Cenni di Programmazione Funzionale

"To Iterate is Human, To Recurse, Divine"

https://docs.python.org/3.6/library/functional.html

Cos'è la Programmazione Funzionale?

La programmazione funzionale è un **paradigma** di programmazione in cui l'operazione fondamentale è l'applicazione di funzioni ai loro argomenti.

Un programma non è altro che una funzione, definita in termini di altre funzioni, che elaborano l'input fornita al programma e restituiscono il loro risultato finale.

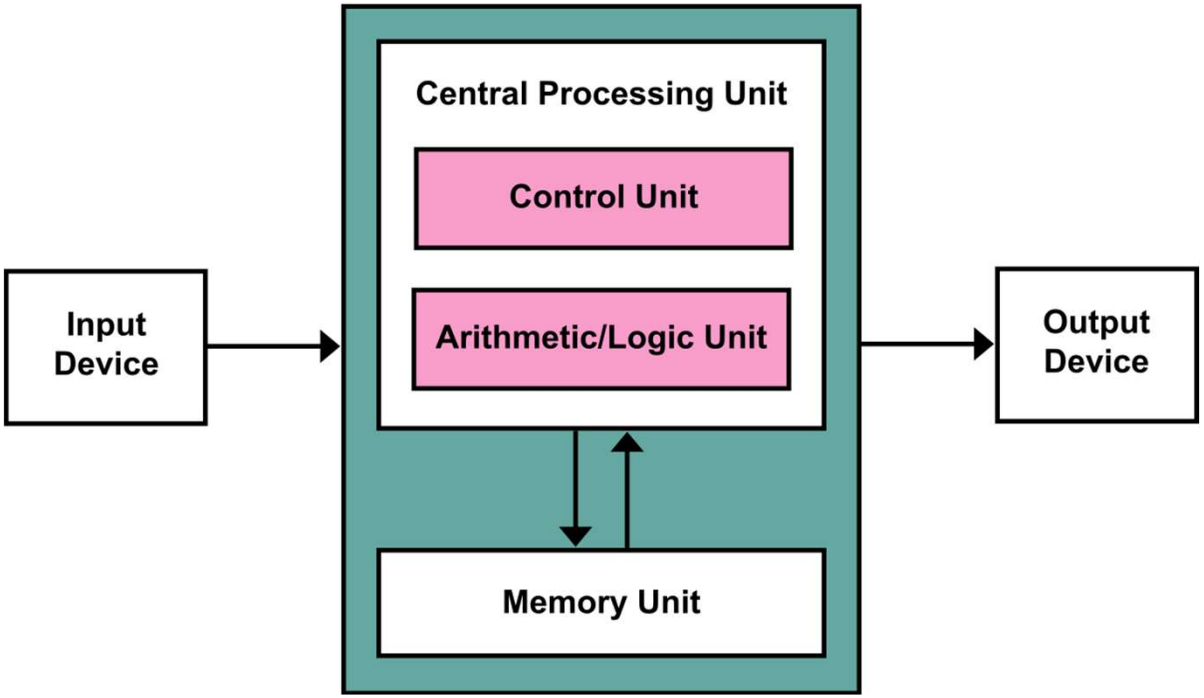
Tutte queste funzioni sono da intendersi come "*funzioni matematiche*", che vengono chiamate **FUNZIONI PURE**: a parità di input restituiscono sempre lo stesso output (no *side effects*)

Caratteristiche della Programmazione Funzionale

1. Le funzioni sono "*first class objects*": sono trattate come qualsiasi altro tipo di dato del linguaggio
2. La **RICORSIONE** è la principale struttura di controllo. Non esistono altri modi di iterare
3. Il focus principale è **processare liste**
4. Per evitare il più possibile effetti collaterali indesiderati, ogni variabile non può cambiare valore, ma è di tipo *read-only*
5. Vengono utilizzate ***High Order Functions***: funzioni che operano su funzioni, che operano su funzioni, che operano su funzioni, …
6. La valutazione di funzioni avviene in modalità **LAZY**. Per esempio è possibile definire funzioni su **liste infinite**

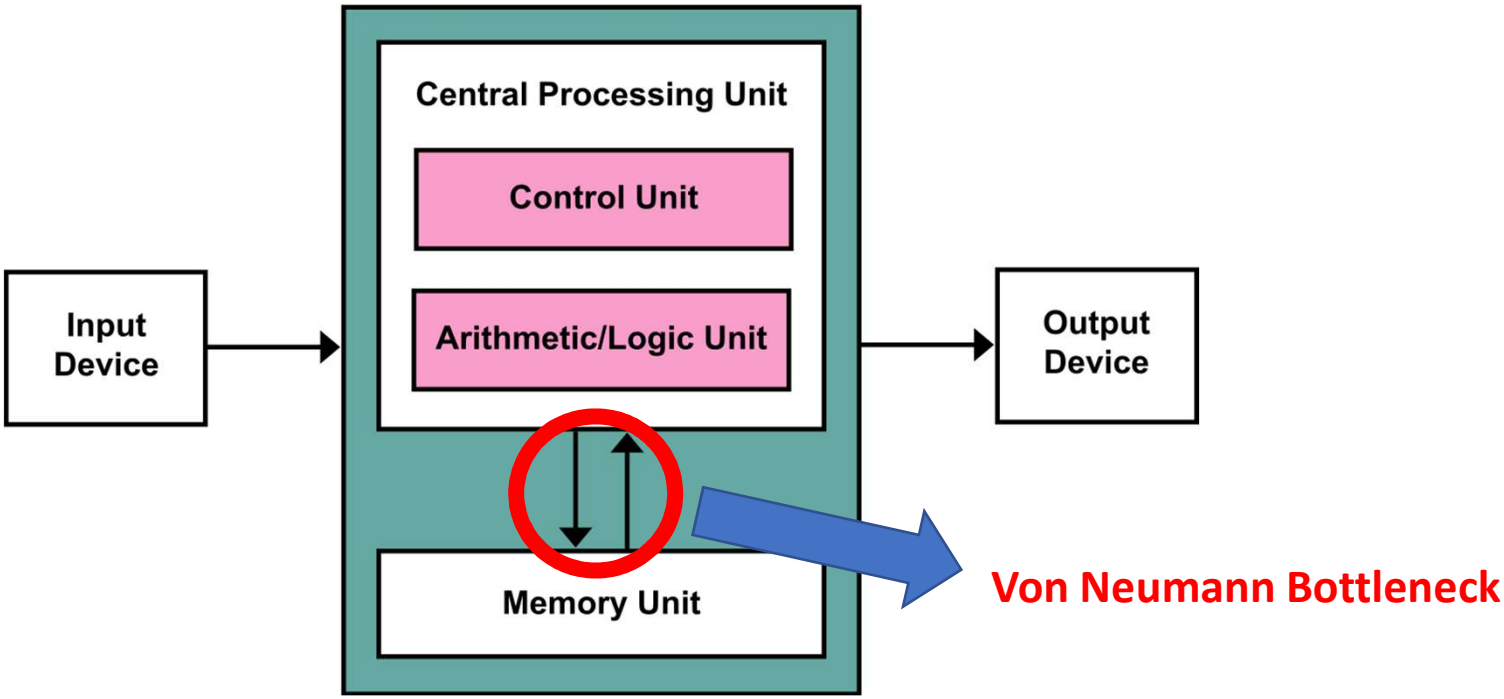
Caratteristiche della Programmazione Imperativa

La maggior parte dei linguaggi di programmazione sono basati su un paradigma di programmazione imperativo.
Questi linguaggi assumono, spesso implicitamente, che saranno eseguiti su un "**Computer di von Neumann**"



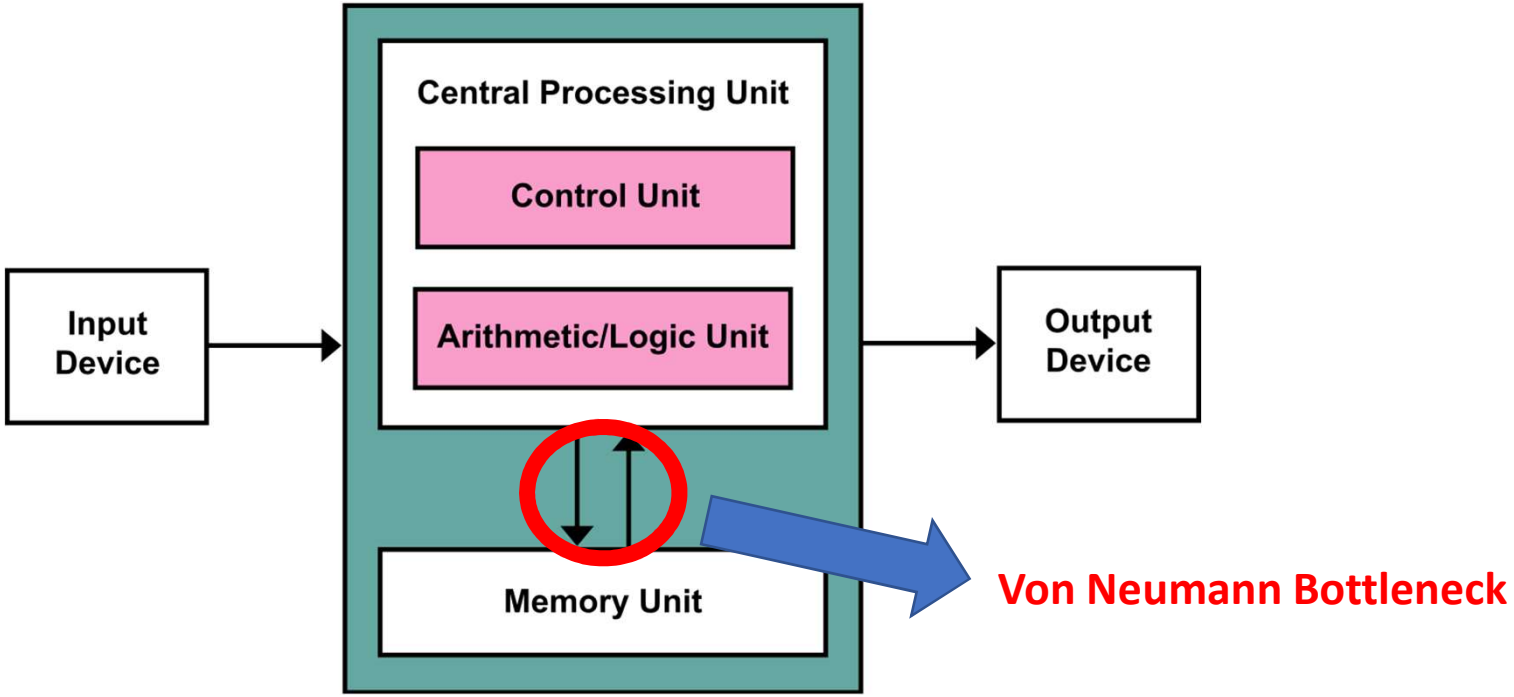
Caratteristiche della Programmazione Imperativa

La maggior parte dei linguaggi di programmazione sono basati su un paradigma di programmazione imperativo.
Questi linguaggi assumono, spesso implicitamente, che saranno eseguiti su un "**Computer di von Neumann**"



Von Neumann Bottleneck

La maggior parte del traffico dati tra la CPU e la memoria sono riguarda i dati, ma dove andare a trovarli e/o memorizzare.
I linguaggi di von Neumann usano le variabili per imitare le celle di memoria di un computer: *le variabili sono trattate come fossero delle celle di memoria*.



Assegnamento di variabili

Un comando di **assegnamento** divide un programma in due mondi diversi:

1. Il primo mondo comprende il ***right hand side*** dell'assegnamento, ed è dove avviene la parte importante dei calcoli. Esempio: c := **c + a[i]*b[i]**

Assegnamento di variabili

Un comando di **assegnamento** divide un programma in due mondi diversi:

1. Il primo mondo comprende il ***right hand side*** dell'assegnamento, ed è dove avviene la parte importante dei calcoli. Esempio: c := **c + a[i]*b[i]**

2. Il secondo è il mondo dei "comandi", di cui l'assegnamento è l'elemento principale. Questo secondo mondo è quello disordinato con **NESSUNA PROPRIETÀ MATEMATICA UTILE**. Per assemblare un risultato come sequenza di "comandi", i linguaggi di von Neumann offrono delle primitive comuni:

   for, **while**, **if-then-else**.

1. Funzioni come First Class

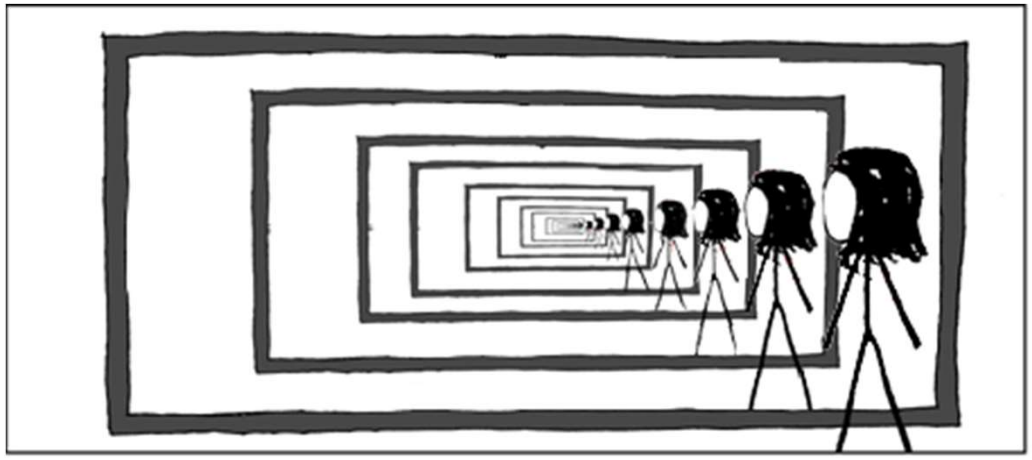
Esempio: Calcolo di un polinomio e di una sua derivata

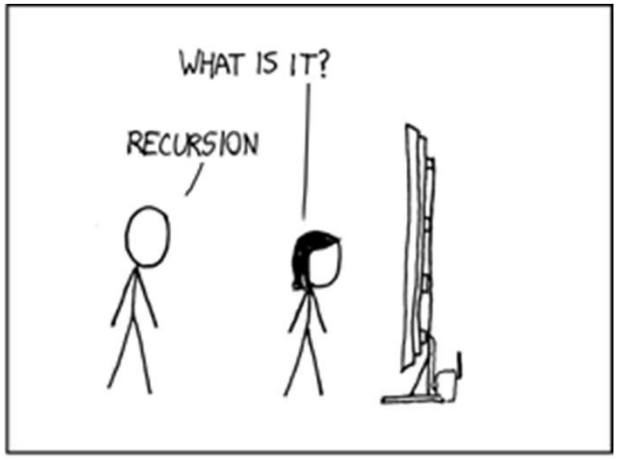
VEDERE NOTEBOOK

2. Funzioni ricorsive

In generale, una definizione **ricorsiva** (o induttiva) è fatta di due parti:

1. Il **caso base** che specifica il risultato per quel caso
2. Il **caso ricorsivo** che specifica lo stesso problema per un suo sottoproblema più semplice da risolvere





2. Funzioni ricorsive: Fattoriale

Definizione di fattoriale con “postfix” operatore **!**:

$$1! = 1$$

$$n! = n*(n-1)!$$

Oppure usando una notazione “prefix” per fattoriale:

```
Fattoriale(1) = 1
Fattoriale(n) = n*Fattoriale(n-1)
```

2. Funzioni ricorsive: Fibonacci

Funzione di Fibonacci:

$$\text{Female}(0) = 1$$

$$\text{Female}(1) = 1$$

$$\text{Female}(n) = \text{Female}(n-1) + \text{Female}(n-2)$$

NOTA:

1. In questo caso ci sono due **casi basi**
2. Nel caso ricorsivo ci sono due chiamate ricorsive

2. Funzioni ricorsive: Palindrome

Esercizio: scrivere una definizione ricorsiva per controllare se una data stringa è una palindrome, ovvero se li legge allo stesso modo sia da sinistra verso destra che da destra verso sinistra.

VEDERE NOTEBOOK

3. Processare liste

- Funzioni elementari sulle liste:

```
Head = lambda Ls: Ls[0]
Tail = lambda Ls: Ls[1:]
```

- Definizione di iteratori: un iteratore definisce un flusso di dati ("*stream*") e ne elabora uno alla volta
- Parole chiave riservate: **`iter`** e **`next`**

```
Ls = [1,2,3,4,3,2,1]
a = iter(Ls)
print(type(a))
next(a), next(a)
A = next(a)
print(A)     # COSA STAMPA??
```

4. Lazy Evaluation e Liste infinite

ESEMPIO:

1. funzione ***counter***
2. funzione ***enumerate***

VEDERE NOTEBOOK

ESERCIZIO: lista di numeri primi (Crivello di eratostene)

https://it.wikipedia.org/wiki/Crivello_di_Eratostene

4. Lazy Evaluation e Liste infinite

Libreria: **itertools**
https://docs.python.org/3.6/library/itertools.html

Da guardare: **groupby** (esercitazione venerdi)

Molto utili:

Iterator	Arguments	Results
`product()`	p, q, … [repeat=1]	cartesian product, equivalent to a nested for-loop
`permutations()`	p[, r]	r-length tuples, all possible orderings, no repeated elements
`combinations()`	p, r	r-length tuples, in sorted order, no repeated elements
`combinations_with_replacement()`	p, r	r-length tuples, in sorted order, with repeated elements
`product('ABCD', repeat=2)`		`AA AB AC AD BA BB BC BD CA CB CC CD DA DB DC DD`
`permutations('ABCD', 2)`		`AB AC AD BA BC BD CA CB CD DA DB DC`
`combinations('ABCD', 2)`		`AB AC AD BC BD CD`
`combinations_with_replacement('ABCD', 2)`		`AA AB AC AD BB BC BD CC CD DD`

5. Tail Recursion

```
def Sum(Ls):
    def SumRec(As, v):
        if As == EmptyList():
            return v
        else:
            return As[0] + SumRec(As[1:], v)
    return Sum(Ls[1:], Ls[0])
```

5. Tail Recursion

```
def Sum(Ls):
    def SumRec(As, v):
        if As == EmptyList():
            return v
        else:
            return As[0] + SumRec(As[1:], v)
    return Sum(Ls[1:], Ls[0])
```

```
def Sum(Ls):
    def TailRec(As, v):
        if As == EmptyList():
            return v
        else:
            return TailRec(As[1:], v+As[0])
    return TailRec(Ls[1:], Ls[0])
```

5. Tail Recursion

```
def ReverseTR(Ls):
    def ReverseRec(As, v):
        if As == EmptyList():
            return v
        else:
            return ReverseRec(As[1:], [As[0]] + v)
    return ReverseRec(Ls[1:], [Ls[0]])
```

5. Tail Recursion

```
def ReverseTR(Ls):
    def ReverseRec(As, v):
        if As == EmptyList():
            return v
        else:
            return ReverseRec(As[1:], [As[0]] + v)
    return ReverseRec(Ls[1:], [Ls[0]])


def Reverse2(Ls):
    def Cat(v, As):
        yield v
        for l in As:
            yield l

    def ReverseRec(As, v):
        if As == EmptyList():
            return [v]
        else:
            return Cat(As[-1], ReverseRec(As[:-1], v))
    return ReverseRec(Ls, [])

As = Reverse2([4,3,2,5,6,3])
print(next(As), next(As), next(As))
```

5. Tail Recursion

- **VANTAGGIO**: Una funzione Tail Recursive chiama immediatamente se stessa con i nuovi parametri sino a quando non raggiunge la fine della lista. Si noti, che ad ogni chiamata di una funzione ricorsiva, tutti i valori delle chiamate precedente (lo stackframe) non sono più necessari: Questo permette ai compilatori di ottimizzare le funzioni Tail Recursive.

- **SVANTAGGIO**: Se la funzione della ricorsione (che non è tail recursive) può produrre una parte del suo output prima di processare tutta la lista, allora può essere implementata in modo da gestire le liste infinite e supportare la **LAZY EVALUATION**

6. Ricerca Lineare

Un **algoritmo di ricerca** (*search algorithm*) è un algoritmo per cercare un dato elemento con specifiche proprietà all'interno di una "collezione" di elementi.

La collezione di elementi viene di solito chiamato lo **spazio di ricerca** (*search space*).

Consideriamo ora 3 algoritmi di ricerca di elementi all'interno di una lista, che abbia la seguente **specifica:**

```
def Search(Ls, e):
    # Si assuma Ls è una lista
    # Restituisce True se "e" appartiene
    # alla lista, False altrimenti
```

6. Ricerca Lineare: Versione 0

```
def Search0(Ls, e):
    # Si assuma Ls è una lista
    # Restituisce True se "e" appartiene
    # alla lista, False altrimenti
    return e in Ls
```

6. Ricerca Lineare: Versione 1

```
def Search0(Ls, e):
    # Si assuma Ls è una lista
    # Restituisce True se "e" appartiene
    # alla lista, False altrimenti
    return e in Ls


def Search1(Ls, e):
    for i in range(len(Ls)):
        if Ls[i] == e:
            return True
    return False
```

DOMANDA: COMPLESSITÀ DI QUESTA IMPLEMENTAZIONE?

6. Ricerca Lineare: Versione 2

```
def Search1(Ls, e):
    for i in range(len(Ls)):
        if Ls[i] == e:
            return True
    return False
```

```
def Search2(Ls, e):
    for l in Ls:
        if l == e:
            return True
    return False
```

DOMANDA: DIFFERENZE?

6. Ricerca Lineare: Versione 2

```
def Search2(Ls, e):
    for l in Ls:
        if l == e:
            return True
    return False
```

DOMANDA: RICORSIVA? E SE LA LISTA FOSSE ORDINATA?

```
def SearchRec(Ls, e):
    if Ls == EmptyList():
        return False
    if Head(Ls) == e:
        return True
    return SearchRec(Tail(Ls, e))
```

6. Ricerca Lineare: Versione 3

```
def Search2(Ls, e):
    for l in Ls:
        if l == e:
            return True
    return False
```

DOMANDA: SE LA LISTA FOSSE ORDINATA?

```
def Search3(Ls, e):
    for l in Ls:
        if l == e:
            return True
        if l > e:
            return False
    return False
```

NOTA: MIGLIORA L'AVERAGE RUNNING TIME, NON IL WORST CASE RUNNING TIME!

7. Ricerca Binaria

Come possiamo generalizzare l'idea di **Bisection Search** vista per trovare la radice quadrata di un numero?

7. Ricerca Binaria

Come possiamo generalizzare l'idea di **Bisection Search** vista per trovare la radice quadrata di un numero, supponendo che la lista sia ordinata in modo crescente?

IDEA PRINCIPALE:

1. Si prenda l'indice "mid" che individua l'elemento mediano della lista
2. Lo si confronta con l'elemento "e" che stiamo cercando: se sono uguali abbiamo finito e ritorna **True**
3. Confrontiamo il mediano con "e": se il mediano è minore, continuamo la ricerca nella metà sinistra della lista, altrimenti in quella destra

7. Ricerca Binaria

```
def BinarySearch(Ls, e):
      def bSearch(Ls, e, low, high):
            if low >= high
                  return e == low
            mid=(low+high)// 2
            if Ls[mid] == e:
                  return True
            if Ls[mid] < e:
                  return bSearch(Ls, e, mid+1, high)
            else:
                  return bSearch(Ls, e, low, mid-1)

      return bSearch(Ls, e, 0, len(Ls)-1)
```

DOMANDA: COMPLESSITÀ WORST-CASE?

8. Libreria functools

- Documentazione:
  https://docs.python.org/3/library/functools.html

- La libreria contiene la funzione:

  `reduce(Op, Ls, z)`

  che non è altro che la nostra FoldRight

- Vedere l'esempio con Fibonacci per **lru_cache**