Informatica

Basi di Python

La parte dopo ">>" è l'input

Definizione di Calcolare: Calcolare è un procedimento meccanico di manipolazione simbolica non interpretata *(cioè significativa)*

Esempio:
L'addizione standard, meccanica e combinata, $321 + 493 = 814$, operazione puramente meccanica e priva di significato.

Già in quest'esempio si possono vedere 3 attori/livelli:

1. *Calcolo* in sé e per sé
2. *Descrizione* di questo processo (data dal maestro in linguaggio normale)
3. *Esecutore* del calcolo

In informatica questa struttura a tre livelli è data dal programma (esecutore), che attraverso il linguaggio di programmazione (descrizione), esegue il calcolo (calcolo).

Definizione di Programma: Descrizione tramite un linguaggio artificiale (linguaggio di programmazione) della collezione di calcoli

Definizione di Esecutore: Macchina astratta di quel linguaggio.
Poiché ogni linguaggio ha la sua macchina, ogni macchina è capace di eseguire un programma solo se è scritto nel suo linguaggio.

Esempio:
L'esempio precedente può essere tradotto secondo lo schema:
- Descrizione: $321 + 493$
- Esecutore: PC
- Calcolo: $321 + 492 \to 814$

La macchine manipolano simboli raggruppati in collezioni omogenei di oggetti, per esempio gli interi (di cui fanno parte anche le operazioni di somma "+", differenza "-", moltiplicazione "*", elevamento a potenza "**", divisione intera "//", operazione modulo "%").

Altre informazioni:
I simboli possono essere infissi (in mezzo agli operandi), prefissi (posti prima $\sqrt{2}$), oppure postfissi (cioè messi dopo $2!$)
Utilizza la notazione posizionale decimale (ossia la posizione della cifra nel numero prendono un significato a seconda di dove sono scritti e sono scritti in base 10)
Inoltre non c'è un massimo per come grandezza di numeri, però c'è un limite nella realizzazione da parte della macchina

Ogni tipo di dato ha due caratteristiche principali, il Valore e le Operazioni:

Definizione di Valore: Con valore si intende il modo in cui sono rappresentati i dati

Esempio:
Per il tipo int il valore è rappresentato dalla usuale scrittura posizionale in base 10

Definizione di Operazioni: Sono le operazioni effettivi che si possono fare sui quei valori

Esempio:
Le operazioni su int sono la Somma, la Sottrazione (unarie e binarie), la Moltiplicazione, la Divisione tra interi, l'elevamento a potenza, l'estrazione di radice e la divisione con modulo.

È poi possibile scrivere in numeri anche in base diversa da 10.
0b serve per scrivere i numeri in base binaria, 0o serve per scrivere i numeri in base ottale e 0x in base 16

Tipo dei numeri con la virgola: Non possono essere definiti dei numeri Reali $\mathbb{R}$ in quanto non è possibile poter definire qualcosa di indefinito, ossia non possono essere definiti in modo effettivo, quindi ci si limita ad una porzione limitata dei numeri razionali $\mathbb{Q}$, più o meno centrata attorno allo 0. Questa è la tipologia float.
Vengono presentati attraverso l'usuale rappresentazione posizionale decimale con il punto.
(*Si potrebbe dire che c'è una iniezione tra numeri in* float *e in* int*, ma in realtà si tratta di cose completamente diverse*)
Possono essere scritti anche con significante/matissa dell'esponente, segnata da una virgola mobile.

Esempio:
$9.88\mathrm{e}{+25}$ che è esattamente equivalente a 9.88×10^{25}

La maggiore utilità di questa scrittura è quella di ridurre al minimo l'errore.
Le sue Operazioni sono la Somma + e la Differenza - (entrambi scritti sia come infissi che come prefissi), la moltiplicazione *, l'elevamento a potenza ** e la divisione tra razionali /.

Domanda: *Perché ci sono errori con i calcoli?*
Perché la macchina Python ragiona in binario e quindi tende a commettere errori con le trasformazioni.

Esempio:
$\frac{1}{10} = 0.1_0 = 0.0\overline{0011}_2$

Domanda: *L'elevamento a potenza funziona normale con i numeri "reali"?* si

Definizione di Espressione: Un'espressione è una frase corretta del linguaggio la cui valutazione produce un valore (oggetto), strutturato in tipi (che possono essere per esempio di tipo float oppure int)
La valutazione procede da sinistra a destra con usuali precedenze: $\mathrm{Elevamento\ a\ potenza} \rightarrow \mathrm{Somma\ e\ differenza\ unarie} \rightarrow \mathrm{Moltiplicazione,\ entrambe\ le\ Divisioni,\ Modulo} \rightarrow \mathrm{Somma\ e\ Differenza\ binarie}$
Poi possono essere modificate attraverso l'eventuale uso delle parentesi tonde.

Esiste anche il Tipo dei numeri complessi che risulta futile per il corso

Se mi serve invece solo la parte intera (attraverso il troncamento) mi basta il comano int:

```
>> int(<numero>)
```

Esempio:
$\mathrm{int}(-56.888) = -56$

Definizione di Overloading: Il fenomeno dell'overloading è quel fenomeno in cui uno stesso simbolo viene utilizzato per diverse funzioni ed in questo caso non è possibile stabilire immediatamente a quale comando fa riferimento

Esempio:
Un esempio di overloading è dato dal segno $+$ che sta a significare la somma fra interi che la somma fra numeri razionali, *apparentemente sono la stessa cosa ma in realtà non lo sono, in quanto sono diversi i tipi dei valori*

Definizione di Tipo str: È la tipologia di simboli che non sono numerici, ossia sono stringhe, da cui str
I suoi valori sono successioni finite di caratteri da opportuno argomento *ossia simboli dall'alfabeto anglosassone, simboli in sé e per sé, numeri, eccetera*

Esempio:
SIMONE è una stringa, ma scritta in questo modo la macchina non è in grado di leggerla, infatti deve essere presentata in modo opportuno.

La presentazione infatti deve avvenire in questi tre modi possibili:

```
"Simone"
'Simone'
'''Simone'''
```

Operazioni in str:

- Concatenazione: da due stringe attraverso il segno $+$ ottengo un'unica scritta composta dall'unione delle due scritte precedenti;

Esempio:

```
>> x = 'Simone'+'Martini'
>> x
'SimoneMartini'
```

- Commando len: commando che restituisce la lunghezza della scritta, in tipo int

Esempio:

```
>> x = len('Ciao')
>> x
4
```

- Ripetizione $*$: Concatenazione con sé stesso n volte, dove n è un numero di tipo int, $\text{'stringa'} * int$

Esempio:

```
>> x = 'Simone'*7
>> x
'SimoneSimoneSimoneSimoneSimoneSimoneSimone'
```

- Selezione: Seleziona il carattere in stringhe in maniera postfissa $\text{'stringa'}[int]$.
  Se il segno del numero è negativo allora parte dal fondo.
  Si può concatenare con len

Esempio:

```
>> x = 'Simone'[2]
>> x
'm'
```

- Comando type: Restituisce il tipo dell'input

Esempio:

```
>> len('ciao')
<class str>
```

Chiaramente nelle operazioni non posso concatenare diversi tipi.

Definizione di Comando: È la base del linguaggio la cui valutazione modifica lo stato della macchina

Definizione di Stato Interno: È la lista di associazioni fra nome e valore/oggetto, che evolve nel tempo grazie ai comandi. Il comando più semplice è quello di Assegnamento:

```
>> <nome> = <espressione>
```

Esempio:

```
>> 7*x
"x non è stato ancora assegnato"
>> x = 8
>> 7*x
56
```

Un nome viene definito quando si trova per la prima volta a sinistra in un assegnamento.
In altri linguaggi esiste una parte che prende il nome di "dichiarazione delle variabili", che in Python non esiste, in quanto è strettamente legato al valore.

Regole di Assegnamento:

1. Si valuta la parte di destra (*Si guarda se è corretta oppure no*)
2. Se $<\text{nome}>$ non è definito, viene messo nello stato interno
3. Crea un legame tra $<\text{nome}>$ e valore definendo un legame tra il nome e il valore determinato al passo 1

Domanda: *È possibile usare a destra lo stesso valore a sinistra (tipo* x=x+1*)*
Si in quanto l'ordine è rispettato

Domanda: *Che succede se nomino lo stesso oggetto due volte? (tipo* x=4; y=x*)?*
Molto semplicemente quello stesso valore/oggetto avrà due nomi (fenomeno dell'Aliasing)

I nomi dati possono essere lettere maiuscole e minuscole, cifre e trattini bassi.
L'unica regola è che non possono iniziare con numero.
Esistono poi dei nomi che non possono essere usati, questi sono dei nomi riservati (per esempio from non può essere utilizzato in quanto è un nome utilizzato già dalla macchina)

Definizione di Script: È un testo in cui ogni riga deve essere una frase legale in Python.
Chiaramente quando si esegue uno script, tutto quello che era presente nella sessione precedente viene eliminato.

Per vedere direttamente un risultato c'è la funzione $\text{print}()$

Esempio:

```
[SCRIPT]
x = 10
y = 15
z = x+y
print(z)

[SHELL]
25
```

Si possono scrivere più elementi di uno stesso print separando i vari elementi con una virgola

Altri tipi sono i booleani bool che servono per le verità (vero o falso) e i numeri complessi complex (al posto di i viene messa j)

Esiste un sito in cui è possibile fare Tracing del programma, ossia è possibile seguire passo per passo i vari comandi del programma Python Tutor.
Visualizza lo Stato Interno della Macchina Python

Servono per spiegare al meglio quello che fa il programma in linguaggio normale
Per poter fare i commenti è sufficiente mettere davanti $\#$, oppure tra tripli doppi apici

Esempio:

```
#Sono un commento felice
"""Sono un Commento"""
```

Ci sono i moduli, che sono delle funzioni già pronte e dei nomi a cui sono già stati assegnati dei valori

Esempio:
In math, pi è già assegnata un'ottima approssimazione di π

Un altro modulo è random per creare numero pseudo randomici (tutto trovato nelle documentazioni)

Per fare già si usa il comando di Importazione "$\mathrm{from}\ <\ldots>\ \mathrm{import} <\ldots>$"

Esempio:

```
>> from math import pi
>> pi
3.1415...
```

È possibile importare sia numeri costanti (per esempio e) sia funzioni (sqrt che risulta essere più precisa rispetto a $**1/2$)

Questi moduli non sono altro che altri file $.\mathrm{py}$, quindi è possibile farlo anche con i nostri file
L'importante è salvare il tutto nello stesso posto

Esempio:

```
[PROVA.py]
secondo = 'nome generico'

[SHELL]
>> from PROVA import secondo
>> secondo
'nome generico'
```

Per poter mandare a capo basta mettere nel comando di print un "$\backslash\mathrm{n}$"

Esiste un funzione che permette di trasferire dallo stato esterno a quello interno Input

Definizione di Input: È una funzione predefinita che permette di trasferire da stato esterno a stato interno: $\mathrm{input}()$. Valutare $\mathrm{input}()$ restituisce la stringa che è stata digitata su dispositivo di ingresso.
Quello che nell'effettivo succede è che la valutazione del programma viene interrotta per aspettare che l'utente inserisca una stringa, per terminare con invio

Esempio:

```
print('Inizio')
test = input()
print('dopo')
print(testo)
```

```
[SHELL]
inizio
{Qui la macchina aspetta che venga inserita una stringa} 'Pippo'
dopo
Pippo
>>
```

Se non assegno un nome ad un valore semplicemente viene buttato via.

Normalmente il valore viene salvato in tipo str, per poter renderlo in un altro tipo, semplicemente mi è sufficiente davanti il tipo del valore.

È possibile scrivere anche all'interno di $\mathrm{input}()$

Esempio:

```
[FILE.py]
text = input('Dammi il raggio')

[SHELL]
Dammi il raggio
```

Definizione di Funzione: Pezzo di programma, a cui è possibile associare un nome, costituito da diversi comandi e restituisce un valore. Posso eseguire una funzione semplicemente chiamandola.
Non è necessario un $\mathrm{print}()$ *per ridare il valore*

Esempio:

```
[FILE.py]
def VolumeSfera(Raggio):
        vol = (4/3)*pi*(raggio)**3
        return vol

[SHELL]
>> VolumeSfera(3)
>> <Valore del volume>
```

La funzione è costituita da due aspetti diversi:

- La definizione di funzione dato dal nome riservato def, poi il nome della funzione, poi una parentesi tonda $($, poi ci possono essere i parametri, poi chiusa e poi due punti:
  def $<\mathrm{nome}>$ ($<\mathrm{parametro\ formale}>$): *Si chiama intestazione della funzione*
- Corpo della funzione: Sono linee di comandi in Python che devono avere una indentazione positiva (*Devono essere spostate dal margine e tutte alla stessa indentazione, quella canonica sono 4 spazi*)
  Il corpo deve necessariamente terminare con la funzione return

È definito come comando composto in quanto è costituito da una intestazione seguito da un corpo a indentazione positiva.

Quando una macchina valuta una funzione, il corpo non viene eseguito, ma viene messo un legame interno tra la funzione e lo Stato Interno e non viene eseguito (se non eseguito esplicitamente)

Per mettere più parametri, basta separarli con un virgola

Solo nel corpo di una funzione ci può stare il comando retrun seguito dal valore che voglio restituire (return $<\mathrm{valore}>$)

Posso chiamare o usare una funzione in questo modo: <nome> (<parametro attuale>)
Come nell'esempio sopra
Chiamare una funzione significa mandare in esecuzione il corpo della funzione

Se ho un determinato numero di Parametri Formali, devo avere lo stesso numero di Parametri Attuali
Parametri Attuali sono espressioni e sono valutate alla chiamata, mentre quelli formali sono Formali sono i "nomi" dei Parametri Attuali.

È possibile anche dare assegnare al valore finale della funzione un nome

Esempio:

```
[FILE.py]
def VolumeSfera(Raggio):
        vol = (4/3)*pi*(raggio)**3
        return vol

volume = vol

[SHELL]
>> VolumeSfera(8)
>> Volume
2144,66...
```

Se ci si dimentica del return è come se ci fosse scritto retun None

Le funzioni aiutano molto perché è possibile spezzare il programma in tanti piccoli programmi

Esempio:
Calcolare il volume della sfera con raggio pari alla diagonale di un quadrato di lato l

```
[FILE.py]
from math import pi, sqrt

def diag_quad(l)
        return l*sqrt(2)

def volsf(raggio)
        return (4/3)*pi*(raggio**3)

def mio_prob(l)
        return volsf(diga_quad(l))

res = volsf(8)

[SHELL]
>> mio_prob(8)
6066,00..
```

L'ideale è importare prima tutto quello che serve per poi far partire il programma

L'associazione tra parametro attuale e tra parametro formale viene distrutta al termine della funzione. (Ossia viene distrutto l'ambiente locale)
Il parametro formale è un nome **locale** alla funzione

Definizione di Frame: Lo stato interno di una si chiama Frame. Il frame del programma prende il nome di Frame Globale, mentre lo stato interno di una funzione prende il nome di Frame Locale

Regola per i Frame Locali: Un nome che compare a sinistra di un assegnamento nel corpo di una funzione è *locale* a quella funzione e quindi ha senso ad una associazione *solo* durante l'esecuzione di quella funzione. *Vero anche se quel nome fosse stato già presente nello Stato Globale, infatti ci sarebbero due nomi uguali ma che facevano riferimento a due cose diverse.*

Se all'interno di un frame locale utilizzo un nome a destra, possono succedere tre casi:
- lo prende dal locale;
- se non c'è nel locale, lo prende dal globale;
- se non c'è nel globale da errore.
Ma se lo usassi a sinistra quel valore non sarebbe più globale ma locale.

È possibile definire una funzione dentro una funzione in questo modo:

```
def a(x)
        def b(y)
                return
        return
```

Solo che la funzione dentro l'altra funzione è locale all'altra, quindi devo prima chiamare la funzione più esterna, poi passare a quella più interna.

Utilizza un processo di Scoping Statico (ossia è in base alla struttura del codice).

Definizione di Numeri Pseudocasuali: Algoritmo che estrae dei numeri che dovrebbero passare dei test casistici. Sembrano casuali ma in realtà non lo sono (giusto un paio di cose in natura lo sono)

Struttura dell'if:

```
if <condizione>
        <comando nel ramo then>
        ...
else:
        <comando nel ramo else>
        ...
<comando fuori dall'if>
```

Questo tipo di struttura è definita struttura booleana, ossia può essere valutata come vera o falsa e, in caso sia vera, vengono eseguite le istruzioni che si trovano all'interno del costrutto if

Ecco la lista di alcuni operatori di confronto:

```
x==y
x!=y
x>y
x<y
x>=y
x<=y
```

Altri tipi di operatori sono gli operatori logici:
- a and b: Se sono entrambe vere allora valgono True
- a or b: Se una tra a e b è vera allora vale True
- not a: Se a vale True, allora vale False

Un caso particolare di else è dato da elif in cui è possibile mettere un'altra condizione:

```
if <condizione 1>
        <comando del then>
        ...
```

```
elif <condizione 2>
        <comando dell'elseif>
        ...
else
        <comando qualora non vengono soddisfatte le condizioni>
        ...
```

Esistono delle sequenze che possono essere immutabili (come stringhe o tuple) o mutabili (cioè che vengono create e poi possono essere modificate).

Le tuple (di tipo tuple) sono delle sequenze di valori (tra parentesi tonde) separate da delle virgole.

Altre operazioni sulle stringhe (oltre a $*$ e $+$) vi sono gli operatori di confronto, come $==, <, >\ldots$
Una particolare operazione è quella di $\mathrm{slicing}$ $([\mathrm{inizio} : \mathrm{fine}])$, che come dice il nome, serve per "tagliare" delle parti da delle stringhe. *Se inizio non è specificato allora inizia dal primo indice, se fine non è specificato arriva fino alla fine*

Esempio:

```
>>> parola = 'Ciao'
>>> parola[2:3]
'ao'
```

Vi è un altro tipo di $\mathrm{slicing}$ che è lo $\mathrm{slicing\ esteso}$ che ha la struttura $[\mathrm{inizio} : \mathrm{fine} : \mathrm{passo}]$ dove il passo serve per indicare il senso di percorrenza (in caso $\mathrm{inizio} > \mathrm{fine}$)

Le tuple sono un tipo strutturato e sono delle sequenze immodificabili di valori separate da stringhe:

Esempio:

```
>>> dati = 'Anna', 'Pannocchia' ,'F' ,20
>>> dati
('Anna', 'Pannocchia', 'F', 20)
```

Le virgole sono indispensabili, in quanto danno la netta separazione tra stringhe e tuple, infatti $('\mathrm{Michael}')$ è una stringa mentre $('\mathrm{Michael},')$ è una tupla con un solo elemento

Le operazioni che si possono fare con le tuple sono $+$ per poter concatenare le tuple, $[\,]$ di selezione e $[\,:\,]$ di slicing.

È possibile anche impacchettare le tuple e spacchettarle:

Esempio:

```
>>> dati = ('Anna', 'Pannocchia', 'F', 20)
>>> dati
('Anna', 'Pannocchia', 'F', 20)
>>> (nome, cognome, sesso, eta) = dati
>>> nome
'Anna'
>>> cognome
'Pannocchia'
```

Per verificare l'appartenenza è possibile usare in e $\mathrm{not\ in}$ se un elemento (o una sequenza di elementi) appartiene o meno ad una sequenza.

Esiste la possibile di creare una iterazione sulle sequenze:

```
for <nome> in <sequenza>
        <blocco>
```

In Python vi è una gran differenza tra "Oggetto" e "Valore". Infatti due oggetti diversi x e y per esempio possono avere lo stesso valore ma essere oggetti diversi:

```
x = 1000
y = 1000
```

Così hanno lo stesso valore

```
x = 1000
y = x
```

Così hanno lo stesso oggetto (e di conseguenza lo stesso valore)

Per verificare effettivamente che sono lo stesso oggetto esiste il comando is

Esempio:

```
>>> x = 1000
>>> y = 1000
>>> x is y
False
>>> y = x
>>> x is y
True
```

Esiste un comando particolare che è accum che permette di "accumulare" dei determinati dati:

Esempio:

```
accum = <valore iniziale>
for e in <sequenza>
        <comandi>
        accum = <op>(accum, e)
        <comandi>
```

$<\mathrm{op}>$ non è altro che l'operazione che si vuole fare con l'accumulazione
Spesso il valore iniziale di accum non è altro che l'elemento neutro dell'operazione

Ogni oggetto è un "valore attivo".
Esistono una serie di comandi (differenti per tipo) che permettono di stimolare gli oggetti e il metodo è una di queste modalità.
Ha la struttura di:

```
<oggetto>.<metodo>(<eventuali parametri>)
```

Chiaramente oggetti dello stesso tipo condividono gli stessi metodi.

Esempi di metodi offerti per str:
https://docs.python.org/3/library/stdtypes.html

Un altro tipo particolare di sequenze è quello del range. Questi sono altro che intervalli di int e ha la stessa notazione dello slice, ossia $<\mathrm{range}>(<\mathrm{inizio}>, <\mathrm{fine}>, <\mathrm{passo}>)$

Durante la scrittura del codice si possono individuare due tipi di errori:
- quelli statici:
$\text{SyntaxError: invalid syntax}$
- quelli dinamici (che vengono chiamati anche eccezioni):
$\text{Traceback (most recent call last): File ””, line 1, in ZeroDivisionError: division by zero}$

È possibile arginare l'eccezione attraverso i comandi $\text{try} - \text{except}$

Esempio:

```
def mediaE(T):
        somma = 0
        for e in T:
                somma = somma + e
        try:
                return somma/len(T)
        except ZeroDivisionError:
                return 0
```

La struttura base di $\text{try} - \text{except}$ è:

```
try:
        <block>
except <exception>:
        <handlerblock>
```

Se non è presente $<\text{exception}>$, allora vengono bloccate tutte le eccezioni

Esiste un altro modo di fare cilci, ossia sfruttando il comando while:

```
<comando>
while <guardia>
        <corpo>
<comando>
```

Notare come la guardia viene valutata ogni volta che si ricomincia il tutto.
Chiaramente la parte prima deve avere un senso e che la parte del $< \text{corpo} >$ modifichi qualcosa, perché sennò il programma andrebbe in loop.

Invariante di un ciclo: proprietà vera quando si entra nel ciclo e che è di nuovo vera quando si rientra nuovamente nel ciclo (Proprietà che sostanzialmente mantenuta vera durante il ciclo).

Esempio:

```
def pari(tup): #True sse esiste pari in tup
        for i in range(len(tup):
                if tup[i]%2==0:
                        return True
        return False
```

L'invariante in questo caso è $\text{Inv(i)} \equiv \neg\exists x \in tup[:i]$ ed è quello che succede
Con $\text{Inv}(0) \neg\exists x \in tup[:0]$ per pari, Vero perché non ci sono elementi pari in nell'insieme vuoto $(\varnothing)$
Poi lo si fa per induzione:
Supponiamo che $\text{Inv}(i)$ sia vera quando iniziamo l'esecuzione del corpo con valore i. Vogliamo dimostrare che vale $\text{Inv}(i+1)$ dopo che abbiamo eseguito una volta il corpo e non abbiamo fatto il return del corpo.
Sappiamo che $\nexists x \in tup[:i]$ che è pari. Se eseguiamo il corpo e **non** viene fatto return vuol dire che $tup[:i]$ è

dispari, cioè $\nexists x \in tup[:i+1]$ che sia pari.
Quindi se il for termina senza mai eseguire il return al suo interno, vale $\text{Inv}(\text{len } tup) \equiv \nexists x \in tup$ x pari

Definizione di Debug: Processo di eliminazione di errori del software (attraverso le varie prove):

Ci sono tre tipi di errore:

- Errori di Sintassi (normalmente facilmente visibili): Codice scritto male
- Errori di run-time: Errori che non saltano fuori durante la fase di debug che l'IDE non riesce a percepire (errore che avvengono durante l'esecuzione) e ne consegue che l'errore si chiude per un'anomalia
- Errori di Semantica: Il programma non si comporta come vorremmo.

Algoritmo di Debug:
- La prima cosa da fare è quello di leggere attentamente il testo del programma (ossia ipotizzando quali sono i risultati che vorrei ottenere);
- Lo si prova a fare "a mano" (ossia provando a spiegarlo a qualcuno - o ad una paperella di gomma, in gergo qualcuno che non è altamente intelligente)
- Lo si prova su diverse prove (per vedere che effetti e risultati).
- Quando c'è un errore provo a sistemarlo, ma se sto peggiorando le cose posso tornare indietro.

Uno dei più grandi errori da principiante è quello di fare lunghe sessioni di programmazione senza mai testare il codice.

Scansione lineare: Quantità di tempo proporzionale alla lunghezza della sequenza (lineare per questo motivo). La scansione lineare, indipendentemente dai dati, il costo è proporzionale a n, perché deve scandire tutta la sequenza.

Esempio:

```
for i in s
        if e in vocali:
                print(e)
```

for i in s è la scansione lineare ed è proporzionale a s (cambia al massimo la costante proporzionale).

Poi c'è la Ricerca Lineare (che non è più come la scansione), in quanto c'è anche il "se c'è" e "in che posizione si trova":

Esempio:

```
for i in s
        if e in vocali:
                return 'Trovata'
return 'No vocale'
```

Qui il tempo della funzione non è strettamente legata alla lunghezza, ma dipende anche dal tipo di dati che inseriamo.
Il caso peggiore qui è che s non abbia vocali (quindi il tempo su una sequenza lunga n è proporzionale a kn, dove n è la lunghezza della stringa).
Il caso ottimale è che $s[0]$ è una vocale e il tempo impiegato non dipende più da n ed è costante.
Poi ci sono tutti i casi intermedi $s[i]$ è una vocale e in $s[:i]$ non ci sono vocali. Qui il tempo è proporzionale a i (nonostante la sequenza sia lunga n).

Nel caso della scansione lineare dipende solo dalla lunghezza della stringa (poco interessante)
Nella ricerca lineare (più interessanti) dipende da lunghezza e posizione. Quello che ci interesserà di più è il caso peggiore.

Volendo potremmo cercare le cose intermedie, tempo e posizione, ma per parlare di media devo sapere come sono fatti i dati. In ipotesi di distribuzione omogenea il tempo sarebbe proporzionale a $\frac{n}{2}$, ma così non lo vedremo quasi mai.

Quanto si tratta di un problema di complessità, non si cerca il tempo in secondi, ma la complessità in sé da quello che richiede (Quindi analisi del caso peggiore).

Dove usare il while?

- Scrivere una funzione in cui si utilizza una ricerca lineare (e dove non si possa usare return)

Esempio:

```
i = 0
while i < len(tup) and tup[i]%2 != 0:
        i = i + 1
if i < len(tup):
        #c'è un pari
else:
        #non ci sono pari
```

- In cicli di Input

Esempio:

```
x = None
while x is None
        try
                x = float(input('Dammi un numero' ))
        except ValueError
                print('Sei duro? Un numero.')
                x = float(input())
print(x)
```

- Liste che cambiamo lunghezza
- Moltissimi algoritmi belli, interessanti e eleganti

Esempio sull'algoritmo naive per $\mathcal{MCD}$:

```
def MCD_naive(a, b):
        if a == 0 or b == 0:
                return max(a, b)
        for k in range(min(a, b), 0, -1):
                if a % k == 0 and b % k == 0:
                        return k
```

Qui il tempo è proporzionale (nel caso peggiore) al minore tra a e b (Ossia quando sono coprimi)

Sfruttando l'algoritmo di Euclide possiamo definire una due successioni $\{a_i\}$ e $\{b_i\}$ tali che:
$a_0 = a,\ b_0 = b$ che diventa $a_{i+1} = b_i,\ b_{b+1} = a_i \% b_i \Rightarrow a_n = \mathcal{MCD}(a, b)$
Bisogna vedere due cose:
1. Se effettivamente il programma termina
2. E se termina effettivamente con $a_n = \mathcal{MCD}(a, b)$

1. poiché $b_1 = a_0 \% b_0 \in [0, b_0[$ e di conseguenza $\forall b_i :\ b_{i+1} = a_i \% b_i \in [0, b_i[$ Quindi b_i è una successione decrescente che eventualmente raggiungerà lo 0. Di conseguenza prima o poi l'algoritmo termina.
2. Data la dimostrazione si ha effettivamente che $a_n = \mathcal{MCD}(a, b)$

Il codice:

```
def Euclide_base(a, b)
        ai = a
        bi = b
        while b != 0
                temp = ai
                ai = bi
                bi = temp % b
        return ai

def Euclide(a, b):
        while b != 0:
                a, b = b, a % b #Assegnamento multiplo mi permette di non usare temp
        return a
```

Tra questo e quello precedente, il caso peggiore è quando sono coprimi.
Il caso peggiore è quando a e b sono due numeri successivi della sequenza di Fibonacci, in quanto è proporzionale al logaritmo del più piccolo dei due $\log min(a,b)$ nel secondo caso e al più piccolo dei due nel primo caso.

Un altro bell'algoritmo è quello dello schema del ricerca binaria/dicotonica (schema che può essere applicato ovunque). È un algoritmo che va a cercare un elemento in una sequenza ordinata.

Funzionamento: $\ell = 0$ (low = indice dell'inizio), $h = \text{indice più alto}$, $m = (\ell + h)//2$ (il punto medio con divisione intera). Se cerco un numero posso partire analizzando l'elemento medio. Se è maggiore allora sta a destra, altrimenti a sinistra. Se sta a destra allora $\ell = m + 1$, se invece sta a sinistra $h = m$ e così via
Se cercassi un valore che non c'è, otterrei un intervallo vuoto, che rappresenta la condizione di terminazione dell'algoritmo.

Ecco un possibile codice:

```
def binsearch(A, item):
        high = len(A)
        low = 0
        while low < high:
                med = (high + low) // 2
                if A[med] == item:
                        return med
                if item < A[med]:
                        high = med
                else:
                        low = med + 1
        return -1
```

-1 è un modo convenzionale di dire che l'elemento non c'è

Il caso peggiore nella ricerca binaria è il caso in cui non c'è l'elemento (ossia quando $\ell = h$).
Se la sequenza A è lunga 2^n ci vogliono n o $n-1$ sequenze, ossia $\log_2 \text{len(A)}$

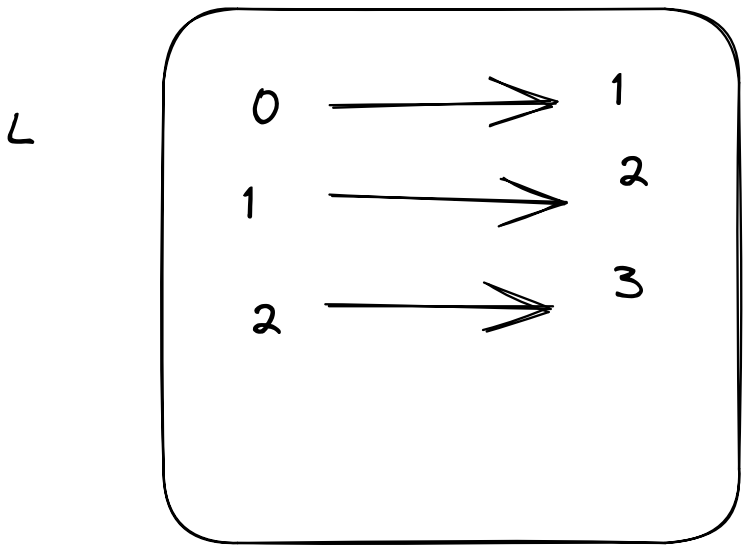
C'è anche un elemento di invarianza, ossia $\text{In(low, high)} = \text{item} \notin A \text{ or item} \in A[\text{low:high}]$

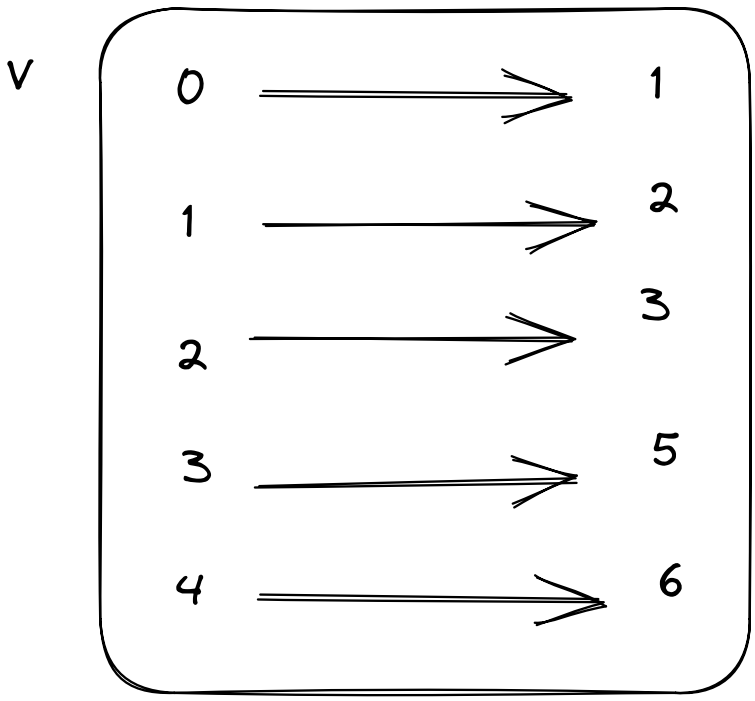
Ripasso Tuple:
- Valore: Sequenze non modificabili di valori:
- Presentazione: $(v_1, v_2, \ldots, v_k)$
- Operazioni: $+, *, len, t[]$, slice

Liste:
- Valore: Sono sequenze modificabili di valori (Modificabili: è possibile modificare/cambiare il **valore** di un oggetto preservandone l'identità)
- Presentazione: $[v_1, v_2, \ldots, v_k]$
- Operazioni: *le stesse delle tuple*

Esempio:

```
>>> L = [1, 2, 3]
>>> V = L + [5, 6]
>>> print(V[2])
3
```





Nelle Liste è possibile fare anche assegnazioni a sinistra:

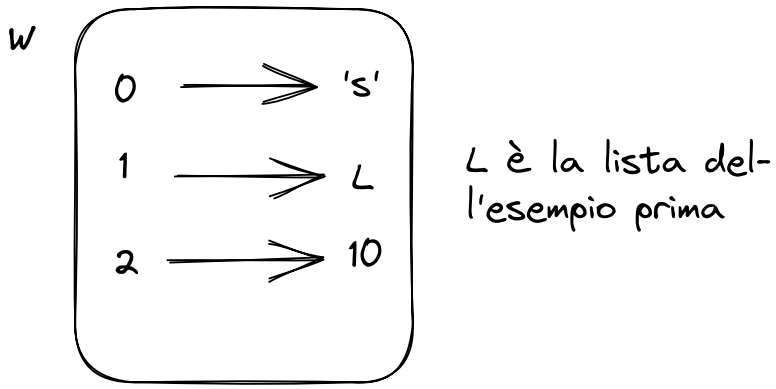
Esempio:

```
>>> V[2] = 200
```

Adesso il valore 3 è diventato 200 ma il comando is continua a dare vero

Altro Esempio:

```
>>> W = ['5', L, 10]
```



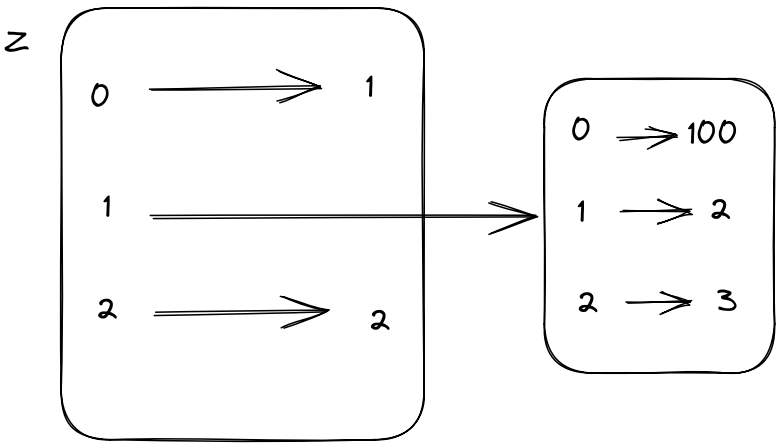

```
>>> L[0] = 100
>>> print(W)
['s', [100, 2, 3], 10]
```

Non è più quello che vedevo prima perché W è L, quindi una modifica su L si ripercuote su tutto W (side effect - effetti collaterali)
Comodo sì ma può creare tantissimi errori.

Per copiare una lista/tupla si fa con le slice.

Continuo esempio:

```
>>> L = [1, L[:], 20]
```




```
>>> print(Z[1][0])
```

Se io facessi:

```
>>> V[2] = V #Sono lo stesso oggetto - caso di Aliasing (2 nomi stesso oggetto)
#Ottengo una lista ciclica che ripete sé stessa nella posizione 2
>>> V[2][0] = 9
>>> V[2][2][2][2][0] = 10
#Posso scriverlo tutte le volte che voglio ma alla fine modifica sempre lo stesso elemento.
```

```
>>> print(V)
[10, 2, [10, 2, [10, 2, [...]]], 5, 6]
```

Python si accorge che c'è un ciclo e mette dei puntini.
Quando ci sono dei puntini nella valutazione della lista significa che c'è un qualche ciclo.

A cosa servono le tuple se abbiamo le liste?
Con liste ci sono degli aspetti delicati e condivisione. Con liste non sono modificabili. Sostanzialmente modifica il ragionamento durante la programmazione

Con le liste ci sono due vincoli affinché non vengano generate eccezioni:

- Devono essere su un oggetto modificabile;
- La sintassi deve essere scritta bene.

```
<selezione> = <espressione>
```

Esempio:

```
#Ho lista non vuota e voglio scambiare primo e ultimo elemento
>>> L = [10, 20, 30, 40]
>>> L[0], L[-1] = L[-1], L[0] #Posso usare assegnamento multiplo con le liste
>>> print(L)
[40, 20, 30, 10]
```

Altro Esempio:

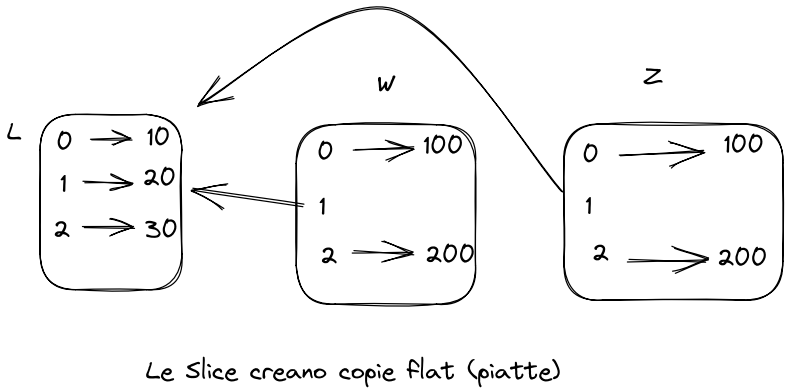
```
L = [10, 20, 30, 40]
def doppio(l)
        #Modifica ogni elemento della lista con il suo doppio
        for i in range(len(L))
                L[i] = L[i] * 2 #La selezione deve essere fatta sugli indici, non su
singoli nomi
                #La selezione inoltre deve essere fatta a sinistra

[SHELL]
>>> doppio(L) #Se facessi direttametne print(doppio(L)) mi avrebbe dato None
>>> L
[20, 40, 60, 80]
```

Non si può cambiare una lista in una tupla, ma è possibile creare una tupla che abbia gli stessi valori della lista.

```
>>> L = ['a', 'b', 'c']
>>> T = tuple(L)
>>> T
(a, b, c)
>>> L[1] = 100
>>> T
('a', 'b', 'c')
```

Non dobbiamo mai supporre l'identità di oggetti anche se hanno lo stesso valore. Le operazioni creano sempre nuovi oggetti.

```
>>> L = [10, 20, 30]
>>> W = [100, L ,200]
>>> Z = W[:]
```



Le Slice creano copie flat (piatte)

Posso chiamare lo stesso oggetto L in tre modi diversi

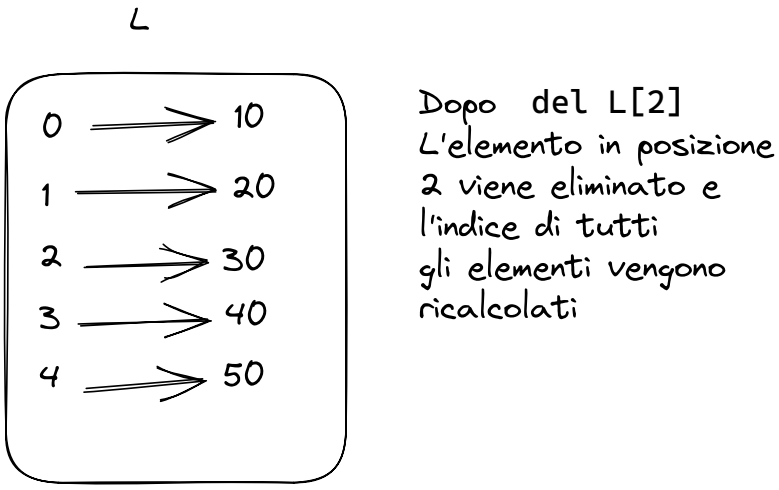
```
>>> L
>>> W[1]
>>> Z[1]
```

Nelle liste è possibile fare anche operazioni in più rispetto alle tuple:

Il comando $\mathrm{del}\ L[i]$ elimina l'elemento della lista L nella posizione i:

```
del L[i]
```

L'identità dell'oggetto è rimasto, ma all'interno della lista qualcosa è cambiato.
La cancellazione di un elemento ha fatto si che gli elementi vengano ricalcolati.



Esempio:

```
L = [10, 0,30 ,40, 0, 50, 0]
i = 0
while i < len(L):
        if i == 0:      #Qua non aumento l'indice perché vengono già cambiati
                del L[i]
        else:
                i = i + 1
```

Con il for non si poteva fare perché il for congela l'oggetto ma non il suo valore (quindi cambia il valore di $\mathrm{len(L)}$)
Quindi il ciclo sarebbe andato oltre la lunghezza della lista.
Per questo **Mai usare** for **per le liste**

Per creare una lista con un solo elemento è possibile fare:

```
>>> L = [2] #Non c'è bisogno di mettere una virgola rispetto alla tupla
```

Un modo in cui si possono usare la lunghezza delle liste è utilizzando lo slice a sinistra dell'assegnamento:

```
>>> L = [10, 20, 30]
>>> L
[10, 20, 30]
>>> L[0:2] = [100, 200, 300] #Parte in cui si aggiunge
>>> L
[100, 200, 300, 30]
```

Allo stesso modo si può anche cancellare

```
>>> L = [100, 200, 300, 30]
>>> L[1:2] = [] #Parte in cui si elimina
>>> L
[100, 300, 30]
```

Esiste anche un altro modo per aumentare e diminuire, ossia attraverso i metodi (in quanto le liste sono attive e possono essere stimolati attraverso metodi).
Il metodo più semplice è $.\text{append}$ (ma restituisce None)

Esempio:

```
>>> L = [100, 200, 300, 30]
>>> L.append(900)
>>> L
[100, 200, 300, 30, 900]
>>> L.append('simo')
>>> L
[100, 200, 300, 30, 900, 'simo']
>>> L.append([3, 4, 5])
>>> L
[100, 200, 300, 30, 900, 'simo', [3, 4, 5]]
```

Per tutto il codice l'identità della lista resta sempre la stessa.
Concatenare due liste invece altera l'identità della lista.

Esempio:

```
>>> L = [10, 20]
>>> L = L + [30, ]
>>> L
[10, 20, 30]
```

Con $.\text{insert}(i, x)$ posso aggiungere un elemento x all'indice i
Esempio:

```
>>> L = [10, 20, 30]
>>> L.insert(0, 100)
>>> L
[100, 10, 20, 30]
>>> L.insert(33, 1000) #Se l'indice supera la lunghezza della lista è come .append
>>> L
[100, 10, 20, 30, 1000]
```

Con $.\mathrm{extend}(\mathrm{Lista})$ si fa una concatenazione che elimina l'oggetto
Con $.clear()$ svuota la lista

Tutti questi metodi restituiscono None in quanto ci interessa soltanto quello che fanno

Altro esempio:

```
>>> L = [10, 20]
>>> W = L
>>> L = L + [30]
>>> L
[10, 20, 30]
>>> W
[10, 20]

#È la stessa cosa di fare
>>> L = [10, 20]
>>> W = L
>>> L.append(30)
>>> L
[10, 20, 30]
>>> W
[10, 20]
```

Ripasso lista: Una lista è una sequenza di oggetti modificabile, ossia posso modificare un elemento all'interno della lista, mantenendone la giusta relazione.

Ripasso for:

```
for n in expr:
        corpo
```

Il for valuta expr, determinando un oggetto s.
0.1 - Se s non è una sequenza eccezione
0.2 - Mette via un riferimento a questo oggetto s, sequenza che controlla il for. Usa quel riferimento per controllare il for

Questo spiega perché in

```
A = 3
for i in range(A):
        print(i)
        A = 10
print(A)
```

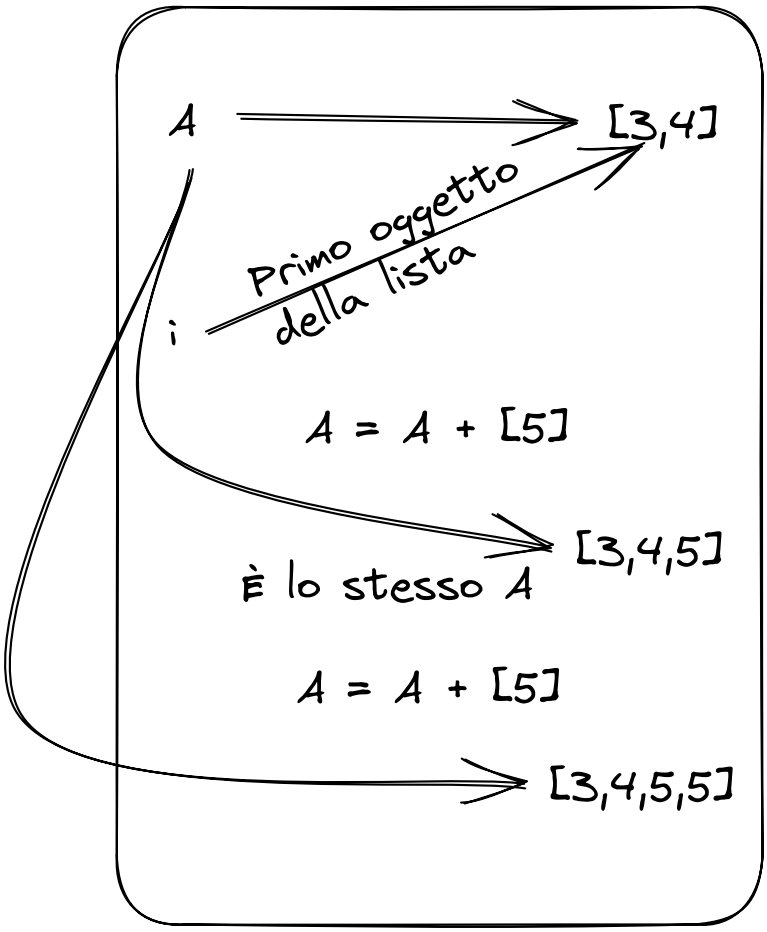
In questo codice, il valore di A viene cambiato ma solamente dopo aver congelato il valore $\mathrm{range}(\mathrm{A})$ (restando costantemente 3) quindi, quello che viene stampato è:

```
0
1
2
10
```

Che succede invece in:

```
A = [3,4]
for i in A:
        print (i)
```

```
        A = A + [5]
print(A)
```

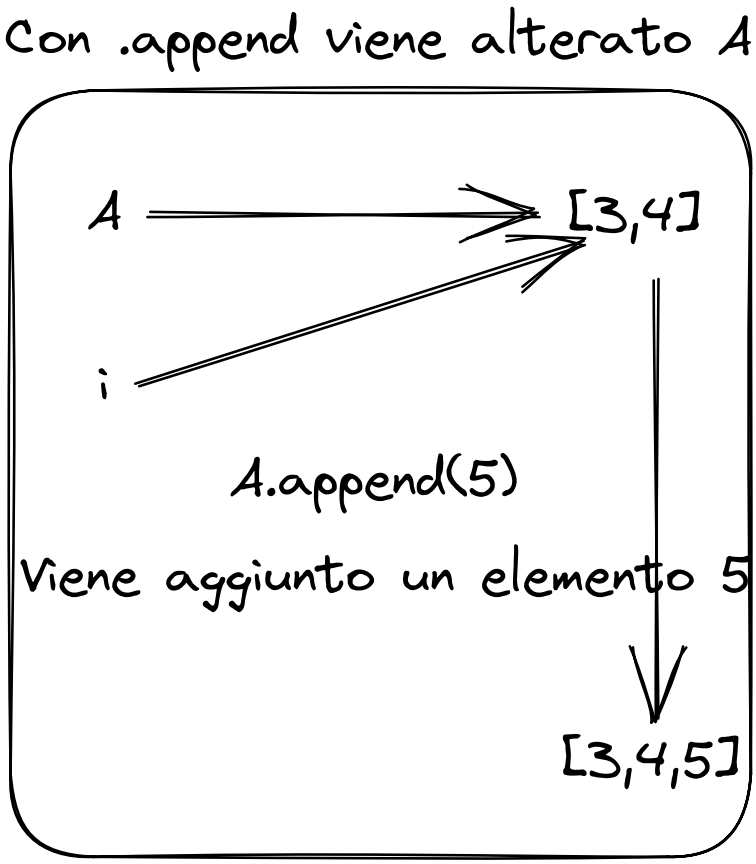


A furia di fare si ottiene una nuova lista con cose nuove
Quello che esce fuori è:

```
3
4
[3,4,5,5]
```

Nuovo codice:

```
A = [3,4]
for i in A:
        print (i)
        A.append(5)
print(A)
```



Quello che viene stampato è:

```
3
4
5
5
5
[...]
```

E il programma non finisce più

Assegnamento Aumentato: Durante while, spesso si è trovato $\text{A} = \text{A} + 1$ oppure $\text{Res} = \text{Res} + \text{Val}$
Python non ha auto incremento/ auto decremento, ma ha gli assegnamenti aumentati.
$\text{A} = \text{A} + 1 \Leftrightarrow \text{A} \mathrel{+}= 1$ e quindi $\text{Res} = \text{Res} + \text{Val} \Leftrightarrow \text{Res} \mathrel{+}= \text{Val}$
Come già visto, il $+$ dipende dal tipo legato ad A, non ha un significato a sé stante.

```
>>> s = 10
>>> s += 20
>>> print(s)
30
>>> T = (10, 20)
>>> T += (30,)
>>> print(T)
(10, 20, 30)
```

Ce ne sono moltissimi di Assegnamenti Aumtentati: $+=, -=, *=, /=, //=, \%=, **=$ *Per le cose basi, si tende a preferirle a quelle estese*

Non sono esattamente la stessa cosa: infatti ci sono tre grosse differenze:

- Se possibile, l'operazione avviene "in place" (con la modifica dell'oggetto senza la creazione in un altro) *Cosa più importante per il corso*

```
L = [10, 20] #Qui la lista viene modificata
L += [30]
#È diversa da
L = [10, 20] #Qui viene creata un'altra lista
L = L + [30]
#La differenza viene vista direttamente dall'id con
id(L)
```

- Il LHS (parte sinistra dell'assegnamento) è valutato prima della parte destra RHS

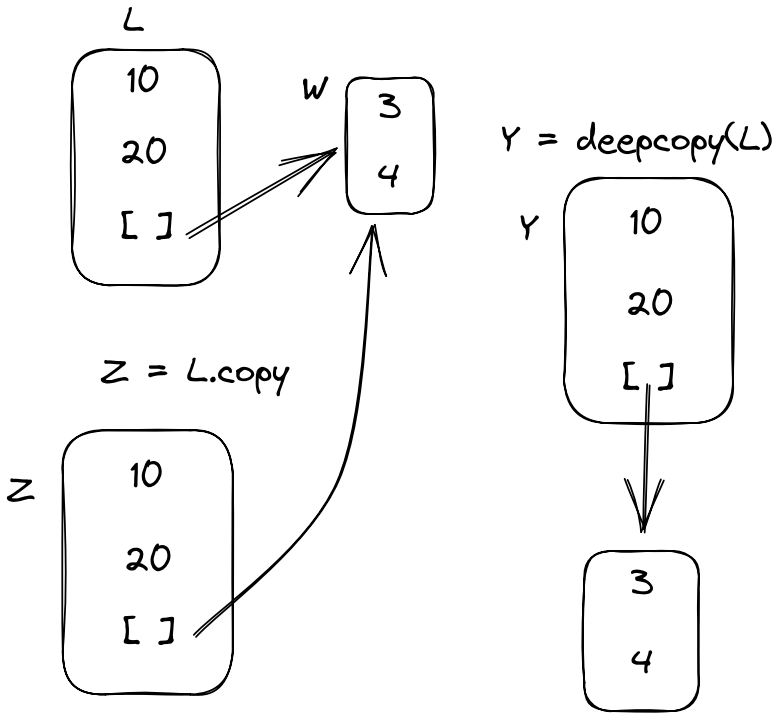
```
X = 3 + 5 + 9 #Viene valutata prima parte destra
X += 39 + 7 #Viene valutata prima la parte sinistra, qui è coerente
L[f(i)] += 29 #Qui non è coerente
L[f(i)] += f(i) #Qui ci sono ancora più effetti collaterali
```

- LHS è valutato una volta soltanto
  Infatti se fosse davvero una abbreviazione si avrebbe la valutazione due volte, ma questo non è il caso

Altri metodi sulle liste:
- $\mathrm{L.remove(x)}$ rimuove da L il primo elemento che ha valore x, da errore se non presente
- $\mathrm{L.copy()}$ restituisce una copia piatta di L, come fare $\mathrm{from\ copy\ import\ copy \setminus copy(L)}$, il primo per le liste e il secondo per tutti gli oggetti. Oppure sempre da copy possiamo importare il modulo $\mathrm{deep\ copy}$
- $\mathrm{L.reverse(L)}$ rovescia L ma non è la stessa cosa di $\mathrm{L = L[::-1]}$. Nel primo l'oggetto non è cambiato, mentre il secondo è un nuovo oggetto rovesciato.
- $\mathrm{L.pop(i)}$ fa due cose, prima restituisce l'elemento ad indice i, poi rimuove l'elemento a quella posizione. Se non si mette indice, restituisce l'ultimo elemento e poi lo rimuove
- $\mathrm{L.index(el)}$ restituisce il primo indice i per cui $\mathrm{L[i] == el}$
- $\mathrm{L.count(el)}$ restituisce quante volte compare el

Esempio:

```
W = [3, 4]
L = [10, 20, W]
Z = L.copy()
Y = deepcopy(L)
```



La sostanziale differenza è che con copy *crea un riferimento alla stessa lista, mentre con* $\mathrm{deepcopy}$ *crea da zero tutto quello precedente, anche le sottoliste*

Non tutte le operazioni/metodi hanno costo unitario (per esempio $\mathrm{x\ in\ S}$, dipende da dove si trova x, oppure è proporzionale a $\mathrm{len(S)}$)

L'operazione $\mathrm{del\ L[i]}$ costa. Infatti data una lista $[10, 20, 30, 40, 50]$ se si elimina l'elemento con indice i viene cancellato e tutti vengono traslati a indice precedente (per assurdo costa di più cancellare un elemento ad indice prima che dopo)
La stessa cosa anche con $\mathrm{L.insert(x)}$.
Entrambe sono proporzionali a $\mathrm{len(lista)}$-i.
Fa eccezione $\mathrm{L.append(e)}$ che ha costo unitario (su una sequenza di append), ossia non dipende dalla lunghezza di L.

Esercizio:
Data una sequenza S di numeri interi ordinata e crescente, viene dato un intero x. Inserire x in S, mantenendo ordine (da realizzare "in place").

```
def insortL(L, x):
        i = 0
        while i < len(L) and L[i] < x: #Tempo lineare
                i += 1
        L.insert(i-1, x) #Tempo lineare
        return L

def ins_ord_b(L, el):
        high = len(L)
        low = 0
        while low < high: #Tempo logaritmico
                med = (low + high) // 2
                if el < L[med]:
                        high = med
                else:
                        low = med + 1
        L.insert(high, el) #Tempo lineare che sovrasta quello lineare
        return L
```

```
L = [10, 20, 30]
```

In sintesi, non tutti gli algoritmi si equivalgono e non tutti sono efficienti allo stesso modo.

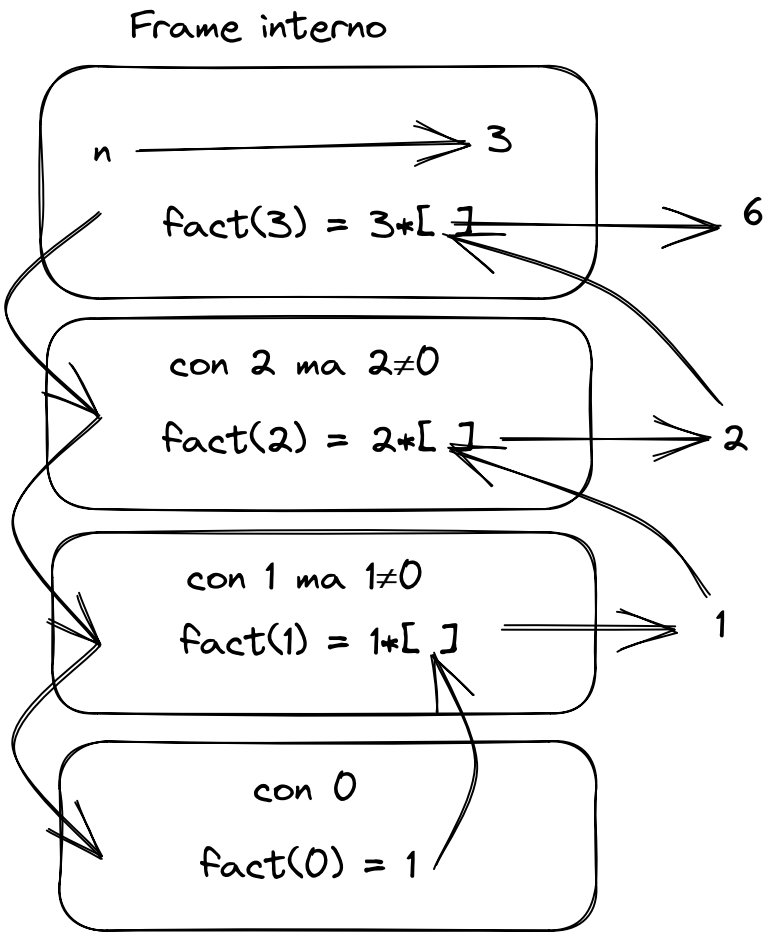
Definizione di Funzione Ricorsiva: È una funzione che nel corpo richiami sé stessa. (per esempio $n!$)

Esempio:

```
def lung(L):
        res = 0
        for e in L:
                res += 1
        return res

def lr(L):
        if L == []:
                return 0
        else:
                return 1 + lr(L[1:]) #Chiamata a funzione stessa
```

Sempre sullo stesso fattoriale:

```
def fact(n):
        if n == 0:
                return 1
        else:
                return n*fact(n-1) #Chiamata a funzione stessa
```



In generale mi servono due ingredienti affinché funzioni:

- Un caos base nella quale non vi sono chiamate ricorsive;
- Casi ricorsivi (L'importante è che la sequenza di chiamate si avvicini al caso base)

Affinché il tutto abbia senso, ogni elemento deve eventualmente capitare nel caso base, tutto questo spetta al programmatore.

La Ricorsione come l'Iterazione esprimono una ripetizione.

Esempio Iterazione:

```
n = 10
for i in range(n): #(1, 2, 3) condensati
        print(i)
```

Abbiamo corpo dell'iterazione (1), modo per passare da una iterazione alla seguente (2) e abbiamo un criterio di terminazione (3).

Lo stesso succederebbe se scrivessimo con la iterazione indeterminata:

```
i = 0
n = 10
while i < n:   #(3)
        print(i)   #(1)
        i += 1     #(2)
```

Lo si può fare anche con la ricorsione:

```
def rip_s(n):
        if x == 10:     #(3)
                return
        else:
                print(x)    #(1)
                rip_s(x+1)  #(2)
```

Non c'è un secondo return perché ci interessano soltanto gli effetti collaterali, ossia che stampi qualcosa e basta. *Metterlo concettualmente è sbagliato, anche perché restituirebbe* None

Sono modi diversi ma sono del tutto analoghi:

1. Abbiamo quello che vogliamo ripetere: $\mathrm{print(x)}$
2. Abbiamo come passare da un caso al successivo: il caso ricorsivo $\mathrm{rip\_s(x+1)}$
3. Abbiamo una condizione di terminazione chiamato caso base: $\mathrm{if\ x == 10: return}$

I due modi, dal punto di vista, sono effettivamente equivalenti.

Supponiamo di avere una funzione che si può calcolare solo con il while, allora la si può scrivere con un modo ricorsivo e viceversa (Ricorsione $\Leftrightarrow$ while)
Tuttavia ci sono anche dei casi in cui una forma è più intuitiva e semplice dell'altra (Con alcuni algoritmi)

Algoritmo di Euclide (Ricorsivo):

```
def MCDr(a, b):
        if b == 0:
                return a
        MCDr(a, a%b)
```

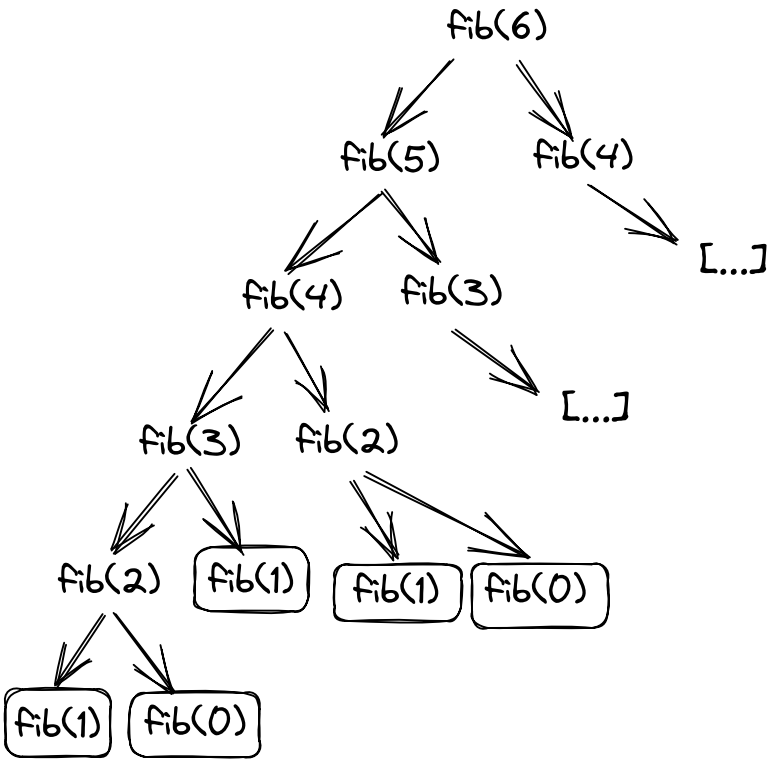
Test di Palindromicità (Esercizio di un laboratorio):

```python
def pali(s): #Senza ricorsività
        return s == s[::-1]

def palr(s): #Con ricorsività
        if len(s) <= 1:   #Caso base con sequenze lunghe 0/1 caratteri
                return True
        if s[0] != s[1]:
                return False
        return palr(s[1:-1])
```

Funzione Fibonacci: ($f_0 = 0,\ f_1 = 1 \Rightarrow f_{n+1} = f_n + f_{n-1}$)

```python
def fib(n): #n >= 0 #Ricorsiva e più lenta
        if n < 2:
                return n
        return fib(n-1) + fib(n-2)

def fibi(n): #Iterativa e più veloce
        if n < 2:
                return n
        a, b = 1, 0
        while n > 1:
                a, b = a + b, a
                n -= 1
        return a
```

Perché è così inefficiente? Proviamo con $\mathrm{fib}(6)$



In questo modo non faccio altro che creare un albero binario che mi va a calcolare tantissime funzioni per niente, in quanto la macchina Python non si rende conto che va a calcolare cose che ha già calcolato.

È possibile quindi fare una funzione ricorsiva senza dover fare il "pastrocchio" della funzione prima:

```python
def fibc(n):
        #Restituisce la coppia fib(n) e fib(n-1)
        if n < 2:
```

```
                return n, 0
        a, b == fibc(n-1)
        return a + b, a
```

Ricerca lineare (Ricorsiva):

```
def ricli(S, e):           #Fatto in maniera Iterativa
        for i in S:
                if x == e:
                        return True
        return False

def ricl(S, e):            #Fatto in maniera Ricorsiva
        'Ricerca lineare di e nella sequenza S'
        if len(S) == 0:
                return False
        if S[0] == e:
                return True
        return ricl(S[1:], e)
```

La stessa funzione ma vogliamo l'indice:

```
#Fatto in modo Iterativo

def ricli(S, e):
        for i in range(len(S)):
                if S[i] == e:
                        return i
        return -1

#Fatto in modo Ricorsivo

def ricl_aux(S, e:
        'Ricerca lineare di e nella sequenza S'
        if len(S) == 0:
                return False, -1
        if S[0] == e:
                return True, i
        return ricl_aux(S[1:], e, i+1)

def ricl(S, e):
        return ricl(S, e, 0)

#Oppure

def ricl(S, e):
        'Ricerca lineare di e nella sequenza S'

        def ricl_aux(S, e, i = 0):
                if len(S) == 0:
                        return False, -1
                if S[0] == e:
                        return True, i
                return ricl_aux(S[1:], e, i+1)

        return ricl(S, e, 0)
```

Così definisco una funzione locale solo per $\mathrm{ricl}(\mathrm{S}, \mathrm{e})$.
Oppure sfruttando una cosa non ancora vista:

```
def ricl2(S, e, i = 0): #Sto definendo un parametro default.
        #Quindi posso chiamare la funzione con 2 parametri in quanto me lo mette da solo a
0
        if len(S) == 0:
                return False, -1
        if S[0] == e:
                return True, i
        return ricl2(S[1:], e, i+1)

#Oppure partendo dal fondo

def riclf(S, e):
        'Ricerca lineare di e nella sequenza S'
        if len(S) == 0:
                return -1
        if S[-1] == e:
                return len(S)-1
        return ricl_aux(S[:-1], e)
```

Abbiamo appena visto 5 modi diversi (di cui 4 ricorsivi) per far vedere che ci sono diverse scuole di pensiero. Tutto questo per imparare a fare le stesse cose in maniere diversi, disastrose o eleganti.

Esercizi:

1. Funzione ricorsiva che restituisce una copia rovesciata di una lista $\mathrm{L} \rightarrow \mathrm{L}[:\text{-}1]$
2. Funzione ricorsiva che modifica "in place" una lista L (la rovescia)

Soluzione:

```
def rev(L):
        'Restituisce una copia di L rovesciata in maniera ricorsiva'
        if len(L) <= 1:
                return L[::]
        return rev(L[1:])+[L[0]] #Potevo fare anche return [L[-1]]+rev([:-1])

def rev2(L):
        if len(L) <= 1:
                return L[::]
        return [L[-1]]+rev(L[1:-1])+[L[0]] #Esattamente come con le parole Palindrome
```

```
def revip(L):
        'Fa la stessa cosa come prima ma "in place"'

        def aux(W, l, h):
                'Fa tutto il lavoro tra l e h-1'
                if h - l <= 1:
                        return None #Non devo fare niente
                W[l], W[h-1] = W[h-1], W[l]
                aux(W, l+1, h-1)

        aux(L, 0, len(L))
```

Si può fare anche la ricerca binaria con la ricorsione (infatti è uno di quelli che si presta meglio per la ricorsione):

```
def ric_bin_r(S, el):
        if len(S) == 0:
                return False
        if len(S) == 1:
```

```
                return S[0] == el

        med = len(S) // 2

        if S[med] == el:
                return True
        if el < S[med]:
                return ric_bin_r(S[:med], el)       #high = med
        else:                                     #el > S[med]
                return ric_bin_r(S[:med + 1], el)   #low = med + 1
```

Complessità Computazionale

Domanda fondamentale: *Quante risorse sono necessarie per eseguire un programma?*
Che può essere *Quanta energia? / Quanta corrente?*

Qui è: *Tempo di Calcolo* e *Spazio di Memorizzazione ausiliario*. (Nel senso di *quanto spazio nascosto uso all'interno del programma per fare i conti*)

Noi ci concentriamo soprattuto sul tempo.

Come già detto, non ci interessa un tempo assoluto, perché i dati assoluti sono dipendenti da una quantità incredibile che cambia da molte cose (OS, carico sul sistema, ...)
Tempo assoluto è tempo troppo variabile per fare una teoria.

A noi ci interessa il tempo di calcolo come $\text{Dipendenza Funzionale}$, ossia come varia al variare di determinati parametri *(Come il consumo di risorse varia al variare di alcuni parametri di dati)*. Questi parametri vengono chiamati $\text{Dimensione dei dati}$

Abbiamo già visti degli esempi (parte sopra):

Nella scansione lineare:

```
def sl(S):
        for e in S:
                print(e)
```

Il tempo varia in maniera lineare rispetto alla lunghezza dei dati (qui di S).
Ossia $\text{Tempo di Calcolo lineare a len(S)}$. *Non interessa il tipo di dati, ma dalla dimensione*

In alcuni casi, come la ricerca lineare, non tutte le esecuzioni hanno lo stesso tempo (alcune di più altre di meno):

```
def sl(S, e):
        for t in S:
                if t == e:
                        return True
                else:
                        return False
```

Tempo di calcolo dipende anche dal fatto se e compare in S oppure no.
A noi però interessa solamente il caso pessimo / peggiore.
Fissata una dimensione ci interessa determinare il caso peggiore. *In questo caso è come se non compare, quindi il tempo di calcolo collassa con quello della scansione lineare e diventa appunto lineare alla dimensione dei dati*.

Quello di cui tratteremo è la $\text{Valutazione del Limite Superiore (Uppedband) di Complessità}$
Abbiamo un algoritmo che risolve un problema e vogliamo capire quanto ci mette.

Limite superiore = caso peggiore

Un altro capitolo della Complessità Computazionale che al posto di prendere un algoritmo e studiarne il tempo, si preoccupa di rispondere alla domanda: *determinare la complessità intrinseca di un problema*. (Che non tratteremo)
Si limita a studiare il minimo delle risorse che un algoritmo spende per ogni situazione.

Cosa conto per determinare il tempo di calcolo?
Nell'analisi contiamo i passi di calcolo, cioè unità convenzionali di tempo che non dipendono dalla dimensione dei dati (Concetto di Base).
Quali sono i possibili passi?

1. Un'operazione tra due float può essere vista come un'unità in quanto non dipende dalla dimensione dei numeri.
2. Un'operazione tra due bool
3. Un'operazione tra due interi piccoli (in Python ci sono degli interi arbitrariamente grande, quindi più tempo)
4. Un'operazione tra caratteri (Stringhe da un elemento)
5. Un assegnamento (La valutazione potrebbe non essere a tempo costante)
6. Un return (Sempre senza valutazione)
7. Un print (Sempre senza valutazione)

Esempio: Ricerca lineare con while e studio di rlw:

```
def rlw(S, e):
        i = 0                                   #1 passo (Assegnazione)
        while i < len(S) and e != S[i]:         #3 passi (I due confronti e l'and) n + 1 volte
                i += 1                              #2 passi (Valutazione e Assegnazione) n
volte
        return i < len(S)                       #2 passi (Return e valutazione)
```

- Dimensione: $\mathrm{len(S)} = n$
- Caso peggiore: $\mathrm{e} \notin \mathrm{S}$
- Passi nel caso peggiore: $\underbrace{3}_{\text{Quello Fuori}} + \underbrace{3(n+1)}_{\text{Guardia}} + \underbrace{2n}_{\text{Corpo}} = 5n + 6$

Se ho fatto tutto bene, cronometrando il tempo impiegato ottengo che il tempo segue la funzione $f(t) = 5t + 6$

Cosa posso supporre che non sia un passo di calcolo:

1. $\mathrm{x\ in\ S}$ (lineare nella lunghezza di S)
2. Un operazione tra interi grandi (gli interi in Python sono illimitati, quindi devo tenere conto al numero delle cifre, di cui il tempo è lineare)
3. Confrontare due sequenze ($\mathrm{L} = \mathrm{LL}$ va a confrontare due liste elemento per elemento)
4. Chiamare una funzione (la chiamata in sé è un passo, ma poi viene eseguita la funzione, che ha un suo tempo)
5. I comandi composti (in cui bisogna valutare la parte interna)

Un'altra cosa che mi interessa nell'analisi non è la formula esplicita, ma il suo ordine di grandezza, ossia è $\mathrm{Asintotica}$, ossia non vado a considerare le costanti che sono in gioco.
Per questo motivo si utilizza la notazione di $O()$ grande.

Definizione di $O()$: Siano $f, g : \mathrm{Dimensione\ dei\ dati} \to \mathbb{N}$. Diciamo che
$f = O(g) \Leftrightarrow \exists c, b, n_0 : \forall n > n_0, f(n) \leq cg(n) + b$

Quindi continuando l'esempio si ha che:

- Dipendenza lineare: $O(n)$

Esercizio:

- $3n^3 = O(n^3) \Rightarrow \exists c, b, n_0 : 3n^3 \leq n^3 + b, \forall n > n_0$, mi basta prendere $c = 3, b = 0, , n_0 = 0$
  Questo per sottolineare che gli $o()$ non servono a niente, ma servono solo le costanti moltiplicative.
- $n\log(n) = O(n^2) \Rightarrow c = 1, n_0 = 1, b = 0$
  Le basi dei logaritmi è assolutamente inutile, perché è una costante che non serve

Per le costanti in informatica:
$k = 2^{10} = 1024 \sim 1000$
$M = 2^{20} \sim 1000000$

Nei metodi per le liste:
- Creazione della lista è lineare o costante
- Tutte le operazioni di concatenazione tra sequenze, aggiunte di valori e altro ancora, possiamo maggiorare con un coso della lunghezza della stringa
- La lunghezza è estratta a tempo costante, in quanto è fuori dalla lista in sé
- Ordinare la lista invece è $0(n \log n)$

Esercizio di Valutazione:

```
def foo(L, a):
        for i in range(len(L)):
                L.insert(0, a)          #Ci dice che deve essere una lista
        return L
```

- Capire cosa fa: Aggiunge in place inserisce alla posizione in indice 0 tante volte quanto è la lunghezza di L
- Dimensione: $\mathrm{len(L)} = \mathrm{n}$
- Caso peggiore: Tutte le istanze sono istanze del caso peggiore (Hanno lo stesso tempo di esecuzione)
- Passi nel caso peggiore: Per quanto riguarda il corpo, alla prima interazione costa n passi (insert su una lista lunga n, alla seconda costa $n+1$, alla terza $n+2$, finché all'ultima non costerà $2n-1$). Il costo globale è di conseguenza la somma di tutti i tempi $\underbrace{\left(\sum_{\ell=0}^{n} n + \ell\right)}_{\text{for}} + \underbrace{1}_{\text{return}} = nn + \sum_{\ell=0}^{n-1} \ell + 1 = n^2 + \frac{n(n-1)}{2} + 1$
  La seconda possibilità è: Poiché il corpo è $O(n)$ ma lo ripeto n volte ottengo $nO(n) = O(n^2)$

Esercizio 2:

```
def primik_2(S, k):
        T = []
        for i in range(k):
                m = max(S)
                S.remove(m)
                T.append(m)
        return T
```

- Capire cosa fa: Data una lista, estrae il massimo dalla lista e poi lo aggiunge alla fine
- Dimensione: $\mathrm{n} = \mathrm{len(S)}, \mathrm{k} \leq \mathrm{len(S)}$ *Parametri che influenzano l'esercizio*
- Caso peggiore: *(Dopo aver fissato n e k)*. Tutte le istanze sono equivalenti e quindi quelle peggiori
- Passi nel caso peggiore:
  Iterazione 1: $\underbrace{n}_{\max} + \underbrace{n}_{\text{remove}} + \underbrace{1}_{\text{append}}$
  Iterazione 2: $\underbrace{n-1}_{\max} + \underbrace{n-1}_{\text{remove}} + \underbrace{1}_{\text{append}}$

Interazione k: $\underbrace{n-k+1}_{\text{max}} + \underbrace{n-k+1}_{\text{remove}} + \underbrace{1}_{\text{append}}$

Quindi il costo totale: $\underbrace{\sum_{i=1}^{k}(2n-2i+2+1)}_{\text{for}} + \underbrace{1}_{\text{L=[]}} = 2kn - 2\sum_{i=1}^{k} i + 3k = 2kn - 2\frac{k(k+1)}{2} + 3k = O(kn)$

Esercizio della ricerca binaria, ricorsiva:

```
def ric_ip_aux(L, el, low, high):
        if low > high:
                return False
        med = (high + low) // 2
        if L[med] == el:
                return True
        if el < L[med]:
                return ric_ip_aux(L, el, low, med)
        else: #el >= [med]:
                return ric_ip_aux(L, el, med + 1, high)
```

- Capire cosa fa: Ricerca Binaria ricorsiva
- Dimensione: $\mathrm{n} = \mathrm{len(L)}$
- Caso peggiore:
- Passi nel caso peggiore:
  Sia $T(n)$ la complessità della funzione per L di lunghezza n
  Sia $T(1)$ il caso base, quindi ha un costo costante c
  Sia $T(n)$ un caso non base, ha un po' di passi costanti b più $T(\frac{n}{2})$
  Quindi $\begin{cases} T(1) = c \\ T(n) = d + T(\frac{n}{2}) \end{cases}$
  In questo caso, si prova ad espandere la relazione, in modo da poter trovare un caso particolare
  Supponiamo $n = 2^k$ e proviamo a vedere cosa si ottiene: $T(2^k) = d + T(2^{k-1})$, così posso continuare ad usare la relazione di ricorrenza ossia $T(2^k) = d + T(2^{k-1}) = 2d + T(2^{k-2}) = 3d + T(2^{k-3})$, ma in generale: $id + T(2^{k-i})$, ci si ferma quando $i = k$, in particolare $kd + T(2^0) = kd + c \Rightarrow k = d\log_2 n + c \Rightarrow O(\log_2 n)$.
  Se non è una potenza di due, posso prendere quella più grande, tanto sempre in $O()$ sono, quindi resta sempre logaritmica.

Esercizio, ricerca binaria con slice:

```
def ric_bin_r(S, el):
        if len(S) == 0:
                return False
        if len(S) == 1:
                return S[0] == el
        med = len(S) // 2
        if S[med] == el:
                return True
        if el < S[med]:
                return ric_bin_r(S[:med], el)
        else:
                return ric_bin_r(S[med + 1:], el)
```

- Tutti i dati precedenti restano lo stesso tranne il numero di passi
- Rifacciamo lo stesso calcolo per il numero di passi.
  $\begin{cases} S(1) = c \\ S(n) = d + S(\frac{n}{2}) + O(\frac{n}{2}) \end{cases}$, dove l'ultimo $\frac{n}{2}$ è dato dal costo di costruzione di una slice.
  Prendiamo $n = 2^k$, quindi diventa $S(2^k) = d + S(2^{k-1}) + O(2^{k-1}) = 2d + S(2^{k-2}) + O(2^{k-1}) + O(2^{k-2})$, quindi

iterando si ottiene che $id + S(2^{k-1}) + O\left(\sum_{j=1}^{i} 2^{k-j}\right) = kd + S(1) + O\left(\sum_{j=1}^{k} 2^{k-j}\right) = kd + S(1) + O(2^k - 1) =$
$= d\log n + c + O(2^{n-1}) = O(n)$

Quale è la complessità dello spazio? È la quantità di spazio aggiuntivo che utilizzo per portare a termine la mia funzione.

Sia $Sp(n)$ lo spazio **di lavoro** utilizzato per completare l'esecuzione
Lo spazio nel caso base $Sp(1) = 0$, lo spazio nel caso n,
$$\begin{cases} Sp(1) = 0 \\ Sp(n) = 1 + Sp(\frac{n}{2}) + \underbrace{O(\frac{n}{2})}_{\text{Creazione della slice (Trascurabile, può essere anche } \frac{n}{2})} + \underbrace{f}_{\text{frame che vado a costruire}} \end{cases}$$
La formula del consumo dello spazio ha la stessa forma di quella del tempo
Questa formula diventa quindi $O(\log n) + O(n)$, quindi la formula

Algoritmi numerici: (Come $\mathcal{MCD}$, Euclide, Fibonacci...)
Dobbiamo chiederci "qual è la dimensione dei dati per gli algoritmi numerici?"
Pensiamo all'algoritmo di somma $32345 + 12372 = 44717$. La sua complessità dipende da numero di cifre.
Anche in generale la dimensione dei dati in un algoritmo numerico è dato dal numero di cifre del dato.
Quindi se il dato è n, la dimensione del dato è $\log n$, si assume $\log_2$ ma non ha importanza.
La dipendenza funzionale del tempo è in funzione del tempo

Prendiamo l'algoritmo di $\mathcal{MCD}$:

```
def primo(n):
        for k in range(2, n-1):
                if n % k == 0:
                        return False
        return True
```

Numero di passi: Il corpo costa un'operazione, il corpo viene eseguito $n - 2$ volte, dunque si eseguono $n - 2 = O(n)$ operazioni elementari *(supposto siano tutti interi piccoli)*.
Poiché però la dimensione dei dati va presa come $\log n$, di conseguenza l'algoritmo di $\mathcal{MCD}$ risulta esponenziale rispetto a $O(n)$ in quanto $n = 2^{\log n}$

Complessità negli algoritmi numerici:
La dimensione che dovrebbero avere è la dimensione di spazio per rappresentare il numero di cifre (ossia il logaritmo del numero del valore). Esempio $\mathcal{MCD}$, test di primarità e logaritmo di Euclide.

Sugli algoritmi numerici c'è un altro problema, oltre a capire la giusta dimensione del dato: Altro problema è dato da numeri troppo grandi. Il primo più grande noto è $2^{89589931} - 1$, nell'ordine di $10^{24862048}$
Per interagire con i numeri troppo grandi ci sono modi diversi a seconda del linguaggio
In Python, dati due numeri a e b e $k = \log(min(a, b))$, allora $a + b$ costa circa k ed è dell'ordine di k, mentre $a * b, a\%b, a/b$ costa k^2, ossia un logaritmo quadrato del valore del numero.
Nel caso del test di primarità ovvio: 2^k passi, ma ciascun passo costa ancora 2^k. Quindi diventa $O(\log^2(k) \cdot 2^k)$
Nel caso dell'algoritmo di Euclide $O(k)$ considerando che ogni operazione costa $O(k^2)$ diventa $O(k^3)$
Nella primarità, poiché è esponenziale è estremamente costoso.
In crittografia, conoscere numeri primi è estremamente importante per gli stessi motivi di Algebra.
Per anni si è chiesto quale sia la complessità dell'algoritmo del test di Primarità, del quale non c'era risposta.
C'erano di limiti inferiori ovvi (lineari in k), ma non c'erano degli upper bound.
Negli anni '70 sono stati proposti degli algoritmi efficienti (Miller-Rabin, algoritmi polinomiali (Da Riemann)). Nel 2002 si è trovato un algoritmo di test primarità, polinomiale che non dipende dall'ipotesi di Riemann.

L'algoritmo AKS (2002), test di primarità è $O(k^{12} \cdot \log^p(k^1 2))$ ora è stata diminuita a $O(k^6 \cdot \log^a(k^6))$.
Problema della fattorizzazione: se n non è primo determina una scomposizione. È un problema è ancora aperto.
Ci sono degli algoritmi ma sono comunque esponenziali, quindi come quello ovvio.

Ma non sappiamo effettivamente se è un limite inferiore di efficienza.
Questo è vero problema della fattorizzazione.
I protocolli sono sicuri perché è difficilissimo fattorizzare numeri troppo grandi. Ma non sappiamo se è veramente così.
Forse potrebbe esiste un algoritmo polinomiale per risolvere.

Esistono anche altri modelli di calcolo come quello quantistico, che sfrutta le sovrapposizioni di eventi quantici, capace di semplificare infinitamente problemi quasi impossibili con i calcolatori standard.

Problema dell'Ordinamento (Sorting):
Abbiamo una lista di oggetti (ordinabili - c'è una relazione d'ordine) e vogliamo determinarne una permutazione ordinata.
Mettiamo un vincolo sulle operazioni: *Le operazioni devono essere basate su confronti*.
Alcuni algoritmi di ordinamento sono alla base di tanti dispositivi di oggi. Vediamone tre:

Insertion Sort:
Sia una sequenza numerica: $9 \quad 1 \quad 8 \quad 7 \quad 2 \quad 5$.
Fisso il primo elemento $9 \mid 1 \quad 8 \quad 7 \quad 2 \quad 5$ e lo confronto con il secondo. Poiché $1 < 9 \Rightarrow 1 \quad 9 \mid 8 \quad 7 \quad 2 \quad 5$. La parte a sinistra della barra è ordinata. Ho creato una *invarianza*: "Fino ad un certo indice i, la sequenza è ordinata"
Cioè, data una sequenza S è invariante e $S[:i]$ è ordinata dove i varia da 0 a $\text{len}(S)$
Quindi poi diventerà $1 \quad 8 \quad 9 \mid 7 \quad 2 \quad 5$, fino ad arrivare alla sequenza completamente ordinata.
Questo tipo di ordinamento si chiama **Insertion Sort**:

```
def Insort(L):
        for i in range(L):     #L[:i] è ordinata
                temp = L[i]        #Libero il posto di L[i]
                k = i - 1
                while k >= 0 and L[k] > temp:
                        L[k+1] = L[k]  #Sposta a destra
                        k -= 1
                L[k+1] = tempo
```

In questo modo si possono fare anche su sequenze non modificabili.

Analisi della Complessità: il caso peggiore è quando gli elementi in fondo sono i più piccoli.
Lasciamo un attimo il for *esterno*.
Il while interno viene eseguito i volte, quindi i confronti
Ossia $0 + 1 + \ldots + n = O(n^2)$ per la formula di Gauss, è un algoritmo quadratico nel caso peggiore.

Teorema: Il problema dell'ordinamento di confronti ha un limite inferiore di complessità di $O(n \cdot \log(n))$ confronti.

Quick Sort: È stato creato da Tony Hoare (1961) e funziona con la scelta, del primo elemento e della separazione degli elementi più grandi o più piccoli dei successivi.

```
def QS(L):
        if len(L) < 2:
                return L
        pivot = L[0]
        L1 = []           #I due mucchi: L1 più piccoli, L2 più grandi
        L2 = []
        for e in L[1:]:
                if e < pivot:
                        L1.append(e)
                else:
                        L2.append(e)
        return QL(L1) + [pivot] + QL(l2)
```

La complessità computazionale tra questa e quella precedente è la stessa $O(n^2)$ nel caso peggiore.
$t_Q(n) = c + \underbrace{O(n)}_{\text{Divisione dei gruppi}} + T_Q(k-1) + T_Q(n-k+1)$

Per calcolo complessità:
- Quali sono i dati e la loro dimensione;
- Quale è il caso peggiore
- Effettuare un'analisi asintotica

Normalmente la funzione da analizzare è $T(n)$ ossia il tempo in relazione al alla dimensione dei dati.

Se ho una costante k e $f(n) = O(g(n)) \Rightarrow k \cdot f(n) = O(g(n))$
Se ho due funzioni $f(n) = O(g(n))$ e $d(n) = O(h(n)) \Rightarrow f(n) + d(n) = O(g(n) + h(n)) = O(max\{g(n), h(n)\})$ e $f(n) \cdot d(n) = O(g(n) \cdot h(n))$

L'ordine degli $O(n)$:
$n^n >> n! >> c^n >> 2^n >> n^3 >> n^2 >> n\log(n) >> n >> \sqrt{n} >> \log(n) >> 1$

Visto che è una valutazione asintotica, conta soltanto quando n è grande

Ci sono altri vari algoritmi naive per l'ordinamento, ma non sono altre che modifiche di quelli iniziali, ma tutti hanno la stessa caratteristica: sono $O(n^2)$ nel caso peggiore e in tante delle possibili permutazioni.

Ponendo $n_1 = \text{len(L1)}, n_2 = \text{len(L2)}$ con $n_1, n_2 < n$
$QS(n) = \underbrace{O(n-1)}_{\text{Partition}} + QS(n_1) + QS(n_2) + \underbrace{O(n)}_{\text{Concatenazione finale}}$

Caso Peggiore: Ogni volta il Pivot è scelto in modo che uno uno tra $L1$ e $L2$ è vuota.
Per esempio se Pivot=$L[0]$ se L è ordinata al contrario
Quindi $QS(n) = \underbrace{O(n)}_{\text{Concatenazione finale}} + \underbrace{QS(0)}_{O(1)} + QS(n-1) \Rightarrow O(n) + O(n-1) = O(n) + O(n) + O(n-2) = \dots$
In generale $QS(n) = \underbrace{O(n) + \dots + O(n)}_{i \text{ volte}} + O(n-i)$. Quando $i = n \Rightarrow \underbrace{O(n) + \dots + O(n)}_{n \text{ volte}} = n \cdot O(n) = O(n^2)$
Ossia è alla pari degli altri sistemi di ordinamento

Caso Migliore: Pivot scelto in modo tale che $n_1 = n_2 = n/2$
Ipotizziamo $n = 2^k$
Allora $QS(2^k) = O(2^k) + QS(2^{k-1}) + QS(2^{k-1}) w O(2^k) + 2QS(2^{k-1}) = O(2^k) + 2O(2^{k-1}) + 2QS(2^{k-2}) =$
$= O(2^k) + 2(O(2^k) + QS(2^{k-2})) = O(2^k) + O(2^k) + 2^2 QS(2^{k-2}) = \dots$
Il caso generale diventa $\underbrace{O(2^k) + \dots + O(2^k)}_{i \text{ volte}} + 2^i QS(2^{k-i})$
L'ultimo caso diventa $\underbrace{O(2^k) + \dots + O(2^k)}_{k \text{ volte}} + O(1) = k \cdot O(2^k) \overset{k=\log(n)}{=\!=} O(n\log(n))$

Teorema:
Supponiamo equiprobabili le $n!$ permutazioni di una sequenza S in cui la lunghezza di S è n
1. Il tempo medio di Insertion Sort (e di conseguenza tutti gli alti ordinamenti naive) è $O(n^2)$
2. Il tempo medio di Quick Sort è $O(n\log(n))$

Merge Sort: Ordinamento per fusione di Von Newmann (1945)
L'essenza è una funzione che ha lo scopo di unire le liste, ma che abbiamo già fatto in laboratorio
Date due sequenze, costruirne una ordinata.
$\text{merge}(L1,\ L2)$ con $L1, L2$ sequenze ordinate, restituisce una sequenza ordinata con gli elementi di $L1$ e di $L2$.

```
def merge(L1, L2):
        len1 = len(L1)
        len2 = len(L2)
        res = []
```

```
        i = j = 0
        while i < len1 and j < len2:
                if L1[i] <= L2[j]:
                        res.append(L1[i])
                        i += 1
                else:
                        res.append(L2[j])
                        j += 1
        if i == len1:
                res.extend(L2[j:])
        else:
                res.extend(L1[i:])
        return res
```

Per farlo nel migliore dei modi serve la ricorsione, con interazione diventa un problema.

```
def mergesort(L):
        if len(L) <= 1:
                return L
        mid = len(L) // 2
        L1 = mergesort(L[:mid])
        L2 = mergesort(L[mid:])
        return merge(L1,L2)
```

Analisi di complessità nel caso peggiore (Supponendo $n = 2^k$)
$MS(2^k) = 2MS(2^{k-1}) + O(2^k)$ che è la stessa equazione di complessità del caso migliore di Quicksort
Quindi la soluzione è $O(n \log(n))$
In realtà non c'è un caso peggiore, perché non è sensibile alla effettiva disposizione dei dati, quindi tutte le istanze si comportano nello stesso modo, quindi è un algoritmo ottimale

Altra caratteristica non ancora analizzata è la stabilità dell'algoritmo, che nel sort è data dalla presenza di elementi uguali. In particolare, è stabile se viene mantenuto l'ordinamento relativo degli elementi (ossia se ci sono degli stessi elementi uguali, viene rispettato l'ordine originario)

Master Theorem: $T(n) = \begin{cases} aT(n/b) + n^c & n > 1 \\ d & n = 1 \end{cases}$, dipende da:
1. $\log_b a < c \Rightarrow T(n) = O(n^c)$
2. $\log_b a = c \Rightarrow T(n) = O(n^c \log n)$
3. $\log_b a > c \Rightarrow T(n) = O(n^{\log_b a})$

Tipo di dato dei Dizionari

Problema dato dal problema delle sequenze
Data una stringa, contare quante lettere ci sono in una stringa

Proviamo a farlo con quello che abbiamo

```
'Contare le frequenze dei caratteri in una stringa'
LC = 'Nel mezzo del cammin di nostra vita'
LL = []
LF = []

for c in LC:
        if c not in LL:
                LL.append(c)
                LF.append(1)
        else:
```

```
                LF[LL.index(c)] += 1

print(LL)
print(LF)
```

Non si può fare meglio, inoltre ci sono problemi di efficienza e leggibilità
Infatti $.\mathrm{index}$ è lineare, quindi pago un prezzo lineare.
E se le lettere fossero usate come indici?

```
Freq = {}                  #Inizializzazione del Dizionario
for c in LC:
        if e not in Freq:
                Freq[c] = 1
        else:
                Freq[c] += 1
print Freq
```

Freq è un dizionario:

- Valore: Una corrispondenza (finita) tra chiavi e valori dove le chiavi sono oggetti di tipo non modificabile e i valori sono oggetti qualsiasi, quindi delle sequenze e sono modificabili *(Le liste possono essere viste come corrispondenze tra* $\mathrm{range}(\mathrm{len}(\mathrm{L}))$ *e oggetti)*
- Presentazione: $\{\}$ (dizionario vuoto) e con $\{k_1 : v_1,\ k_2, v_2,\ \ldots\ , k_n : v_n\}$
- Operazioni: Accesso all'elemento (Selezione) ($< \mathrm{Nome\ DIzionario} > [\mathrm{Chiave}]$), posso modificare le associazioni tramite assegnamento $< \mathrm{Nome\ Dizionario} > [< \mathrm{Chiave} >] =< \mathrm{Valore} >$. Per i dizionari non conta l'ordine con cui sono messi i valori, infatti l'uguaglianza è da essere considerata a meno dell'ordine. (Tuttavia non c'è un modo canonico per ottenere la chiave a partire dal valore)

Le chiavi devono essere tutte distinte, ma che succede se sono tutte uguali?
Mi prende solo l'ultima elencata alla posizione del primo elemento.

```
>>> D = {1:100, 2:200, 1:1000}
>>> D
{1:1000, 2:200}
```

Se associo un elemento del dizionario ad una chiave che non è presente all'interno del dizionario, l'azione che faccio equivale ad aggiungere un elemento in più

```
>>> D = {1:100, 2:200}
>>> D['Mao'] = True
>>> D
{1:100, 2:200, 'Mao':True}
```

Inoltre le chiavi devono poter essere scritte in Hash, altrimenti da errore (posso usare una tupla, ma non una lista)
Non posso usare oggetti modificabili o oggetti che hanno al loro interno oggetti modificabili

Nel dettaglio le operazioni sui dizionari (D nel nostro esempio):

- $D.\mathrm{len}$ numero di elementi nel dizionario;
- Selezioni con $D[a]$ come espressione (a destra di un assegnamento) restituisce il valore associato ad a, ma se non ci sono valori associati a a da errore (Nell'assegnazione crea una nuova chiave e vi associa il valore)
- Cancellazione $\mathrm{del}\ D[k]$, ma c'è una chiave soltanto soltanto;
- $D1 == D2$ restituisce True se sono presenti gli stessi item a meno dell'ordine
- $k\ \mathrm{in}\ D$ restituisce True se e solo se k è una chiave di D

- Si può fare il for che varia sulle chiavi di D (for e in D), ma come varia? (In ordine di inserimento, ossia lo stesso ordine con cui si presentano quando si chiede la stampa del dizionario - però la cosa migliore è immaginare che il dizionario sia ordinato secondo la macchina Python)
- Non ci sono né concatenazione né slice

L'esempio canonico di utilizzo dei dizionari è con quello delle frequenze:

Esempio di Fibonacci con memoization, cioè mi salvo i valori all'interno del dizionario:

```
Fib = {0:1, 1:1}

def fibo(k):
        if k in Fib:
                return Fib[k]
        Fib[k] = fibo(k-1) + fibo(k-2)
        return Fib[k]
```

Questa tecnica sfrutta valori complessi che mi sono già calcolato e che mi sono segnato

Confronto tra liste e dizionari: a livello di informazione sono la stessa identica cosa, ma sui dizionari ho l'accesso per chiave (più efficiente), mentre con le liste ho l'accesso per indice (devo scorrere per trovare l'indice a cui è associato l'elemento che voglio)

Ripasso con le liste:

```
L = [3, 0, 27, 0, 0, 9, -1]
for i in range(len(L)):
        if L[i] == 0:
                del L[i]
```

Questo for fa casino, perché cambia la lunghezza durante l'esecuzione, per questo non va mai usato

La stessa cosa per i dizionari:

```
D = {1:0, 2:3, 4:0}
for k in D:
        if D[k] == 0:
                del D[k]
```

Sui dizionari il divieto non viene dal prof (come nel caso delle liste), ma direttamente dalla macchina Python

Morale: Sulle strutture non si usa il for se la loro lunghezza cambia durante l'esecuzione.

Anche qui ci sono i metodi:

- Surrogato di append: .update:

```
>>> D = {1: 100, 2:200, 3:300}
>>> D.update({'a':10, 'b':20, 'c':30})
>>> D is D
True                #Inplace
```

- Visto che il for restituisce le chiavi, per avere i valori si usa il metodo .values(), mentre per le chiavi c'è il metodo .key() ma è sottointeso direttamente con il for, con .items() restituisce le coppie

```
for i in D.key() #D.values() oppure D.items()
```

Quello che i metodi restituiscono delle Viste dinamiche per spiegare ecco un esempio:

```
>>> E = {1:100, 2:200}
>>> val = E.values()
>>> val
dict_values([100, 200])
>>> E[3] = 300
>>> E
{1:100, 2:200, 3:300}
>>> val
dict_values([100, 200, 300])
```

La vista varia al variare di E. Questi metodi restituiscono una vista su quei tre componenti
Il metodo è chiamato solo una volta e in ogni momento il valore effettivo del Dizionario

È possibile anche congelare il dizionario attraverso una tupla:

```
#Dall'esempio precedente
>>> T = tuple(val)
>>> T
(100, 200, 300)
>>> del D[1]
>>> D
{200, 300}
>>> T
(100, 200, 300)
```

Un altro metodo ancora è $.\text{get}()$
In particolare $D.\,\text{get}(k, \text{default}) = \begin{cases} D[k] & \text{se } k \text{ è in } D \\ \text{default} & \text{Altrimenti} \end{cases}$

Riprendendo il primo esempio sui dizionari: $\text{Freq}[e] = \text{Freq.get}(e, 0) + 1$

Cuore del Linguaggio

Concetti di Oggetti e Classi.
Tutto quello che abbiamo visto non sono altro che manifestazioni di Oggetti / Classi.
Ogni valore viene con il suo tipo Type. Ogni oggetto è di una certa classe e il tipo ci dice la struttura dell'elemento
Abbiamo le classi che qualcuno ha definito per noi che stabiliscono le strutture, i metodi e le operazioni.

Gli oggetti di una classe vengono chiamati **Istanze**, mentre le **Classi** ci definiscono i metodi comuni, le operazioni comuni eccetera per le istanze. Ma se qualcuno ha fatto qualcosa per noi, allora lo possiamo creare anche noi (Ossia il programmatore può definire lui stesso delle certe classi, definendo struttura comune, operazioni e metodi di quella classe)

Lo facciamo con un esempio:

```
'''Una classe per i Punti Cartesiano
- Avrà due informazioni x e y
- un punto è creato sull'origine
- whoareyou: metodo che restituisce le coordinate x e y
- move(delta): metodo che sposta il punto di delta su entrambi gli assi'''

class Point:
        def __init__(self, xx, yy):       #Dunderinit #Crea la classe con i due punti
                self.x = xx
                self.y = yy
```

```
        def whoareyou(self):
                return self.x, self.y #Metodo che su una istanza restituisce il valori di x
e di y
        def move(self, delta):
                self.x += delta
                self.y += delta

[SHELL]
>>> Point()
#Quello che restituisce è la conferma della creazione di una istanza con x = 0 e y = 0
>>> p = Point()
>>> p.whoareyou()
(0, 0)
>>> p.move(10)
>>> p.whoareyou()
(10, 10) #Sempre stesso oggetto ma ha cambiato l'informazione che contiene
>>> q = Point()
>>> q.whoareyou()
(0, 0)
>>> p.whoareyou
(10, 10)
```

I metodi di tipo $\_\_\text{init}\_\_$ sono nomi particolari che determinano cose particolari
$\_\_\text{init}\_\_$ questo metodo stabilisce la struttura comune di tutte le istanze.
Questo metodo ha parametro self che è l'oggetto che viene creato (in realtà quando creo un oggetto dandogli il nome, viene evocato $\_\_\text{init}\_\_$ su self)
Usando self.x indica che nella creazione di tale classe viene creato l'Attributo x, ossia x e y sono istanze di Point
Quando creiamo q di classe Point, q ha il suo attributo con un suo valore (ma comunque necessariamente presente in $\_\_\text{init}\_\_$, in quanto stabilisce la struttura della classe)
Se non metto $\text{self.}<\text{qualcosa}>$, quel valore viene perso, perché resta comunque una funzione e il valore viene salvato localmente

Il corpo della classe può avere sia comandi che definizioni di metodi, ma sono puramente opzionali
Per le istanze dobbiamo usare il nome come costruttore.

Produrre istanze di una classe:
Sia class nomeC. Se prendiamo $\text{nomeC}()$ crea un'istanza della classe nomeC, mediante un'invocazione del metodo $\_\_\text{init}\_\_$ sull'istanza che viene creata *(esempio precedente)*.
Creare istanza = la macchina riserva dello spazio per quell'oggetto e assegna la struttura data dalla definizione della classe (nell'esempio le coordinate x, y)
Quando un metodo definito come: $\text{def met(self}, x_1, \ldots, x_n)$ è invocato su un oggetto obj, ossia $\text{obj.met}(v_1, \ldots, v_n)$
La struttura delle istanze di una classe è descritta nel metodo $\_\_\text{init}\_\_$ se questo esiste (infatti non è obbligatorio)
Nel metodo $\_\_\text{init}\_\_$ creiamo gli attributi della classe.

A volte però il metodo $\_\_\text{init}\_\_$ potrebbe non esserci

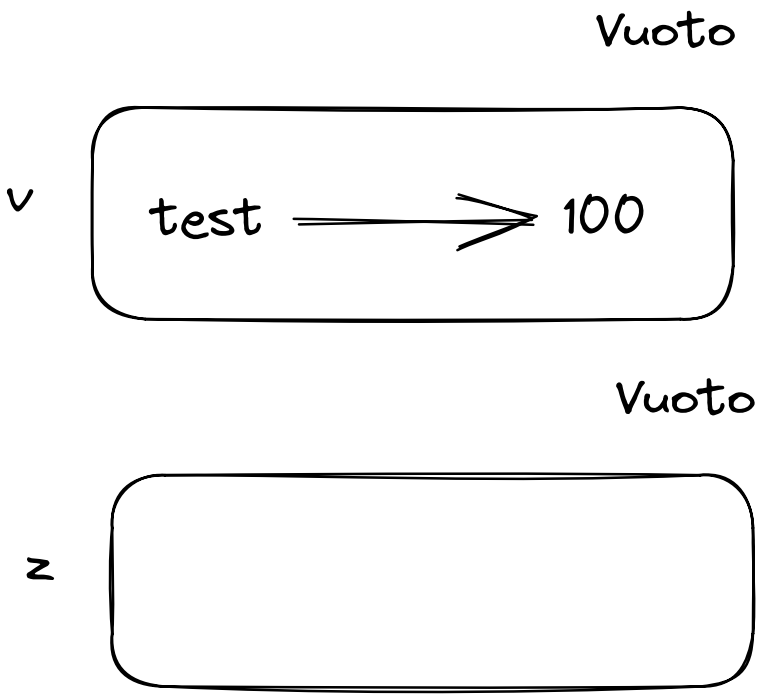
```
class Vuoto
        pass
```

La più semplice delle classi. È una classe che molto raramente viene utilizzata.
Ma posso comunque istanziare la classe vuota.

```
class Vuoto
        pass

[SHELL]
>>> a = Vuoto()
>>> type(a)
```

```
<class '__main__.Vuoto'>
>>> v
<__main__.Vuoto at (id)>
>>> v.test = 100
>>> v
100
>>> z = Vuoto()
```

Nel comando $\mathrm{v.test} = 100$ assegno io l'attributo test solo su v, ma non su z. Infatti la struttura interna sarà:



È possibile ottenere le funzioni passando per il "nome completo" della funzione:
Riprendendo l'esempio della classe Point:

```
>>> Point.whoareyou
<function Point.whorareyou at ...>
>>> p = Point()
>>> p.whoareyou()
(0, 0)
>>> Point.whoareyou(p)
(0, 0)
```

A che serve tutto ciò? Perché mettere due modi per far la stessa cosa?

```
class A:
        def __init__(self):
                self.x = 0
        def move(self):
                self.x += 100

class B:
        def __init__(self):
                self.x = 0
        def move(self):
                self.x += 10000

x = input()

if x == 'A':
```

```
        inst = A()
else:
        inst = B()
inst.move()
```

Con quest'ultima linea non sappiamo quale .move sarà eseguito. Con i metodi fa sì che la cosa sia molto più fluida
In caso contrario dovremmo mettere il metodo all'interno dell'if (è lo stesso principio di $\mathrm{a+b}$, di cui non sappiamo il tipo)

Questa si chiama **Selezione Dinamica dei Metodi** (cosa più importante di un linguaggio orientato agli oggetti)

Con le funzioni, ogni funzione fa la stessa cosa. Con i metodi, più metodi possono avere più nomi e quindi sono intercambiabili e possiamo sapere quale metodo viene utilizzato con lo stesso nome **solo all'esecuzione**, ossia non è possibile staticamente (guardando il codice) che tipo di metodo è.

Esercizio Guidato:
Una classe per rettangoli con il punto in basso a sinistra, la base e l'altezza, con tre metodi che restituisce l'area del rettangolo, uno che muove il rettangolo e il terzo che restituisce l'angolo in alto a destra:

```
class Point:
        def __init__(self, xx = 0, yy = 0):
                self.x = xx
                self.y = yy
        def whoareyou(self):
                return self.x, self.y
        def move(self, delta):
                self.x += delta
                self.y += delta

class Rect:
        def __init__(self, c, b, h):
                self.cornerSW = c
                self.base = b
                self.height = h
        def area(self):
                return self.base * self.hegiht
        def move(self, delta):
                self.cornerSW.move(delta) #Metodo move di Point
        def cornerNE(self):
                return Point(self.cornerSW.x + self.base, self.cornerSW.y + self.height)

[SHELL]
>>> p = Point(2, 3)
>>> r = Rect(p, 4, 1)
>>> r.area()
4
>>> r.cornerNE().whoareyou()
(6, 4)
```

Supponiamo io voglia estendere la classe Point con la somma di due punti:

```
class Point:
        def __init__(self, xx = 0, yy = 0):
                self.x = xx
                self.y = yy
        def whoareyou(self):
                return self.x, self.y
        def move(self, delta):
```

```
                self.x += delta
        def somma(self, other)
        return Point(self.x + other.x, self.y + other.y)

[SHELL]
>>> p = Point(2, 1)
>>> q = Point(4, 4)
>>> r = p.somma(q)
>>> r.whoareyou()
(6, 5)
>>> print(p)
<restituisce il fatto che p è di classe Point>
```

Però mi da errore fare $\mathrm{q} + \mathrm{p}$
Si può fare attraverso un metodo magico ossia attraverso $\_\_\mathrm{add}\_\_$
Se facessi $\mathrm{print(p)}$ mi ritorna soltanto che p è un punto, ma si può usare $\_\_\mathrm{str}\_\_$

```
class Point:
        def __init__(self, xx = 0, yy = 0):
                self.x = xx
                self.y = yy
        def whoareyou(self):
                return self.x, self.y
        def move(self, delta):
                self.x += delta
        def __add__(self, other)
                return Point(self.x + other.x, self.y + other.y)
        def __str__(self):
                return 'P(' + str(self.x) + ', ' + str(self.y) + ')'

[SHELL]
>>> p = Point(2, 1)
>>> q = Point(4, 4)
>>> r = p + q
>>> r.whoareyou()
(6, 5)
>>> r = p.__add__(q)
>>> r.whoareyou()
(6, 5)
>>> print(p)
P(2, 1)
>>> q = Point(4, 4)
>>> p is q
False
>>> p == q
False
```

Ponendo l'uguaglianza tra istanze, fino a prova contraria da sempre falso, perché la macchina Python a priori non sa se indica la stessa cosa.
Si può cambiare la definizione di $==$ attraverso il metodo $\_\_\mathrm{eq}\_\_$

```
class Point:
        def __init__(self, xx = 0, yy = 0):
                self.x = xx
                self.y = yy
        def whoareyou(self):
                return self.x, self.y
        def move(self, delta):
                self.x += delta
```

```
        def __add__(self, other)
                return Point(self.x + other.x, self.y + other.y)
        def __str__(self):
                return 'P(' + str(self.x) + ', ' + str(self.y) + ')'
        def __eq__(self, other):
                return self.x == other.x and self.y == other.y

[SHELL]
>>> p = Point(4, 4)
>>> q = Point(4, 4)
>>> p == q
True
```

I metodi magici sono tutti i metodi con $\_\_<\text{nome}>\_\_$ che vengono chiamati in alcune circostanze.
È quello che avviene in automatico.

Poi ci sono altri metodi magici che si possono trovare sulla documentazione ufficiale

Attributi Privati:
Esempio:

```
from random import randint

class Coin:
        def __init__(self):
                self.face = 'H'
        def toss(self)
                if randint(0, 1) == 0:
                        self.face = 'H'
                else:
                        self.face = 'T'
                return self.face

[SHELL]
>>> c.__face
'H'
```

Restituisce a volte testa o a volte croce:
Con il comando $\text{moneta.face} = \text{'H'}$ ottengo sempre Testa, modificandolo da fuori.
Però posso cercare di segnalare che un attributo da fuori sarebbe meglio da non cambiare:
Posso cambiare il nome da face a $\_\_\text{face}$

```
from random import randint

class Coin:
        def __init__(self):
                self.__face = 'H'
        def toss(self)
                if randint(0, 1) == 0:
                        self.__face = 'H'
                else:
                        self.__face = 'T'
                return self.__face

[SHELL]
>>> c.__face
<Coin has no attribute called face>
```

Dall'esterno apparentemente non c'è.

Gli attributi privati servono per nascondere dall'esterno gli attributi che non vogliamo vengano modificati.
È una cosa puramente metodologica, ma può essere trovato all'esterno con $\_Coin\_\_c$

```
from random import randint

class Coin:
        def __init__(self):
                self.__face = 'H'
        def toss(self)
                if randint(0, 1) == 0:
                        self.__face = 'H'
                else:
                        self.__face = 'T'
                return self.__face

[SHELL]
>>> c.__face
<Coin has no attribute called face>
>>> _Coin__c
'H'
```

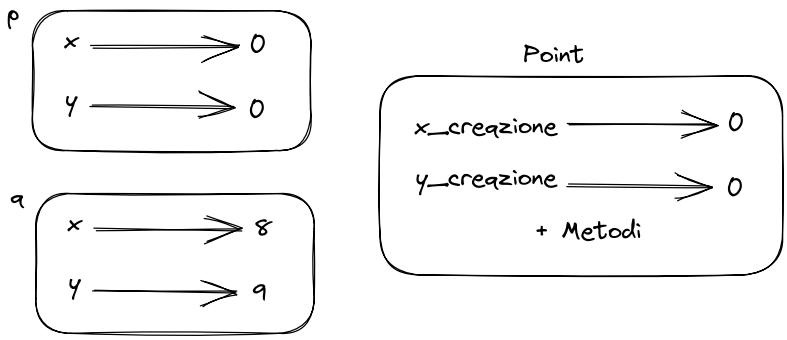
Quest'operazione si chiama **Name Mangling** (letteralmente storpiatura del nome).
Questo perché siamo tutti adulti consenzienti: una cosa non dovrebbe essere fatta, ma se è necessaria...

Gli attributi possono essere attributi di classe o di istanza.
Gli attributi di classe sono gli stessi per ogni istanza

```
class Stud:
        Uni = 'Bologna' #Attributo della classe, non dell'istanza
        def __init__(self):
                self.n = nome
        def __str__(self):
                return 'Studente: ' + self.nome + ' di ' + Stud.Uni

[SHELL]
>>> s = Stud('Maria')
>>> print(s)
Studente: Maria di Bologna
```

Il nome è legato alla classe e non alle diverse istanze
Se nell'esempio volessi fare più università (Uni nell'esempio) devo o creare una nuova classe o non usare Uni come attributo di classe.

```
class Point:
        x_creazione = 0         #Attributo di istanza
        y_creazione = 0
        def __init__(self, xx = x_creazione, yy = y_creazione):
                self.x = xx
                self.y = yy
        def move(self, delta):
                self.x, self.y = self.x + delta, self.y + delta

[SHELL]
>>> p = Point()
>>> q = Point(8, 9)
```



Voglio definire una nuova classe con un metodo che riporta all'origine

```
class Point:
        def __init__(self, xx = x_creazione, yy = y_creazione):
                self.x = xx
                self.y = yy
        def move(self, delta):
                self.x, self.y = self.x + delta, self.y + delta

class NewPoint:
        def __init__(self, xx = x_creazione, yy = y_creazione):
                self.x = xx
                self.y = yy
        def move(self, delta):
                self.x, self.y = self.x + delta, self.y + delta
        def origin(self):
                self.x, self.y = 0

[SHELL]
>>> np = NewPoint(8, 9)
>>> np.x
8
>>> np.y
9
>>> np.origin
>>> np.x
0
>>> np.y
0
```

Il fatto che $\mathrm{NewPoint}$ sia Point più un qualcosa è detto solo a parole. Infatti sembra qualcosa direttamente di nuovo.
"Leggendolo" si può vedere che sono la stessa cosa, ma la cosa migliore è farlo direttamente in maniera più concisa (Non solo come chiarezza del programma ma anche come manutenzione del programma).
Del tipo, supponiamo vogliamo cambiare la $move$ di $Point$, bisogna fare la modifica due volte, anche in $NewPoint$, ma non sempre è sicuro e diventa ingestibile quando si hanno numerose classi.
Proprio per questo motivo ci sono le sottoclassi:

```
class Point:
        def __init__(self, xx = x_creazione, yy = y_creazione):
                self.x = xx
                self.y = yy
        def move(self, delta):
                self.x, self.y = self.x + delta, self.y + delta

class NewPoint(Point):  #Definizione della sottoclasse
```

```
        def origin(self):
                self.x, self.y = 0
```

In questo modo quello che succede è che in $\mathrm{NewPoint}$ vengono copiate (eredita) tutte le informazioni di Point più un nuovo metodo ($.\mathrm{origin}$). Possiamo dire che $\mathrm{NewPoint}$ è la derivata di Point, che è definita anche come la base di $\mathrm{NewPoint}$

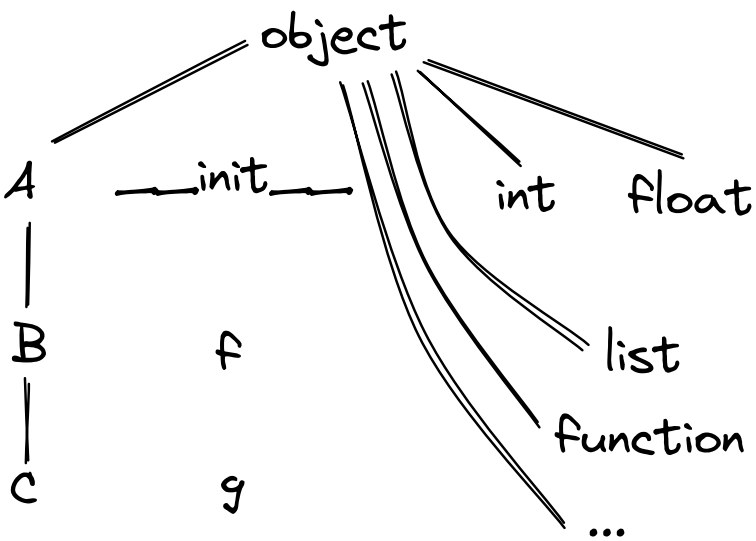
De. inizione di Sottoclasse: È una classe che eredita ($\mathrm{inherits}$) dalla sua sovraclasse i suoi attributi e i suoi metodi che non ridefinisce ($\mathrm{overrights}$)

Ma se invece volessi modificare?

```
class Point:
        def __init__(self, xx = x_creazione, yy = y_creazione):
                self.x = xx
                self.y = yy
        def move(self, delta):
                self.x, self.y = self.x + delta, self.y + delta

class NewPoint(Point):
        def origin(self):
                self.x, self.y = 0
        def move(self, delta):
                self.x = 100
                self.y = 100
```

Altro esempio:

```
class A:
        def __init__(self):
                self.x = 100

class B(A):
        def f(self):
                return self.x

class C(B):
        def g(self):
                self.x = 0
```



Tutti i metodi a partire da A vengono ereditati su C (seguendo la gerarchia delle classi) e termina sulla classe più alta e viene chiamata object

La nozione di sottoclasse estende la nozione di tipo, infatti riprendendo l'esempio precedente si ha che:

```
[SHELL]
>>> c = C(1, 2)
>>> type(C)
<class '__main__.C'>
```

Ma è anche istanza di B che a sua volta è istanza anche di A (il tipo è fissato alla creazione dell'oggetto)

```
[SHELL]
>>> c = C(1, 2)
>>> type(C)
<class '__main__.C'>
>>> isistance(c, C) #c è una istanza di C?
True
>>> isistance(c, B) #c è una istanza di B?
True
>>> isistance(c, A) #c è una istanza di A?
True
>>> isistance(c, object) #c è una istanza di object?
True
```

Tutte le cose sono istanze di object

```
class Point:
        def __init__(self, xx = x_creazione, yy = y_creazione):
                self.x = xx
                self.y = yy
        def move(self, delta):
                self.x, self.y = self.x + delta, self.y + delta

class NewPoint(Point):
        def origin(self):
                self.x, self.y = 0
        def move(self, delta):
                self.x = 100
                self.y = 100
```

La selezione del move è stabilita principalmente e solamente dalla classe dell'elemento.
Questa si chiama **Selezione dinamica di metodi - Dinamic Method Lookup**

```
class Point:
        def __init__(self, xx = x_creazione, yy = y_creazione):
                self.x = xx
                self.y = yy
        def move(self, delta):
                self.x, self.y = self.x + delta, self.y + delta

def CPoint(Point):
        def __init__(self, xx, yy, cc):
                self.x = xx
                self.y = yy
                self.c = cc
```

Non è scritto nel migliore dei modi in quanto ho riscritto tutto il metodo $\_\_init\_\_$ della classe Point
La cosa migliore sarebbe prima inizializzare l'$\_\_init\_\_$ di Point e poi aggiungere la parte nuova:

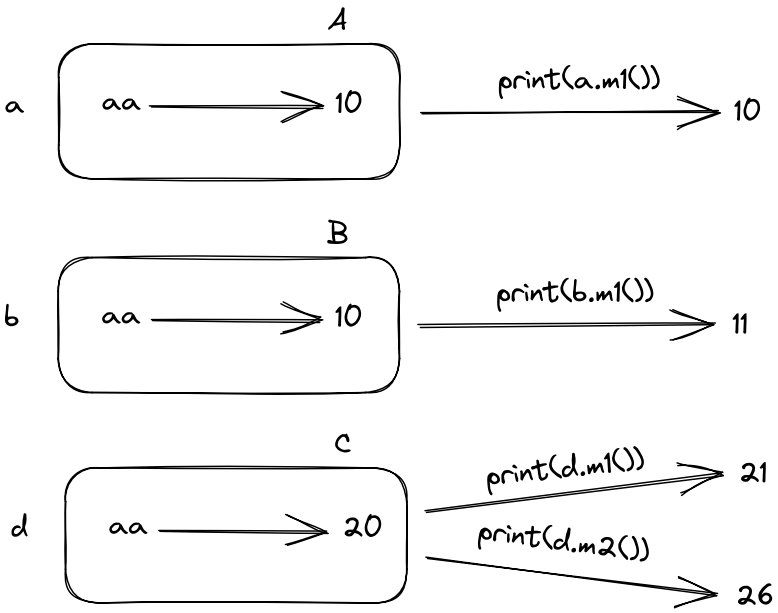
```
class Point:
        def __init__(self, xx = x_creazione, yy = y_creazione):
                self.x = xx
                self.y = yy
        def move(self, delta):
                self.x, self.y = self.x + delta, self.y + delta

def CPoint(Point):
        def __init__(self, xx, yy, cc):
                super().__init__(xx, yy)  #Questa linea di codice serve per chiaramre
l'__init__ di Point
                self.c = cc
```

$\mathrm{super}()$ serve per prendere il metodo della sovraclasse immediata.

Se B è sottoclasse immediata di A, dentro B $\mathrm{super}()$ è self visto in A

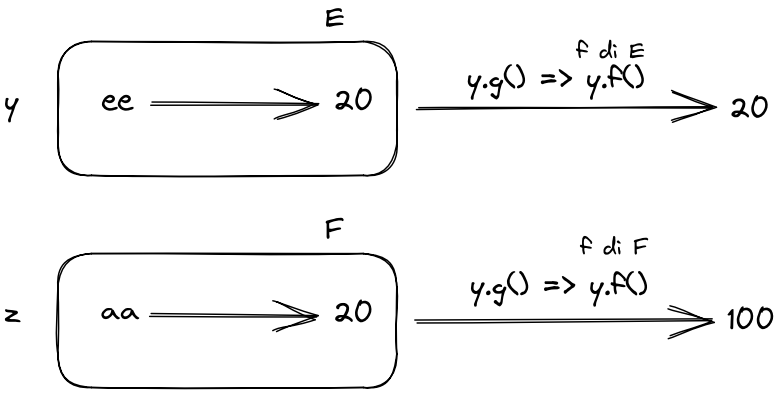
Altro esempio:

```
class A:
        def __init__(self, x):
                self.aa = x
        def m1(self):
                return self.aa

class B(A):
        def m1(self):
                return super().m1() + 1
        def m2(self, x):
                return self.aa + x

class C(B):
        def m2(self, z):
                return self.aa + 2 * Z

a = A(10)
print(a.m1())
b = B(10)
print(b.m1())
d = C(20)
print(d.m1())
print(d.m2(3))
```



Un altro esempio:

```
class E:
        def __init__(self, y):
                self.ee = y
        def f(self):
                return self.ee
        def g(self):
                return self.f()
class F(E):
        def f(self):
                return 100

[SHELL]
>>> z = F(20)
>>> y = E(20)
>>> y.g()
20
>>> z.g()
100
```



È possibile anche fare:

```
class E:
        def __init__(self, y):
                self.ee = y
        def g(self):
```

```
                return self.f()
class F(E):
        def f(self):
                return 100

[SHELL]
>>> z.F(20)
>>> z.g()
100
>>> y.E(20)
>>> y.g()
<Errore>
```

Questo è il meccanismo di Delega (ossia, qualche sottoclasse di una sottoclasse definirà un metodo).
In questo caso g in E delega F, F è definita come sottoclasse astratte

L'utilizzo della scrittura $<\text{istanza}>.<\text{attributo}>$ si chiama **Dot Notation**

Eccezioni

È un evento che normalmente dovrebbe causare degli errori.
Per evitare le eccezioni is usa:

```
try:
        <blocco protetto>
except <eccezione>:
        <gestore>
```

Esempio:

```
try:
        x = 0
        y = 10 / x
        z = 100
except ZeroDivisionError:
        print('Bischero, nun se divide per zero')
```

Per sollevare le eccezioni c'è il comando $\text{raise}(<\text{Errore}>)$
Se non c'è scritto nulla allora vengono calcolate tutte le eccezioni:
È possibile anche ottenere direttamente il valore dell'eccezione:

```
try:
        print(A)
expect ErrorName as X:
        print('Bischero, nun se divide per zero')
        print(X)
```

Volendo si può implementare un else: in caso non vengano sollevate eccezioni
Esiste anche un blocco finally: che viene eseguito indipendentemente dall'esito di try

Chiaramente tutte le eccezioni si possono trovare sulla documentazione.

Poiché ogni cosa sono degli oggetti, allora lo sono anche le eccezioni e sono istanze di (una sottoclasse) della classe Exception che qualcuno ha definito per noi.
Il linguaggio fa si che qualsiasi istanza di Exception siano estenzioni, quindi oggetti e fa si che possano essere manipolati e utilizzati (Di solito non si usa mai la classe Exception) direttamente, ma si utilizza la sottoclasse:

```
class MiaEccezione(exception):
        pass

try:
        x = 0
        raise MiaEccezione
        z = 100                 #z non è definito in quanto l'eccezione interrompe prima
except MiaEccezione:
        print('Eccezione catturata')
print('Dopo il try')
```

Se mettessi $\text{raise MiaEccezione}(10)$ passa anche valore
A quel punto possiamo fare

```
class MiaEccezione(exception):
        pass

try:
        x = 0
        raise MiaEccezione(100)
        z = 100
except MiaEccezione as n:
        print('Eccezione catturata con valore:', n)
print('Dopo il try')
```

Come tutte le altre classi è possibile definire definire sottoclassi anche alle sottoclassi di Exception

Quindi la struttura di Eccezione diventa

```
try:
        <blocco protetto>
except ClasseEccezione:
        <gestore>
```

Quello che fa ClasseEccezione è intrappolare le eccezioni di ClasseEccezione incluse le istanze delle sue sottoclassi.
Per cui nell'esempio precedente, porre $\text{raise MiaEccezione}$ e $\text{raise AltraEccezione}$ non cambia, in quanto vengono catturate entrambe:

Esempio:

```
class B(Exception):
        pass

class C(B):
        pass

class D(C):
        pass

for c in [B, C, D]:
        try:
                raise c()
        except D:
                print('D')
        except C:
                print('C')
```

```
        except B:
                print('B')
```

Esempio:

```
def prod():
        'Restituisce il prodotto di una lista di numeri'
        res = 1
        for e in L:
                if e == 0:
                        return 0
                res *= e
        return res
```

Utilizzando le eccezioni diventa
Per interrompere una computazione in tempi opportuni.

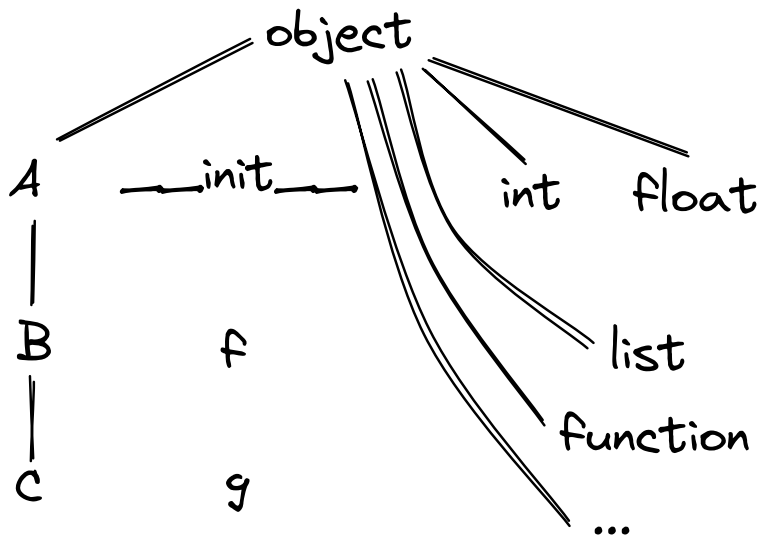
```
class Zero(Exception):
        pass
try:
        res = 1
        for e in L:
                if e == 0:
                        raise Zero
                res *= e
except
        res = 0

print(res)
```

Il tutto è ancora più efficiente quando ci sono le ricorsioni.

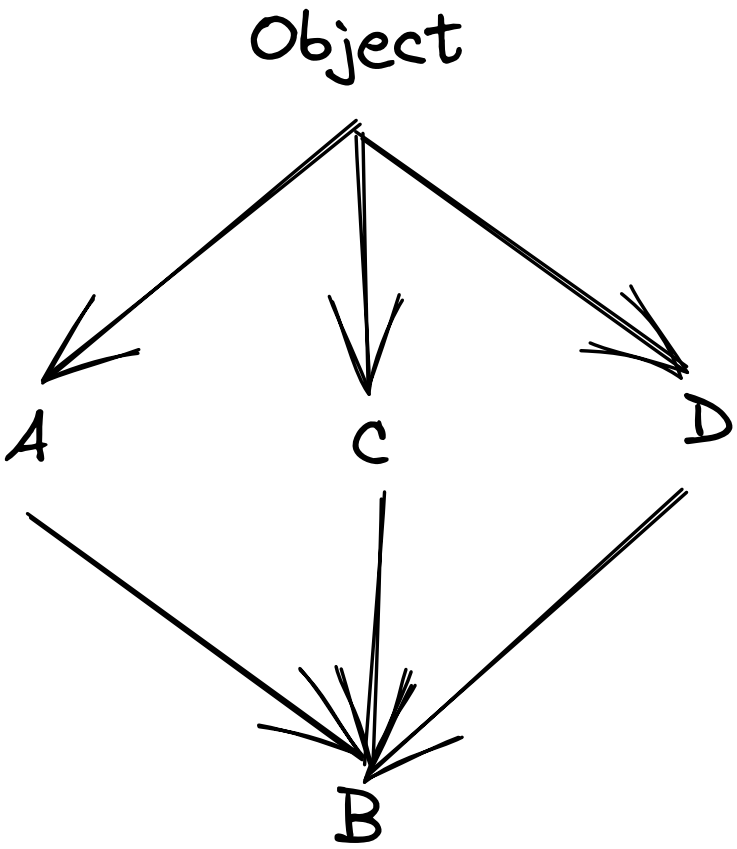
Le eccezioni sono comode anche per poter uscire da dei cicli annidati, potenzialmente infiniti.

Alberi

Dallo scherma seguente può sembrare la gerarchia delle classi sia un albero / grafo



In realtà non è così perché (per esempio) una classe può essere sottoclasse di molte: class B(C,D,A)

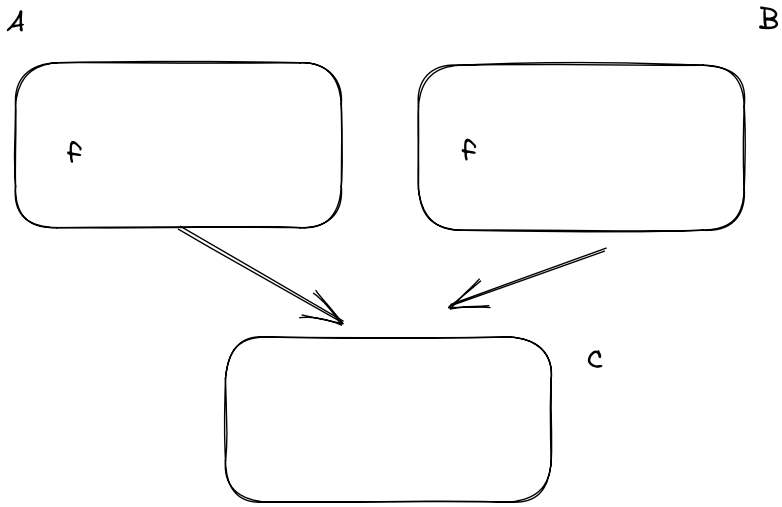


Questo tipo di linguaggio viene chiamata **Ereditarietà Multipla** e in questo caso B eredita tutti gli attributi di A, C, D

Esempio:

```
class A:
        def g(self):
                return 0

class C:
        def f(self):
                return 1

class D:
        def h(self):
                return 3

class B(A,C,D):
        def f(self):
                return 100

[SHELL]
>>> bb = B()
>>> bb.f()
100
>>> bb.g()
1
>>> bb.h()
3
```

Altro Esempio:




```
class A:
        def f(self):
                return 'a'

class B:
        def f(self):
                return 'b'

class C(A, B):
        pass

[SHELL]
>>> c = C()
>>> c.f()
```

In questo caso viene fatto prima f di A perché nella definizione della sottoclasse abbiamo prima invocato A e poi B.

Prendiamo un'altra situazione:

```
class top:
        def f(self):
                return 'top'

class A(top):
        def f(self):
                return 'a'

class B(top):
        pass

class C(A, B):
        pass

[SHELL]
>>> c = C()
>>> c.f()
'a'
```

Ma se provassimo ad invertire, ossia:

```
class top:
        def f(self):
```

```
                return 'top'

class A(top):
        pass

class B(top):
        def f(self):
                return 'b'

class C(A, B):
        pass

[SHELL]
>>> c = C()
>>> c.f()
'b'
```

Ossia, quando viene chiamata una funzione di una sottoclasse (in cui la stessa funzione non è stata definita) va prima a vedere la vicinanza della funzione e se si trova a "distanza 1" (ossia deriva direttamente), lo prende dal primo chiamato.

Esiste un ordine con cui vengono prese le funzioni, che è quello di $M.R.O.$ ossia **Method Resolution Order**, che stabilisce l'ordine con cui vengono cercate le classi. In particolare con $\mathrm{c.mro}$ si ottiene la lista con la gerarchia delle classi.

Chiaramente ci sono delle applicazioni in cui sembra non esserci un M.R.O., ossia l'algoritmo non produce un risultato preciso.

```
class A:
        pass
class B:
        pass
class C(A, B):
        pass
class D(B, A):
        pass
class E(C, D):
        pass
```

Tipi Astratti

Un esempio saranno i numeri complessi:
Prima soluzione = Convenzione: Un complesso è una tupla di float lunga due e dunque possiamo definire le operazioni sui complessi

```
def Re(c):
        return c[0]

def Im(c):
        return c[1]

def somma(c1, c2):
        return (Re(c1) + Re(c2), Im(c1) + Im(c2))

c1 = (1, 1)
c2 = (2, 3)
```

```
s = somma(c1, c2) #Fa la somma
t = c1 + c2       #Concatema le tuple
```

I linguaggi di programmazione ad alto livello come Python sono linguaggi utilizzati nella costruzione.

I linguaggi di programmazione servono a costruire *modelli computazionali* (strutture formali, non solo di programmi, ma anche di descrizione dei dati che descrivono situazioni del mondo reale - non solo procedurale ma anche la struttura, in modo tale che manifesta un aspetto computazionale)

```
class Complex:
        def __init__(self, re, im):
                self.__real = re
                self.__img = im
        def Re(self)
                return self.__real
        def Im(self):
                return self.__img
        def __add__(self, other):
                return Complex(self.__real + other.__real, self.__img + other.__img)
                #Oppure return Complex(Re(self) + Re(other), Im(self) + Im(other))
                #Ma è meglio scrivendolo direttamente perché sono ancora nella definizione
della
                #classe
        def __str__(self):
                return str(self.__real) + '+' + str(self.__img) + 'i'
        def __repr__(self):
                return self.__str__
```

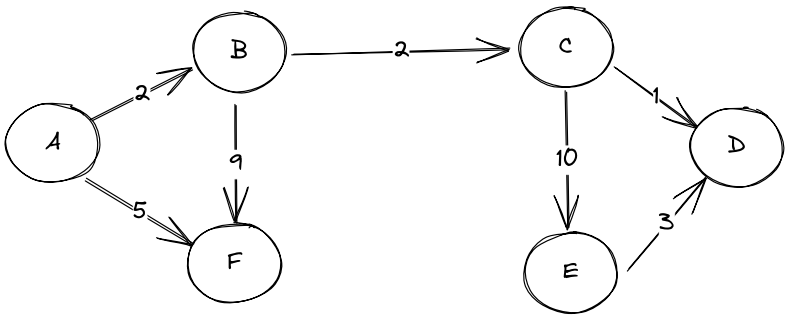
Lo scopo è: l'utilizzatore utilizza la classe dei complessi, senza sapere cosa c'è all'interno dei vari metodi magici e vengano soltanto utilizzati gli operatori predefiniti (questa è la definizione di Tipo Astratto)

I metodi Re e Im vengono chiamati "getter" in quanto restituisco degli attributi privati.
Così ci sono i metodi "setter" che modificano i metodi privati.

Tra i tanti tipi di tipi astratti ci sono gli alberi i binari ("serie di palle collegate da linee")

Nodi

Definizione di Grafo: Un grafo è una coppia $G = (V, E)$ dove V è un insieme arbitrario, chiamato **Insieme dei Vertici o dei nodi** e $E \subseteq V \times V$, chiamati **Archi** (visti che risultati del prodotto cartesiano - coppie unite). G è finito $\Leftrightarrow$ E e V sono insiemi finiti.
Se la relazione è simmetrica, il grafo si chiama "Grafo non orientato". Altrimenti si chiama "Orientato" o "Diretto"

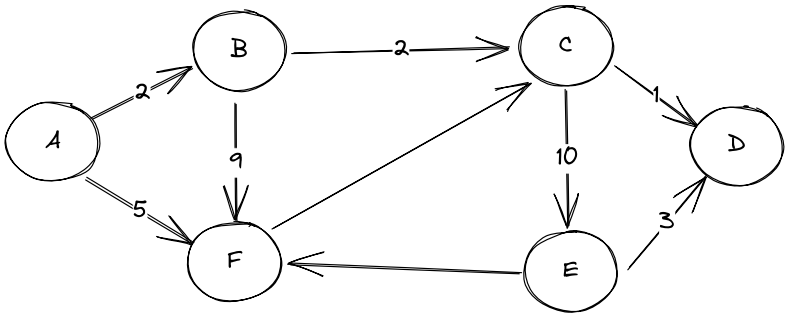


$V = \{A, B, C, D, E, F\}$
$E = \{(A, B), (B, F), (B, C), (A, F), (C, E), (C, D), (E, F)\}$

Un grafo è echitettato quando ci sono due funzioni $lab_V : V \to D_1$ e $lab_E : E \to D_2$

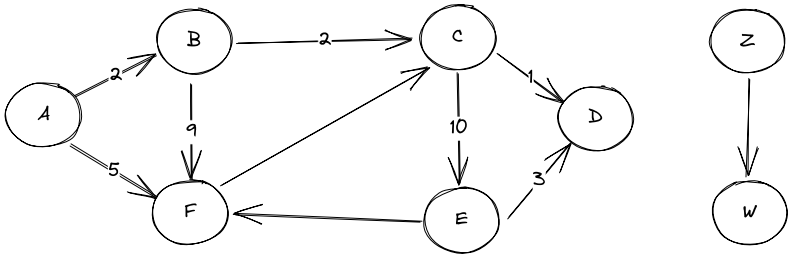
Definizione di Cammino: Un cammino in $G = (V, E)$ è una sequenza di vertici $v_1, v_2, \ldots, v_n$ tale che $\exists v_i, v_{i+1} \in E$
Un ciclo è un cammino $v_1, \ldots, v_n$ con $v_1 = v_n$ senza né archi né nodi ripetuti

Nell'esempio sopra non ci sono cicli
In questo appena sotto si:



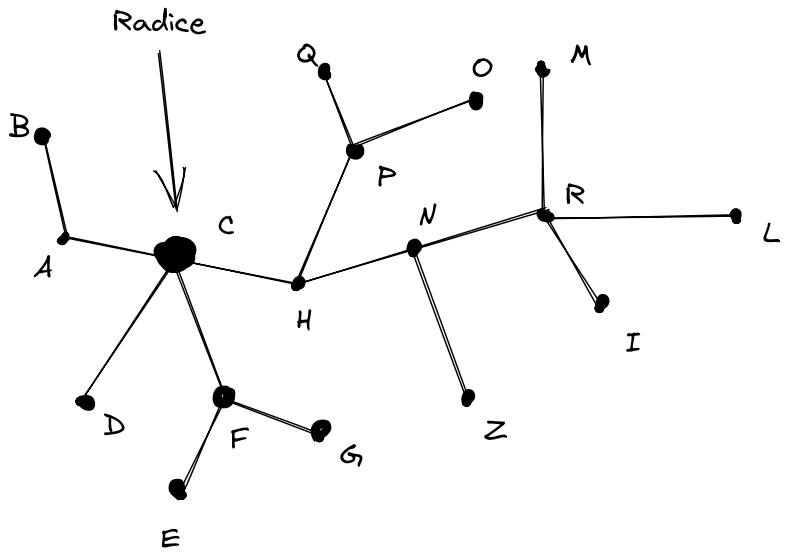
Definizione di Connesso: Un grafo è connesso $\Leftrightarrow \forall v_1, v_2 \in V, \exists$ cammino da v_1 a v_2

Questi due sopra sono connessi, mentre questo appena sotto no



La teoria dei grafi è molto importante soprattutto per la teoria degli algoritmi, del tipo "dato un cammino, ci sono cammini in cui i punti vengono toccati una volta soltanto? E se ci sono, quale è quella con lunghezza minima?" A questi problemi abbiamo risposte solamente all'ordine esponenziale (esponenziale al numero di nodi)

Definizione di Albero (per matematica discreta): Un albero è un grafo non orientato, connesso e senza cicli
Si dice che è radicato quando un suo vertice è scelto come nodo distinto che viene chiamato radice.



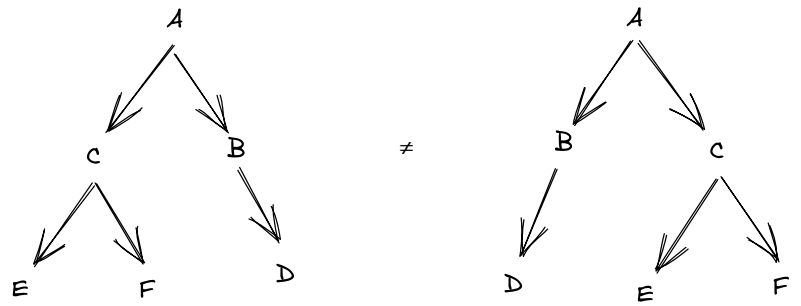
Che diventerà (scritto in Latex):

$$C \to \begin{cases} A \to B \\ D \\ F \to \begin{cases} E \\ G \end{cases} \\ \\ H \to \begin{cases} P \to \begin{cases} Q \\ O \end{cases} \\ N \to \begin{cases} Z \\ R \to \begin{cases} M \\ L \\ I \end{cases} \end{cases} \end{cases} \end{cases}$$

Qui si mischiano linguaggi di botanica e genealogia. Dato un nodo quello prima viene chiamato "padre" e quello dopo "figlio". Se dopo un nodo non c'è un figlio, allora viene chiamato "Foglia".
Dopo aver fissato una radice la genealogia viene naturale.
Una volta scelta una radice, tutti i nodi rappresentano radici di sottoalberi

Definizione di Ordinato: Un albero è ordinato se per ogni vertice v i figli di v sono un insieme ordinato

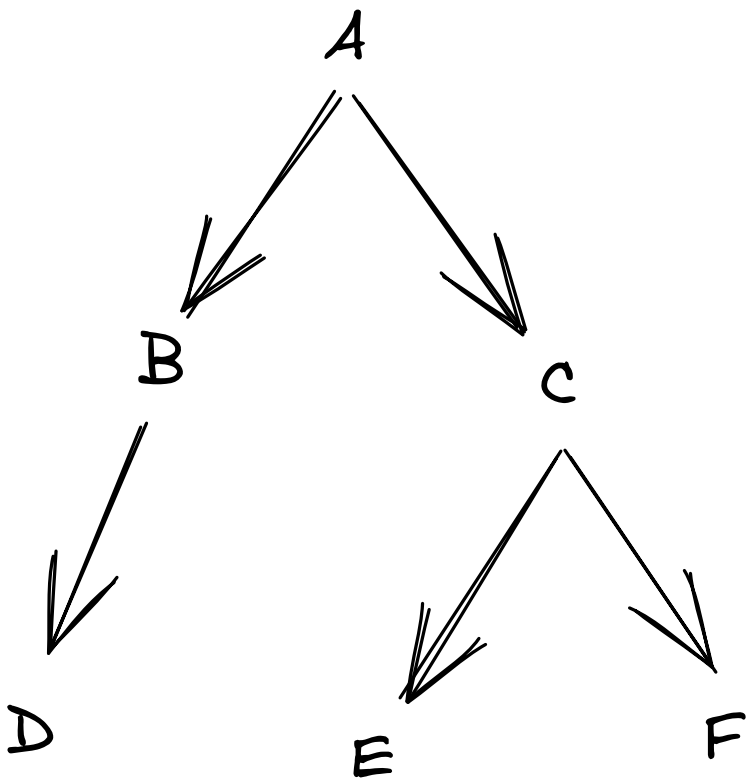
Definizione di Albero Binario: Un albero si dice binario se è radicato, ordinato e ogni vertice ha al più due figli, chiamati "Figlio Sinistro" (o primo figlio) e "Figlio Destro" (o secondo FIglio)
Destro o sinistro se scritto in latex va guardato come ruotato, cioè a in basso = a sinistra e viceversa.



Definizione di Profondità di un nodo: Si chiama profondità di un nodo v il numero di archi da v alla radice
Definizione di Altezza di un nodo: Si chiama altezza di un nodo v è il numero di archi da v alla foglia più profonda (altezza del sottoalbero con vertice v). L'altezza della radice prende anche il nome di **Altezza dell'Albero**.

Definizione Ricorsiva di Albero Binario: Un albero binario è:
- l'albero vuoto
- (v, t_1, t_2) dove v è un nodo che chiamo radice, t_1 è l'albero binario chiamato figlio sinistro e t_2 è un albero binario chiamato figlio destro.
Una foglia è un albero binario dove entrambi i figli sono l'albero vuoto.

Esempio:
Chiamo av è un albero binario vuoto ($av \in AB$)
Ho che $d = (D, av, av) \in AB$ e in modo analogo $e = (E, av, av) \in AB$ e $f = (F, av, av)$
Posso costruire $b = (B, D, av)$ e $c = (C, E, F)$
E infine $a = (A, B, C)$
Questo è esattamente



Con questa definizione riesco a gestire meglio il concetto di Albero.

Definiamo la classe di un albero binario in termini di tipo:

```
class tree:
        def __init__(self, lab = None, fs = None, fd = None):

                '''Albero vuoto lab, fs, fd sono None
                In albero non vuoto fs e fd devono essere di classe tree'''

                self.__label = lab
                self.__sx = fs
                self.__dx = fd

        def is_empty(self): #Metodo per vedere se l'albero è vuoto
                return self.__label == None

        def label(self):
                return self.__label

        def first_child(self):
                return self.__sx

        def second_child(self):
                return self.__dx

        def is_leaf(self):
                if self.__label is None:
                        return False
                return self.__sx.is_empty() and self.__dx.is_empty()
```

Esempi di alberi binari sono:

- $\mathrm{tree}()$ che coincide con l'albero vuoto

- 5 che coincide con 5
- $\text{tree}(7, \text{tree}(5), \text{tree}())$ che coincide con $7 \to \begin{cases} 5 \end{cases}$
- $T = \text{tree}(8, \text{tree}(7, \text{tree}(5), \text{tree}()), \text{tree}(9, \text{tree}(), \text{tree}(4)))$ che coincide con $8 \to \begin{cases} 7 \to \begin{cases} 5 \end{cases} \\ 9 \to \begin{cases} 4 \end{cases} \end{cases}$

Sugli alberi ci sono determinati metodi, in base anche all'esempio:

```
>>> T.label()
8
>>> T.first_child()
7
>>> T.is_empty()
False
>>> tree().is_empty()
True
>>> T.first_child().label()
7
>>> T.first_child().second_child().is_empty()
True
```

Useremo la classe in maniera totalmente astratta

Esercizi sugli alberi: Dato un albero stampare tutti i nodi di un albero

```
from esempio_sopra.py import tree

def stampa_a(alb):
        if alb.is_empty() == True:
                return
        print(alb.label())
        stampa_a(alb.first_child)
        stampa_a(alb.second_child)
```

Se prendiamo $T = \text{tree}(8, \text{tree}(7, \text{tree}(5), \text{tree}()), \text{tree}(9, \text{tree}(), \text{tree}(4)))$, otterremo:

```
8
7
5
9
4
```

Questa si chiama $\text{Visita dell'Albero}$ ossia ottengo tutte le etichette dei nodi.
In particolare si chiama Visita Anticipata in quanto prima stampo l'etichetta e poi accedo al valore del nodo.
Si può anche fare una stampa in ordine posticipato (ottengo prima l'informazione sui figli e poi stampo il valore)

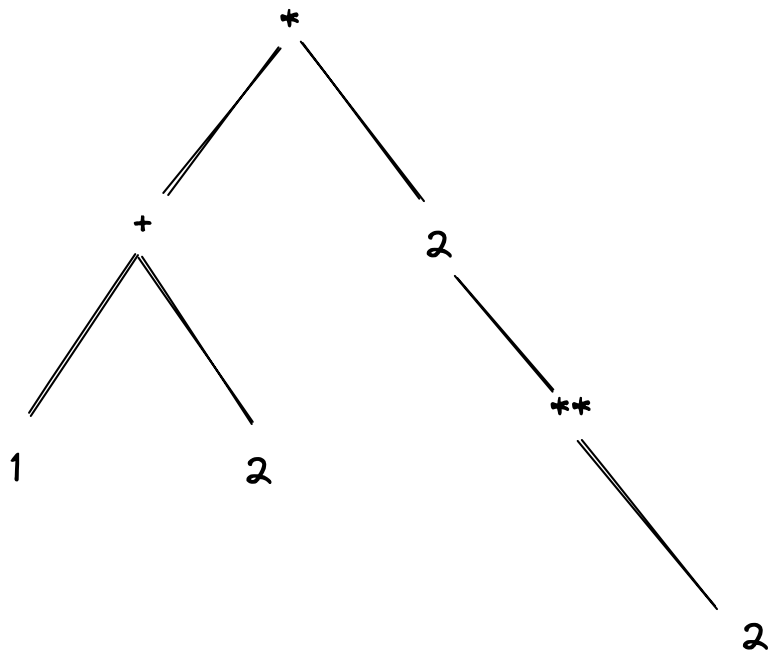
```
from esempio_sopra.py import tree

def stampa_p(alb):
        if alb.is_empty() == True:
                return
        stampa_p(alb.first_child)
        stampa_p(alb.second_child)
        print(alb.label())
```

In questo caso sono perfettamente equivalenti.

Si può fare anche la Visita in Ordine Simmetrico

```
from esempio_sopra.py import tree

def stampa_s(alb):
        if alb.is_empty() == True:
                return
        stampa_s(alb.first_child)
        print(alb.label())
        stampa_s(alb.second_child)
```

Con un albero binario si possono rappresentare anche le espressioni matematiche:



Con la Visita simmetrica perde completamente significato.
Con quella Anticipata no, perché si ottiene tutta una rappresentazione prefissa (Chiamata Polacca) delle operazioni.
Infatti stampa: $*+122**2$ f
Esiste anche una notazione polacca postfissa che si ottiene con una visita Posticipata (e si ottiene $12+2**2*$)

Altro Esercizio: Somma delle etichette di un albero:

```
"Dall'esempio prima"
def somma(alb):
        "L'albero ha solo etichette numeriche e l'albero vuoto = 0"
        if alb.is_empty():
                return None
        return alb.label() + somma(alb.fist_child()) + somma(alb.second_child())
```

Questa è una visita dell'albero Anticipata

Per questioni di completezza si può estendere il codice a:

```
def somma(alb):
        if alb.is_empty():
                return None:
        if alb.first_child().is_empty():
                sommaFC = 0
```

```
        else:
                sommaFC = somma(alb.fist_child())
        if alb.second_child().is_empty():
                sommaSC = 0
        else:
                sommaSC = somma(alb.second_child())
        return alb.label() + sommaFC + sommaSC
```

Fare qualcosa di simile anche per la produttoria.

Esercizio: Scrivere una funzione che indichi quanti nodi ci sono ad una profondità h

```
def numeronodi(alb, h):
        'h interno e non negativo'
        if alb.is_empty():
                return 0
        if h == 0:
                return 1
        return numeronodi(alb.first_child, h-1) + numeronodi(alb.second_child, h-1)
```

Altro Esercizio: Dato un albero, restituire un altro con la stessa struttura ma con tutte le etichette uguali a 0:

```
def dupl(alb):
        if alb.is_empty():
                return tree()
        return dupl(0, alb.first_child(), second_child())
```

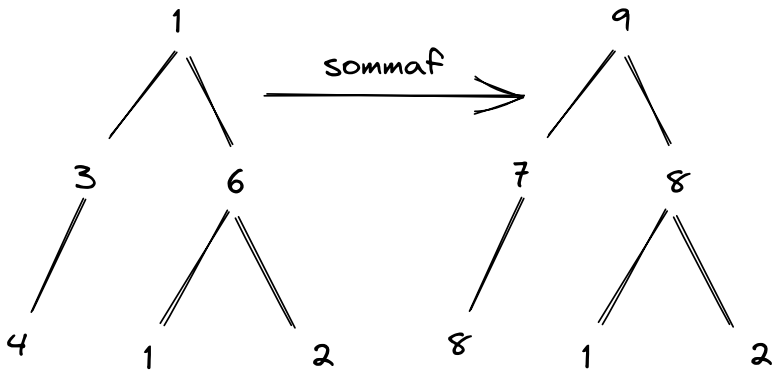
Per noi i $\mathrm{tree}()$ sono immodificabili, però si può fare un esercizio in cui si crea una classe nuova di tree modificabili

Altro esercizio:

```
def sommaf(a):
        if a.is_empty():
                return tree()
        if a.is_leaf():
                return tree(a.label())
        fs = sommaf(a.first_child())
        fd = sommaf(a.second_child())
        if fs.is_empty():
                newlab = a.label() + fd.label()
        elif fd.is_empty():
                newlab = a.label() + fs.label()
        else:
                newlab = a.label + max(fs.label(), fd.label())
        return tree(newlab, fs, fd)
```

Concettualmente questo va bene, ma mi devo assicurare che tutto funzioni - devo arginare gli errori

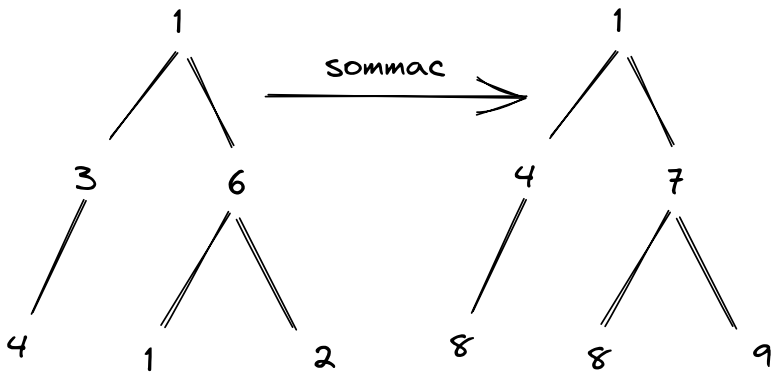
Esercizio: Scrivere una funzione tale che dato un albero binario con etichette intere restituisce un albero con la stessa struttura ogni nodo etichettato con la massima delle etichette sui cammini nodo-foglia dell'albero originale
Il massimo della somma dei cammini che vengono dal basso




```
def sommaf(alb):
        if alb.is_empty():
                return tree()
        fs = sommaf(alb.fist_child())
        fd = sommaf(aln.second_child())
        if fs.is_empty():
                nefs = 0
        else:
                nefs = fs.label()
        if fd.is_empty():
                nefd = 0
        else:
                nefd = fd.label()
        return tree(alb.label() + max(nefs, nefd), fs, fd)
```

Esercizio: Scrivere una funzione tale che dato un albero binario con etichette intere restituisce un albero con la stessa struttura ogni nodoetichettato con la massima delle etichette sui cammini nodo-foglia dell'albero originale
Il massimo della somma dei cammini che vengono dal basso

Questa non è una cosa ottimale o naturale, perché non ho le informazioni del prima, ma del dopo:
Con un esempio, non si comporta come il fattoriale dove $n!$ = n(n-1)!$ e dal fondo ottengo quello che sto in cima*
Come si può fare? Con l'utilizzo dei parametri




```
def sommac(alb, parz = 0):
        if alb.is_empty():
                return tree()
        newlabel = alb.label() + parz
        return tree(newlabel, sommac(alb.first_child(), newlabel),
sommac(alb.second_child(),
                                                                newlabel))
```

Questo tipo di visite vengono chiamate visite in profondità, in quanto prima arrivo in fondo e poi torno indietro
Differente è quella per livelli, però non è così facile con la ricorsione

Esercizio: Scrivere una funzione che restituisce una lista dei nodi che hanno solo etichette dispari su cammino radice-nodo, eccetto il nodo stesso.

```
def tuttidisp(alb):
        if alb.is_empty():
                return []
        if alb.label() % 2 == 0:
                return [alb.label()]
        else:
                return [alb.label()] + tuttisip(alb.first_child() +
tuttidisp(alb.second_child))
```

Questo tipo di visita è una visita posticipata

Struttura della Comprehension

Viene utilizzata nella teoria degli insiemi

Esempio:
$\{i \in [0,100] \mid \underbrace{i\%2 == 0}_{\text{Predicato}}\}$

Tutto questo serve per la compressione e serve per la logica

Python da a disposizione una tecnica simile:

```
'''Costruisco la lista degli int pari tra 0 e 100'''
res = []
for i in range(0, 101):
        if i % 2 == 0:
                res.append(i)
```

Questo è il modo standard semplice

Con la comprehension si può scrivere:

```
ress == [i for i in range(0, 101) if i%2 == 0]
```

Questa teoria viene chiama $\text{List Comprehension}$ perché riprende molto dalla Set Comprehension della matematica

```
resss = [j for j in range(0, 101, 2)]
```

Sono tutti esattamente la stessa cosa
Per fare un confronto:

Esercizio: Scrivi una funzione foo(f, n, k) che data una funzione di interi f e due interi n e k restituisce il grafico di f tra n e k come lista di coppie
Sia succ la funzione successore, foo(succ, 1, 5) deve restituire $[(1,2),(2,3),(3,4),(4,5),(5,6)]$

```
def foo(f, n k):
        return [(i, f(i)) for in range (n, k+1)]

def succ(k):
```

```
        return k + 1

[SHELL]
>>> foo(succ, 1, 5)
[(1, 2), (2, 3), (3, 4), (4, 5), (5, 6)]
```

foo si chiama funzione di ordine superiore in quanto da come risultato un'altra funzione
foo è applicabile a qualsiasi funzione

Nel metodo al quale eravamo abituati diventa:

```
def foo(f, n, k):
        res = []
        for i in range(n, k + 1):
                res.append((i, f(i)))
        return res
```

La definizione generale è estremamente variabile ma quella più semplice è:

$$[\text{ espressione } for \text{ nome } in \text{ sequenza }]$$

Il nome k non può più essere utilizzato in quanto è una variabile locale

Poi si può anche potenziare, ossia non solo tutti i valori della sequenza, ma devono soddisfare una certa condizione, ossia implementando un if
Volendo posso mettere non solo if (anche più di uno) ma anche altri for

Esempio: Lista delle coppie (i, j) con i e j tra n e k con i quadrato perfetto e j dispari

```
n = 2
k = 20
R = [(i**2, j) for i in range(n, k + 1) for j in range(m, k + 1) if j % 2 == 1 if i**2 <=
k]

[SHELL]
>>> R
[Lista di coppie con i = 4, 9, 16 e j = 3, 5, 7, 9, 11, 13, 15, 17, 19]
```

Quicksort:

```
def quicksort(L):
        if len(L) < 2:
                return L
        pivot = L[0]
                return quicksort([x for x in L if x < pivot]) + [pivot] + quicksort([x for
x in L if x >=
                                                                        pivot])
```

La si può utilizzare anche su diverse strutture logiche:
Per esempio un dizionario in cui i valori sono i quadrati delle chiavi:

```
D = {i : i**2 for i in range(21)}
```

Non si può fare le stesse cose con le tuple

```
>>> T = (i**2 for in range(11))
>>> T
<generator object> #Generatore
```

```
>>> tuple(T)
(0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100)
>>> tuple(T)
()
```

List Comprehension: Lista di determinati valori che rispettano una condizione

```
>>> Q = [i**2 for i in range(10)]
>>> Q
[1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Poi si possono mettere anche delle condizioni:

```
>>> Q = [i**2 for i in range(10) if i % 2 == 0]
>>> Q
[4, 16, 36, 64]
```

Altri esempi:

```
def inentrambe(W1, W2):
        'Restituisce una lista con gli elementi in comune'
        res = []
        for i in W1:
                if i in W2:
                        res.append(i)
        return res

def inentrambe2(W1, W2):
        'Restituisce una lista con gli elementi in comune'
        res [e for e in W1 if e in W2]
```

```
def primi(n):
        return [k for k in range(2, n + 1) if len([i for i in range(2, k - 1) if k % i ==
1]) == 0]

def primi(2):
        return [k for k in range(2, n + 1) if all([k % i != 0 for i in range(2, k-1)])]
```

La funzione all sopra usata restituisce True se tutti gli elementi sono True

Bisogna però non abusarne in maniera da rendere il codice illeggibile (e quindi difficile da correggere):

Esempio di abuso:

```
def inentrambe(W1, W2):
        'Restituisce una lista con gli elementi in comune'
        res = []
        for i in W1:
                if i in W2:
                        if i not in res:
                                res.append(i)
        return res

def inentrambe2(W1, W2):
        return [e for e in W1 if e in W2 if not in this list]
        #NO
        #Anche se fosse scritto bene, restituirebbe None, cioè niente
```

```
def inentrambe(W1, W2):
        res = []
        [res.append() for e in W1 if e in W2 if e not in res]
        return res
```

Esiste una cosa simile anche per i dizionari:

```
>>> D = {i : i**2 for i in range(10)}
>>> D
{0:0, 1:1, 2:4, 3:9, [...]}
```

Ma non con le tuple, infatti è un generatore.

```
>>> Q = (i : i**2 for i in range(10))
>>> Q
(1, 4, 9, 16, ...)
>>> Q
(0, 0, 0, 0, ...)
```

Se lo trasformassi in una lisa, ottengo una lista con i quadrati
Ma posso farlo una volta, dopo mi da una lista vuota

Un Generatore è una sequenza in potenza, che può generare ciò che gli chiediamo
Esiste una funzione che si chiama next che mi restituisce il primo (poi i successivi uno alla volta) di un generatore

Non ci sono modi per tornare a quello precedente, solo al successivo con next.
Da un'eccezione se dopo l'ultimo elemento non non ci sono altri elementi $<$ StopIteration $>$
I generatori sono coinvolti nell'interazione con for

Se provassi a fare una lista con un generatore finito ottengo una lista vuota [ ] e se facessi un print di tutti gli elementi della sequenza, non stampo niente

Convertire un generatore in una tupla significa svuotare il generatore con la funzione next, e di conseguenza diventa un generatore esaurito.

```
>>> H = (i for i in range(10**200))
>>> next(H)
0
>>> next(H)
1
#Eccetera
```

Non si può utilizzare un indice per avere un determinato lavoro

Si può usare un generatore per fare delle operazioni.

```
>>> sum([e for e in range(6)]) #Direttamene con una lista
15
>>> from math import prod
>>> prod([e for e in range(1, 6)])
120
>>> sum(e for e in range(6)) #Direttamente con il generatore in potenza, non lista
```

Un generatore non è altro che una variante di una definizione di funzione

```
G = (i**2 for i in range(10)) #Forma Implicita

def Gen():
```

```
        for i in range(10): #Forma Esplicita
                yield i**2
```

yield non è altro che un return particolare

La funzione restituisce un generatore. Se io facessi:

```
>>> G = Gen()
```

Chiamo G il generatore dato dalla funzione

La funzione fatta in questo modo non esegue il corpo, ma restituisce un generatore che può essere chiamato da next, cioè, come valore del generatore c'è l'elemento del generatore

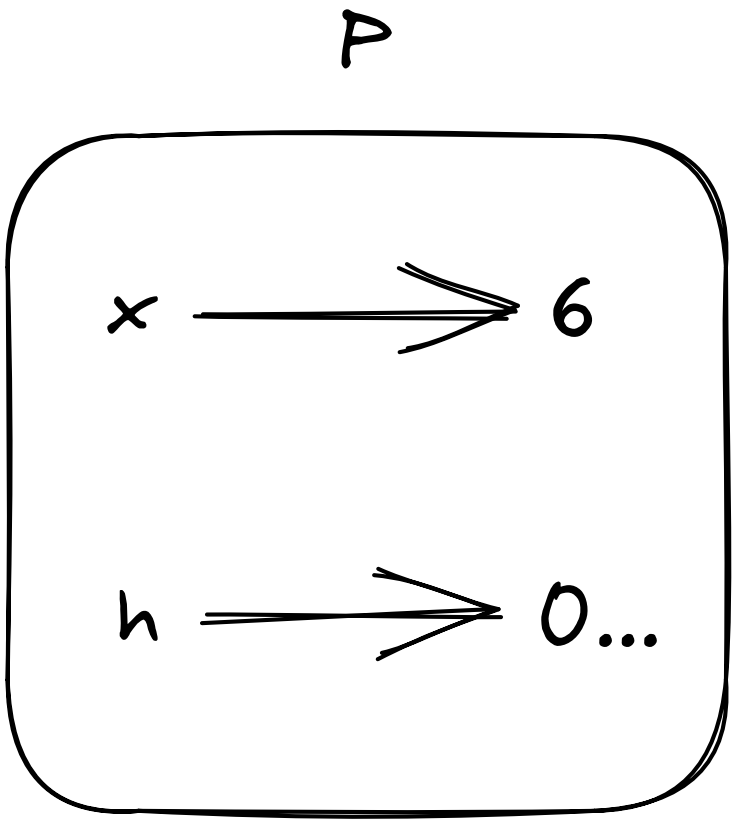
Ma, mentre return distrugge il frame su cui si stava lavorando, yield no, anzi si ricorda degli indici degli elementi e a quali elementi erano associati.

Altro generatore:

```
def Prova(x):
        h = 0
        while x <= x:
                if h % 2 == 0:
                        yield h
                print('Pippo')
                yield h
                h += 1
```

So che questa funzione restituisce un generatore perché c'è un yield nel corpo e non esegue i comandi nel corpo

```
>>> P = Prova(6)
>>> next(P)
0
>>>
```



Ogni volta che viene fatto un next viene sospesa la definizione della funzione.

```
>>> next(P)
'Pippo'
0
>>> next(P)
'Pippo'
1
>>> next(P)
2
>>> next(P)
'Pippo'
2
>>>
```

```
def Omega():
        i = 1
        while True:
                yield i**2
                i += 1

[SHELL]
>>> Q = Omega()
>>> next(Q)
0
>>> next(Q)
1
>>> next(Q)
4
>>>
```

Generatori utili per creare dei flussi potenzialmente infiniti di oggetti, chiamati Stream

Cenni di Informatica Teorica

Cosa significa calcolare?
Manipolazione dei simboli per ottenere un nuovo valore

Studio viene ben prima dei primi calcolatori elettronici

La prima definizione risale da 1936 dal matematico Alan Turing

La domanda che si pone fa parte del titolo "On Computable Numbers". Cosa significa che un numero è calcolabile con numeri reali?
Abbiamo capito che sono alcuni delle bestie perché sono una quantità non definibile, infinita e non periodica
In particolare, per ogni numero reale c'è la possibilità di creare un programma capace di creare tutti i numeri?

No, ci sono dei numeri reali per cui non è possibile stampare dei numeri reali
Oggi è molto semplice arrivarci, è un argomento elementare

Definiamo $\mathrm{Python} = \{p \mid p \text{ è un programma legale in Python}\}$ è l'insieme di tutti i programmi possibili
Qual è la sua cardinalità? Certamente è infinito ma è un infinito numerabile perché c'è una sequenza finita di simboli composti da caratteri finiti. Per cui $|\mathrm{Python}| = |\mathbb{N}|$
Sappiamo che $|\mathbb{R}| > |\mathbb{N}|$ quindi abbiamo la certezza matematica che non esiste un programma Python per ogni numero in $\mathbb{R}$ perché abbiamo un insieme numerabile di programmi
È vero che lavoriamo con dei numeri reali, come π e e, ma sono dei numeri reali con una quantità di informazioni limitata.

Turing fa tuttavia un ragionamento molto più interessante rispetto a quanto appena fatto.
Costruisce un numero reale che non è calcolabile (Non possiamo stampare l'esito preciso del numero reale)
Lo facciamo attraverso un detour dimostrando che "il problema della fermata non è decidibile" ossia:

Esiste un programma $\mathrm{Halt(P, x)}$ *di due argomenti che preso un qualsiasi programma Python* P *e oggetto* x*:*
1) Termina sempre
2) Restituisce True *se e solo se* $P(x)$ *termina,* False *se* $P(x)$ *termina*

Chiaramente senza eseguirlo

Dimostriamolo per assurdo, cioè se esistesse crerei un programma che non può esistere, in quanto sarebbe un assurdo logico:

Per farlo mi servono tre ingredienti:

1. So scrivere funzioni divergenti (cicli)

```
def Omega():           #Programma che va in ciclo
        while True:
                pass
```

2. Posso passare delle funzioni come argomento di altre funzioni:

```
from math import sqrt, sin
L = [1, 2, 3, 4, 5, 6]
def map(f, S):
        return [f(e) for e in S]
```

3. Non è vietato applicare una funzione a sé stessa:

```
def I(x):
        return x
```

Supponiamo quindi per assurdo di avere una funzione $\mathrm{Halt(P, x)}$ come sopra.

```
def Halt(P, x):
        < comandi >
        pass

def K(P):
        if Halt(P, P):
                Omega() #Da sopra, cioè va in ciclo
        else:
                return 0

Assurdo = K(K)
```

Perché $K(K)$ non può esistere?

Chiamando (K, K) si va a vedere $\mathrm{Halt}(K,K) = \begin{cases} \mathrm{Omega}() & \mathrm{Halt}(K,K) == True \quad \Leftrightarrow K(K) \text{ termina} \\ 0 & \mathrm{Halt}(K,K) == False \quad \Leftrightarrow K(K) \text{ non termina} \end{cases}$

Cioè se $\mathrm{Halt}(K,K)$ non termina, allora $K(K)$ termina e viceversa, ossia $K(K)$ termina $\Leftrightarrow$ $K(K)$ non termina

L'unica cosa supposta era l'esistenza di questo programma, quindi era l'unico assurdo

In sintesi ha invertito convergenza e divergenza per l'assurdo

Su alcune classi di programmi ci si può accorgere se termina o meno

Altri invece vanno in ciclo per motivi complessi, non un banale ciclo ripetitivo sulle stesse configurazioni.

Da tutto questo ci sono dei problemi che non hanno soluzioni

Il problema che sorge è la teminazione di un programma (in Python) per il quale non siamo in grado di restituire un programma che non sappiamo risolvere. (Non ci sono abbastanza programmi per risolvere tutti i problemi)

Chi mi dice che in altri linguaggi non ci sono modi per arginare il tutto?

Ho usato if, then, else, i cicli e passare una funzione ad un'altra funzione, cose che si possono fare sostanzialmente in quasi tutti i linguaggi di programmazione (l'utilizzo in Python è solo un'istanza)

Ogni formalismo per la descrizione del calcolo ha gli ingredienti per riprodurre questa dimostrazione

Posso farlo in Java, C, $\mathrm{Macchine\ di\ Turing}$,...

Non è una limitazione di Python, ma della nostra capacità di descrivere il calcolo, una sorta di filosofia naturale

"Non ci sono abbastanza programmi per descrivere tutti i problemi"

Posso dire di avere una biezione $r : \mathbb{N} \to \mathrm{Python}$ (del tipo, dato un numero posso trovare un programma e viceversa)

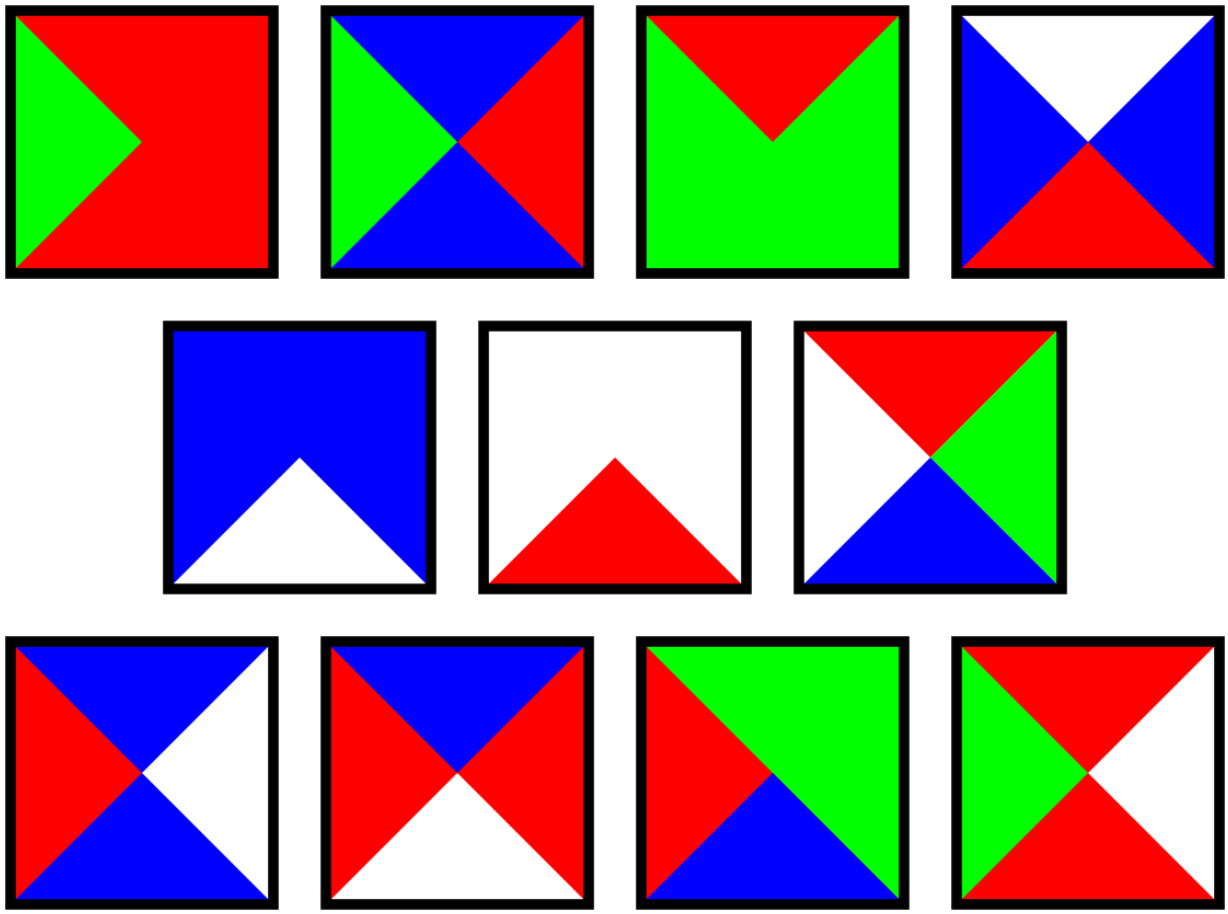
Definisco un numero $t = 0, d_0 d_1 d_2 \dots d_i$ dove $d_i = \begin{cases} 0 & \text{se } r(i)(r(i)) \text{ termina} \quad \Leftrightarrow \mathrm{Halt}(r(i), r(i)) = T \\ 1 & \text{se } r(i)(r(i)) \text{ diverge} \quad \Leftrightarrow \mathrm{Halt}(r(i), r(i)) = F \end{cases}$

Ho ridotto il problema della stampa del numero al problema della dedicibilità, per cui non ha soluzione.

Chiaramente t dipende dal linguaggio di programmazione e dalla scelta della biezione r, ma comunque ottengo lo stesso risultato, ossia che è impossibile

Altri problemi per cui non abbiamo algoritmi - soluzioni: problema di Piastrellature del Piano

Le Piastrelle di Wang sono delle piastrelle con dei colori sui bordi.



L'idea è che devono ricoprire il piano e possono essere attaccate se e solo se si toccano con lo stesso colore (senza ruotarle o ribaltarle)

Ci sono delle piastrellature periodiche e non periodiche (ossia che non fanno uno stesso stile in maniera uguale)

Per questo motivo il problema non è decidibile (dato per alcuni insiemi di piastrelle) e di conseguenza un programma capace di fare tutto ciò in maniera meccanica non esiste

Tutto questo per sottolineare che non è solo legato all'informatica

Ma c'è anche altro
Teorema: Sia $f : \mathbb{N} \to \mathbb{N}$. Se esiste un programma in Python che la calcola, esiste anche un programma in C, in Java e altro ancora. (Sono tutte equivalenti)

Non tutte le funzioni sono numerabili, ma quelle che lo sono, lo sono indipendentemente dal formalismo che io scelgo, infatti $\mathscr{C} = \{f : \mathbb{N} \to \mathbb{N} \mid f \text{ calcolabile}\}$

Tesi di Church-Turing: una funzione da $\mathbb{N}$ a $\mathbb{N}$ è intuitivamente calcolabile se e solo se è calcolabile in modo formale da una macchina di Turing (se e solo se è calcolabile anche in altri linguaggi di programmazione - è un principio di filosofia naturale, sono cose di cui non si può fare un teorema)

Questa tesi è al cuore della teoria della computazione moderna, se riconosciamo che qualcosa è calcolabile, allora ci si può scrivere un programma. *Ci sono stati dei problemi per scalfirlo, ma nessuno ci è mai riuscito*

Questi risultati di indicibilità sono in un certo senso assoluti, nel concetto di calcolo nell'universo.
Le cose cambiano quando vengono implementati dei vincoli (che vengono imposti a seconda della dimensione dei dati). Si passa da programmi con un indice di computazione pari a $O(1),\ O(n),\ O(n \log n)$, altri invece richiedono una lunghezza incredibile pari a $O(n!),\ O(n!)$ o addirittura $O(\infty)$

Esiste una classe di problemi chiamata NP-completi con certe proprietà:
1) Ha un potenziale di tipo Polinomiale
2) Ci sono solo soluzioni di tipo Esponenziale
3) Riesco a capire se una soluzione può essere considerata tale solo se uso la forza-bruta, provando direttamente

Un esempio continua ad essere quello delle piastrelle di Wang (l'unico modo che ho è di provare direttamente)

Poi ci sono problemi di classificazione ancora più difficili (tipo non si sa neanche se sono polinomiali)
Tra questo c'è il problema della fattorizzazione di numeri primi e l'isomorfismo tra grafi

Il più grande problema per tutti i problemi NP è $P \stackrel{?}{=} P$ cioè, è possibile trovare un programma a livello polinomiale per problemi non polinomiali?