Rauhan Doga

Canonical Code: Project Euler Challenge

Eckel, TJHSST AI1, Fall 2019

```
# To handle prime numbers, I keep a global prime list:
global_primes = [2, 3]

# ...and a next_prime function to append the next prime to the list, testing consecutive odds until success:
def next_prime():
    test = global_primes[-1] + 2
    while not is_prime(test):
        test += 2
    global_primes.append(test)

# ...and an is_prime function that makes sure the prime list is complete up to the square root of the desired
# number, then tests every prime in that range against it.  Dual recursion!
def is_prime(x):
    while global_primes[-1] ** 2 < x:
        next_prime()
    for prime in global_primes:
        if x % prime == 0:
            return False
        if prime ** 2 >= x:
            return True
    return True

#1: List comprehension!
print("#1: %s" % sum([x for x in range(1000) if x % 3 == 0 or x % 5 == 0]))

#2: An iterative Fibonacci implementation, for speed; note that the comprehension ignores the last number in the fib list
fib = [1, 1]
while fib[-1] < 4000000:
    fib.append(fib[-1] + fib[-2])
print("#2: %s" % sum([x for x in fib[:-1] if x % 2 == 0]))

#3: One line, given is_prime above; note the [-1] at the end that only prints the last number in the list
print("#3: %s" % [x for x in range(2, int(600851475143 ** 0.5) + 1) if 600851475143 % x == 0 and is_prime(x)][-1])

#4: This solution relies on knowing that successive differences of squares are smaller, so it's guaranteed to search
#   largest to smallest; also note that the is_pal function at the top is unnecessary, but I think it's cleaner looking
#   code to separate it out and it means I don't have to convert low * high to a string twice
def is_pal(string):
    return string == ''.join(reversed(string))

def find_pal():
    for target in range(999, 100, -1):
        low, high = target, target
        while high <= 999:
            if is_pal(str(low * high)):
                return low*high
            low -= 1
            high += 1

print("#4: %s" % find_pal())

#5: One of the optional challenges, implemented as directed on the paper
def gcd(x, y):
    while y != 0:
        (x, y) = (y, x % y)  # see Wikipedia article for Euclidean Algorithm; also, note tuple packing/unpacking!
    return x

answer = 1
for num in range(1, 21):
    answer = answer * num // gcd(answer, num) # because lcm(x, y) = x * y // gcd(x, y)
print("#5: %s" % answer)

#6: One of the optional challenges; in Python, this is almost too easy
print("#6: %s" % (sum([x for x in range(1, 101)])**2 - sum([x**2 for x in range(1, 101)])))

#7: See the next_prime function above
while len(global_primes) < 10001:
    next_prime()
print("#7: %s" % global_primes[10000])
```

```
#8: Not much unique here, but note the use of "max" and note \ to make single-line statements multiline for readability
num_string = "73167176531330624919225119674426574742355349194934969835203127745063262395783180169848018694788518438 5" +\
             "86156078911294949545950173795833195285320880551112540698747158523863050715693290963295227443043557668 9" +\
             "66489504452445231617318564030987111217223831136222989342338030813533627661428280644448664523874930358 9" +\
             "07296290491560440772390713810515859307960866701724271218839987979087922749219016997208880937766572733 3" +\
             "00105336788122023542180975125454059475224352584907711670556013604839586446706324415722155397536978179 7" +\
             "78461740649551492908625693219784686224828397224137565705605749026140797296865241453510047482166370484 4" +\
             "03199890008895243450658541227588666881164271714799244429282308634656748139191231628245861786645835912 4" +\
             "56652947654568284891288314260769004224219022671055626321111109370544217506941658960408071984038509624 5" +\
             "54443629812309878799272442849091888458015616609791913387549920052406368991256071760605886116467109405 0" +\
             "77541002256983155200055935729725716362695618826704282524836008232575304207529634 50"
biggest_prod = 1
for count in range(len(num_string)-12):
    nums = [int(x) for x in num_string[count:count + 13]]
    prod = 1
    for num in nums:
        prod *= num
    biggest_prod = max(prod, biggest_prod)
print("#8: %s" % biggest_prod)

#9: Note c can be defined in relation to a and b, so we only need two loops, not three; also, counting down is faster if
#   you're looking for the biggest answer
def find_trip():
    for a in range(998, 0, -1):
        for b in range(999 - a, 0, -1):
            c = 1000 - a - b
            if a**2 + b**2 == c**2:
                return a * b * c

print("#9: %s" % find_trip())

#10: An optional challenge; not especially efficient - to do this really fast you need more advanced prime detection
while global_primes[-1] < 2000000:
    next_prime()
index = 1
while global_primes[index] < 2000000: #I need this loop in case global_primes is already longer than needed
    index += 1
print("#10: %s" % sum(global_primes[:index]))

#29: Solution presented just to give an example of a set comprehension; this works because sets remove duplicates
print("#29: %s" % len({a**b for a in range(2, 101) for b in range(2, 101)}))
```