Alle README's en code voorbereiding

April 4, 2022

Part I

Voorbereide oefeningen

1 Bowling

Niet per se theoretisch moeilijk, maar veer regels met uitzonderingen -> moeilijk om goed te krijgen.

```python
#!/usr/bin/env python3

from typing import List

def parse_game(data: List[int]) -> str:
    result = [0] * 10
    is_valid = True
    len_data = len(data)

    score = 0
    data_i = 0
    for i in range(len(result)):
        if data_i >= len_data:
            is_valid = False
            break
        if data[data_i] == 10:  # strike
            score += 10
            if i == 9:  # laatste
                if data_i+1 >= len_data:
                    is_valid = False
                    break
                score += data[data_i+1]  # enkel bonus
                data_i += 1  # add one extra, so we can check if we have not too many eleme
```

```
            else:
                if data_i+2 >= len_data:
                    is_valid = False
                    break
                score += data[data_i+1] + data[data_i+2]  # in het midden
            data_i += 1
        else:
            if data_i+1 >= len_data:
                is_valid = False
                break
            frame_score = data[data_i] + data[data_i + 1]
            if frame_score == 10:  # spare
                if data_i+2 >= len_data:
                    is_valid = False
                    break
                frame_score += data[data_i+2]

            score = score + frame_score
            data_i += 2

        result[i] = score

    if not data_i in [len_data-2, len_data-1, len_data]:  # check if not too many
        is_valid = False

    if is_valid:
        return " ".join(map(str, result))
    else:
        return "INVALID"

if __name__ == "__main__":
    number_of_games = int(input())

    for _ in range(number_of_games):
        print(parse_game(list(map(int, input().split(" ")))))

# vim: set ft=python
```

2 Achtbaan

```
world_size = int(500**(1/3))
world = [[[('.', False) for x in range(world_size)] for y in range(world_size)] for z in ran

def dft_oplossing(available_structures):
```

```
    pass

if __name__ == "__main__":
    N = int(input())

    for n in range(N):
        opdracht = input()
        (aantal, available_structures) = opdracht.strip().split(" ")
        oplossing = dft_oplossing(available_structures.split(""))
        for line in oplossing:
            print(f"{n} {line}")
```

3 Cluedo

```
def get_number_of_carts_per_player(p, l, w):
    return ((p-1) + (l-1) + (w-1))//4


def get_heuristics(steps):
    distributions = [{} for _ in range(4)]
    for step in steps:
        p_from = step[0]
        question = step[1]
        p_to = step[2].rstrip()

        if p_to == "X":
            for pi in (pi for pi in range(4) if pi != int(p_from)-1):
                for qi in question:
                    distributions[pi][qi] = -10000
        else:
            p_to = int(p_to)
            for pi in range(4):
                for qi in question:
                    if pi == p_to-1:
                        distributions[pi][qi] = distributions[pi].get(qi, 0) + 3
                    elif pi < p_to-1:
                        distributions[pi][qi] = distributions[pi].get(qi, 0) - 2
                    elif pi > p_to - 1:
                        distributions[pi][qi] = distributions[pi].get(qi, 0) - 1
    return distributions


def reduce_heuristic(heuristics, number_of_carts_per_player):
    result = []
```

```
    for i in range(len(heuristics)):
        heuristic = sorted(heuristics[i].items(), key=lambda item: item[1], reverse=True)
        result.append("".join(sorted(h[0] for h in heuristic[:number_of_carts_per_player])))
    return result


if __name__ == "__main__":
    N = int(input())

    for n in range(N):
        p, l, w = list(map(int, input().split(" ")))
        number_of_steps = int(input())
        steps = [input().split(" ") for _ in range(number_of_steps)]
        heuristics = get_heuristics(steps)
        reduced = reduce_heuristic(heuristics, get_number_of_carts_per_player(p, l, w))
        print(f"{n+1} " + " ".join(reduced))
```

4 Naomees

Simpele vraag met aan patern matching moet gedaan worden.

2 mogelijke oplossingsmethoden: * Handmatig gaan checken -> `oplossing.py` * Converteer de taal naar regex -> `alt_oplossing.py`

Regex lijkt op het eerste zicht sneller omdat het intern versneld kan worden, maar heeft het probleem met te algemeen te zijn.

Benchmarks:

```
$> hyperfine "python alt_oplossing.py <bench.input" "python oplossing.py <bench.input"
Benchmark 1: python alt_oplossing.py <bench.input
  Time (mean ± ):      67.6 ms ±   8.4 ms    [User: 58.2 ms, System: 9.1 ms]
  Range (min ... max):    55.9 ms ...  88.3 ms    42 runs

Benchmark 2: python oplossing.py <bench.input
  Time (mean ± ):      54.4 ms ±   8.9 ms    [User: 46.3 ms, System: 7.9 ms]
  Range (min ... max):    41.8 ms ...  70.3 ms    56 runs

Summary
  'python oplossing.py <bench.input' ran
    1.24 ± 0.26 times faster than 'python alt_oplossing.py <bench.input'
```

```
allowed = ["ba", "di", "du"]


def check_sentence(line):
    for i in range(0, len(line) // 2, 2):
```

```
        word = line[i:i+2]
        if not word in allowed:
            return "onzin"
        last_word = line[-2-i:-i] if i != 0 else line[-2-i:]
        if word != last_word:
            return "onzin"
    return "naomees"


def naomees(test_id, in_data):
    return str(test_id) + " " + " ".join(check_sentence(line.rstrip()) for line in in_data)


if __name__ == "__main__":
    t = int(input())

    for ti in range(1, t+1):
        in_data = [input() for _ in range(5)]
        print(naomees(ti, in_data))
```

5 NightCrawler

Published in The Startup

Leonidas Boutsikaris Follow

Oct 22, 2019 · 3 min read · Listen

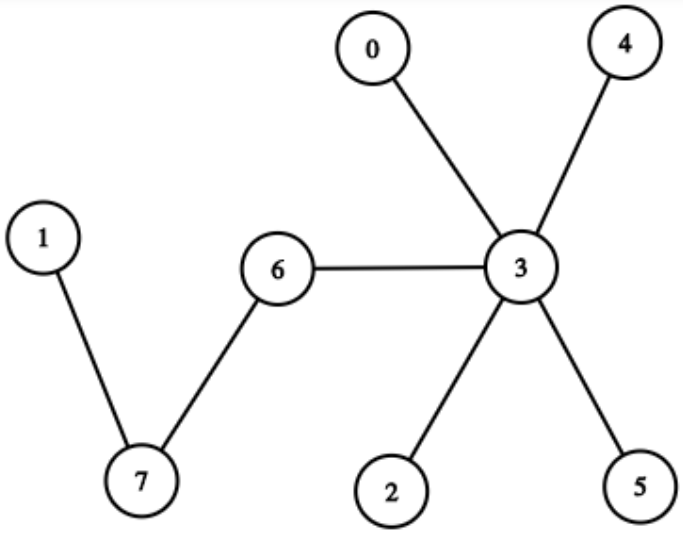
A facility location problem. Where is the optimal placing?

The study of facility location problems (FLP), also known as location analysis, is a branch of operations research and computational geometry concerned with the optimal placement of facilities to minimize transportation costs while considering factors like avoiding placing hazardous materials near housing, and competitors' facilities.

Consider the following scenario

A rural area of a country has decided to improve their health and medicine access. They decided to build a new hospital and a pharmacy but everyone argues about the optimal location of these facilities. Some argue about the population of the cities and others about the distance.

The first step is to create a model of the rural area so we can apply some logic. Using graph theory we will create a non-directed graph of the cities of our rural area. Each vertex will be a city/village and the edges are the connections between them. For example let's take the following graph.



Graph representing the cities of the rural area

Before we try to find the optimal location of the hospital/pharmacy lets define some graph theory terms.

Eccentricity

Eccentricity is the maximum distance from vertex v to any other vertex u of the graph. e(v)= max$\{d($ v, u) $\mid u \in V(G)\}$. For example the Eccentricity of the node 3 is 3 if we consider that every edge has a weight of 1.

The Center of the Graph

The subgraph of a graph G induced by the vertices of G with the minimum eccentricity is called Center of graph G (denoted as center(G)).

In our graph we can see clearly that the center of the graph is the node 3.

Vertex Distance

Vertex Distance is defined as the sum of the distances between a specific vertex and all the vertices of the graph.

distance of v = SUM ($dist(v, u)$) for every $u \in V(G)$

In our case the median of the graph is the node 6.

What about the hospital? what about the pharmacy? Where we should put them?

The hospital must be near the quickest routes. **Take note that an ambulance has to make the same path twice**, one dispatching the vehicle to the location and one returning the patient back to the hospital.

In the other hand the pharmacy just has to be accessible by the citizens.

> The median of the graph can minimize the time because it has the minimum eccentricity. Thats where the hospital must be placed. As you imagined, the center of the graph can be used for the pharmacy because it's more accessible by everyone.

Let's code it.

First thing to do is to transfer the graph in some beep boop computer language. We will use Java.

```
int[][] inGraph = new int[8][8];

// fill zer0s everywhere
for (int l=0; l<8; l++) {
  for (int c=0; c<8; c++){
      inGraph[l][c] = 0;
  }
}

//fill ones according to the neighbors of the graph

inGraph[0][3]=1;
inGraph[1][7]=1;
inGraph[2][3]=1;
```


```
inGraph[4][3]=1;
inGraph[5][3]=1;
inGraph[6][3]=1;
inGraph[6][7]=1;
inGraph[7][1]=1;
inGraph[7][6]=1;

//we now have the adjacency matrix

printGraph(inGraph);
```

We need to use the Dijkstra algorithm so we can find the all the shortests paths for every node.

```
public static ArrayList<Integer> getShortestPaths(int src, int[][] graph) {
  ArrayList<Integer> totalCosts = new ArrayList<Integer>();
  ArrayList<Integer> prevNodes = new ArrayList<Integer>();
  ArrayList<Integer> minPQ = new ArrayList<Integer>();
  ArrayList<Boolean> visited = new ArrayList<Boolean>();

  minPQ.add(src);
  //1. Set all distances to INFINITY except src node
  for (int i=0; i<graph.length; i++) {
    if (i == src) {
      totalCosts.add(i, 0);
      prevNodes.add(src);
    } else {
      totalCosts.add(i, graph.length + 10);
      prevNodes.add(-1);
    }
    visited.add(false);
  }
  /*
  for (int i=0; i< totalCosts.size(); i++) {
    System.out.println("Index: " + i + " Distance: " + totalCosts.get(i));
  }
  */
  while (!minPQ.isEmpty()) {
    int newSmallest = minPQ.remove(getArrayListMinIndex(minPQ));

    for (int i=0; i<graph[newSmallest].length; i++){
```

```
            visited.set(i, true);
            int altPath = totalCosts.get(newSmallest) + neighborDist;

            if (altPath<totalCosts.get(i)) {

              totalCosts.set(i, altPath);
              prevNodes.set(i, newSmallest);
              minPQ.add(i);

            }
          }
        }
      }
    }

    for(int i=0;i<8;i++){
      System.out.print("Cost for "+i+":"+totalCosts.get(i));
      System.out.print(" ////  Previous node visited:"+prevNodes.get(i));
      System.out.println(" ");
    }
    System.out.println(" ");
    return totalCosts;
  }
```

These two helper functions will be used by the Dijkstra algorithm to find the minimum and maximum value of the array.

```
  public static int getArrayListMaxValue(ArrayList<Integer> srcArray){
    int max = -1;
    int maxIndex = -1;
    for(int i=0; i<srcArray.size(); i++){
      if (srcArray.get(i)>max){
        maxIndex = i;
        max = srcArray.get(i);
      }
    }
    return max;
  }

  public static int getArrayListMinIndex(ArrayList<Integer> srcArray){
    int min = srcArray.size() + 10;
```

```
20      }
21    }
22    return minIndex;
23  }
```

helpers.java hosted with ❤ by GitHub view raw

After getting the paths we can find the eccentricity the center and the median as follows

```
1   public static ArrayList<Integer> getVertexEccentricity( ArrayList<ArrayList<Integer>> v
2     ArrayList<Integer> vertexEccentricity = new ArrayList<Integer>();
3     for (int i=0; i<vertexCosts.size(); i++){
4       vertexEccentricity.add(getArrayListMaxValue(vertexCosts.get(i)));
5     }
6     return vertexEccentricity;
7   }
8
9   public static int getCenter(ArrayList<Integer> eccentricities) {
10    return getArrayListMinIndex(eccentricities);
11  }
12
13  public static ArrayList<Integer> getVertexDistances(ArrayList<ArrayList<Integer>> verte
14    ArrayList<Integer> vertexDistances = new ArrayList<Integer>();
15    for (int i=0; i<vertexCosts.size();i++){
16      vertexDistances.add(0);
17      for (int j=0; j<vertexCosts.get(i).size(); j++){
18        vertexDistances.set(i, vertexDistances.get(i) + vertexCosts.get(i).get(j));
19      }
20    }
21    return vertexDistances;
22  }
23  public static int getMedian(ArrayList<Integer> vertexDistances) {
24    return getArrayListMinIndex(vertexDistances);
25  }
```

case.java hosted with ❤ by GitHub view raw

The main of the class will be as follows.

```
1   public static void main(String[] args) {
2     ArrayList<ArrayList<Integer>> vertexCosts = new ArrayList<ArrayList<Integer>>();
```

```
    for (int i=0; i<inGraph.length; i++){
      vertexCosts.add(getShortestPaths(i, inGraph));
    }
    for (int i=0; i<vertexCosts.size(); i++){
      System.out.println("Vertex: "+ i + " Costs :"+ vertexCosts.get(i));
    }
    vertexEccentricity = getVertexEccentricity(vertexCosts);
    System.out.println("Vertex Eccentricities: " + vertexEccentricity);
    center = getCenter(vertexEccentricity);
    System.out.println("We should place the pharmacy at the center of the graph which is
    vertexDistances = getVertexDistances(vertexCosts);
    System.out.println("Vertex Distances: " + vertexDistances);
    median = getMedian(vertexDistances);
    System.out.println("We should place the hospital at the median of the graph which is

}
```

main.java hosted with ♥ by GitHub view raw

Conclusion

We checked a simplified version of a problem that can be very tedious if many factors are acknowledged and connected it with graph theory. At first we made a model of the rural cities with a graph. Using the center and the median of a graph we created an automated way of placing these two facilities in the optimal location. Graph theory helps us model and solve problems using existing theorems and applying everyday logic to them. Every application of the graph theory properties depends solely on the problem we're facing, in our case the hospital and the pharmacy locations.

You can check the live demo of the code here

This blog post was made to feed and share our curiosity in collaboration with Nikos Pantelidis.

6 Bergwandeling

```
# -*- coding: utf-8 -*-
"""
Created on Mon Mar 21 08:55:30 2022

@author: Fox Srk
"""

DimH = 5
DimD = 5
Raster = [32, 30, 40, 26, 25, 44, 7, 12, 15, 19,10, 8, 5, 7, 57,9, 1, 3, 6, 25,7, 5, 2, 44,
Pos = []

I = lambda x : (x%DimD,x//DimD)
O = lambda x : x[0] + x[1]*DimD

test = 1

path = r'wedstrijd.invoer.txt'



def FindRaster():
    Min_p = Raster.index(min(Raster))
    Pos.append(Min_p)

    while(Pos[-1] != -1):
        last_v = Raster[Pos[-1]]
        i = I(Pos[-1])
        Pos.append(-1)
        if(i[0] > 0 and Raster[Pos[-2]-1] > last_v):
            Pos[-1] = Pos[-2]-1
        if(i[0] < DimD-1 and (temp := Raster[Pos[-2]+1]) > last_v and Raster[Pos[-1]]>temp):
            Pos[-1] = Pos[-2]+1
        if(i[1] > 0 and (temp :=Raster[Pos[-2]-DimD]) > last_v and Raster[Pos[-1]]>temp):
            Pos[-1] = Pos[-2]-DimD
        if(i[1] < DimH-1 and (temp := Raster[Pos[-2]+DimD]) > last_v and Raster[Pos[-1]]>tem
            Pos[-1] = Pos[-2]+DimD
        Raster[Pos[-2]] = 99
    Pos.pop()
    L = 0x41
    prt = [chr(L + Pos.index(e)) if e in Pos else '.' for e in range(DimH * DimD)]
    for y in range(0,DimH):
        print(test , end=' ')
        for x in range(0,DimD):
```

```
            print(prt[x + y*DimH], end='')
        print('\n')

with open(path) as FT:
    tot = int(next(FT))
    for T in range(tot):
        Raster = []
        Pos = []
        DimD, DimH = [int(e) for e in next(FT).strip('\n').split(' ')]
        for t in range(DimH):
            Raster.extend([int(e) for e in next(FT).strip('\n').split(' ')])
        FindRaster()
        test += 1
```

7 Buurland

```
import numpy as np


Raster = []

path = r'voorbeeld.invoer.txt'

H, D = 0,0

top = lambda x,y : Raster[y-1][x] if y > 0 else None
left  = lambda x,y : Raster[y][x-1] if x > 0 else None

bottom = lambda x,y : Raster[y+1][x] if y < H-1 else None
right  = lambda x,y : Raster[y][x+1] if x < D-1 else None

def initer(x,y):
    ltr = Raster[y][x]
    Raster[y][x] = nrs
```

```
    if ltr == top(x,y):
        initer(x,y-1)

    if ltr == left(x,y):
        initer(x-1,y)

    if ltr == right(x,y):
        initer(x+1,y)

    if ltr == bottom(x,y):
        initer(x,y+1)


def pth(N):
    global nrs
    nrs = 0
    for y in range(H):
        for x in range(D):
            if type(Raster[y][x]) == str:
                initer(x,y)
                nrs = nrs + 1
    NRS = [{None} for _ in range(max(max(Raster))+1)]
    for y in range(H):
        for x in range(D):
            b = Raster[y][x]
            NRS[b].add(a if (a:= top(x,y)) is not b else None)
            NRS[b].add(a if (a:= left(x,y)) is not b else None)
            NRS[b].add(a if (a:= right(x,y)) is not b else None)
            NRS[b].add(a if (a:= bottom(x,y)) is not b else None)
    for i,e in enumerate(NRS):
        e.remove(None)
    for y in range(H):
        print(N, end=' ')
        for x in range(D):
            print(len(NRS[Raster[y][x]]), end=' ')
        print('\n')


def readFile():
    with open(path) as FT:
        tot = int(next(FT))
        global H, D, Raster
        for i in range(tot):
            D, H = [int(e) for e in next(FT).strip('\n').split(' ')]
            NRS = []
```

```
            Raster = []
            for i in range(H):
                Raster.append([])
                Raster[-1].extend(next(FT).strip('\n'))
            pth(i+1)

readFile()
```

8 Buurland (SamClercky)

```
import queue


def print_kaart(kaart):
    for y in range(len(kaart)):
        print(" ".join(kaart[y]))
    print("-----")


def nbs_to_kaart(kaart, nbs, width, height):
    for y in range(height):
        for x in range(width):
            kaart[y][x] = nbs[kaart[y][x]]


def make_unique(kaart, width, height):
    agenda = queue.PriorityQueue(maxsize=width*height)
    agenda.put((kaart[0][0], kaart[0][0] + "0", 0, 0))  # x, y, from_country
    indices = {kaart[0][0]: 1}

    while not agenda.empty():
        to_country, from_country, x, y = agenda.get()
        if len(kaart[y][x]) == 1:  # not yet visited
            curr_name = to_country
            if curr_name == from_country[0]:  # same color, same country
                kaart[y][x] = from_country
                curr_name = from_country
            else:  # we found a new country
                new_index = indices.get(curr_name, 0)
                indices[curr_name] = new_index + 1
                curr_name = curr_name + str(new_index)
                kaart[y][x] = curr_name
            # expand
            if x+1 < width and len(kaart[y][x+1]) == 1:
                agenda.put((kaart[y][x+1], curr_name, x+1, y))
```

```
            if x-1 >= 0 and len(kaart[y][x-1]) == 1:
                agenda.put((kaart[y][x-1], curr_name, x-1, y))
            if y+1 < height and len(kaart[y+1][x]) == 1:
                agenda.put((kaart[y+1][x], curr_name, x, y+1))
            if y-1 >= 0 and len(kaart[y-1][x]) == 1:
                agenda.put((kaart[y-1][x], curr_name, x, y-1))
        elif kaart[y][x][0] == from_country[0]:
            if kaart[y][x] > from_country:
                kaart[y][x] = from_country
            elif kaart[y][x] < from_country:  # found better way, backtrack
                curr_name = kaart[y][x]
                if x + 1 < width and kaart[y][x + 1] == from_country:
                    agenda.put((kaart[y][x + 1], curr_name, x + 1, y))
                if x - 1 >= 0 and kaart[y][x - 1] == from_country:
                    agenda.put((kaart[y][x - 1], curr_name, x - 1, y))
                if y + 1 < height and kaart[y + 1][x] == from_country:
                    agenda.put((kaart[y + 1][x], curr_name, x, y + 1))
                if y - 1 >= 0 and kaart[y - 1][x] == from_country:
                    agenda.put((kaart[y - 1][x], curr_name, x, y - 1))


def count_neightbours(kaart, width, height):
    nbs = {}
    for y in range(height):
        for x in range(width):
            curr_country = kaart[y][x]
            curr_nbs = nbs.get(curr_country, {})
            if x-1 >= 0 and kaart[y][x-1] != curr_country:
                adj = kaart[y][x-1]
                curr_nbs[adj] = 1
            if y-1 >= 0 and kaart[y-1][x] != curr_country:
                adj = kaart[y-1][x]
                curr_nbs[adj] = 1
            if x+1 < width and kaart[y][x+1] != curr_country:
                adj = kaart[y][x+1]
                curr_nbs[adj] = 1
            if y+1 < height and kaart[y+1][x] != curr_country:
                adj = kaart[y+1][x]
                curr_nbs[adj] = 1
            nbs[curr_country] = curr_nbs
    return {k: len(v.keys()) for (k, v) in nbs.items()}


if __name__ == "__main__":
    N = int(input())
```

```
    for n in range(N):
        width, height = list(map(int, input().rstrip().split(" ")))

        kaart = []
        for h in range(height):
            kaart.append([c for c in input().rstrip()])

        make_unique(kaart, width, height)
        nbs = count_neightbours(kaart, width, height)
        nbs_to_kaart(kaart, nbs, width, height)

        for y in range(height):
            print(n+1, " ".join(map(str, kaart[y])))
```

9 Slinger

```
def correct_chain(chain):
    chain_striped = chain.rstrip().split("*")
    pattern = {}
    for part in chain_striped:
        pattern[part] = pattern.get(part, 0) + 1
    main_pattern = max(pattern.items(), key=lambda item: item[1])[0]
    len_main = len(main_pattern)

    new_chain = []
    for i in range(len(chain_striped)):
        if len(chain_striped[i]) == len_main or (len(chain_striped[i]) < len_main and i == (
            new_chain.append(chain_striped[i])
        else:
            if len(chain_striped[i]) < len_main:  # mistake, to short -> change *
                part_len = 0
                part = []
                while part_len < len_main and i < len(chain_striped):  # accumulate parts it
                    part.append(chain_striped[i])
                    part_len += len(chain_striped[i])
                    i += 1
                if part_len > len_main:  # check if not accumulated too much
                    part.pop()
                    i -= 1
                new_part = [c for c in ".".join(part)]
                if len(new_part) > len_main:  # When having too much handle it by seeing it
                    if len(new_part) == 2*len_main+1 or i == len(chain_striped):  # At * if
                        new_part[len_main] = "*"
                    else:  # if not, weird edge condition -> place one . to the front
                        new_chain[0] = "." + new_chain[0]
```

```
                    new_part.pop()
                new_chain.append("".join(new_part))  # At to new chain
            else:  # mistake, to long -> change .
                part = [c for c in chain_striped[i]]
                if i == 0:
                    part[-len_main-1] = "*"
                else:
                    part[len_main] = "*"
                new_chain.append("".join(part))
                i += 1
            new_chain.extend(chain_striped[i:])  # We found the mistake -> copy all the rest
            break

    return "*".join(new_chain)


if __name__ == "__main__":
    N = int(input())

    for n in range(N):
        chain_len = int(input())
        chain = input()
        print(f"{n+1} " + correct_chain(chain))
```

10 Beleefd

Programmeer technisch niet al te moeilijk. Moeilijk door wiskundige theorie.

Interessante linken: * https://nrich.maths.org/2074 * https://en.wikipedia.org/wiki/Polite_number

Beleefd tot 20:

```
5 = 2 + ... + 3
6 = 1 + ... + 3
7 = 3 + ... + 4
9 = 2 + ... + 4
9 = 4 + ... + 5
10 = 1 + ... + 4
11 = 5 + ... + 6
12 = 3 + ... + 5
13 = 6 + ... + 7
14 = 2 + ... + 5
15 = 1 + ... + 5
15 = 4 + ... + 6
15 = 7 + ... + 8
```

```
17 = 8 + ... + 9
18 = 3 + ... + 6
18 = 5 + ... + 7
19 = 9 + ... + 10
20 = 2 + ... + 6
```

10.1 Vinden van hoeveelheid beleefde nummers

Het aantal oneven delers groter dat 1 = aantal te vinden sequenties

10.1.1 Aantal sequenties

1. Decompositie in priem getallen
2. Neem de machten van de priemfactoren groter dan 1 en +1 voor elk
3. Vermenigvuldig alle en dan -1

Vb. 90:

```
90 = 2 * 3^2 * 5^1
2,1 => (2+1) * (1+1) -1 = 5
```

10.1.2 Constructie sequentie

```
# x is totaal van som
# y is oneven deler
x = sum(i for i in range(x/y - (y-1)/2, x/y + (y+1)/2+1))
```

Het volstaat dus om enkel een priemfactorisatie te doen en de uiterste te berekenen:

```
eerste = x/y - (y-1)/2
laatste = x/y + (y+1)/2
```

```
# https://stackoverflow.com/questions/567222/simple-prime-number-generator-in-python
def is_prime(num):
    if num < 2:          return False
    elif num < 4:        return True
    elif not num % 2:    return False
    elif num < 9:        return True
    elif not num % 3:    return False
    else:
        for n in range(5, int(math.sqrt(num) + 1), 6):
            if not num % n:
                return False
            elif not num % (n + 2):
                return False

    return True
```

NRICH

Polite Numbers

Age 16 to 18
Challenge Level

A polite number is a number which can be written as the sum of two or more consecutive positive integers.
For example, $21 = 10 + 11$ is polite as it is the sum of 2 consecutive positive integers, and $10 = 1 + 2 + 3 + 4$ is polite as it is the sum of four consecutive positive integers.
Here are some questions to think about:

Is 63 a polite number?

If you add up three consecutive integers, what sort of answers do you get?

Are all multiples of 5 polite?

An impolite number is one that cannot be written as a sum of two or more consecutive positive integers.
Can you find an impolite number?

Can an impolite number be odd?

Can you find a rule for identifying impolite numbers?

Show hint

Can you explain why your rule works?

Show hint

When you have explored this problem, you might like to take a look at the different proofs offered in the problem Impossible Sums https://nrich.maths.org/14954 .

Did you know ... ?

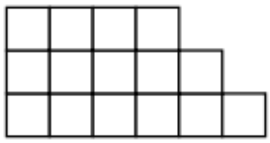
Although number theory - the study of the natural numbers - does not typically feature in school curricula it plays a leading role in university at first year and beyond. Having a good grasp of the fundamentals of number theory is useful across all disciplines of mathematics. Moreover, problems in number theory are a great leisure past time as many require only minimal knowledge of mathematical 'content'.

We are very grateful to the Heilbronn Institute for Mathematical Research https://heilbronn.ac.uk/ for their generous support for the development of this resource.

WIKIPEDIA

Polite number

In number theory, a **polite number** is a positive integer that can be written as the sum of two or more consecutive positive integers. A positive integer which is not polite is called **impolite**.[1][2] The impolite numbers are exactly the powers of two, and the polite numbers are the natural numbers that are not powers of two.



A Young diagram representing visually a polite expansion 15 = 4 + 5 + 6

Polite numbers have also been called **staircase numbers** because the Young diagrams which represent graphically the partitions of a polite number into consecutive integers (in the French notation of drawing these diagrams) resemble staircases.[3][4][5] If all numbers in the sum are strictly greater than one, the numbers so formed are also called **trapezoidal numbers** because they represent patterns of points arranged in a trapezoid.[6][7][8][9][10][11][12]

The problem of representing numbers as sums of consecutive integers and of counting the number of representations of this type has been studied by Sylvester,[13] Mason,[14][15] Leveque,[16] and many other more recent authors.[1][2][17][18][19][20][21][22][23] The polite numbers describe the possible numbers of sides of the Reinhardt polygons.[24]

Contents

Examples and characterization

The first few polite numbers are

> 3, 5, 6, 7, 9, 10, 11, 12, 13, 14, 15, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, ... (sequence A138591 in the OEIS).

The impolite numbers are exactly the powers of two.[13] It follows from the Lambek–Moser theorem that the nth polite number is $f(n + 1)$, where

$$f(n) = n + \lfloor \log_2(n + \log_2 n) \rfloor .$$

Politeness

The *politeness* of a positive number is defined as the number of ways it can be expressed as the sum of consecutive integers. For every x, the politeness of x equals the number of odd divisors of x that are greater than one.[13] The politeness of the numbers 1, 2, 3, ... is

0, 0, 1, 0, 1, 1, 1, 0, 2, 1, 1, 1, 1, 1, 3, 0, 1, 2, 1, 1, 3, ... (sequence A069283 in the OEIS).

For instance, the politeness of 9 is 2 because it has two odd divisors, 3 and itself, and two polite representations

$9 = 2 + 3 + 4 = 4 + 5;$

the politeness of 15 is 3 because it has three odd divisors, 3, 5, and 15, and (as is familiar to cribbage players)[25] three polite representations

$15 = 4 + 5 + 6 = 1 + 2 + 3 + 4 + 5 = 7 + 8.$

An easy way of calculating the politeness of a positive number by decomposing the number into its prime factors, taking the powers of all prime factors greater than 2, adding 1 to all of them, multiplying the numbers thus obtained with each other and subtracting 1. For instance 90 has politeness 5 because $\mathbf{90 = 2 \times 3^2 \times 5^1}$; the powers of 3 and 5 are respectively 2 and 1, and applying this method $\mathbf{(2+1) \times (1+1) - 1 = 5}$.

Construction of polite representations from odd divisors

To see the connection between odd divisors and polite representations, suppose a number x has the odd divisor $y > 1$. Then y consecutive integers centered on x/y (so that their average value is x/y) have x as their sum:

$$x = \sum_{i=\frac{x}{y}-\frac{y-1}{2}}^{\frac{x}{y}+\frac{y-1}{2}} i.$$

Some of the terms in this sum may be zero or negative. However, if a term is zero it can be omitted and any negative terms may be used to cancel positive ones, leading to a polite representation for x. (The requirement that $y > 1$ corresponds to the requirement that a polite representation have more than one term; applying the same construction for $y = 1$ would just lead to the trivial one-term representation $x = x$.) For instance, the polite number $x = 14$ has a single nontrivial odd divisor, 7. It is therefore the sum of 7 consecutive numbers centered at $14/7 = 2$:

$14 = (2 - 3) + (2 - 2) + (2 - 1) + 2 + (2 + 1) + (2 + 2) + (2 + 3).$

The first term, −1, cancels a later +1, and the second term, zero, can be omitted, leading to the polite representation

$14 = 2 + (2 + 1) + (2 + 2) + (2 + 3) = 2 + 3 + 4 + 5.$

Conversely, every polite representation of x can be formed from this construction. If a representation has an odd number of terms, x/y is the middle term, while if it has an even number of terms and its minimum value is m it may be extended in a unique way to a longer sequence with the same sum and an odd number of terms, by including the $2m - 1$ numbers $-(m - 1), -(m - 2), \ldots, -1, 0, 1, \ldots, m - 2, m - 1$. After this extension, again, x/y is the middle term. By this construction, the polite representations of a number and its odd divisors greater than one may be

placed into a one-to-one correspondence, giving a bijective proof of the characterization of polite numbers and politeness.[13][26] More generally, the same idea gives a two-to-one correspondence between, on the one hand, representations as a sum of consecutive integers (allowing zero, negative numbers, and single-term representations) and on the other hand odd divisors (including 1).[15]

Another generalization of this result states that, for any n, the number of partitions of n into odd numbers having k distinct values equals the number of partitions of n into distinct numbers having k maximal runs of consecutive numbers.[13][27][28] Here a run is one or more consecutive values such that the next larger and the next smaller consecutive values are not part of the partition; for instance the partition $10 = 1 + 4 + 5$ has two runs, 1 and $4 + 5$. A polite representation has a single run, and a partition with one value d is equivalent to a factorization of n as the product $d \cdot (n/d)$, so the special case $k = 1$ of this result states again the equivalence between polite representations and odd factors (including in this case the trivial representation $n = n$ and the trivial odd factor 1).

Trapezoidal numbers

If a polite representation starts with 1, the number so represented is a triangular number

$$T_n = \frac{n(n+1)}{2} = 1 + 2 + \cdots + n.$$

Otherwise, it is the difference of two nonconsecutive triangular numbers

$$i + (i+1) + (i+2) + \cdots + j = T_j - T_{i-1} \quad (j > i \geq 2).$$

This second case is called a trapezoidal number.[12] One can also consider polite numbers that aren't trapezoidal. The only such numbers are the triangular numbers with only one nontrivial odd divisor, because for those numbers, according to the bijection described earlier, the odd divisor corresponds to the triangular representation and there can be no other polite representations. Thus, non-trapezoidal polite number must have the form of a power of two multiplied by an odd prime. As Jones and Lord observe,[12] there are exactly two types of triangular numbers with this form:

1. the even perfect numbers $2^{n-1}(2^n - 1)$ formed by the product of a Mersenne prime $2^n - 1$ with half the nearest power of two, and
2. the products $2^{n-1}(2^n + 1)$ of a Fermat prime $2^n + 1$ with half the nearest power of two.

(sequence A068195 in the OEIS). For instance, the perfect number $28 = 2^{3-1}(2^3 - 1)$ and the number $136 = 2^{4-1}(2^4 + 1)$ are both this type of polite number. It is conjectured that there are infinitely many Mersenne primes, in which case there are also infinitely many polite numbers of this type.

References

1. Adams, Ken (March 1993), "How polite is *x*?", *The Mathematical Gazette*, **77** (478): 79–80, doi:10.2307/3619263 (https://doi.org/10.2307%2F3619263), JSTOR 3619263 (https://www.jstor.org/stable/3619263).
2. Griggs, Terry S. (December 1991), "Impolite Numbers", *The Mathematical Gazette*, **75** (474): 442–443, doi:10.2307/3618630 (https://doi.org/10.2307%2F3618630), JSTOR 3618630 (https://www.jstor.org/stable/3618630).
3. Mason, John; Burton, Leone; Stacey, Kaye (1982), *Thinking Mathematically*, Addison-Wesley, ISBN 978-0-201-10238-3.

4. Stacey, K.; Groves, S. (1985), *Strategies for Problem Solving*, Melbourne: Latitude.
5. Stacey, K.; Scott, N. (2000), "Orientation to deep structure when trying examples: a key to successful problem solving", in Carillo, J.; Contreras, L. C. (eds.), *Resolucion de Problemas en los Albores del Siglo XXI: Una vision Internacional desde Multiples Perspectivas y Niveles Educativos* (https://web.archive.org/web/20080726085811/http://staff.edfac.unimelb.edu.au/~kayecs/publications/2000/ScottStacey-OrientationTo.pdf) (PDF), Huelva, Spain: Hergue, pp. 119–147, archived from the original (http://staff.edfac.unimelb.edu.au/~kayecs/publications/2000/ScottStacey-OrientationTo.pdf) (PDF) on 2008-07-26.
6. Gamer, Carlton; Roeder, David W.; Watkins, John J. (1985), "Trapezoidal numbers", *Mathematics Magazine*, **58** (2): 108–110, doi:10.2307/2689901 (https://doi.org/10.2307%2F2689901), JSTOR 2689901 (https://www.jstor.org/stable/2689901).
7. Jean, Charles-É. (March 1991), "Les nombres trapézoïdaux" (http://www.recreomath.qc.ca/art_trapezoidaux_n.htm) (French), *Bulletin de l'AMQ*: 6–11.
8. Haggard, Paul W.; Morales, Kelly L. (1993), "Discovering relationships and patterns by exploring trapezoidal numbers", *International Journal of Mathematical Education in Science and Technology*, **24** (1): 85–90, doi:10.1080/0020739930240111 (https://doi.org/10.1080%2F0020739930240111).
9. Feinberg-McBrian, Carol (1996), "The case of trapezoidal numbers", *Mathematics Teacher*, **89** (1): 16–24.
10. Smith, Jim (1997), "Trapezoidal numbers", *Mathematics in School*, **5**: 42.
11. Verhoeff, T. (1999), "Rectangular and trapezoidal arrangements" (http://www.emis.de/journals/JIS/trapzoid.html), *Journal of Integer Sequences*, **2**: 16, Bibcode:1999JIntS...2...16V (https://ui.adsabs.harvard.edu/abs/1999JIntS...2...16V), Article 99.1.6.
12. Jones, Chris; Lord, Nick (1999), "Characterising non-trapezoidal numbers", *The Mathematical Gazette*, **83** (497): 262–263, doi:10.2307/3619053 (https://doi.org/10.2307%2F3619053), JSTOR 3619053 (https://www.jstor.org/stable/3619053).
13. Sylvester, J. J.; Franklin, F (1882), "A constructive theory of partitions, arranged in three acts, an interact and an exodion", *American Journal of Mathematics*, **5** (1): 251–330, doi:10.2307/2369545 (https://doi.org/10.2307%2F2369545), JSTOR 2369545 (https://www.jstor.org/stable/2369545). In The collected mathematical papers of James Joseph Sylvester (December 1904) (https://archive.org/details/collectedmathem04sylvrich), H. F. Baker, ed. Sylvester defines the *class* of a partition into distinct integers as the number of blocks of consecutive integers in the partition, so in his notation a polite partition is of first class.
14. Mason, T. E. (1911), "On the representations of a number as a sum of consecutive integers", *Proceedings of the Indiana Academy of Science*: 273–274.
15. Mason, Thomas E. (1912), "On the representation of an integer as the sum of consecutive integers", *American Mathematical Monthly*, **19** (3): 46–50, doi:10.2307/2972423 (https://doi.org/10.2307%2F2972423), JSTOR 2972423 (https://www.jstor.org/stable/2972423), MR 1517654 (https://www.ams.org/mathscinet-getitem?mr=1517654).
16. Leveque, W. J. (1950), "On representations as a sum of consecutive integers", *Canadian Journal of Mathematics*, **2**: 399–405, doi:10.4153/CJM-1950-036-3 (https://doi.org/10.4153%2FCJM-1950-036-3), MR 0038368 (https://www.ams.org/mathscinet-getitem?mr=0038368),
17. Pong, Wai Yan (2007), "Sums of consecutive integers", *College Math. J.*, **38** (2): 119–123, arXiv:math/0701149 (https://arxiv.org/abs/math/0701149), Bibcode:2007math.....1149P (https://ui.adsabs.harvard.edu/abs/2007math.....1149P), MR 2293915 (https://www.ams.org/mathscinet-getitem?mr=2293915).
18. Britt, Michael J. C.; Fradin, Lillie; Philips, Kathy; Feldman, Dima; Cooper, Leon N. (2005), "On sums of consecutive integers", *Quart. Appl. Math.*, **63** (4): 791–792, doi:10.1090/S0033-569X-05-00991-1 (https://doi.org/10.1090%2FS0033-569X-05-00991-1), MR 2187932 (https://www.ams.org/mathscinet-getitem?mr=2187932).

19. Frenzen, C. L. (1997), "Proof without words: sums of consecutive positive integers", *Math. Mag.*, **70** (4): 294, JSTOR 2690871 (https://www.jstor.org/stable/2690871), MR 1573264 (https://www.ams.org/mathscinet-getitem?mr=1573264).
20. Guy, Robert (1982), "Sums of consecutive integers" (http://www.fq.math.ca/Scanned/20-1/guy.pdf) (PDF), *Fibonacci Quarterly*, **20** (1): 36–38, Zbl 0475.10014 (https://zbmath.org/?format=complete&q=an:0475.10014).
21. Apostol, Tom M. (2003), "Sums of consecutive positive integers", *The Mathematical Gazette*, **87** (508): 98–101, JSTOR 3620570 (https://www.jstor.org/stable/3620570).
22. Prielipp, Robert W.; Kuenzi, Norbert J. (1975), "Sums of consecutive positive integers", *Mathematics Teacher*, **68** (1): 18–21.
23. Parker, John (1998), "Sums of consecutive integers", *Mathematics in School*, **27** (2): 8–11.
24. Mossinghoff, Michael J. (2011), "Enumerating isodiametric and isoperimetric polygons", *Journal of Combinatorial Theory*, Series A, **118** (6): 1801–1815, doi:10.1016/j.jcta.2011.03.004 (https://doi.org/10.1016%2Fj.jcta.2011.03.004), MR 2793611 (https://www.ams.org/mathscinet-getitem?mr=2793611)
25. Graham, Ronald; Knuth, Donald; Patashnik, Oren (1988), "Problem 2.30", *Concrete Mathematics*, Addison-Wesley, p. 65, ISBN 978-0-201-14236-5.
26. Vaderlind, Paul; Guy, Richard K.; Larson, Loren C. (2002), *The inquisitive problem solver*, Mathematical Association of America, pp. 205–206, ISBN 978-0-88385-806-6.
27. Andrews, G. E. (1966), "On generalizations of Euler's partition theorem", *Michigan Mathematical Journal*, **13** (4): 491–498, doi:10.1307/mmj/1028999609 (https://doi.org/10.1307%2Fmmj%2F1028999609), MR 0202617 (https://www.ams.org/mathscinet-getitem?mr=0202617).
28. Ramamani, V.; Venkatachaliengar, K. (1972), "On a partition theorem of Sylvester", *The Michigan Mathematical Journal*, **19** (2): 137–140, doi:10.1307/mmj/1029000844 (https://doi.org/10.1307%2Fmmj%2F1029000844), MR 0304323 (https://www.ams.org/mathscinet-getitem?mr=0304323).

External links

- *Polite Numbers* (http://nrich.maths.org/public/viewer.php?obj_id=2074), NRICH, University of Cambridge, December 2002
- An Introduction to Runsums (https://web.archive.org/web/20030227224508/http://www.mcs.surrey.ac.uk/Personal/R.Knott/runsums/index.html), R. Knott.
- Is there any pattern to the set of trapezoidal numbers? (https://archive.today/20130415051118/http://www.intellectualism.org/questions/QOTD/oct03/20031002.php) Intellectualism.org question of the day, October 2, 2003. With a diagram showing trapezoidal numbers color-coded by the number of terms in their expansions.

11 Regels afbreken

11.1 Formule rafeligheid

```
lines = ["Lijn 1 tekst", "Lijn 2 tekst"]
max_len_regel = max(len(line) for line in lines)
rafel = sum((max_len_regel - len(line)**2 for line in lines))
```

11.2 Constructie met minimale rafeligheid

```
max_len = 80
input_text = "bla bla bla bla bla"
input_len = len(input_text)
result = []
index = 0
while index < input_len:
    start_index = index
    # advance max_len
    index += 80
    if index < input_len:
        # backtrack to prev whitespace
        while input_text[index] != " ":
            index -= 1
        result.append(input_text[start_index:index])
```

Opm: Het is waarschijnlijk mogelijk om de maximum en lengte van alle aparte onderdelen te berekenen in vorig stuk code. Er wordt niet gevraagd naar de geformateerde tekst, enkel de rafeligheid!

12 Minmax

```
def collect_list():
    n = int(input())
    return [int(input()) for _ in range(n)]


def minmax(volgnummer, lijst):
    mini = 9999999999999
    maxi = -9999999999999
    for i in lijst:
        if i < mini:
            mini = i
        if i > maxi:
            maxi = i
    return volgnummer, mini, maxi
```

```
if __name__ == "__main__":
    k = int(input())

    for ki in range(1, k+1):
        print(" ".join(map(str, minmax(ki, collect_list()))))

#test
```

Part II

Python documentatie

5. Data Structures

This chapter describes some things you've learned about already in more detail, and adds some new things as well.

5.1. More on Lists

The list data type has some more methods. Here are all of the methods of list objects:

`list.`**`append`**(*x*)
Add an item to the end of the list. Equivalent to `a[len(a):] = [x]`.

`list.`**`extend`**(*iterable*)
Extend the list by appending all the items from the iterable. Equivalent to `a[len(a):] = iterable`.

`list.`**`insert`**(*i*, *x*)
Insert an item at a given position. The first argument is the index of the element before which to insert, so `a.insert(0, x)` inserts at the front of the list, and `a.insert(len(a), x)` is equivalent to `a.append(x)`.

`list.`**`remove`**(*x*)
Remove the first item from the list whose value is equal to *x*. It raises a `ValueError` if there is no such item.

`list.`**`pop`**([*i*])
Remove the item at the given position in the list, and return it. If no index is specified, `a.pop()` removes and returns the last item in the list. (The square brackets around the *i* in the method signature denote that the parameter is optional, not that you should type square brackets at that position. You will see this notation frequently in the Python Library Reference.)

`list.`**`clear`**()
Remove all items from the list. Equivalent to `del a[:]`.

`list.`**`index`**(*x*[, *start*[, *end*]])
Return zero-based index in the list of the first item whose value is equal to *x*. Raises a `ValueError` if there is no such item.

The optional arguments *start* and *end* are interpreted as in the slice notation and are used to limit the search to a particular subsequence of the list. The returned index is computed relative to the beginning of the full sequence rather than the *start* argument.

`list.`**`count`**(*x*)
Return the number of times *x* appears in the list.

`list.`**`sort`**(**, key=None, reverse=False*)
Sort the items of the list in place (the arguments can be used for sort customization, see `sorted()` for their explanation).

`list.`**`reverse`**()

`list.`**`copy`**`()`

Return a shallow copy of the list. Equivalent to `a[:]`.

An example that uses most of the list methods:

```
>>> fruits = ['orange', 'apple', 'pear', 'banana', 'kiwi', 'apple', 'banana']
>>> fruits.count('apple')
2
>>> fruits.count('tangerine')
0
>>> fruits.index('banana')
3
>>> fruits.index('banana', 4)  # Find next banana starting a position 4
6
>>> fruits.reverse()
>>> fruits
['banana', 'apple', 'kiwi', 'banana', 'pear', 'apple', 'orange']
>>> fruits.append('grape')
>>> fruits
['banana', 'apple', 'kiwi', 'banana', 'pear', 'apple', 'orange', 'grape']
>>> fruits.sort()
>>> fruits
['apple', 'apple', 'banana', 'banana', 'grape', 'kiwi', 'orange', 'pear']
>>> fruits.pop()
'pear'
```

You might have noticed that methods like `insert`, `remove` or `sort` that only modify the list have no return value printed – they return the default `None`. [1] This is a design principle for all mutable data structures in Python.

Another thing you might notice is that not all data can be sorted or compared. For instance, `[None, 'hello', 10]` doesn't sort because integers can't be compared to strings and *None* can't be compared to other types. Also, there are some types that don't have a defined ordering relation. For example, `3+4j < 5+7j` isn't a valid comparison.

5.1.1. Using Lists as Stacks

The list methods make it very easy to use a list as a stack, where the last element added is the first element retrieved ("last-in, first-out"). To add an item to the top of the stack, use `append()`. To retrieve an item from the top of the stack, use `pop()` without an explicit index. For example:

```
>>> stack = [3, 4, 5]
>>> stack.append(6)
>>> stack.append(7)
>>> stack
[3, 4, 5, 6, 7]
>>> stack.pop()
7
>>> stack
[3, 4, 5, 6]
>>> stack.pop()
6
>>> stack.pop()
5
>>> stack
[3, 4]
```

It is also possible to use a list as a queue, where the first element added is the first element retrieved (“first-in, first-out”); however, lists are not efficient for this purpose. While appends and pops from the end of list are fast, doing inserts or pops from the beginning of a list is slow (because all of the other elements have to be shifted by one).

To implement a queue, use `collections.deque` which was designed to have fast appends and pops from both ends. For example:

```
>>> from collections import deque
>>> queue = deque(["Eric", "John", "Michael"])
>>> queue.append("Terry")           # Terry arrives
>>> queue.append("Graham")          # Graham arrives
>>> queue.popleft()                 # The first to arrive now leaves
'Eric'
>>> queue.popleft()                 # The second to arrive now leaves
'John'
>>> queue                           # Remaining queue in order of arrival
deque(['Michael', 'Terry', 'Graham'])
```

5.1.3. List Comprehensions

List comprehensions provide a concise way to create lists. Common applications are to make new lists where each element is the result of some operations applied to each member of another sequence or iterable, or to create a subsequence of those elements that satisfy a certain condition.

For example, assume we want to create a list of squares, like:

```
>>> squares = []
>>> for x in range(10):
...     squares.append(x**2)
...
>>> squares
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Note that this creates (or overwrites) a variable named `x` that still exists after the loop completes. We can calculate the list of squares without any side effects using:

```
squares = list(map(lambda x: x**2, range(10)))
```

or, equivalently:

```
squares = [x**2 for x in range(10)]
```

which is more concise and readable.

A list comprehension consists of brackets containing an expression followed by a `for` clause, then zero or more `for` or `if` clauses. The result will be a new list resulting from evaluating the expression in the context of the `for` and `if` clauses which follow it. For example, this listcomp combines the elements of two lists if they are not equal:

```
>>> [(x, y) for x in [1,2,3] for y in [3,1,4] if x != y]
[(1, 3), (1, 4), (2, 3), (2, 1), (2, 4), (3, 1), (3, 4)]
```

and it’s equivalent to:

```
...     for y in [3,1,4]:
...         if x != y:
...             combs.append((x, y))
...
>>> combs
[(1, 3), (1, 4), (2, 3), (2, 1), (2, 4), (3, 1), (3, 4)]
```

Note how the order of the `for` and `if` statements is the same in both these snippets.

If the expression is a tuple (e.g. the `(x, y)` in the previous example), it must be parenthesized.

```
>>> vec = [-4, -2, 0, 2, 4]
>>> # create a new list with the values doubled
>>> [x*2 for x in vec]
[-8, -4, 0, 4, 8]
>>> # filter the list to exclude negative numbers
>>> [x for x in vec if x >= 0]
[0, 2, 4]
>>> # apply a function to all the elements
>>> [abs(x) for x in vec]
[4, 2, 0, 2, 4]
>>> # call a method on each element
>>> freshfruit = ['  banana', '  loganberry ', 'passion fruit  ']
>>> [weapon.strip() for weapon in freshfruit]
['banana', 'loganberry', 'passion fruit']
>>> # create a list of 2-tuples like (number, square)
>>> [(x, x**2) for x in range(6)]
[(0, 0), (1, 1), (2, 4), (3, 9), (4, 16), (5, 25)]
>>> # the tuple must be parenthesized, otherwise an error is raised
>>> [x, x**2 for x in range(6)]
  File "<stdin>", line 1, in <module>
    [x, x**2 for x in range(6)]
               ^
SyntaxError: invalid syntax
>>> # flatten a list using a listcomp with two 'for'
>>> vec = [[1,2,3], [4,5,6], [7,8,9]]
>>> [num for elem in vec for num in elem]
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

List comprehensions can contain complex expressions and nested functions:

```
>>> from math import pi
>>> [str(round(pi, i)) for i in range(1, 6)]
['3.1', '3.14', '3.142', '3.1416', '3.14159']
```

5.1.4. Nested List Comprehensions

The initial expression in a list comprehension can be any arbitrary expression, including another list comprehension.

Consider the following example of a 3x4 matrix implemented as a list of 3 lists of length 4:

```
>>> matrix = [
...     [1, 2, 3, 4],
...     [5, 6, 7, 8],
...     [9, 10, 11, 12],
... ]
```

```
>>> [[row[i] for row in matrix] for i in range(4)]
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```

As we saw in the previous section, the nested listcomp is evaluated in the context of the `for` that follows it, so this example is equivalent to:

```
>>> transposed = []
>>> for i in range(4):
...     transposed.append([row[i] for row in matrix])
...
>>> transposed
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```

which, in turn, is the same as:

```
>>> transposed = []
>>> for i in range(4):
...     # the following 3 lines implement the nested listcomp
...     transposed_row = []
...     for row in matrix:
...         transposed_row.append(row[i])
...     transposed.append(transposed_row)
...
>>> transposed
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```

In the real world, you should prefer built-in functions to complex flow statements. The `zip()` function would do a great job for this use case:

```
>>> list(zip(*matrix))
[(1, 5, 9), (2, 6, 10), (3, 7, 11), (4, 8, 12)]
```

See Unpacking Argument Lists for details on the asterisk in this line.

5.2. The `del` statement

There is a way to remove an item from a list given its index instead of its value: the `del` statement. This differs from the `pop()` method which returns a value. The `del` statement can also be used to remove slices from a list or clear the entire list (which we did earlier by assignment of an empty list to the slice). For example:

```
>>> a = [-1, 1, 66.25, 333, 333, 1234.5]
>>> del a[0]
>>> a
[1, 66.25, 333, 333, 1234.5]
>>> del a[2:4]
>>> a
[1, 66.25, 1234.5]
>>> del a[:]
>>> a
[]
```

`del` can also be used to delete entire variables:

```
>>> del a
```

5.3. Tuples and Sequences

We saw that lists and strings have many common properties, such as indexing and slicing operations. They are two examples of *sequence* data types (see Sequence Types — list, tuple, range). Since Python is an evolving language, other sequence data types may be added. There is also another standard sequence data type: the *tuple*.

A tuple consists of a number of values separated by commas, for instance:

```
>>> t = 12345, 54321, 'hello!'
>>> t[0]
12345
>>> t
(12345, 54321, 'hello!')
>>> # Tuples may be nested:
... u = t, (1, 2, 3, 4, 5)
>>> u
((12345, 54321, 'hello!'), (1, 2, 3, 4, 5))
>>> # Tuples are immutable:
... t[0] = 88888
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
>>> # but they can contain mutable objects:
... v = ([1, 2, 3], [3, 2, 1])
>>> v
([1, 2, 3], [3, 2, 1])
```

As you see, on output tuples are always enclosed in parentheses, so that nested tuples are interpreted correctly; they may be input with or without surrounding parentheses, although often parentheses are necessary anyway (if the tuple is part of a larger expression). It is not possible to assign to the individual items of a tuple, however it is possible to create tuples which contain mutable objects, such as lists.

Though tuples may seem similar to lists, they are often used in different situations and for different purposes. Tuples are immutable, and usually contain a heterogeneous sequence of elements that are accessed via unpacking (see later in this section) or indexing (or even by attribute in the case of `namedtuples`). Lists are mutable, and their elements are usually homogeneous and are accessed by iterating over the list.

A special problem is the construction of tuples containing 0 or 1 items: the syntax has some extra quirks to accommodate these. Empty tuples are constructed by an empty pair of parentheses; a tuple with one item is constructed by following a value with a comma (it is not sufficient to enclose a single value in parentheses). Ugly, but effective. For example:

```
>>> empty = ()
>>> singleton = 'hello',    # <-- note trailing comma
>>> len(empty)
0
>>> len(singleton)
1
>>> singleton
('hello',)
```

The statement `t = 12345, 54321, 'hello!'` is an example of *tuple packing*: the values `12345`, `54321` and `'hello!'` are packed together in a tuple. The reverse operation is also possible:

This is called, appropriately enough, *sequence unpacking* and works for any sequence on the right-hand side. Sequence unpacking requires that there are as many variables on the left side of the equals sign as there are elements in the sequence. Note that multiple assignment is really just a combination of tuple packing and sequence unpacking.

5.4. Sets

Python also includes a data type for *sets*. A set is an unordered collection with no duplicate elements. Basic uses include membership testing and eliminating duplicate entries. Set objects also support mathematical operations like union, intersection, difference, and symmetric difference.

Curly braces or the `set()` function can be used to create sets. Note: to create an empty set you have to use `set()`, not `{}`; the latter creates an empty dictionary, a data structure that we discuss in the next section.

Here is a brief demonstration:

```
>>> basket = {'apple', 'orange', 'apple', 'pear', 'orange', 'banana'}
>>> print(basket)                      # show that duplicates have been removed
{'orange', 'banana', 'pear', 'apple'}
>>> 'orange' in basket                 # fast membership testing
True
>>> 'crabgrass' in basket
False

>>> # Demonstrate set operations on unique letters from two words
...
>>> a = set('abracadabra')
>>> b = set('alacazam')
>>> a                                  # unique letters in a
{'a', 'r', 'b', 'c', 'd'}
>>> a - b                              # letters in a but not in b
{'r', 'd', 'b'}
>>> a | b                              # letters in a or b or both
{'a', 'c', 'r', 'd', 'b', 'm', 'z', 'l'}
>>> a & b                              # letters in both a and b
{'a', 'c'}
>>> a ^ b                              # letters in a or b but not both
{'r', 'd', 'b', 'm', 'z', 'l'}
```

Similarly to list comprehensions, set comprehensions are also supported:

```
>>> a = {x for x in 'abracadabra' if x not in 'abc'}
>>> a
{'r', 'd'}
```

5.5. Dictionaries

Another useful data type built into Python is the *dictionary* (see Mapping Types — dict). Dictionaries are sometimes found in other languages as “associative memories” or “associative arrays”. Unlike sequences, which are indexed by a range of numbers, dictionaries are indexed by *keys*, which can be any immutable type; strings and numbers can always be keys. Tuples can be used as keys if they contain only strings, numbers, or tuples; if a tuple contains any mutable object either directly or indirectly, it cannot be used as a key. You can’t use lists as keys, since lists can be modified in place using index assignments, slice assignments, or methods like `append()` and `extend()`.

key:value pairs within the braces adds initial key:value pairs to the dictionary; this is also the way dictionaries are written on output.

The main operations on a dictionary are storing a value with some key and extracting the value given the key. It is also possible to delete a key:value pair with `del`. If you store using a key that is already in use, the old value associated with that key is forgotten. It is an error to extract a value using a non-existent key.

Performing `list(d)` on a dictionary returns a list of all the keys used in the dictionary, in insertion order (if you want it sorted, just use `sorted(d)` instead). To check whether a single key is in the dictionary, use the `in` keyword.

Here is a small example using a dictionary:

```
>>> tel = {'jack': 4098, 'sape': 4139}
>>> tel['guido'] = 4127
>>> tel
{'jack': 4098, 'sape': 4139, 'guido': 4127}
>>> tel['jack']
4098
>>> del tel['sape']
>>> tel['irv'] = 4127
>>> tel
{'jack': 4098, 'guido': 4127, 'irv': 4127}
>>> list(tel)
['jack', 'guido', 'irv']
>>> sorted(tel)
['guido', 'irv', 'jack']
>>> 'guido' in tel
True
>>> 'jack' not in tel
False
```

The `dict()` constructor builds dictionaries directly from sequences of key-value pairs:

```
>>> dict([('sape', 4139), ('guido', 4127), ('jack', 4098)])
{'sape': 4139, 'guido': 4127, 'jack': 4098}
```

In addition, dict comprehensions can be used to create dictionaries from arbitrary key and value expressions:

```
>>> {x: x**2 for x in (2, 4, 6)}
{2: 4, 4: 16, 6: 36}
```

When the keys are simple strings, it is sometimes easier to specify pairs using keyword arguments:

```
>>> dict(sape=4139, guido=4127, jack=4098)
{'sape': 4139, 'guido': 4127, 'jack': 4098}
```

5.6. Looping Techniques

When looping through dictionaries, the key and corresponding value can be retrieved at the same time using the `items()` method.

```
>>> knights = {'gallahad': 'the pure', 'robin': 'the brave'}
>>> for k, v in knights.items():
...     print(k, v)
```

```
robin the brave
```

When looping through a sequence, the position index and corresponding value can be retrieved at the same time using the `enumerate()` function.

```
>>> for i, v in enumerate(['tic', 'tac', 'toe']):
...     print(i, v)
...
0 tic
1 tac
2 toe
```

To loop over two or more sequences at the same time, the entries can be paired with the `zip()` function.

```
>>> questions = ['name', 'quest', 'favorite color']
>>> answers = ['lancelot', 'the holy grail', 'blue']
>>> for q, a in zip(questions, answers):
...     print('What is your {0}?  It is {1}.'.format(q, a))
...
What is your name?  It is lancelot.
What is your quest?  It is the holy grail.
What is your favorite color?  It is blue.
```

To loop over a sequence in reverse, first specify the sequence in a forward direction and then call the `reversed()` function.

```
>>> for i in reversed(range(1, 10, 2)):
...     print(i)
...
9
7
5
3
1
```

To loop over a sequence in sorted order, use the `sorted()` function which returns a new sorted list while leaving the source unaltered.

```
>>> basket = ['apple', 'orange', 'apple', 'pear', 'orange', 'banana']
>>> for i in sorted(basket):
...     print(i)
...
apple
apple
banana
orange
orange
pear
```

Using `set()` on a sequence eliminates duplicate elements. The use of `sorted()` in combination with `set()` over a sequence is an idiomatic way to loop over unique elements of the sequence in sorted order.

```
>>> basket = ['apple', 'orange', 'apple', 'pear', 'orange', 'banana']
>>> for f in sorted(set(basket)):
...     print(f)
...
```

```
orange
pear
```

It is sometimes tempting to change a list while you are looping over it; however, it is often simpler and safer to create a new list instead.

```
>>> import math
>>> raw_data = [56.2, float('NaN'), 51.7, 55.3, 52.5, float('NaN'), 47.8]
>>> filtered_data = []
>>> for value in raw_data:
...     if not math.isnan(value):
...         filtered_data.append(value)
...
>>> filtered_data
[56.2, 51.7, 55.3, 52.5, 47.8]
```

5.7. More on Conditions

The conditions used in `while` and `if` statements can contain any operators, not just comparisons.

The comparison operators `in` and `not in` are membership tests that determine whether a value is in (or not in) a container. The operators `is` and `is not` compare whether two objects are really the same object. All comparison operators have the same priority, which is lower than that of all numerical operators.

Comparisons can be chained. For example, `a < b == c` tests whether `a` is less than `b` and moreover `b` equals `c`.

Comparisons may be combined using the Boolean operators `and` and `or`, and the outcome of a comparison (or of any other Boolean expression) may be negated with `not`. These have lower priorities than comparison operators; between them, `not` has the highest priority and `or` the lowest, so that `A and not B or C` is equivalent to `(A and (not B)) or C`. As always, parentheses can be used to express the desired composition.

The Boolean operators `and` and `or` are so-called *short-circuit* operators: their arguments are evaluated from left to right, and evaluation stops as soon as the outcome is determined. For example, if `A` and `C` are true but `B` is false, `A and B and C` does not evaluate the expression `C`. When used as a general value and not as a Boolean, the return value of a short-circuit operator is the last evaluated argument.

It is possible to assign the result of a comparison or other Boolean expression to a variable. For example,

```
>>> string1, string2, string3 = '', 'Trondheim', 'Hammer Dance'
>>> non_null = string1 or string2 or string3
>>> non_null
'Trondheim'
```

Note that in Python, unlike C, assignment inside expressions must be done explicitly with the walrus operator `:=`. This avoids a common class of problems encountered in C programs: typing `=` in an expression when `==` was intended.

5.8. Comparing Sequences and Other Types

Sequence objects typically may be compared to other objects with the same sequence type. The comparison uses *lexicographical* ordering: first the first two items are compared, and if they differ this determines the outcome of the comparison; if they are equal, the next two items are compared, and so on, until either

sequences are considered equal. If one sequence is an initial sub-sequence of the other, the shorter sequence is the smaller (lesser) one. Lexicographical ordering for strings uses the Unicode code point number to order individual characters. Some examples of comparisons between sequences of the same type:

```
(1, 2, 3)              < (1, 2, 4)
[1, 2, 3]              < [1, 2, 4]
'ABC' < 'C' < 'Pascal' < 'Python'
(1, 2, 3, 4)           < (1, 2, 4)
(1, 2)                 < (1, 2, -1)
(1, 2, 3)             == (1.0, 2.0, 3.0)
(1, 2, ('aa', 'ab'))   < (1, 2, ('abc', 'a'), 4)
```

Note that comparing objects of different types with < or > is legal provided that the objects have appropriate comparison methods. For example, mixed numeric types are compared according to their numeric value, so 0 equals 0.0, etc. Otherwise, rather than providing an arbitrary ordering, the interpreter will raise a `TypeError` exception.

Footnotes

[1] Other languages may return the mutated object, which allows method chaining, such as `d->insert("a")->remove("b")->sort();`.

queue — A synchronized queue class

Source code: Lib/queue.py

The queue module implements multi-producer, multi-consumer queues. It is especially useful in threaded programming when information must be exchanged safely between multiple threads. The Queue class in this module implements all the required locking semantics.

The module implements three types of queue, which differ only in the order in which the entries are retrieved. In a FIFO queue, the first tasks added are the first retrieved. In a LIFO queue, the most recently added entry is the first retrieved (operating like a stack). With a priority queue, the entries are kept sorted (using the heapq module) and the lowest valued entry is retrieved first.

Internally, those three types of queues use locks to temporarily block competing threads; however, they are not designed to handle reentrancy within a thread.

In addition, the module implements a “simple” FIFO queue type, SimpleQueue, whose specific implementation provides additional guarantees in exchange for the smaller functionality.

The queue module defines the following classes and exceptions:

class queue.**Queue**(*maxsize=0*)

Constructor for a FIFO queue. *maxsize* is an integer that sets the upperbound limit on the number of items that can be placed in the queue. Insertion will block once this size has been reached, until queue items are consumed. If *maxsize* is less than or equal to zero, the queue size is infinite.

class queue.**LifoQueue**(*maxsize=0*)

Constructor for a LIFO queue. *maxsize* is an integer that sets the upperbound limit on the number of items that can be placed in the queue. Insertion will block once this size has been reached, until queue items are consumed. If *maxsize* is less than or equal to zero, the queue size is infinite.

class queue.**PriorityQueue**(*maxsize=0*)

Constructor for a priority queue. *maxsize* is an integer that sets the upperbound limit on the number of items that can be placed in the queue. Insertion will block once this size has been reached, until queue items are consumed. If *maxsize* is less than or equal to zero, the queue size is infinite.

The lowest valued entries are retrieved first (the lowest valued entry is the one returned by `sorted(list(entries))[0]`). A typical pattern for entries is a tuple in the form: `(priority_number, data)`.

If the *data* elements are not comparable, the data can be wrapped in a class that ignores the data item and only compares the priority number:

```
from dataclasses import dataclass, field
from typing import Any

@dataclass(order=True)
class PrioritizedItem:
    priority: int
    item: Any=field(compare=False)
```

tracking.

New in version 3.7.

exception queue.**Empty**

Exception raised when non-blocking `get()` (or `get_nowait()`) is called on a `Queue` object which is empty.

exception queue.**Full**

Exception raised when non-blocking `put()` (or `put_nowait()`) is called on a `Queue` object which is full.

Queue Objects

Queue objects (`Queue`, `LifoQueue`, or `PriorityQueue`) provide the public methods described below.

`Queue`.**qsize**()

Return the approximate size of the queue. Note, qsize() > 0 doesn't guarantee that a subsequent get() will not block, nor will qsize() < maxsize guarantee that put() will not block.

`Queue`.**empty**()

Return `True` if the queue is empty, `False` otherwise. If empty() returns `True` it doesn't guarantee that a subsequent call to put() will not block. Similarly, if empty() returns `False` it doesn't guarantee that a subsequent call to get() will not block.

`Queue`.**full**()

Return `True` if the queue is full, `False` otherwise. If full() returns `True` it doesn't guarantee that a subsequent call to get() will not block. Similarly, if full() returns `False` it doesn't guarantee that a subsequent call to put() will not block.

`Queue`.**put**(*item*, *block=True*, *timeout=None*)

Put *item* into the queue. If optional args *block* is true and *timeout* is `None` (the default), block if necessary until a free slot is available. If *timeout* is a positive number, it blocks at most *timeout* seconds and raises the `Full` exception if no free slot was available within that time. Otherwise (*block* is false), put an item on the queue if a free slot is immediately available, else raise the `Full` exception (*timeout* is ignored in that case).

`Queue`.**put_nowait**(*item*)

Equivalent to `put(item, False)`.

`Queue`.**get**(*block=True*, *timeout=None*)

Remove and return an item from the queue. If optional args *block* is true and *timeout* is `None` (the default), block if necessary until an item is available. If *timeout* is a positive number, it blocks at most *timeout* seconds and raises the `Empty` exception if no item was available within that time. Otherwise (*block* is false), return an item if one is immediately available, else raise the `Empty` exception (*timeout* is ignored in that case).

Prior to 3.0 on POSIX systems, and for all versions on Windows, if *block* is true and *timeout* is `None`, this operation goes into an uninterruptible wait on an underlying lock. This means that no exceptions can occur, and in particular a SIGINT will not trigger a `KeyboardInterrupt`.

Two methods are offered to support tracking whether enqueued tasks have been fully processed by daemon consumer threads.

Queue.**task_done**()

Indicate that a formerly enqueued task is complete. Used by queue consumer threads. For each `get()` used to fetch a task, a subsequent call to `task_done()` tells the queue that the processing on the task is complete.

If a `join()` is currently blocking, it will resume when all items have been processed (meaning that a `task_done()` call was received for every item that had been `put()` into the queue).

Raises a `ValueError` if called more times than there were items placed in the queue.

Queue.**join**()

Blocks until all items in the queue have been gotten and processed.

The count of unfinished tasks goes up whenever an item is added to the queue. The count goes down whenever a consumer thread calls `task_done()` to indicate that the item was retrieved and all work on it is complete. When the count of unfinished tasks drops to zero, `join()` unblocks.

Example of how to wait for enqueued tasks to be completed:

```
import threading, queue

q = queue.Queue()

def worker():
    while True:
        item = q.get()
        print(f'Working on {item}')
        print(f'Finished {item}')
        q.task_done()

# Turn-on the worker thread.
threading.Thread(target=worker, daemon=True).start()

# Send thirty task requests to the worker.
for item in range(30):
    q.put(item)

# Block until all tasks are done.
q.join()
print('All work completed')
```

SimpleQueue Objects

`SimpleQueue` objects provide the public methods described below.

SimpleQueue.**qsize**()

Return the approximate size of the queue. Note, qsize() > 0 doesn't guarantee that a subsequent get() will not block.

SimpleQueue.**empty**()

SimpleQueue.**put**(*item*, *block=True*, *timeout=None*)

Put *item* into the queue. The method never blocks and always succeeds (except for potential low-level errors such as failure to allocate memory). The optional args *block* and *timeout* are ignored and only provided for compatibility with `Queue.put()`.

CPython implementation detail: This method has a C implementation which is reentrant. That is, a `put()` or `get()` call can be interrupted by another `put()` call in the same thread without deadlocking or corrupting internal state inside the queue. This makes it appropriate for use in destructors such as `__del__` methods or `weakref` callbacks.

SimpleQueue.**put_nowait**(*item*)

Equivalent to `put(item)`, provided for compatibility with `Queue.put_nowait()`.

SimpleQueue.**get**(*block=True*, *timeout=None*)

Remove and return an item from the queue. If optional args *block* is true and *timeout* is `None` (the default), block if necessary until an item is available. If *timeout* is a positive number, it blocks at most *timeout* seconds and raises the `Empty` exception if no item was available within that time. Otherwise (*block* is false), return an item if one is immediately available, else raise the `Empty` exception (*timeout* is ignored in that case).

SimpleQueue.**get_nowait**()

Equivalent to `get(False)`.

See also:

Class `multiprocessing.Queue`

A queue class for use in a multi-processing (rather than multi-threading) context.

`collections.deque` is an alternative implementation of unbounded queues with fast atomic `append()` and `popleft()` operations that do not require locking and also support indexing.