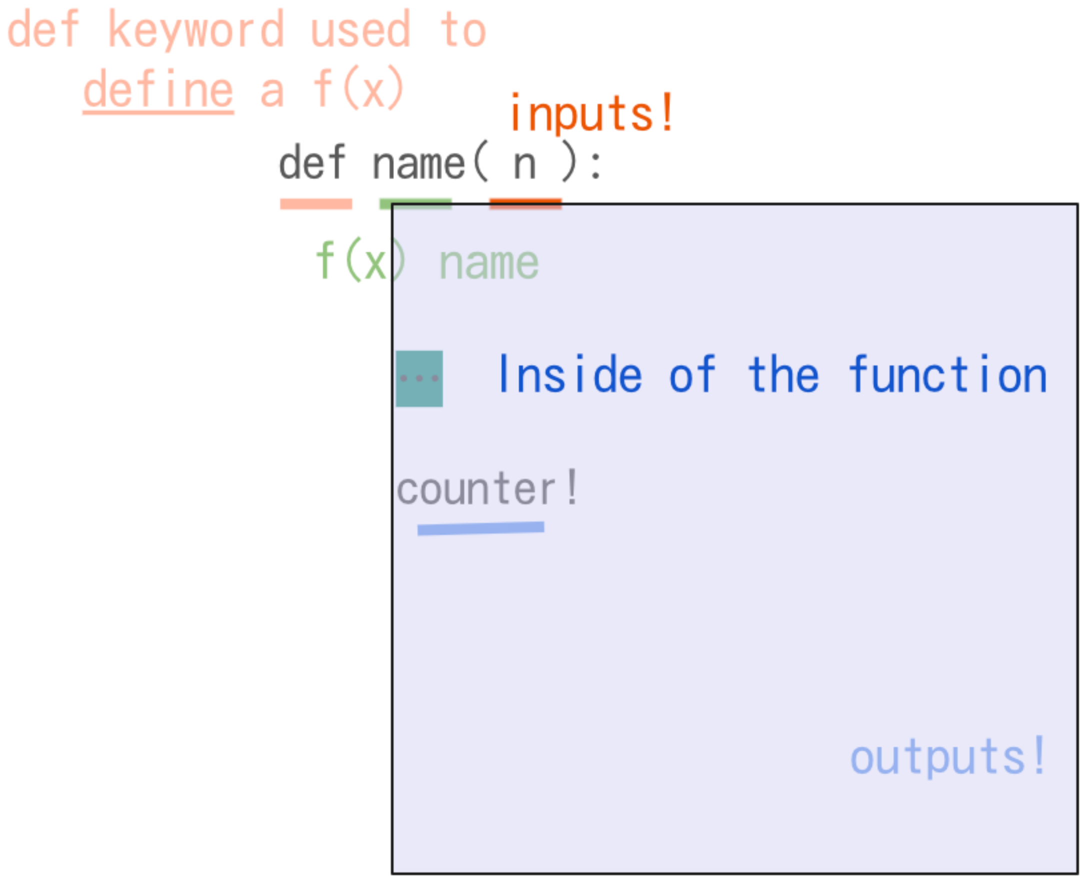
What the <u>fuck</u> (watch my profanity) do we want out of a function? That’s a good question, and the aggression you feel is justified. Let it out. It’s true, we’ve done this problem a lot.

Truthfully, functions are just **inputs** and **outputs**. Nothing more, nothing less. How do we *create* functions? We define them by using the “def” keyword.

```
def function_name(inputs):

    ...

    outputs!
```

Our inputs/outputs can be anything, any data type or a variable itself: `anything=5` `“anything”` `[‘anything’, ‘anything’]` `{‘anything’}` `{‘any’: ‘thing’}` `anything=[ [‘any’], [‘thing’]]`

That `...` may be making you a bit nervous – *that's* where our outputs are being computed. Our outputs are being computed *inside* the function.



Let's start thinking in terms of inputs and outputs. When I say, I want you to take a given letter and tell me the next two letters of the alphabet, what are my inputs and outputs? When we say "given", we mean someone is giving it to you, someone is *inputting* it. "Tell me", "give me back", "return", "yield" all are ways of saying *output*.

Active solve: Let's say I wanted you to write a function that takes in a given integer and returns the square root. What is the input and what is the output? What if I say I wanted you to take a given name and return 'Hello, name!'

The key thing is that given name implies that `def give_name(name):` takes in a variable called “name”. Or, it doesn’t, maybe it’s just `def give_name(“Liam”)` !

Playground: just mess around with the below code. What happens when you move stuff *inside* and *outside* of the f(x)?

```
y = 5
x = 'def'

def f(x):
    x = 'abc'
    print(x*2)
    return y**2

print(x)
print(f(x))
```

Active solve: Can you write a f(x) that takes in a given integer, n, and returns the sum of all numbers less than n that are divisible by 3 and 5?

Active solve: Can you write a f(x) that takes in a given letter, n, and a string, s, and returns the **first** index position of n?

```
#Example input:
n = ‘a’
s = ‘bbbaacdefghisl’
#Output 3
n = ‘x’
s = ‘tyuiox’
#Output: 5
```

Bonus: can you write a f(x) that returns *all* instances of n in s?

```
#Example input:
n = ‘a’
s = ‘bbbaacdefghisl’
#Output:
[3, 4] or “3, 4” both are acceptable
```

You will probably get stuck on this since we haven’t covered `list methods` . Please take a look @ `hints.ipynb` or, if you prefer, you can read the documentation itself!

Let's practice fixing some error in our code**.** Let's say I have written a f(x) that *takes in* a number n and returns the sum of all numbers divisible by 3 and 5:

```
def sum_multiples(n):

    sum = 0
    for i in range(n):
        if i %3 ==0 or i %5 ==0:
           sum += i
           return sum


n = 10

print(sum_multiples(n))
#Expected output: 23
#Actual output: 0
```

Can you point out where the error is, and why it is happening? This is called **debugging**.

Let's try another one!

```
def sum_multiples(n):

    sum = 0
    for i in range(n):
        if n %3 ==0 or n %5 ==0:
           sum += i
    return sum


n = 10

print(sum_multiples(n))
#Expected output: 23
#Actual output: 45
```

Space for your answer:

Functions within functions

Functions **return or yield a value or values to its caller**. When we "run" a function like this:

```
sum_multiples(n))
```

we can say we are "calling" the function "sum_multiples" *on* our input value "n".

Within functions, we can *call* other functions. It's like, adding someone to a group chat.

```
def group_chat([‘Liam’, ‘Hao’, ‘Rachel’]):

    add_new_person(‘Real Ji’)

    add_new_person(‘Hu Yang’)

    return;
```

You can even write a function within a function!

```
def group_chat([‘Liam’, ‘Hao’, ‘Rachel’]):

    def restrict_chat_privileges(person):
         person.chat_privileges = False
         return person

    add_new_person(‘Real Ji’)

    add_new_person(‘Hu Yang’)

    restrict_chat_privleges(‘Liam’)

    return;
```

Active solve: Can you write a f(x) that takes in a given integer, n, and returns the sum of all numbers less than n that are prime? Work off of the starter code below!

```
def sum_below_primes(n):

    def check_prime(integer):




         ...




        return True/False



    sum all primes below n



    return sum;
```

Functions that call themselves

We’ve seen that a f(x) can, within a f(x), make another f(x) and call it.

Check this out:

```
s = ‘Hello’

def chop_by_letter(s):

    print(s)

    chop off the last letter of s and set that equal to new_s

    chop_by_letter(new_s)

chop_by_letter(s)

#Output:
Hello
Hell
Hel
He
H
```

This is called **recursion** , a phenomenon where a function calls itself. This is where we can flirt with the infinite, where we can go down a never-ending rabbit hole, like this:

```
s = ‘Hello’

def infinite_print(s):

    print(s)

    if s:
         infinite_print(s)

infinite_print(s)

#Output:
Hello
Hello
Hello
...

RecursionError: maximum recursion depth exceeded while calling a Python object
```

The concept of **depth** here is just that, as we recursively call a function from itself, each time we are entering a new level of “depth”. We are going deeper.

Active solve: In mathematics, the factorial of a non-negative integer *n*, denoted by *n!*, is the product of all positive integers less than or equal to *n*,

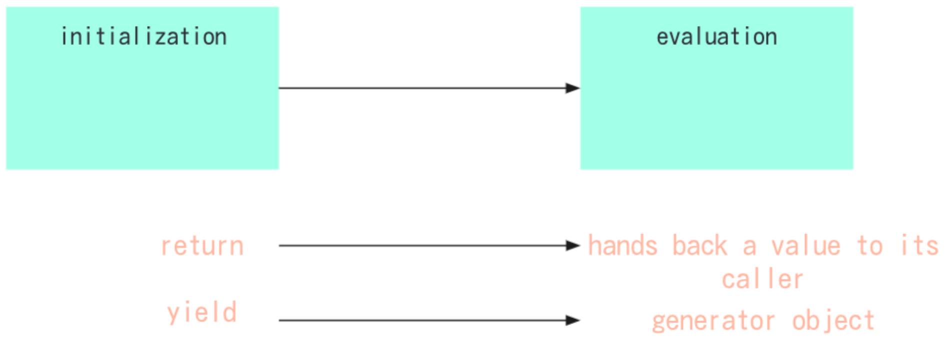
$n! = n * (n-1) * (n-2) * (n-3) \ldots 3 * 2 * 1$

For example,
5! = 5*4*3*2*1 = 120

Can you write a **recursive** function that returns the factorial of a given integer, *n*?

Yield and return

When we talk a function's input and output, steps 1 and 2, we can also call that **initialization** and **evaluation**. In Python, functions can either *return* a value to its caller, or *yield* a value to its generator.



Return only returns one value, and yield can return multiple, picking up each time where it left off. Let's pretend we're reading a book. Return is like reading a book from start to finish in one sitting – you store everything in memory. Yield is like reading a book over multiple days: each time, you pick up where you left off at, at page 80 or 100 or whatever. You only remember where you left off at.

So, when's it useful to use generators? When we have sequences!!

Let's take the Fibonacci sequence, for example:

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

0, 1 … 0+1 =1, 1+1=2, 1+2=3, 2+3=5

Let's say I wanted to **ask you** , what is the *next* number after 34?

Well, what is it? => write it down ! ____

In your head, you probably went, oh 21+34 = 55, right ?

Wait! You're telling me you *didn't* go 0 1 1 2 3 5 8 13 21 34 .. then 55?

You are like a **generator object**.

See this?

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

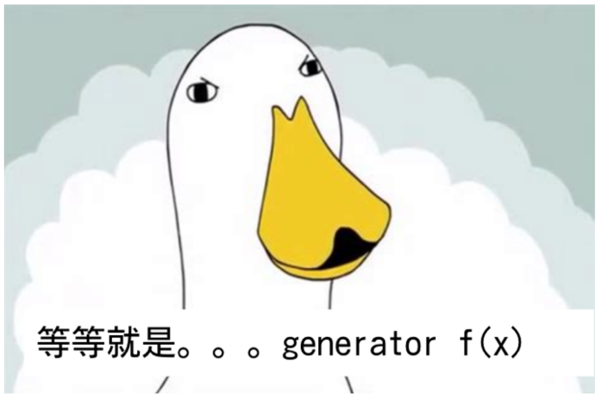
That's you. That's us, really.

When we look @ the Fibonacci sequence and try to calculate the last number, we *could* sit here and go over everything from start to finish

…

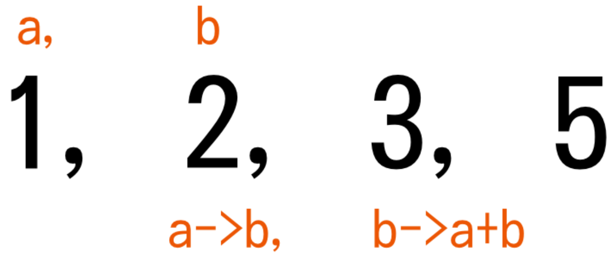
You'd inevitably end up feeling like:

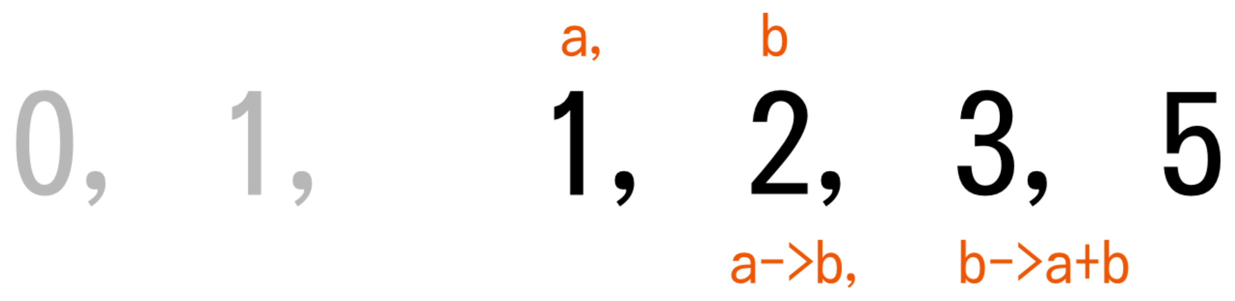
Eventually, hopefully you’d get here:



Trying to remember the whole thing **blows**. It sucks.

Better than that? Just remembering the *last two* numbers of the sequence! All we need to know to know the next number in the Fibonacci sequence is the previous two.





Now, trying to write a **function** to do this is hard. Let’s say we want to print out the Fibonacci sequence of all values less than an input integer.

```
Input:   10
Output:  0 1 1 2 3 5 8
Because: 0, 1, 1, 2, 3, 5, and 8 are all elements of the Fibonacci sequence below the input
integer 10
```

Let’s start with the **pseudo-code**:

```
def fib(n):

   #Kickstart our sequence
   current_val, next_val = 0,1

   so long as we are less than n:

         yield current_val

         assign next two numbers of the Fibonacci sequence
```

What is this *so long as* mess we’ve got ourselves in?

So long as (while loops)

What if I say, *so long as* it’s raining, I am going outside and dancing. That is almost like checking a condition, like an `if` statement, but I want to keep dancing until it STOPS raining, right?

If I just wrote

```
def dance():

   wave hands back and forth

def human_dancer(me):

   if raining:

       return dance()
```

how many times do I end up dancing? **Once!** Only **one measly time** ! Because, this *isn’t* – and this should be kind of your big “WOW” moment – a *so long as* condition, where I am continuing to dance until it STOPs raining, I am just checking <u>one</u> time if it’s raining, and if it is I dance <u>one time</u> .
So, how do I keep dancing while it’s raining?

```
def dance():

   wave hands back and forth

def human_dancer(me):

   while raining:

       return dance()
```

Boom.

This is called a “while loop”. It’s a different kind of iterator than a **for loop**. It does something *so long as* a condition is True, and stops when it’s false.

Active solve: Can you write a **while** loop that counts all the beans in a jar?

```
jar = [‘bean’, ‘bean’, ‘bean’, ‘bean’]

def count_beans(jar):

    ...



#Output:
4
```

You might have thought to write some kind of loop like this:

```
jar = [‘bean’, ‘bean’, ‘bean’, ‘bean’]

def count_beans(jar):

    num_beans = 0

    while num_beans < 4:

        num_beans += 1
```

But how do we *think* about what that `4` truly is? What is that `4` huh? The **number of beans in our jar**, right? It’s one other thing. There’s one another way of referring to it.

…

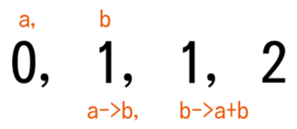
It’s the *length* of our jar list, right? Our jar has 4 things in it.

```
jar = [‘bean’, ‘bean’, ‘bean’, ‘bean’]

def count_beans(jar):

    num_beans = 0

    while num_beans < ~~length~~len(jar):

        num_beans += 1
```

A final side-note is that I will call the built-in f(x) that returns # of items in a given data-type as `len`, not length. OK!

Back to Fibonacci

```
def fib(n):

   #Kickstart our sequence
   current_val, next_val = 0,1

   ~~so long as we are less than length n:~~
   while current_val < n:

         yield current_val

         assign next two numbers of the Fibonacci sequence
```



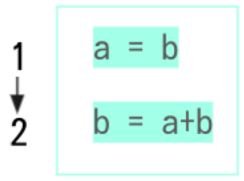
So, how do we get this *assignment* part of it going? How do we make a->b, and b->a+b?

Well, let’s just try:

```
def fib(n):
   #Kickstart our sequence
   a, b = 0,1

   ~~so long as we are less than length n:~~
   while a < n:

        yield a

        a = b
        b = a+b

generator_output = fib(10)
#<generator object fib at 0x1092b2f20>

generator_to_list = list(generator_output)
#convert generator to list

print(generator_to_list)

#Expected output:
[0, 1, 1, 2, 3, 5, 8]
#Actual output:
[0, 1, 2, 4, 8]
```

Crap! Why doesn’t this work?

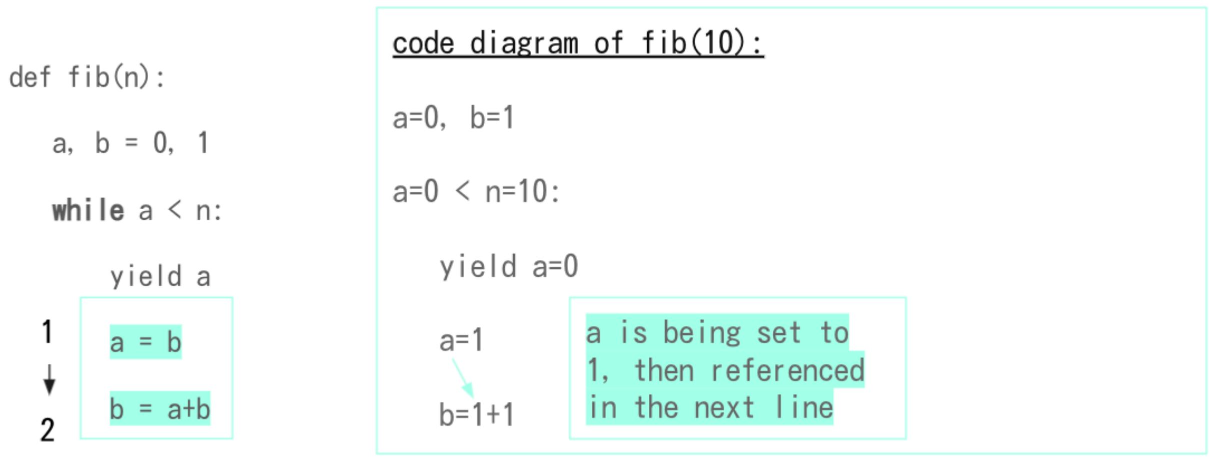
In python, code is executed or ran **synchronously**, one line after the other.



As a result, what do you think happens?

Active solve: Can you take a guess at why our code above is failing? Try diagramming how the code runs over each iteration!

That's right! Maybe you got something like this:



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

<u>code diagram of fib(10):</u>

```
a=0, b=1
a=0 < n=10:
    yield a=0
    a=1
    b=1+1
```

The problem is that *a* is being set to *b* **then** *b* references *a* . We need to find a way to "preserve" *a*'s value <u>before</u> changing it to *b* .

The way you can do this is by using a temporary variable, let's call temp:

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

In python, we can *set* a and b as new values <u>asynchronously</u> via the below syntax:

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

Active solve: In mathematics, the Fibonacci numbers, commonly denoted F_n, form a **sequence**, called the Fibonacci sequence, such that each number is the sum of the two preceding ones, starting from 0 and 1. That is,

$F_0 = 0, F_1 = 1$

and

$F_n = F_{n-1} + F_{n-2}$

for $n > 1$.

The beginning of the sequence is thus:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144 …

List the sum of all even numbers in the Fibonacci sequence below 1000000000000.

Output: 1099511627774

Advanced string indexing

Let's say someone asked us to write a f(x) that, for an input string, **reversed** it.

For example,

```
 def reverse_str(s):

    ...

    output

#Input: "abc"
#Output: "cba"
```

Active solve: Can you write a function that reverses an input string?

```
def reverse_str(s):





    ...





    output

#Input: "abc"
#Output: "cba"
```

You are probably pretty quick at using a `for` loop or `enumerate()` to do that. You might have done something like:

```
def reverse_str(s):

   new_s = ‘’

   for i in s:

       new_s = i + new_s

#Input: “abc”
#Output: “cba”
```

There is a way faster way of indexing strings, though, and those ways are called **positive indexing**, **negative indexing**, and string **slicing** .

I think, for these, it's easier if I just show you:

Playground: just mess around with the below code!

```
s = ‘Happiness, like a flower, blooming in winter and in summer, held eternal in an hour’

s[0]

s[-1]

s[:3]

s[start : stop: step]

s[-1: -4: -2]

s[: : -1]
```

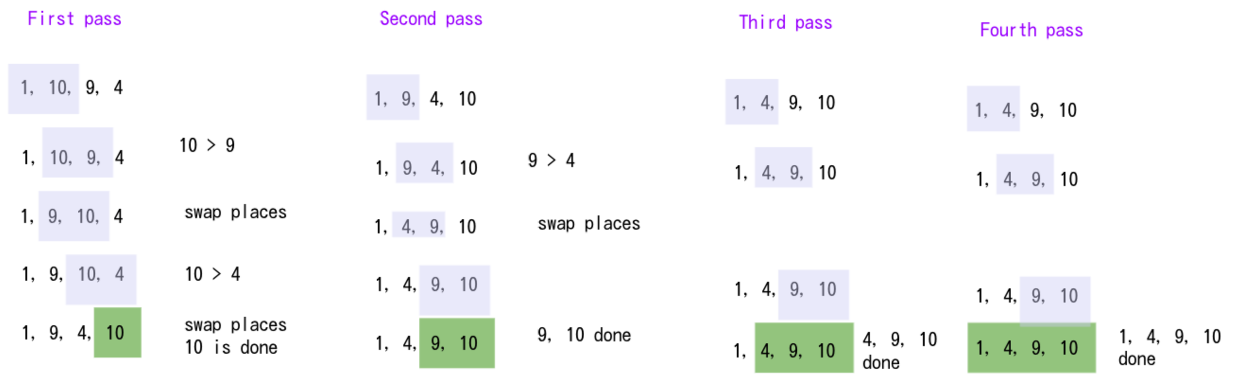
Now, can you revisit the string reversal problem? What do you come up with?

Nested loops

Let’s say I wanted to take a list and sort it, like my input would be an unsorted list `[1, 10, 9, 4]` and my output is a sorted list, `[1, 4, 9, 10]`

How can you think we can do this?

I think the most intuitive way of thinking about it, let’s compare consecutive numbers. Let’s just compare every number with the number next to it, and we will swap places if we need to.



While you may be nodding your head at this, there’s something funky going on here. We understand the “first pass” and “second pass” to be iterations of a *single* for loop, but within each loop there is another loop!

For each pass, we can every number again until our list is ordered.

Let’s code this out:

```
l = [1, 10, 9, 4]
def b(l):
    for i in range(len(l)):
        print(l)
        for j in range(len(l)-1):
            if l[j+1] < l[j]:
                l[j+1], l[j] = l[j], l[j+1]
    return l

print(b(l))
```

Active solve: Can you take a shot at explaining how this sorting algorithm works?

The key is that there is a **nested** for loop. Let’s take a step back. If I write

```
adjective = [‘very’, ‘a lot’]
noun = [‘banana stand’, ‘tree’]

def nested_loop(l1, l2):
    for i in l1:
        for j in l2:
            print(i, j)

print(nested_loop(adjective, noun))

#Output:
very banana stand
very tree
a lot banana stand
a lot tree
```

Active solve: Let’s say I have a string `‘abc’` and want to, using a nested loop, print out for each iteration the remaining characters *after* my current character.

```
a = ‘abc’

...

#Output:
0 a
0 b
0 c
1 b
1 c
2 c
```

Dictionaries

I think, for these, it’s easier if I just show you:

Playground: just mess around with the below code!

```
d = {
        ‘0: ‘zero’,
        ‘1’: ‘one’,
        ‘2’: ‘two’,
     }

d[0]

d[‘2’]

len(d)
```

Homework Problems

Due: Thursday, Jul 1 12:00 AM

Please turn in via email to liamnisaacs@gmail.com . Please complete all the stuff on `noteshomework.ipynb` and submit your answers via a Jupyter notebook file.

1. **All the "Active Solves"**

2. **Divisible 1 to 20**

2520 is the smallest number that can be divided by each of the numbers from 1 to 10 without any remainder.

What is the smallest positive number that is evenly divisible by all of the numbers from 1 to 20? Write your answer as a f(x)

3. **Nested parenthesis depth**

Given a string input, write a f(x) that returns the *nesting depth* of the string.

```
Input: s = "(1+(2*3)+((8)/4))+1"
Output: 3
Explanation: Digit 8 is inside of 3 nested parentheses in the string.
```

4. **Palindromes**

A palindromic number reads the same both ways. The largest palindrome made from the product of two 2-digit numbers is $9009 = 91 \times 99$.

Find the largest palindrome made from the product of two 3-digit numbers.

5. **Digits**

The four adjacent digits in the 1000-digit number that have the greatest product are $9 \times 9 \times 8 \times 9 = 5832$.

Find the thirteen adjacent digits in the 1000-digit number that have the greatest product. What is the value of this product?

6. **Max sum of subarray**

Given an array `a` find the maximum sum of subarray of size 3.

```
a = [4, 2, 1, 1, 1, 1, 80, 10, 10, 4, 2, 1,]

…

#Output:
100
#Explanation: the maximum of subarray size 3 is the sum of [80, 10, 10], which is 100.
```

7. **Say hello**

Write a function that takes in a dictionary *d* that maps "Hello" to a given language of your choice, and prints out a translation.

```
d = {
        'Hello': ...

     }

...

#Example output:
你好
```

Bonus

8. **Longest substring without repeating characters**

Given an array `s` find the length of the **longest substring** without repeating characters.

```
s = "abcabcbb"

#Output:
3
#Explanation: the answer is "abc" which has length 3.

s = "bbbb"

#Output:
1
#Explanation: the answer is "b" which has length 1.

s = "pwwkew"

#Output:
3
#Explanation: the answer is "wke" which has length 3.
```