**Strings**

1.Slicing [start:stop:howmanysteps]

2. New line si tab : \n sau \t

3. Slicing pentru tot cuvantul variable[::]

4. Slicing si sarituri peste index [::2] – aici itis are cu cate 2 pozitii

5. Inversarea stringului – [::-1] , adica start, stop si cu cate un -1

6. Eroare: str object does not support item assignment – ptc strings are immutable, nu accepta sa le schimbi valoarea facand reassignment pe un index cu un alt caracter decat cel stability initial.

7. Concat = “p” + variable name (after applying a slicer to remove the initial letter, in case you would want to replace “Sam” with “Pam”)

8. Insert a string in a string cu **.format:** print(“this is a string {}”.format(“something”)) – **string interpolation**

Poti pune mai multe strings si sa mentionezi index position: print(“this is a string {1}{0}{2}”.format(“something”,”to”,”do”))

9. Float formatting {value:width.precision f}

Result = 100/777

Print(“the result was {r:1.3f}”.format(r=result))

10. f format :

Name = “jose”

Print(f”Hello, his name is {name}”)

**Lists**

**Dictionaries**

1.Retrieve all keys:

D={“key1” =1, “key2” = 2}

D.keys()

2. Retrieve all values

d.values()

3.d.items() – all the pairs

4.Unpacking:

For key,value in myvariab.items():

Print(value) or if you want print(key)

**Tuples**

Similar to lists, but they are immutable, cannot reassign values

3.Unpacking for x,y in mytuple:

Print(a)

Print(b)

Iti da toate valorile unui tuplu

**Sets**

Unordered collections of unique elements – no duplicates

1. How to create: myvariable = set()

2.Adaugare: variablename.add(2)

**Working with files**

1.Creezi un nou notebook file : %%writefile myfile.text

This **is** the first line

2. Cum deschizi un fisier : myfile = open(“myfile.text”)

3. Cum afli in ce path e fisierul? : pwd

4. Cum afisezi continutul in python: myfile.read()

Nota: daca dupa ce apare textul ii mai dam odata .read, o sa dea none, pentru ca este cursorul la final de pagina

Workaround: myfile.seek(0) – ca sa iti puna cursorul la inceput, iar apoi putem da din nou read.

5. Printare rand cu rand: myfile.readlines() – din nou treb facut seek la 0 daca e cazul

6. Deschiderea unui fisier dintr-o anumita locatie : myfile = open(“C:\\Users\\etc”)

7. Inchidere myfile.close()

8. ca sa nu mai am treaba cu close file, sa nu am erori:

With open(“myfilename.txt”) as my\_new\_file:

Contents = my\_new\_file.read()

Print(cotents)

9.Write: with open(“myfile.txt”, mode = “w”) as myfile:

Contents = myfile.read()

R+ means reading and writing , w+ means writing and reading

**For and while loops**

1.Range(start,stop,steps)

2. Enumerate – functie ca sa it idea un tuple cu perechi, de ex 0, 2 – adica 2 ul care se afla la index position 0

2. Zip – zips together 2 lists

3. In – x in [“x”,”y”,”z”] – output true

4. iterate and check the previous element for r in list, [r-1]

5. Pointer la finalul listei len(lista) -1

6. Loop backwards – pui un pointer care sa trimita la ultimul element din lista cu len(lista) -1 si decrementezi

**List comprehension**

Te ajuta in situatiile in care vrei sa faci o lista cu mai multe elemente si sa eviti sa scrii maim ult cod cu for loop si crearea unei listen oi

Exemplu:

Mylist = “abcd”

Result = [letter for letter in Mylist]

Se pot folosi si formule matematice si else if si nisted for loops

Result = [x if x%2 ==0 else “Odd” for x in range(0,11)]

**Methods and documentation**

Ca sa descoperi toate methods posibile, pui variaila. Si apoi apesi Tab. Ca sa primesti definitia, pui numele metodei si apoi shift tab.

Mai merge si cu help(mylist.insert) – si imi da definitia.

Sau accesam pe internet docs.python.org

**Functions**

Ele sunt facute sa impachetam un cod intr-o functie, ca sa evitam sa il rescriem acelasi cod de 70 de ori.

Ca sad ai call unei functii:

Name\_of\_function(“argument”) – daca functia continue un printing statement, nu mai e nevoie sa zici din nou print la capat, dar daca nu are, trebuie.

**Return** vs **Print** – print doar inchide functia si iti printeaza rezultatul, dar return iti da voie sa salvezi rezultatul ca variabila, poate fi pus in variabila.

**Return si For loop**:

Se da urmatorul cod:

Def check\_even(num\_list):

For num in num\_list:

If num % 2 == 0:

Return True

Else:

Return False

Aici e gresit pentru ca daca un numar nu e true, o sa verifice imediat a doua conditie, iar daca numarul even este la final, o sa dea tot fals, desi exista un numar par in lista.

Return false se pune in linie cu for loopul, pentru ca trebuie sa se verifice abia la terminarea for loopului.

**Function interaction**

Te lasa sa iei variabile deja setate in alte functii, deci ai voie sa faci asta

**\*args and \*\*kwargs –**

**args** se pune intr-o functie ca sa stabilesti o infinitate de argumente de pus intr-o functie, ptc daca pui argumentele separate si pui 3, iar userul baga 4, da eroare. – scopul lui e sa evite erori. Args da tuplu, ca sa fie date de nemodificat.

Un alt lucru de luat in seama e c cuv args se poate modifica, de fapt Steaua \* e notatia obligatory, deci args poate fi si clatita, daca vrei. Iar acest args se poate folosi ca variabila, merge cu un loop de ex,

For item in args 🡪 print (item) si deci args e tratat ca un fel de lista de valori.

**\*\*kwargs** – permite tot asa mai multe argumente, la alegerea userului, dar ca return ofera dictionar.

Se permite si folosirea lor simultana, dar e obligatoriu sa urmezi ordinea oferita, ori kwargs, ori args, este obligatory ordinea.

**Lambda map and flter**

Lambda = functie anonima pe care o folosesti doar odata si dupa pa.

**Map** – te ajuta cand vrei sa verifici mai multe obiecte dintr-o lista pe baza unei functii:

Def square(nums):

Return Num\*\*2

My\_nums = [2,3,4,5]

For item in MAP(square, my nums)

Print(item)

Te ajuta ca sa nu folosesti un simplu for loop care ar costa mult cod

In interiorul parantezei nu e nevoie sa punem () la numele functiei, ptc map executa singur functia, altfel iti da eroare.

**Filter** te ajuta sa verifici valoarea de adevar a unor elemente, ea iti da true sau false.

La fel ca map ,verifica o functie, exemplu:

Def check\_even(num):

Return num%2 ==0

Mynums = [1,2,3,4,5,6]

List(map(check\_even, mynums))

**Lambda** – este o functie destinate folosirii singulare, e mai aplicabila la functii scurte si rapide de tipul n \*\* sau calcule rapide. Poate fi folosita cu map and filter

Lambda num : num\*\*2 – si e exact ca functia de ridicare la patrat.

List(filter(lambda num:num%2 == 0, mynums))

**Nested statements and scope**

Variabilele sunt salvate intr-un namespace, iar variabilele au si un scope (local sau global), si in functie de acst scope se determina vizibilitatea variabilei in celelalte parti ale codului.

**Object oriented programming**

Functiile cu def create dupa metoda init, au self in paranteza ca sa arate ca sunt conectate la clasa.

Instance of class = adica sa salvezi clasa cu paranteze, intr-o variabila.

Init – self – self reprezinta instance of the object.

Class object attribute – este o variabila care poate fi pusa intre class si init, si asta e valabila pt toate instantele clasei.

Atributele nu au paranteza dupa, dar metodele au, deci ca sad ai call metodei, trebuie sa pui paranteza

Lungime sau string dintr-o clasa- se face cu metodele \_\_len\_\_(self) sau \_\_str\_\_(self).

Stergerea unui obiect cu del variable name – variabila unde salvezi clasa

\_\_del\_\_(self) – ca sa arati un mesaj cand stergi ceva

**Return self** = The idea is that each method returns an object, which allows the calls to be chained together in a single statement without storing the results in intermediary variables.

Example:

class Calc():

def \_\_init\_\_(self, number=0):

self.number = number

def add(self, value):

self.number = self.number + value

return self

def subtract(self, value):

self.number = self.number - value

return self

calc = Calc()

calc.add(5).subtract(2).add(5)

print(calc.number) *# 👉️ 8*

\_\_name\_\_ == \_\_main\_\_ - asta iti verifica daca faci import sau daca rulezi codul direct de pe fisierul .py current., e pentru code organization

**Exception handling**

Try, except si finally, finally continua sa execute codul mai departe, chiar daca a aparut o eroare.

**Decorators** – te ajuta sa modifici functiile create ca sa nu stai sa recopiezi codul, sit e ajuta si sa stergi chestii

@new\_decorator si abia apoi pui functia

**Generator** – o metoda memory efficient de a lua numere multiple, adica in loc sa le salvezi intr-o lista, le scoti direct din functie pe masur ace ai nevoie de ele.. Se face cu **Yield** keyword.

Poti printa pe rand valorile cu print(**next**(variabila\_in\_care\_am\_salvat\_functia))

**Advanced Data Structures**

**Linked list**

O lista care are un head node si un pointer next care duce la urmatorul nod.

Intr-un array obisnuit, daca vrei sa bagi un element, trebuie sa muti memory location pentru celelalte elemente, cee ace te costa timp in plus.

La linked list, doar schimbi linkul catre adresa de memorie a noului element, deci nu mai trebuie sa le muti pe celelalte in alta adresa.

Insertia la inceput e o1, dar la final e On, pentru ca trebuie sa treci prin toate nodurile.

Daca intr-o problema ti se da head ul, ca lista, nu mai trebuie sa creez eu un head, doar pot isa faci loop prin el.

Ca sa treci la urm nod trebuie updatat pointerul la urm pointer, egalandu-l cu variabila/obiectul next.

**Binary search tree**

Alternativa buna pentru Sets

In order traversal – root node is between the left and right

Preorder you start with root 🡪 left 🡪right

Postorder start with left 🡪 right🡪 root

Strings au valoare dupa ordine alfabetica, de ex SUA > Moldova ptc S vine dupa M

**Heap**

Binary tree compus din numere unde se face prin ordine crescatoare, adica head node e val cea mai mica si evolueaza pana la cea mai mare

**Trie**

Copac bun pentru searching care iti cauta litere si cuvinte

Faci un nod el are children de tip dictionar

Mai se initializeaza si un root node

**Graph**

Este non linear si are nodes/vertices si edges, edges fac conexiunea dintre nodes, poate sa reprezinte valoarea distantei de la un nod la altul.

Diferenta fata de tree este ca poate conecta pe mai multe rute un nod la altul

**Stack**

First in – last out – ca un pachet de carti

Dequeue – from collections import deque

Stack = deque()

Dir(myvariable) – asta te ajuta sa vezi toate metodele posibile care exista intr-un object, cum e asta deque imprumutat din collections

In deque sunt mai multe functii care te ajuta cu un stack, mai rapide decat daca ai folosi listele de baza

**Que**

First in – first out

Se scot metodele tot cu pop, dar se introduce la index 0 mereu elementele, cu metoda insert

queue = []  
  
queue.insert(0, 5)  
queue.insert(0, 6)  
queue.insert(0, 7)  
  
queue.pop()  
  
print(queue)

Folosim si aici metodele din deque, dar cu append left

**ALGORITHMS**

1. **Binary search**

All values must be sorted

Trebuie sa ai 2 puncte de referinta, unul care e lowest sau in left si unul highest sau in right.

Mai trebuie un mid point , asta se ia prin pozitia cea mai mare + pozitia cea mai mica impartita la 2

Faci verificarea, daca cee ace cauti nu e mid, trebuie schimbat mid value. Daca valoarea pe care o caut e maim are ca midul, noup low bound o sa fie midul actual, air vice versa, vice versa

Algoritmul asta te ajuta sa cauti mai repede, fast search

2. **Breadth first search algorithm**

Ajuta sa gasesti calea cea mai simpla dintr-un punct in altul

E aplicabila la graf , breadth ptc se refera la largime, intai cauta orizontal, nu vertical

Iti trebuie un q ca sa ai node tracking, sa vezi in ce nod esti.

Se tine cont de nodurile vizitate deja si cele care mai trebuie vizitate ( astea care mai treb vizitate se pun in queue). Dup ace parasim un punct, il scoatem din q si il marcam ca visited, deoarece inainte sa intram in punct toate nodurile sunt deja in coada

3. **Depth-first search**

Fix inversul lui breadth, intai vine verificarea verticala, si abia apoi cea orizontala, depth se refera la adancime

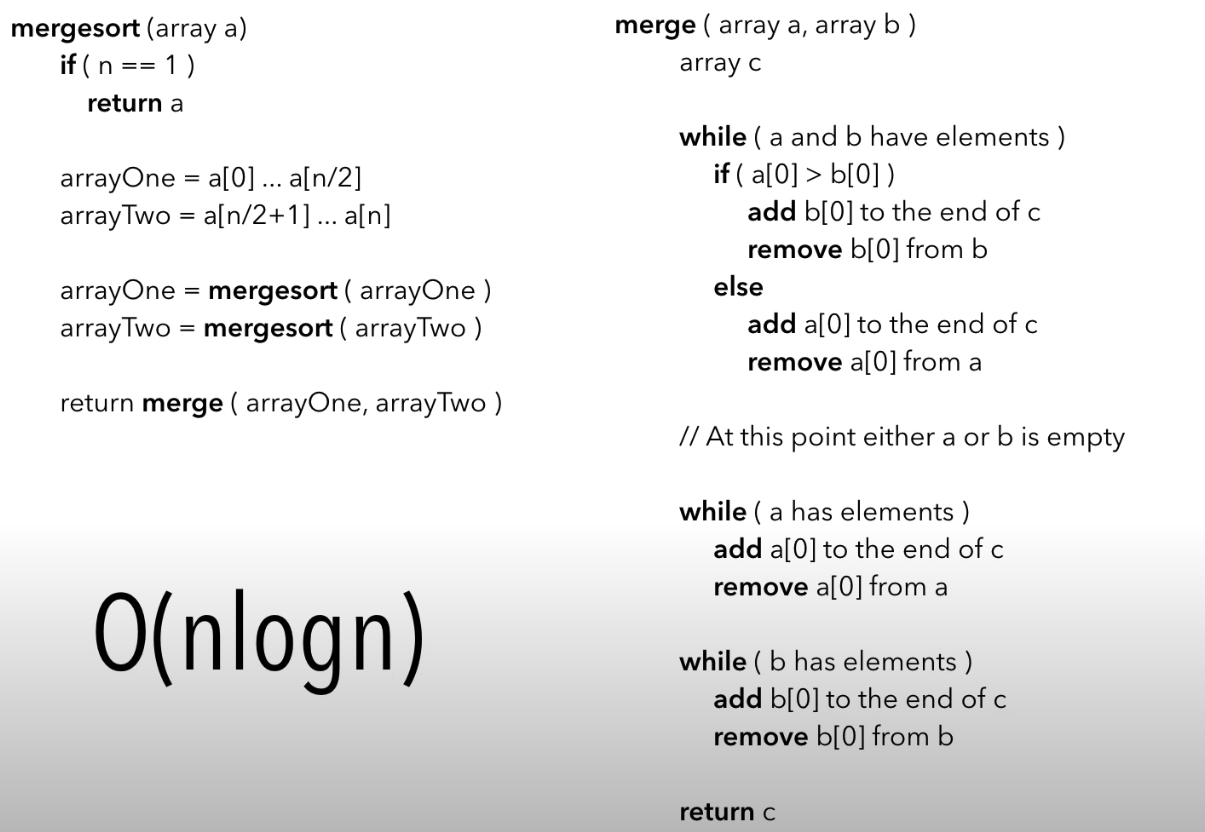
Aici se foloseste stack, nu queue si aici se foloseste deque

4. **Merge sort**

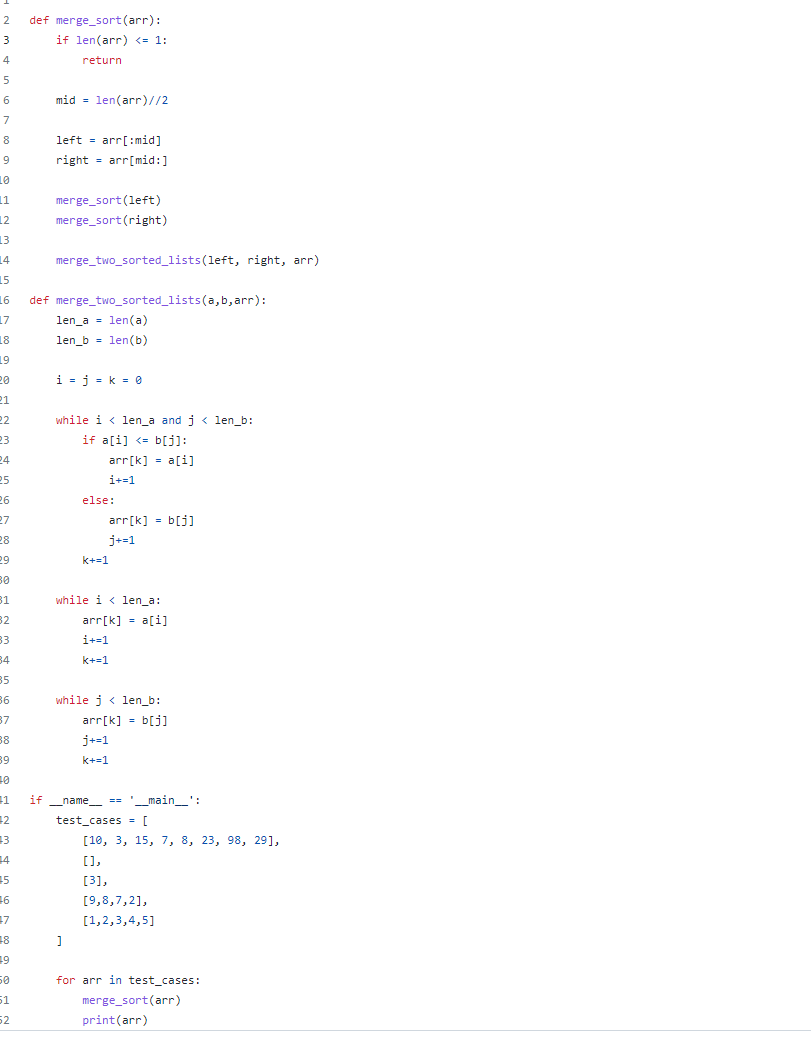
Se face recursive. Te ajuta sa sortezi si sa impreunezi liste

Se imparte lista in jumatati pana cand ramanem cu elementele singular care formeaza arrays

Apoi se examineaza valorile si se combina in temporary arrays cate 2. Apoi treptat se refac listele si se face inserarea in ordinea corecta



E un algoritm rapid, dar mananca spatiu

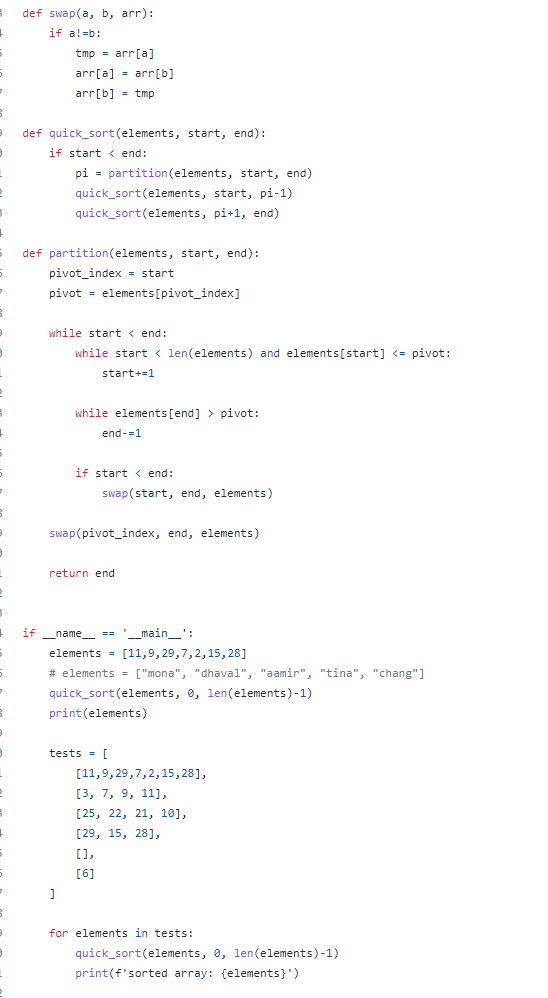


5. **Quick sort**

Pivot – elementele din stanga lui sunt mai mici si alea din dreapta mai mari, are o pozitie corecta in array ul sortat

Se allege un pivot si se muta la finalul array ului

Se allege un element primul din stanga care e mai mare ca pivotal si unul primul din dreapta care e mai mic decat pivotal. Se inverseaza item from left cu asta din right si continuam sa repetam procesul, ne oprim cand item from left e mai mare ca item from right, apoi inversam item from left cu pivotal, adica ala care e mai mare



6. **Kruskal’s algorithm** – algoritm de cautarea a celor mai mici conexiuni dintr-un graf si crearea unui spanning tree – sa nu se faca circles, adica laturile sa nu fie formate din puncta conectate intre ele

7. Floys-warshall algorithm – algoritm care te ajuta sa gasesti cea mai mica distanta intre perechile de noduri din graf

8. **Dijkstras algorithm** – iti arata ea mai mica dinstanta de la un noc catre toate celelalte noduri din graf

Treci prin fiecare nod luan in considerare cea mai mica valoare mereu, cea mai mica distanta

9. **Bellman-Ford** – tot pentru graf, da-l dracu momentan, dar faci node number -1 iteratii, verifica toate nodurile si se uita la marginile fiecaruia si constant cauta directii mei bune catre alte noduri

10. **Kadane’s algorithm** – pentru maximum subarray care este un array dintr-un array, chiar si intregul array se pune ca subarray, ba da-ma dracu, si e format din contiguous elements, nu pe sarite. Merge sa faci si prin brute force, verificand fiecare subarray, dar e mai lent decat Kadane

**def** max\_subarray(numbers):

*"""Find the largest sum of any contiguous subarray."""*

best\_sum = 0

current\_sum = 0

**for** x **in** numbers:

current\_sum = max(0, current\_sum + x)

best\_sum = max(best\_sum, current\_sum)

**return** best\_sum

11. **Flood fill algorithm** porneste dintr-un pixel si continua si coloreze cu aceeasi culoare cate alti 4 pixeli pana cand ajunge la un border cu o culoare diferita , e combinatie intre dfs sib fs

12. **Floyd cycle detection algorithm** sau numit si hare and tortoise – ai 2 algortimi, unul se misca mai repede ca broasca testoasa.

E de regula utilizata in linked lists unde pointerul rapid traverseaza lista si daca da de null, insm ca e capat de lista si nu e loop, dar daca se intalneste iar cu pointerul mai lent, insm ca exista un loop

13. **Union find algorithm**

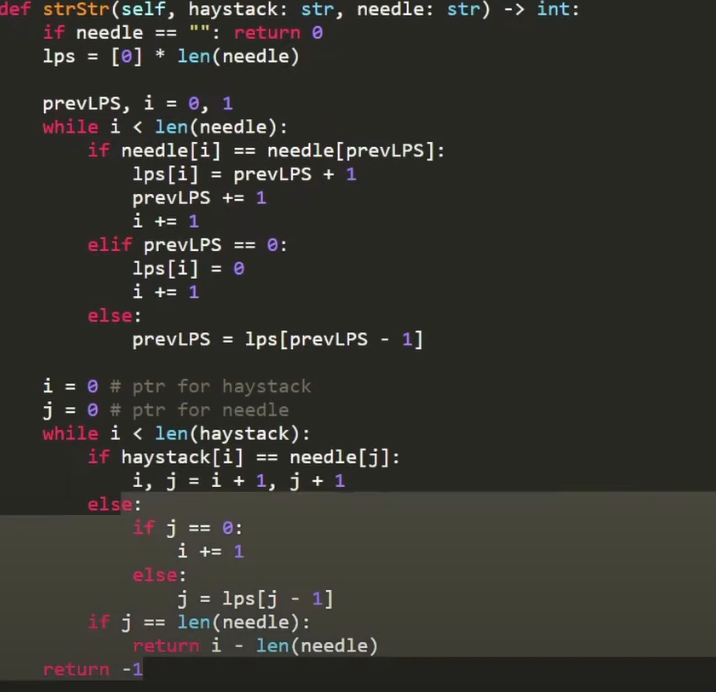
Ajuta in probleme cu graph, imbina intr-un grup elemental din alte subgrupe, te ajuta sa detectezi cycle intr-un graf si sa unesti totul intr-o singura familie

14. **Topological sorting**

Trecem prin nodurile unui graf si marcam ca visited. Daca nodul asta are drum spre altul, trecem la altul, daca nu, nodul care nu are drum spre alt nod, trebuie pus intr-un stack si ne intarcem la cel visited anterior si apoi continuam cu ce mai ramane unvisited si se pun in stack toate

15. **KMP ALGORITHM**

Algoritm de pattern searching, pe strings. Iti da numarul indexului de la care incepe patternul sa fie valid



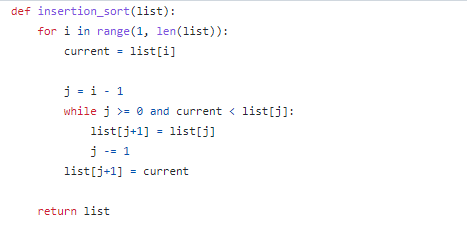
Iti trebuie un LPS ca array care va avea lungimea patternului dat care trebuie gasit intr-un text

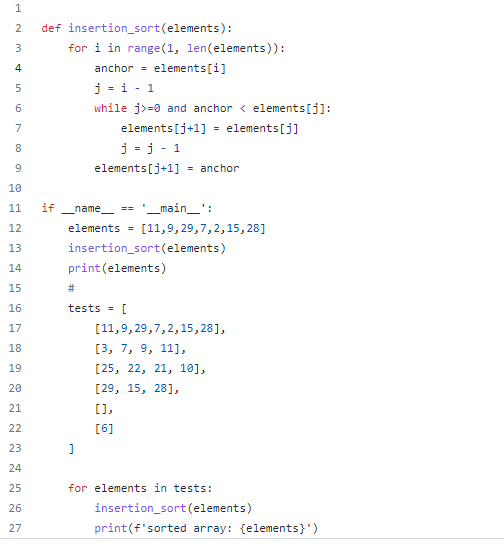
In LPS se pune indexul ori de cate ori se face un match intre litere, daca se gaseste mis match, se va decrementa pozitia indexului din LPS. Stim ca am gasit un match doar atunci cand ajungem la end of LPS, ptc daca nu se face match intre caractere, scadem din pointerul de pe LPS

16. **Insertion sort**

Ideea e sa se sorteze obiectele in ordine crescatoare, se incepe de la stanga la dreapta, se verifica fiecare obiect si se compara cu ce are la stanga. Incepem de la primul din stanga, si il marcam cu ceva ca fiind sorted, apoi trecem la urm si daca e mai mare, il lasam asa si va fi marcat tot ca sorted, daca urm e mai mic, se schimba cu el si se pune la stanga.

Asta e sortare in place:

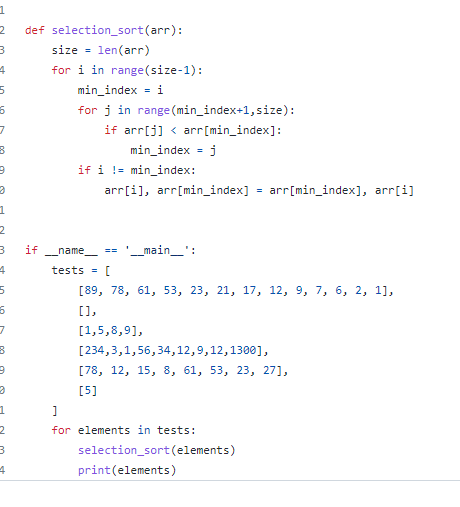




17. **Selection sort**

La fiecare iteratie se allege cel mai mic numar din aprtea unsorted si se muta in partea sorted

Se pune un pointer la inceputul listei, si cu celalalt pointer te mut iin lista pana dai de un numar mai mic. Il muti in partea sorted si faci schimbul de pozitie.



18. **Counting sort**

O porcarie

19. **Heap sort**

Se construieste un heap tree ca sa sortezi numerele, se efface un max heap. Si se inlocuieste nodul cel mai mare cu cel mai mic si se scoate

20. **Quickselect algorithm**

Se cauta cel mai mic element si tot ce e mai mic decat el se pune in stanga si tot ce e mai mare, in dreapta

21. **Boyer moore** – cauta elemental majoritar in constant time and space

Elmentul majoritar este considerat ala care apare in lista de maim ult de lungimea listei impartite la 2 ori.

Se face o variabila unde salvam posibilul candidat si alta variabila unde numaram de cate ori apare candidatul

Se trece prin elementele listei si se sccade countul daca urm element nu ii este egal si daca scade si count pana la 0, deci ai grija ca candidatul se schimba doar daca count devine 0, facem update candidatului cu indexul curent si count ii creste cu 1

22. In-place algorithms – loops si pointers

23. **Bubble sort**

[**https://github.com/codebasics/data-structures-algorithms-python/blob/master/algorithms/2\_BubbleSort/bubble\_sort.py**](https://github.com/codebasics/data-structures-algorithms-python/blob/master/algorithms/2_BubbleSort/bubble_sort.py)

Exercise:

<https://github.com/codebasics/data-structures-algorithms-python/blob/master/algorithms/2_BubbleSort/bubble_sort_exercise.md>

Se iau perechi de numere, 1 cu 2, apoi 2 cu 3 etc, se compara si se aranjeaza cel mai mic si apoi cel mai mare. Scopul e ca numarul cel mai mare din lista sa iasa la suprafata

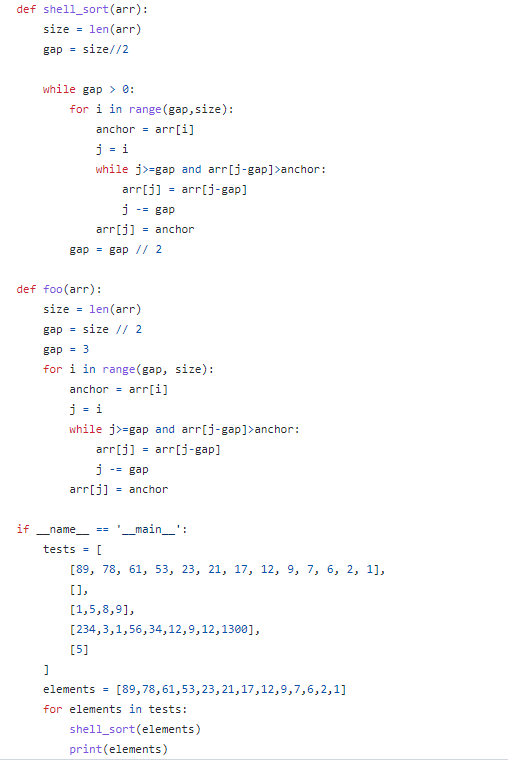
24. **Shell sort**

Cauta sa resolve problema de la Insertion short unde daac sunt mai multe elemente mici in dreapta trebuie sa faci prea multe comparatii si swaps.

Shell foloseste un Gap si auta sa mute cele mai mari elemente in Dreapta. Gap sunt limite care stabilesc subarrays

Gap se tot reduce si cand ajunge la 1, deja lista e pregatita pentru insertion short, cand ai elementele mari in dreapta, iar cele mai mici in stanga.

Gap este egal cu len(list) // 2, gap se tot reduce si in acelasi timp se si sorteaza elementele, iar ultima iteratie este insertion sort



**Methods**

Zip() – aduna toate cuvintele unui string, care pot fi iterate, si le pune intr-un tuple, cu keyword (\*numele listei) iti aduna fiecare litera

**Time complexity**:

1. Constant – indifferent de cat de mare e lista, rezultatul e acelasi in cazuri in care scoatem un element specific gasit imediat

2. Linear – cu cat creste lista, cu atat creste si timpul – de obicei la for loops care treb sa verifice fiecare element

3. Logaritmic – timpul se reduce exponential, opusul ridicarii la putere- in algoritmi de tip binary search sau merge sort etc, unde scapi dintr-o data de jumatate din lista ca sa gasesti mai repede ceva

**Binary conversion**

**Decimal to binary**

1. imparti numarul la 2 si rezultatul continuu, pana ajungi la 0 iar rezultatele daca au reminder, adica rest, pui 1, daca nu, pui 0 si le numari de la capat. La capat este the most significant bit, si in top este the least significant.

2. ridici la puterea 2 numerele pana nu depaseste numarul pe care il ai. Apoi scazi din fiecare rezultat de la cel mai mare la cel mai mic si rezultatele le pastrezi intr-o parte. Scazi cu numerele rezultate din ridicarea la puterea 2, alea care sunt mai mici decat numarul curent.

**Binary to decimal**



**Int(x,2) – binary to decimal**

**Probleme de rezolvat**

**Linked List**

1. Insert method

2.Print

3. Insert at the beggining

4. Insert values from a list

5. Return the length of the Linked List

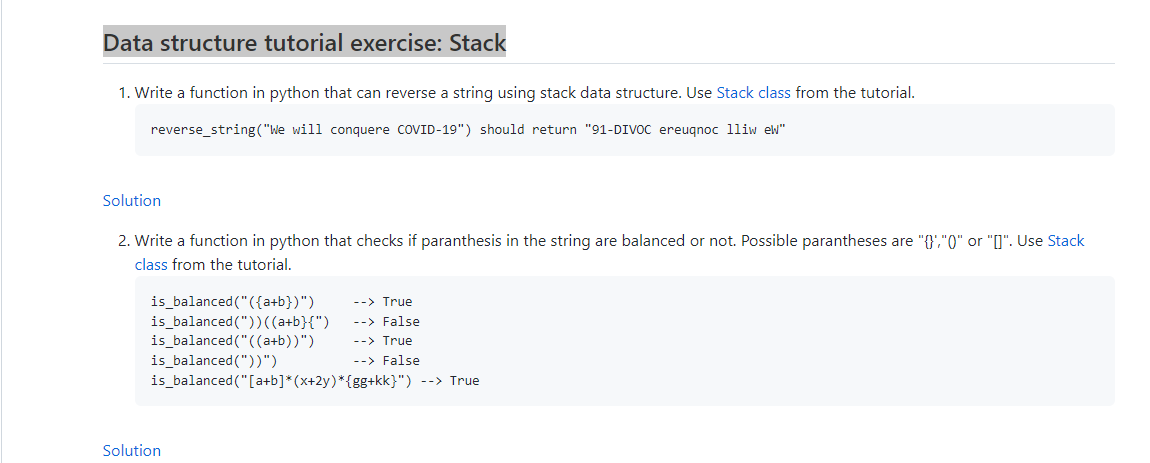
6.Remove an element at a given index

7. Insert a specific value at a certain index

8. Implement doubly linked list

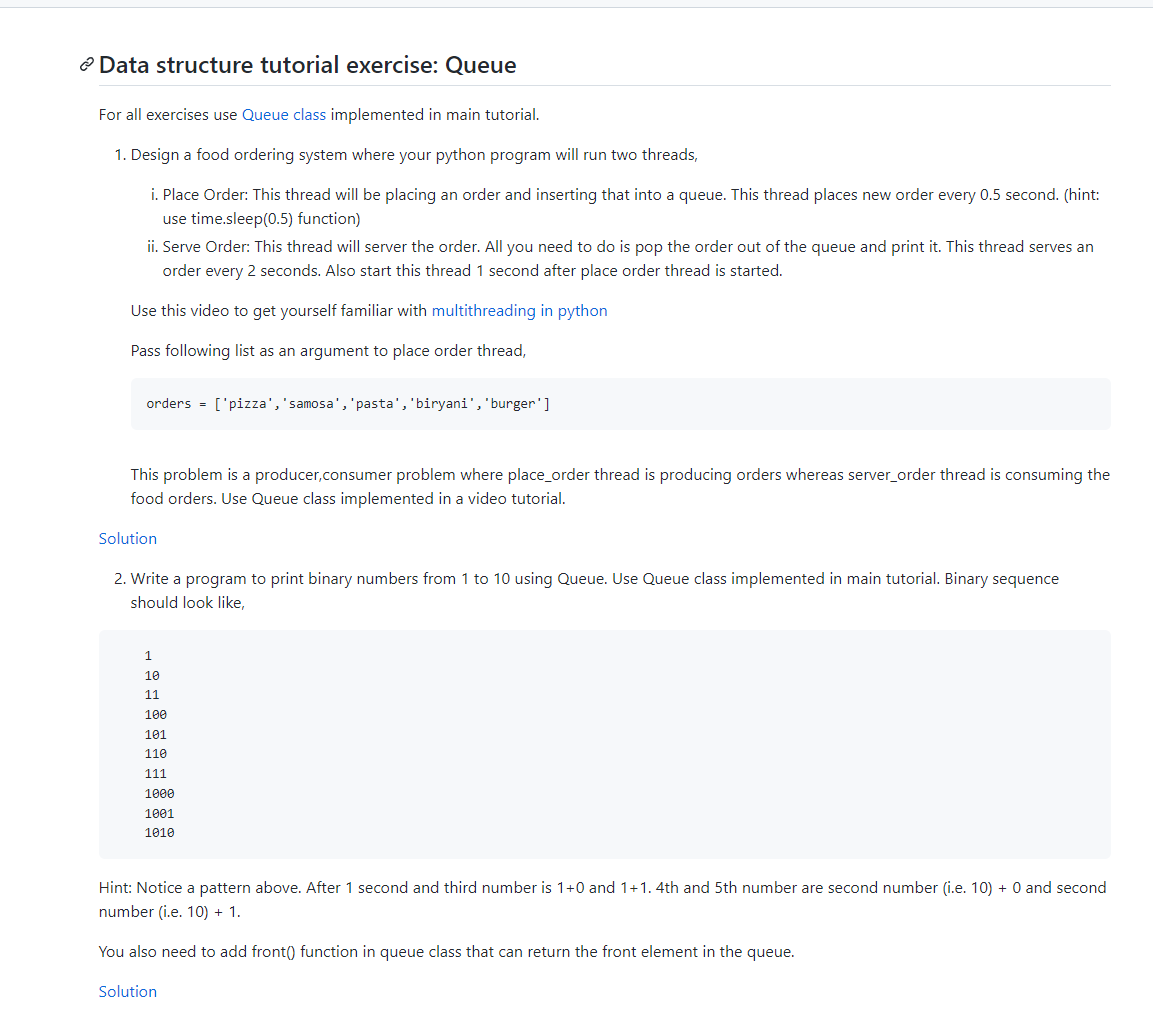
**Stack**

<https://github.com/codebasics/data-structures-algorithms-python/blob/master/data_structures/5_Stack/5_stack_exercise.md>



**Queue**

<https://github.com/codebasics/data-structures-algorithms-python/blob/master/data_structures/6_Queue/6_queue_exercise.md>



**Tree**

Implement a hierarchical structure for Electronics, laptop, TV, phone in a tree

**Binary Search Tree**

1.Implement BST

2. Add children

3. Search

4. In order traversal

5. preorder

6.postorder

7. Build tree = sa faci o lista cu elementele

8. Find min

9. Find Max

10. Sum of tree

11. Delete node

**Recursion**

1. Exercises from W3 school

2. Fibonacci

**Graph**

1.Implement Graph

2. Get Paths

3. Paths

4. Shortest Path

**Binary Search**

<https://github.com/codebasics/data-structures-algorithms-python/blob/master/algorithms/1_BinarySearch/binary_search_exercise.md>

**Bubble sort**

Exercise referenced above

**Quicksort**

Implemented as above

**Insertion sort**

<https://github.com/codebasics/data-structures-algorithms-python/blob/master/algorithms/4_InsertionSort/insertion_sort_exercise.md>

**Merge sort**

<https://github.com/codebasics/data-structures-algorithms-python/blob/master/algorithms/5_MergeSort/merge_sort_exercise.md>

**Shell Sort**

<https://github.com/codebasics/data-structures-algorithms-python/blob/master/algorithms/6_ShellSort/shell_sort_exercise.md>

**Selection Sort**

<https://github.com/codebasics/data-structures-algorithms-python/blob/master/algorithms/7_SelectionSort/selection_sort_exercise.md>

**Trie**

1. Insert

2.Search

**KMPB Algorithm**

Hay stack leetcode problem

**Backtracking algorithm**

N queens from Leetcode

**Easy level - Leetcode**

1. Two sum

2. Roman to integer

3. Longest common prefix.

4. Valid parantheses

5. Merge two sorted lists

6. Duplicate number

7. Remove element

8. search index position

9. Length of last word

10. Plus one

11. SQRT