Jul 1, 2021
PS 2 Solutions

Active Solves

1. Sum all numbers below 100,000 that are divisible by 414 or 4543.

I broke this problem down into a few steps:

- *sum* --> instantiating a counter variable *c* & incrementing …
- *all numbers below* … --> a range up to a given stop integer, where *class* **range**(stop) is an immutable sequence of numbers up to *stop*, i.e. `list(range(10))` gives the list `[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]` .
- *divisible by* … --> the remainder of *x* divided by *y* is 0, i.e. `x % y == 0`[1]

Now it's just putting our parts together. We'll want to (1) instantiate a counter variable *c* , (2) loop over a range of 100,000 , (3) write a conditional if statement that checks if each iteration of our sequence range is (4) "divisible by" (the remainder is 0) of 414 or 4543:

```
c = 0
for i in range(100000):
       if i %414 ==0 or i %4543 ==0:
              c += i
print(c)
#Output: 13222033
```

2. 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!"

In the example of a square root, the *input* is a given integer and the *output* is its square root.

The keyword **def** is used to introduce a function definition, followed by a function name and parenthesized list of inputs. The **return** statement leaves the f(x) with the f-string `f 'Hello, {name}!'` as its return value; optionally, you can choose to simply **print** the same formatted string.

```
def say_hello(name):
       '''Return f-string Hello, name!'''
       return f 'Hello, {name}!'

print(say_hello('Liam'))
#Output: Hello Liam!

def say_hello(name):
       '''Prints Hello, name!'''
       print(f 'Hello, {name}!')
```

[1] where == is the *equality operator*.

```
say_hello('Liam')
#Output: Hello Liam!
```

3. Can you write an 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?

   Same logic as *Active Solve* (1), now implemented with parenthesized input *n* .

```
def sum_upto(n):
	'''Sums all integers in range of n divisible by 3 and 5'''

	c = 0
	for i in range(n):
		if i %3 ==0 and i %5 ==0:
			c += i
	print(c)
	return c

#Call function
sum_upto(1000)
#Output: 33165
```

4. Can you write an f(x) that takes in a given letter *n* and a string *s* and returns the **first** index position of *n* ? (Assume: *n* is in *s*.)

```
def first_index(n, s):
	'''Returns first index of n in s'''
	for ind, val in enumerate(s):
		if val == n:
			return ind;

#Call function
print(first_index('a', 'bbbbba'))
#Output: 5
```

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

   We instantiate an empty list *l* and iterate over the *enumerate object* of s[2] to append (add to our list) the index of each occurrence of *n* in *s* .

```
def all_index(n, s):
	'''Returns all index of n in s'''
	l = []
	for ind, val in enumerate(s):
		if val == n:
			l.append(ind)
	return l;

#Call function
print(all_index('a', 'aaaa')
#Output: [0, 1, 2, 3]
```

[2] The built-in f(x) *enumerate* takes in an iterable and spits out an enumerate object `<enumerate at 0x7fe8b804fcc0>` where 0xf … is an address in memory. Calling `list()` on the iterator returned by enumerate returns a tuple containing index, value pairs.

If you wanted it to be shortened (this does not run any faster), you can use a list comprehension:

```
def all_index(n, s):
    '''Returns all index of n in s'''
    return [i for i, v in enumerate(s) if v == n]
```

Alternatively, without using enumerate:

```
def all_index(n, s):
    '''Returns all index of n in s'''

    l = []
    for i in range(len(s)):
        if s[i] == n:
            l.append(i)
    return l;
```

6. Debugging practice

(i) The code has an error on line 6, where the return statement is nested within the for loop's if statement. Upon reaching the first number divisible by 3 or 5 in the range of *n*, the function will be exited, and the value of *sum* will be returned. Since `0%3==0` and `0%5==0`, the function will instantly return 0. To correct this, the return statement may be un-indented to be outside of the for loop.

Incorrect:

```
1   def sum_multiples(n):
2       sum = 0
3       for i in range(n):
4           if i%3==0 or i%5==0:
5               sum += i
6               return sum;
```

Correct:

```
1   def sum_multiples(n):
2       sum = 0
3       for i in range(n):
4           if i%3==0 or i%5==0:
5               sum += i
6       return sum;
```

(ii) Line 4 of the code uses a control-flow if statement to check if the input integer *n* is divisible by 3 or 5, s.t. the remainder would be 0. Instead of checking if the input integer *n* is divisible by 3 or 5, the problem statement is to sum all multiples *below* *n*, as in each number in the range of *n*. We can correct this by changing line 4 to use the variable *i* as opposed to *n* .

Incorrect:

```
1    def sum_multiples(n):
2        sum = 0
3        for i in range(n):
4            if n%3==0 or n%5==0:
5                sum += i
6        return sum;
```

Correct:

```
1    def sum_multiples(n):
2        sum = 0
3        for i in range(n):
4            if i%3==0 or i%5==0:
5                sum += i
6        return sum;
```

7. Can you write an f(x) that takes in a given integer *n* and returns the sum of all numbers less than *n* that are prime?

   We can break this problem up into steps: (i) write check-if-prime function, (ii) instantiate counter, (iii) iterate over all numbers below *n* , i.e. range(n), (iv) check if each number below *n* is prime, and (v) if it is, increment our counter.

   A prime number is a number only divisible by 1 or itself. So, that means if the number is divisible by anything other than 1 or itself, it’s not prime. Another way of saying that is, if a number is divisible by anything between 2 and itself. Programmatically, that “between 2 and itself” sounds like range(2, itself), since the *class* range can be range(stop) or range(start, stop, step).

```
def sum_primes(n):
    '''Sums all primes below n'''

    def check(m):
        for i in range(2, m):
            if m%i ==0:
                return False
        return True

    s = 0
    for num in range(n):
        if check(num): #if check(num) == True: ← this is the same thing
            s += num

    return s;

print(sum_primes(1000)) #Output → 76128
```

8. 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,

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

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

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

The **iterative** (looping) algorithm's code used to do this can be written as:

```
def factorial(n):

       factorial = 1
       for i in range(1, n+1):
              factorial *= i
       return factorial;
```

Let's say we run factorial(5). We first set factorial to 1 (this will be our running total). `for i in range(1, n+1):` well that's a loop, one that runs from 1 to *n inclusive* of *n* (range stops one short). Then we say `factorial *= i`, in other words we multiply the numbers 1 to *n* by our running total, *factorial* . Our running total starts at 1, then we multiply by 2, then 3, then 4, then 5, then we *exit* the loop and return the value 120 as factorial.

A **recursive** algorithm breaks the problem down into smaller problems and calls itself for each of the smaller problems. It includes a base case (or terminal case) and a recursive case.

Let's take a closer look at *5!*

$5! = 5 * 4 * 3 * 2 * 1$

When we look at that, we see that $4 * 3 * 2 * 1$ is actually equal to $4!$ So,

$5! = 5 * 4!$

Now, let's break down *4!*

$4! = 4 * 3 * 2 * 1$

Look: $3 * 2 * 1$ is actually equal to $3!$ So, we can rewrite this as:

$4! = 4 * 3!$

We can continue doing that until we get to our terminal case, that is $0! = 1$ and $1! = 1$.

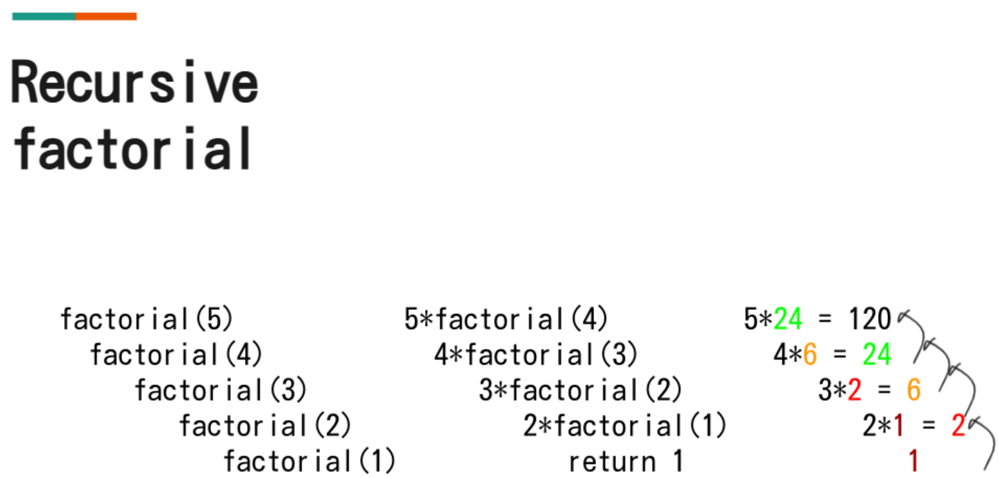
*3! = 3 * 2 * 1*
*3! = 3 * 2!*
*2! = 2 * 1*
*2! = 2 * 1!*
1! = 1 <-- this is our **base/terminal** case, where we will just return 1.

By peeling off the highest number and passing in the number again to the function, factorial lends itself quite well to recursion. If we were to write out our recursive algorithm, we get:

```
#Function calls:

factorial(5)                                    5*factorial(4)
      factorial(4)                              4*factorial(3)
            factorial(3)                        3*factorial(2)
                  factorial(2)                  2*factorial(1)
                        factorial(1)            return 1
```

Then, we start to work backwards from there:



We dig ourselves deep into a hole and compute our way out of it. We have a lot of function calls, then close out the function calls when we get to our base case.

We can write the code for this as:

```
def recursive_factorial(n):

    #Base case 1! = 1, 0! = 1
    if n < 2: return 1
    else: return n*recursive_factorial(n-1)
```

Note on drawbacks of recursion: typically, no calculations are done until the base case is reached. If you have millions of recursive calls, i.e. `recursive_factorial(1000000)` you will probably run out of memory, since you'll have millions of open function calls. It does not **scale up** that well.

9. Can you write a while loop that counts all the beans in a jar?

   While loops are like for loops, but instead of looping *n* number of times, they only loop until a specific condition is met. In this case, our condition is that so long as *c* is less than the number of beans in the jar, otherwise expressed as `len(jar)`.

```
jar = ['bean', 'bean']
c = 0
while c < len(jar):
	c += 1
print(c) #Output: 2
```

10. Debugging Fibonacci

Incorrect:

```
1	def fib(n):
2		a, b = 0, 1
3		while a < n:
4			yield a
5			a = b
6			b = a+b
```

This error happens because python executes code **synchronously**, this means we might just as well write:

```
1	def fib(n):
2		a, b = 0, 1
3		while a < n:
4			yield a
5			a = b
6			b = b+b
```

as the moment line 5, `a = b` , is ran, the value of `a` has been changed to that of `b`.

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

```
def fib(n):
	a, b = 0, 1
	while a < n:
		if a %2 ==0:
			yield a
		a, b = b, a+b

print(sum(list(fib(1000000000000))) #Output: 478361013020
#I think I must've ran the wrong number for the HW, sorry about that...
```

12. The sorting algorithm is called **bubble sort**.

13. Can you write a function that reverses an input string?

    We write a function that steps backwards (-1) from the start, i.e. abc will be read as cba.

```
def reverse_str(s):
	return s[::-1]
```

14. Let's say I have a string 'abc' and want to, using a nested loop, print out for each iteration the <u>remaining</u> characters *after* my current character.

    For our string *txt* we iterate over the enumerate object of *txt* as to preserve our index value *i* and write a nested loop to loop over the characters in *txt* following *i* represented as the string splice txt[i:]. We are then able to use an f-string to print out *i* and *t* adjacent.

```
txt = 'abcdefg'
for i, _ in enumerate(txt):
	for t in txt[i:]:
		print(f'{i} {t}')
```

Alternatively, you can think to use a counter:

```
i = 0
for _ in txt:
	for t in txt[i:]:
		print(f'{i} {t}')
	i += 1
```

In the event we wanted to output on *one line* the characters remaining in a string, e.g.

```
#Output for input 'abcdefg'
bcdefg --> because after a, the string abcdefg has bcdefg left to go
cdefg
defg
efg
fg
g
```

We can write a for loop that loops over the range of the length of txt, initializes an empty string *temp* and iterates over the string splice txt[i+1]: , adding each letter to *temp* . In our first loop we then print out *temp* .

```
txt = 'abcdefg'
for i in range(len(txt)):
	temp_char = ''
	for remaining in txt[i+1:]:
		temp += remaining
	print(temp)
```

Homework Questions

1. Divisible 1 to 20

2520 is the smallest number that can be divided by each of the numbers 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?

Problem courtesy of *Project Euler*.

The **brute force** approach is

```
i = 1
while (i %2 != 0 or i %3 != 0 or i %4 != 0 or i %5 != 0 or i %6 != 0
       or i %7 != 0 or i %8 != 0 or i %9 != 0 or i %10 != 0 or
       i %11 != 0 or i %12 != 0 or i %13 != 0 or i %14 != 0 or
       i %15 != 0 or i %16 != 0 or i %17 != 0 or i %18 != 0 or
       i %19 != 0 or i %20 !=0 ):

       i+= 1;

print(i);
```

```
232792560
[Finished in 28.092s]
```

If something is not divisible by 20, who cares if it divisible by 16, 15. Since 20 is the largest number in the range 1 to 20, we can just start incrementing *i* by 20 each time.

```
i = 1
while (i %2 != 0 or i %3 != 0 or i %4 != 0 or i %5 != 0 or i %6 != 0
       or i %7 != 0 or i %8 != 0 or i %9 != 0 or i %10 != 0 or
       i %11 != 0 or i %12 != 0 or i %13 != 0 or i %14 != 0 or
       i %15 != 0 or i %16 != 0 or i %17 != 0 or i %18 != 0 or
       i %19 != 0 or i %20 !=0 ):

       i+= 20;

print(i);
```

```
232792560
[Finished in 28.092s]
```

That worked, *sort of*. This is where we get pretty math heavy.

We want to find the recursive least common multiple (lcm) of range 1-21:

$$f(n) = lcm(n, f(n-1)), f(1) = 1$$

This is a fancy way of saying, we want *lcm* 20, 21; *lcm* 19, 20.

If *a* and *b* are both non-zero, the *lcm* can be represented as:

$$\gcd(a, b) = \frac{|a*b|}{lcm(a,b)}$$

Where *gcd* is the greatest common denominator.

```
def gcd(x, y):

    if x > y: small = y
    else: small = x
    for i in range(1, small+1):
        if( (x%i==0) and (y%i==0) ):
            gcd = i
    return gcd


def f():
    b = 1
    for i in range(1, 21):
        b *= i //gcd(i, b)
    return str(b)

print(f())

232792560
[Finished in 0.276s]
```

We can shorten this to:

```
import math

def f():
    b = 1
    for i in range(1, 21):
        b *= i // math.gcd(i, b)
    return str(b)

print(f())

232792560
[Finished in 0.203s]
```

2. Nested parenthesis depth

We can instantiate a counter c at zero. We initialize a list l and, by iterating over the string s , we choose to increment c by 1 if a (character is seen, appending c to l , or choose to decrement c by 1 if a) character is seen, appending c to l . We can then take the max value of c and return that as our answer.

```
s = " (((2)))"

# Count +1 if (, -1 if )
# For, ( ( ( 2 ) ) ) , this would look like:
#      1 2 3   2 1 0
# Keep [1, 2, 3, 2, 1, 0] stored in list variable.
# We then call max on our list to return 3 as our answer.

def f(s):

    l = []
    c = 0
    for i in range(len(s)):
          if s[i] == '(':
                c += 1
                l.append(c)
          if s[i] == ')':
                c -= 1
                l.append(c)
    return max(l)

print(f('((2) ((32)))'))
```

A cool trick to keep a running sum instead is,

```
from itertools import accumulate

def f(s):

        return max(list(accumulate(filter(None, map({'(': 1, ')': -1}.get,
s))))

    print(f('((2) ((32)))'))
```

3. Palindromes

We can think to use a nested loop to iterate over the range of 1000 as *i* and *j* respectively, using their product, *i*j* to represent every possible product of 3-digit numbers. To check if the product is palindromic, we convert the *int* data-type to a *str* and check if the *str* read backwards is the same as the *str* . If it is, we can keep track of that in a *list* data-type, and take the *max* of the list to return our answer.

```
l = []
for i in range(1000):
      for j in range(1000):
            if str(i*j) == str(i*j)[::-1]:
                  l.append(i*j)
print(max(l))

906609
[Finished in 0.686s]
```

4. Digits

Here, we can loop through our string in “windows” of 13 characters length. We instantiate an answer at 0, and if the product of each digit in our 13-long window is greater than answer, we set that to answer.

```
s  =  "73167176531330624919225119674426574742355349194934969835203127745
06326239578318016984801869478851843858615607891129494954595017379583319
52853208805511125406987471585238630507156932909632952274430435576689664
89504452445231617318564030987111217223831136222989342338030813533627661
42828064444866452387493035890729629049156044077239071381051585930796086
67017242712188399879790879227492190169972088809377665727333001053367881
22023542180975125454059475224352584907711670556013604839586446706324415
72215539753697817977846174064955149290862569321978468622482839722413756
57056057490261407972968652414535100474821663704844031998900088952434506
58541227588666881164271714799244429282308634656748139191231628245861786
64583591245665294765456828489128831426076900422421902267105562632111110
93705442175069416589604080719840385096245544436298123098787992724428490
91888458015616609791913387549920052406368991256071760605886116467109405
07754100225698315520005593572972571636269561882670428252483600823257530
420752963450"

largest_product = 0
for i in range(len(s)):
      window = s[i:i+13]

      window_prod = 1
      for j in window:
            window_prod *= j

      if window_prod > largest_product: largest_product = window_prod

print(largest_product)

23514624000
[Finished in 0.335s]
```

5. Max sum of subarray

Here, we are using a **brute force** solution to calculate the largest sum of the subarray.

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

def max_sum(array, size=3):

      s = 0
      for i in range(0, len(array)-2):
            t = array[i] + array[i+1] + array[i+2]
            if t > s: s = t
      return s

print(max_sum(a))
#100
```

6. Say hello

```
d = {
      'hello': 'heelo'
}

def map(d, word):
      return d[word]

print(map(d, 'hello'))
#Out: heelo
```