PDSA Notes

by

Gagneet Kaur

Date
December 27, 2021

www.markdownguide.org
www.kaggle.org

PAGE NO.
DATE:

WEEK 1

→ PYTHON RECAP ←

COMPUTING GCD

- gcd (m,n) - greatest common divisor
  - → largest k that divides both m & n
  - → gcd (8,12) = 4
  - → gcd (18,25) = 1
  - → Also hcf - highest common factor

- gcd (m,n) always exists
  - → 1 divides both m and n

- Computing gcd (m,n)
  - → gcd (m,n) ≤ min (m,n)
  - → Compute list of common factors from 1 to min (m,n)
  - → Return the last such common factor

CODE :

```
def gcd (m,n):
    cf = []                    # List of common factors
    for i in range (1, min(m,n) + 1):
        if (m % i) == 0 and (n % i) == 0:
            cf.append(i)
    return (cf[-1])
```

Shrinath
PAGE NO.
DATE:

Points to note : -> Need to initialize cf for cf.append() to work
→ variables (names) derive their type from the value they hold

-> Control flow
→ conditionals (if)
→ loops (for)

-> range (i,j) runs from i to j-1

-> List indices run from 0 to len(l)-1 and backwards from -1 to -len(l)

Eliminate the list :

-> Only the last value of cf is important

-> Keep track of most recent common factor (mrcf)

-> Recall that 1 is always a common factor
→ No need to initialize mrcf

CODE:

```
def gcd(m,n):
    cf = []
    for i in range(1, min(m,n)+1):
        if (m%i) == 0 and (n%i) == 0:
            mrcf = i
    return (mrcf)
```

Efficiency : -> Both versions of gcd take time proportional to min(m,n)
-> Can we do better ?

CHECKING PRIMALITY

- A prime number n has exactly two factors, 1 and n
  → Note that 1 is not a prime no.
- Compute the list of factors of n
- n is a prime if the list of factors is precisely [1,n]

CODE:

```
def factors(n):
    fl = []                    # factor list
    for i in range(1, n+1):
        if (n%i) == 0:
            fl.append(i)
    return(fl)

def prime(n):
    return(factors(n) == [1,n])
```

COUNTING PRIMES

- List all primes upto m
- List the first m primes (multiple simultaneous assignment)

- for vs while
  - → Is the no. of iterations known in advance
  - → Ensure progress to guarantee termination of while

CODE →

```
def primesupto(m):
    pl = []                          # prime list
    for i in range(1, m+1):
        if prime(i):
            pl.append(i)
    return(pl)

def firstprimes(m):
    (count, i, pl) = (0, 1, [])
    while (count < m):
        if prime(i):
            (count, pl) = (count+1, pl+[i])
        i = i+1
    return(pl)
```

COMPUTING PRIMES

- Directly check if n has a factor between 2 and n-1
- Terminate check after we find first factor
  - Breaking out of a loop
- Alternatively, use while

Shriansh
PAGE NO.
DATE:

- Speeding things up slightly
  - → factors occur in pairs
  - → sufficient to check factors upto $\sqrt{n}$
  - → if n is prime, scan $2, \ldots, \sqrt{n}$ instead of $2, \ldots, n-1$.

CODE 1:

```
def prime(n):
    result = True
    for i in range(2,n):
        if (n%i) == 0:
            result = False
            break                    #Abort loop
    return(result)
```

CODE 2:

```
def prime(n):
    (result, i) = (True, 2)
    while (result and (i<n)):
        if (n%i) == 0:
            result = False
        i = i+1
    return(result)
```

CODE 3:

```
import math
def prime(n):
    (result, i) = (True, 2)
    while (result and (i< math.sqrt(n))
        if (n%i) == 0:
            result = False
        i = i+1
    return(result)
```

Shriansh
PAGE NO.
DATE:

PROPERTIES OF PRIMES

- There are infinitely many primes
- How are they distributed
- Twin primes : p, $p+2$
- Twin prime conjecture (There are infinitely many twin primes)
- Compute the differences between primes
- Use a dictionary
- Start checking from 3, since 2 is the smallest prime

CODE:

```
def primediffs(n):
    lastprime = 2
    pd = {}                    #dict for prime differences
    for i in range (3, n+1):
        if prime(i):
            d = i - lastprime
            lastprime = i
            if d in pd.keys():
                pd[d] = pd[d] + 1
            else:
                pd[d] = 1
    return(pd)
```

Better Code (Efficiency)

COMPUTING GCD

CODE 1:

```
def gcd(m,n):
    cf = []                    # List of common factors
    for i in range(1, min(m,n)+1):
        if (m%i)==0 and (n%i)==0:
            cf.append(i)
    return(cf[-1])
```

CODE 2:

```
def gcd(m,n):
    for i in range(1, min(m,n)+1):
        if (m%i)==0 and (n%i)==0:
            mrcf = i
    return(mrcf)
```

-> Both versions of gcd take time proportional to min(m,n) !

-> Can we do better?

-> Suppose d divides m and n
- m = ad , n = bd
- m-n = (a-b)d
- d also divides m-n

-> Recursively defined function
- Base case : n divides m, answer is n

→ Otherwise, reduce gcd(m,n) to gcd(n, m-n)

- Unfortunately this takes time proportional to max(m,n)

CODE:

```
def gcd(m,n):
    (a,b) = (max(m,n), min(m,n))
    if a%b == 0:
        return(b)
    else:
        return(gcd(b, a-b))
```

- Consider gcd(2,9999)
  - → gcd(2,9997)
  - → gcd(2,9995)
  - .....
  - → gcd(2,3)
  - → gcd(2,1)
  - → 1

- Approximately 5000 steps
- Can we do better?

Euclid's Algorithm

- Suppose n does not divide m
- Then, $m = qn + r$
- Suppose d divides both m & n

- Then $m = ad$, $n = bd$
- $m = qn + r \rightarrow ad = q(bd) + r$
- r must be of the form cd
- Euclid's Algorithm :
  - → If n divides m, $gcd(m,n) = n$
  - → Otherwise, compute $gcd(n, m \bmod n)$

CODE :

```
def gcd (m,n):
    (a,b) = (max(m,n), min(m,n))
    if a%b == 0 :
        return (b)
    else :
        return (gcd (b, a%b))
```

- Can show that this takes time proportional to number of digits in max (m,n).
- One of the first non-trivial algorithms

December

PAGE NO.
DATE

EXCEPTION HANDLING

When things go wrong

- Our code could generate many types of errors
  - → y = x/z , but z has value 0
  - → y = int(s), but string s does not represent valid integer
  - → y = 5*x , but x does not have a value
  - → y = l[i], but i is not a valid index for list l
  - → Try to read from a file, but the file does not exist
  - → Try to write a file, but the disk is full

Recovering gracefully

- Try to anticipate errors
- Provide a contingency plan
- Exception Handling

TYPES OF ERRORS

→ Python flags the type of each error

→ Most common error is a syntax error
  - → SyntaxError: invalid syntax
  - → Not much we can do !

→ We are interested in errors when the code is running

NameError occurs in full situations :
- variable is not defined
- calling a function before declaration
- misspelled built-in functions name

Shrinedi
PAGE NO.
DATE:

1. Name used before value is defined
NameError : name 'x' is not defined

2. Division by zero in arithmetic expression
ZeroDivisionError : division by zero

3. Invalid list index
IndexError : list assignment index out of range

TERMINOLOGY

- Raise an exception
  - Run time error → signal error type, with diagnostic information
  NameError : name 'x' is not defined

- Handle an exception
  - Anticipate and take corrective action based on error type

- Unhandled exceptions aborts execution

Handling Exceptions :

```
try :
    ....
    ....   ← code where error may occur

except IndexError :
    ....   ← Handle IndexError
```

→ When we try to change a float point number stored as string to integer it gives VALUE ERROR! Eg: x = '10.5' print(int(x)) → will throw Value Error

PAGE NO.
DATE:

except (NameError, KeyError):
    ..... ← Handle multiple exception types

except:
    ..... ← Handle all other exceptions

else:
    .... ← Execute if try runs without errors

USING EXCEPTIONS 'positively'

- Collect scores in dictionary

scores = {"Shefali": [3, 22], "Harmanpreet": [200, 3]}

- Update the dictionary
- Battery b already exists, append to list

scores[b].append(s)

- New batter, create a fresh entry

scores[b] = [s]

Traditional Approach:

```
if b in scores.keys():
    scores[b].append(s)
else:
    scores[b] = [s]
```

Using exceptions:

```
try:
    scores[b].append(s)
except KeyError:
    scores[b] = [s]
```

Also, we can't convert list type into integer.
It gives Type Error: Eg: x = [10] → TypeError
print(int(x))

PAGE NO.
DATE:

CLASSES & OBJECTS

- Abstract datatype
  - → stores some information
  - → designated functions to manipulate the information
  - → for instance, stack : last-in, first-out, push(), pop()
- Separate the (private) implementation from the (public) specification
- Class
  - → Template for a data type
  - → How data is stored
  - → How public functions manipulate data
- Object
  - → concrete instance of template

Example : 2D points

- A point has coordinates (x, y)
  - → __init__() initializes internal values x, y
  - → First parameter is always self
  - → Here by default a point is at (0,0)
- Translation : shift a point by (Δx, Δy)
  - → $(x, y) \mapsto (x + \Delta x, y + \Delta y)$
- Distance from the origin
  - → $d = \sqrt{x^2 + y^2}$

PAGE NO.
DATE:

CODE:

```
class Point:
    def __init__(self, a=0, b=0):
        self.x = a
        self.y = b

    def translate(self, deltax, deltay):
        self.x += deltax
        self.y += deltay

    def odistance(self):
        import math
        d = math.sqrt(self.x*self.x + self.y*self.y)
        return(d)
```

Example : POLAR COORDINATES

- (r, θ) instead of (x, y)
  - $r = \sqrt{x^2 + y^2}$
  - $\theta = \tan^{-1}(y/x)$
- Distance from origin is just r
- Translation :
  - Convert (r, θ) to (x, y)
  - $x = r\cos\theta$, $y = r\sin\theta$
  - Recompute r, θ from $(x + \Delta x, y + \Delta y)$
- Interface has not changed
  - User need not to be aware whether representation is (x, y) or (r, θ)

Shriansh
PAGE NO.
DATE:

CODE:

```
import math
class Point:
    def __init__(self, a=0, b=0):
        self.r = math.sqrt(a*a + b*b)
        if a==0:
            self.theta = math.pi/2
        else:
            self.theta = math.atan(b/a)

    def odistance(self):
        return(self.r)

    def translate(self, deltax, deltay)
        x = self.r* math.cos(self.theta)
        y = self.r* math.sin(self.theta)
        x += deltax
        y += deltay
        self.r = math.sqrt(x*x + y*y)
        if x==0:
            self.theta = math.pi/2
        else:
            self.theta = math.atan(y/x)
```

Special Functions

- `__init__()` → constructor
- `__str__()` → convert object to string
  - `str(o) == o.__str__()`
  - Implicitly invoked by `print()`

Srivienth
PAGE NO.
DATE:

- __add__()
  - → implicitly invoked by +
- Similarly,
  - → __mult__() invoked by *
  - → __lt__() invoked by <
  - → __ge__() invoked by >=
  - ...

CODE:

```
class Point :

    ....

    def __str__(self):
        return ('('+str(self.x)+','+str(self.y)+')')

    def __add__(self,p):
        return (Point(self.x + p.x, self.y + p.y))
```

Dated December 30, 2021

TIMING OUR CODE

- How long does our code take to execute ?
  - → Depends from one language to another
- Python has a library time with various useful functions
- perf_time() is a performance counter
  - → Absolute value of perf_time() is not meaningful
  - → Compare two consecutive readings to get an interval
  - → Default unit is seconds

CODE:

```
import time
start = time.perf_counter()
.....
# Execute some code
.....
end = time.perf_counter()
elapsed = end - start
```

A TIMER OBJECT

- Create a timer class
- Two internal values
  - → _start_time
  - → _elapsed_time
- start starts the timer

- stop records the elapsed time
- More sophisticated version in the actual code
- Python executes 10^7 operations per second

CODE:

```
import time
class Timer:

    def __init__(self):
        self._start_time = 0
        self._elapsed_time = 0

    def start(self):
        self._start_time = time.perf_counter()

    def stop(self):
        self._elapsed_time = time.perf_counter -
                             self._start_time

    def elapsed(self):
        return (self._elapsed_time)
```

Sat
January 1, 2022

Shrinath
PAGE NO.
DATE:

WHY EFFICIENCY MATTERS ?

A real world problem

- Every sim card needs to be linked to an Aadhaar card
- Validate the Aadhaar details for each SIM card
- Simple nested loop
- How long will this take ?
  - → M sim cards , N Aadhaar cards
  - → Nested loop iterate M·N times
- What are M and N ?
  - → Almost everyone in India has Adhaar card : $N > 10^7$
  - → No. of sim cards registered is similar : $M > 10^7$

CODE →

```
for each SIM card S :
    for each Aadhaar number A :
        check if Aadhaar details of S match A
```

- Assume $M = N = 10^9$
- Nested loops execute 10^{18} times
- We calculated that Python can perform 10^7 operations in a second
- This will take at least 10^{11} seconds
  - → $10^{11} / 60 \approx 1.67 \times 10^9$ minutes
  - → $(1.67 \times 10^9) / 60 \approx 2.8 \times 10^7$ hours
  - → $(2.8 \times 10^7) / 24 \approx 1.17 \times 10^6$ days
  - → $(1.17 \times 10^6) / 365 \approx 3200$ years !
- How can we fix this ?
- Let's take birthday example.

Shriansh
PAGE NO.
DATE:

Guess My Birthday

- You propose a date
- I answer, Yes, Earlier, Later
- Suppose my birthday is 12 April
- A possible sequence of questions
  - → Sept 12 ? Earlier
  - → Feb 23 ? Later
  - → July 2 ? Earlier
  - ...
- What is the best strategy ?
- Interval of possibilities

→ Query Midpoint - halves the interval !!

- → June 30 ? Earlier
- → March 31 ? Later
- → May 15 ? Earlier
- → April 22 ? Earlier
- → April 11 ? Later
- → April 16 ? Earlier
- → April 13 ? Earlier
- → April 12 ? Yes

- Interval shrinks from 365 → 182 → 91 → 45 → 22 → 11 → 5 → 2 → 1
- Under 10 questions !

Real World Problem (Continued...)

- Assume Aadhaar details are sorted by Aadhaar no.
- Using the halving strategy to check each SIM card

- Halving 10 times reduces the interval by a factor of 1000, because $2^{10} = 1024$

- After 10 queries, interval shrinks to 10^6
- After 20 queries, interval shrinks to 10^3
- After 30 queries, interval shrinks to 1
- Total time $\approx 10^9 \times 30$

→ 3000 seconds, or 50 minutes

- From 3200 years to 50 minutes!
- Of course, to achieve this we have to first sort the Aadhaar cards
- Arranging the data results in a much more efficient solution
- Both algorithms and data structures matter.

To convert list to string : use join operator

Eg:
```
l = ['c', 'a', 't']
new = ''.join(l)
print(new)
```

Output : cat