# PermPy_1

July 10, 2022

## 1 Permutazioni (con un tocco di python) - parte I

### 1.1 Introduzione

Per permutazione si intende una disposizione di $n$ elementi in uno dei modi possibili. Un esempio è dato dai possibili anagrammi di una parola: **ROMA, RAMO, AMOR,** etc. comprendendo anche tutte le disposizioni prive di significatocome **MROA, MAOR, ...** e così via.
Una definizione formale di permutazione su un insieme di n elementi è la seguente:
*Si definisce permutazione una funzione bijettiva da un insieme finito di n elementi in sè.*

Solitamente, senza perdere di generalità, gli elementi dell'insieme sono indicati con la sequenza

$$0, 1, 2, \cdots, n-1$$

una permutazione viene descritta in forma esplicita con una matrice di due righe e n colonne:

$$S_{(4)} = \begin{pmatrix} 0 & 1 & 2 & 3 \\ 1 & 2 & 3 & 0 \end{pmatrix}$$

in cui la prima riga contiene gli elementi del dominio e la seconda le immagini corrispondenti secondo la funzione S(n)

Alternativamente si può omettere la prima riga e considerare solo la seconda (sottointendendo la prima riga che definisce la posizione di ciascun elemento.

### 1.2 Quante sono le permutazioni?

Poichè l'insieme di elementi sul quale applichiamo la permutazione è finito riesce naturale chiedersi qual è il numero di permutazioni diverse che è possibile definire su un insieme di n elementi.
Cerchiamo di capirlo con un esempio pratico:
Consideriamo un insieme con 3 elementi: {a, b, c} e cerchiamo di descrivere l'insieme di tutte le permutazioni, ossia tutte le possibili disposizioni delle tre lettere.
Considerando le tre posizioni disponibili possiamo sicuramente affermare che nella posizione iniziale (0 per intenderci) potremo mettere una lettera qualsiasi delle tre disponibili, proseguendo in seconda posizione abbiamo due sole possibilità di scelta (avendo già occupato la prima posizione con una delle tre lettere) infine, in ultima posizione, metteremo l'ultimo elemento rimasto disponibile.

Quindi le possibili permutazioni su tre elementi saranno:

| | |
|---|---|
| 1 | $abc$ |
| 2 | $acb$ |
| 3 | $bac$ |
| 4 | $bca$ |
| 5 | $cab$ |
| 6 | $cba$ |

Il numero di permutazioni possibili risulta quindi essere uguale a 6, valore che si ottiene moltiplicando tra loro le alternative diusponibili per ciascuna posizione: 3 per la posizione 0, 2 per la posizione 1 e 1 per l'ultima posizione.

Il numero così ottenuto si chiama fattoriale di 3 e si indica **3!**

In generale il fattoriale di un numero intero positivo n si indica con n! ed è definito come il prodotto degli n fattori interi da 1 a n.

$$n! = n * (n-1) * \cdots * 2 * 1$$

È possibile dare anche una definizione ricorsiva del fattoriale di n:

$$n! = \begin{cases} 1 & \text{se } n = 0 \\ n * (n-1)! & \text{se } n > 0 \end{cases}$$

### 1.3 Calcolare il fattoriale in python

Per calcolare il fattoriale in python, usiamo un costrutto for ripetuto su un range da 1 a n

[2]:
```
n = int(input('Inserisci un numero intero maggiore di 0: '))
fatt = n
for x in range(1,n):
    fatt *= x
print('Il fattoriale di',n,'è',fatt)
```

```
Inserisci un numero intero maggiore di 0: 4
Il fattoriale di 4 è 24
```

[3]:
```
# In realtà il package math mette a disposizione una serie
# di metodi tra i quali il metodo che restituisce il fattoriale
# del numero intero passato come parametro:

from math import factorial
n = int(input('Inserisci un numero intero maggiore di 0: '))
print('Il fattoriale di',n,'è',factorial(n))
```

```
Inserisci un numero intero maggiore di 0: 4
Il fattoriale di 4 è 24
```

## 1.4 Permutazioni in pratica

Spesso nei problemi combinatori e/o in altri tipi di problemi occorre generare delle permutazioni e sono stati sviluppati diversi metodi per rispondere a queste esigenze.
Quali sono le richieste più frequenti riguardo alle permutazioni?

(a) generare una permutazione casuale

(b) generare tutte le permutazioni

(c) ordinare le permutazioni

(d) generare una permutazione specifica (dato un ordinamento)

Poichè ci siamo proposti di utilizzare python per affrontare questi problemi, cominciamo a considerare quale potrebbe essere la struttura più adatta a rappresentare una permutazione.
In python esistono tre strutture principali (simili):

(a) liste

(b) tuple

(c) dizionari

La **lista** è una struttura molto flessibile e si presta bene all'uso che dobbiamo farne.
Una *lista*, in python, è un insieme di elementi, anche di tipo diverso, al quale è possibile aggiungere, eliminare singoli elementi ed accedere a qualsiasi elemento contenuto nell'insieme specificandone la posizione (come normalmente si fa con gli array).
Una *lista* è rappresentata come un elenco degli elementi che ne fanno parte racchiuso tra parentesi quadre.
La lista, inoltre, espone diversi metodi utili per la sua manipolazione e gestione.

## 1.5 Inizializzazione di una lista

[30]:
```
#Lista vuota
l=[]
l
```

[30]:
```
[]
```

[31]:
```
#Lista con elementi omogenei
l = ['mela','pera', 'banana']
l
```

[31]:
```
['mela', 'pera', 'banana']
```

[32]:
```
#Lista con elementi eterogenei (stringa, intero e boolean)
l = ['mela', 1, True]
l
```

[32]:
```
['mela', 1, True]
```

Python mette a disposizione una funzione, che ci tornerà utile, che permette di generare sequenze di numeri interi: la funzione **range()**.
La funzione *range()* si può richiamare con uno, due o tre parametri:

[33]:
```
# Un parametro
# range(n)
# genera la sequenza di numeri interi da 0 a n-1
# la sequenza può essere convertita in una lista utilizzando la funzione␣
 ↪predefinita list()
l = list(range(5))
l
```

[33]:
```
[0, 1, 2, 3, 4]
```

[34]:
```
# Due parametri
# range(a, b)
# genera la sequenza di numeri interi compresi tra a (incluso) e b (escluso)

l = list(range(-2, 6))
l
```

[34]:
```
[-2, -1, 0, 1, 2, 3, 4, 5]
```

[35]:
```
# Tre parametri
# range(a, b, step)
# genera la sequenza di numeri interi compresi tra a e b (escluso) con passo␣
 ↪fisso (step)

l = list(range(0,10,2))
l
```

[35]:
```
[0, 2, 4, 6, 8]
```

### 1.6 Generare una permutazione di n elementi casuale

Per generare una permutazione casuale abbiamo bisogno di utilizzare il package random che espone diversi metodi tra i quali **shuffle()** è il metodo che fa al caso nostro (mischia gli elementi della lista casualmente)

[36]:
```
from random import shuffle
l = list(range(5))
l
```

[36]:
```
[0, 1, 2, 3, 4]
```

[37]:
```
shuffle(l)
l
```

[37]:
```
[1, 3, 2, 0, 4]
```

### 1.7 Generare tutte le permutazioni di n elementi

Il problema di generare **tutte** le permutazioni di n elementi è un problema che può essere risolto applicando la *ricorsione*.
Se la sequenza è formata da **un solo** elemento, allora c'è un'unica permutazione e la stampo, altrimenti ripeto (per ciascun elemento della sequenza) i seguenti passi:
1. scambio l'elemento attuale con l'ultimo
2. richiamo la funzione sulla sottosequenza ottenuta escludendo l'ultimo elemento
3. rimetto a posto l'elemento precedentemente scambiato

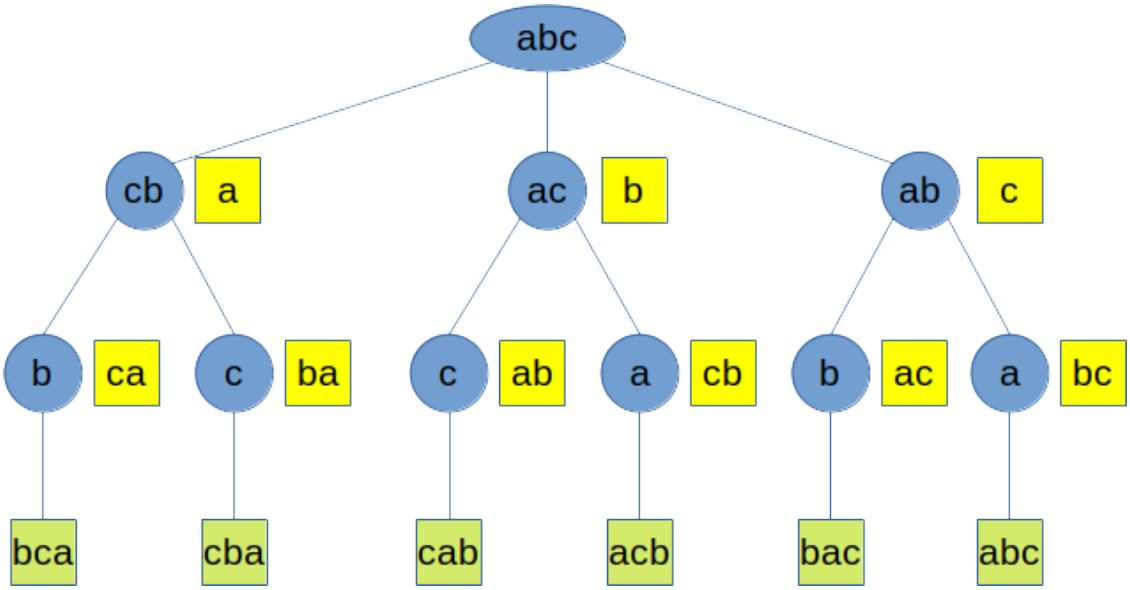
La prima chiamata va fatta con la sequenza di partenza e, come secondo parametro, la lunghezza della sequenza

[38]:
```
# definizione della funzione ricorsiva che genera tutte le permutazioni di p
def perm(p,n=1):
    if n==1:
        print(p)
    else:
        for i in range(n):
            p[i],p[n-1] = p[n-1],p[i]
            perm(p,n-1)
            p[i],p[n-1] = p[n-1],p[i]
```

[39]:
```
l = ['a','b','c']
perm(l,len(l))
```

```
['b', 'c', 'a']
['c', 'b', 'a']
['c', 'a', 'b']
['a', 'c', 'b']
['b', 'a', 'c']
['a', 'b', 'c']
```

Di seguito l'albero delle chiamate ricorsive della funzione perm(p,n) con p=[‘a’,‘b’,‘c’] e n=3:




### 1.8 Facciamo ordine!

La procedura *ricorsiva* ha generato tutte le permutazioni dell'insieme, ma non le ha generate in *ordine*.
Definiamo, quindi, un criterio di ordinamento e mettiamo in ordine le permutazioni.
Osservando il risultato ottenuto precedentemente è facile definire un ordinamento sulle *sequenze* prodotte: trattandosi di singoli caratteri l'ordinamento alfabetico è quello naturale da applicare, quindi, volendo riscrivere *ordinatamente* le sei permutazioni si perviene facilmente al seguente risultato:

0. abc
1. acb
2. bac
3. bca
4. cab
5. cba

Anche se gli elementi dell'insieme fossero *parole* si può applicare lo stesso criterio.
Per esempio se l'insieme fosse costituito dalle parole {arancia, banana, cavolo}, le sei permutazioni *ordinate* sarebbero:

0. arancia, banana, cavolo
1. arancia, cavolo, banana
2. banana, arancia, cavolo
3. banana, cavolo, arancia
4. cavolo, arancia, banana
5. cavolo, banana, arancia

Analogamente, senza perdere di generalità, considerando l'insieme costituito dai tre elementi {0, 1, 2} potremmo ottenere facilmente la sequenza delle sei permutazioni ordinate.
Ovviamente, lo stesso criterio, con un pò di pazienza è facilmente applicabile ad insiemi più numerosi.
Sarà più chiaro successivamente il motivo per il quale la prima permutazione è associata allo 0.

## 1.9 Scopriamo il *factoradic()*!

Dovrebbe essere nota la procedura con la quale è possibile passare dalla usuale rappresentazione decimale ad una base qualsiasi diversa da 10 e, quindi, diamo per acquisita questa abilità.
Solo per rinfrescare la memoria, il numero decimale 12 può essere rappresentato in base 2 (binario) come *1100* oppure in base 8 come *14* o in base 16 come *C* (in base 16 bisogna *aggiungere* le cifre da 10 a 15 rappresentandole con le prime sei lettere dell'alfabeto: A, B, C, D, E, F.
Scopriamo cos'è il *factoradic*: è una rappresentazione di un numero intero usando, però, una base **variabile**!

Per capirci, supponiamo di avere la seguente rappresentazione: *1100* e convertiamola in decimale associando a ciascuna posizione il valore del **fattoriale** corrispondente, cioè, a partire dalla posizione a sinistra della meno significativa (l'ultima cifra di un numero rappresentato con questa convenzione è sempre 0) il valore della cifra deve essere moltiplicato per il fattoriale della posizione corrispondente:

$$1 * 3! + 1 * 2! + 0 * 1! + 0 * 0! = 1 * 6 + 1 * 2 + 0 * 1 + 0 * 1 = 8$$

Altrettanto semplice è il passaggio inverso: passare, cioè da una rappresentazione decimale al *factoradic*
Supponiamo di voler rappresentare il numero 21 decimale.
Poichè 21 è *minore* di 4! = 24, utilizzeremo solo 4 posizioni (ricordando che l'ultima cifra sarà **sempre** 0) e le divisioni partono da 3!

21 : 3! = 21 : 6 = **3** con resto 3, prendiamo il resto e continuiamo la divisione:
3 : 2! = 3 : 2 = **1** con resto 1, continuiamo
1 : 1! = 1 : 1 = **1** con resto 0.
Finito!! (aggiungiamo uno **0** finale, sempre presente)

Il *factoradic* di 21 è **3110**
Semplice, no?

## 1.10 *factoradic* e permutazioni

Come sono legati il factoradic e le permutazioni?
Cominciamo con il considerare quante permutazioni dobbiamo generare, per esempio, partendo da un insieme di 4 elementi genereremo **4!** permutazioni, cioè 24 permutazioni.
Se assegniamo un numero a ciascuna di esse partendo da 0, le permutazioni saranno associate ai numeri da 0 a 23 (ricordate? Le permutazioni *ordinate* sono associate ai numeri da 0 a n! - 1).
Iniziamo con il considerare una funzione che, dato un numero **n** di elementi da permutare e la posizione **k**, genera il factoradic di **k** (a condizione che k < n!.

[1]:
```
# definizione della funzione python che restituisce il factoradic di un numero␣
↪intero k su n posizioni
from math import factorial
def factoradic(n, k):
    # n è il numero di posizioni
    # k è il numero che vogliamo rappresentare come factoradic
    # k dovrà essere compreso tra 0 e (n! - 1)
    s = []
```

```
    if k < factorial(n):
        while n>1:
            n = n - 1
            q = k // factorial(n)
            k = k % factorial(n)
            s.append(q)
        s.append(0) # aggiunge lo 0 finale
    return s
```

[2]:
```
factoradic(4,14)
```

[2]:
```
[2, 1, 0, 0]
```

[3]:
```
for x in range(0,24):
    print(x,':', factoradic(4,x))
```

```
0 : [0, 0, 0, 0]
1 : [0, 0, 1, 0]
2 : [0, 1, 0, 0]
3 : [0, 1, 1, 0]
4 : [0, 2, 0, 0]
5 : [0, 2, 1, 0]
6 : [1, 0, 0, 0]
7 : [1, 0, 1, 0]
8 : [1, 1, 0, 0]
9 : [1, 1, 1, 0]
10 : [1, 2, 0, 0]
11 : [1, 2, 1, 0]
12 : [2, 0, 0, 0]
13 : [2, 0, 1, 0]
14 : [2, 1, 0, 0]
15 : [2, 1, 1, 0]
16 : [2, 2, 0, 0]
17 : [2, 2, 1, 0]
18 : [3, 0, 0, 0]
19 : [3, 0, 1, 0]
20 : [3, 1, 0, 0]
21 : [3, 1, 1, 0]
22 : [3, 2, 0, 0]
23 : [3, 2, 1, 0]
```

[ ]: