LABORATORIO DI PYTHON

DEFINIZIONE ED USO DELLA RICORSIONE IN PYTHON

10 Aprile 2019

Ricorsione

PER CAPIRE LA RICORSIONE, BISOGNA PRIMA CAPIRE LA RICORSIONE

Una funzione è ricorsiva se, **nella sua definizione**, compare (direttamente o indirettamente) una chiamata a se stessa.

Risolvere un problema ricorsivamente significa:

- Identificare il caso / i casi base: quelli per i quali la soluzione è banale, e la possiamo restituire direttamente
- **Supporre** di avere, come "`per magia`", già la soluzione al problema ma in un caso "più semplice" (es. se come parametro abbiamo un numero n, possiamo **supporre** di avere già la soluzione nel caso si abbia $n - 1$ come parametro; se abbiamo una stringa come parametro, possiamo supporre di avere già la soluzione per una sua sottostringa...)
- Usiamo la soluzione del problema più semplice, in aggiunta a qualche operazione, per risolvere il problema generale.

PER CAPIRE LA RICORSIONE, BISOGNA PRIMA CAPIRE LA RICORSIONE

Vogliamo calcolare il fattoriale di `n`

- Identificare il caso base: per definizione, $0! = 1$

```
    if n == 0:
        return 1
```

PER CAPIRE LA RICORSIONE, BISOGNA PRIMA CAPIRE LA RICORSIONE

Vogliamo calcolare il fattoriale di `n`

- Identificare il caso base: per definizione, $0! = 1$

```
    if n == 0:
        return 1
```

- **Supporre** di avere, come "`per magia`", già la soluzione al problema ma in un caso "più semplice". Supponiamo dunque di avere già `fattoriale(n−1)`

PER CAPIRE LA RICORSIONE, BISOGNA PRIMA CAPIRE LA RICORSIONE

Vogliamo calcolare il fattoriale di n

- Identificare il caso base: per definizione, $0! = 1$

```
    if n == 0:
        return 1
```

- **Supporre** di avere, come "`per magia`", già la soluzione al problema ma in un caso "più semplice". Supponiamo dunque di avere già `fattoriale(n-1)`
- Usiamo la soluzione del problema più semplice, in aggiunta a qualche operazione, per risolvere il problema generale. Se ho già il `fattoriale(n-1)`, mi basterà moltiplicarlo per `n` per ottenere `n!`.

```
return n*fattoriale(n-1)
```

PER CAPIRE LA RICORSIONE, BISOGNA PRIMA CAPIRE LA RICORSIONE

Vogliamo calcolare il fattoriale di n

- Identificare il caso base: per definizione, $0! = 1$

```python
    if n == 0:
        return 1
```

- **Supporre** di avere, come "`per magia`", già la soluzione al problema ma in un caso "più semplice". Supponiamo dunque di avere già `fattoriale(n-1)`
- Usiamo la soluzione del problema più semplice, in aggiunta a qualche operazione, per risolvere il problema generale. Se ho già il `fattoriale(n-1)`, mi basterà moltiplicarlo per n per ottenere $n!$.

```python
return n*fattoriale(n-1)
```

- Complessivamente:

```python
def fattoriale(n):
    if n == 0:
        return 1
    return n*fattoriale(n-1)
```

Vogliamo scrivere una funzione **ricorsiva** `potenza(b,e)` che presi due numeri b ed e calcoli b^e (senza usare l'operatore potenza `**`)

- Identificare il caso base: per definizione, $b^0 = 1$

```
    if e == 0:
        return 1
```

PER CAPIRE LA RICORSIONE, BISOGNA PRIMA CAPIRE LA RICORSIONE

Vogliamo scrivere una funzione **ricorsiva** `potenza(b,e)` che presi due numeri b ed e calcoli b^e (senza usare l'operatore potenza `**`)

- Identificare il caso base: per definizione, $b^0 = 1$

```
    if e == 0:
        return 1
```

- **Supporre** di avere, come "`per magia`", già la soluzione al problema ma in un caso "più semplice". Supponiamo dunque di avere già `potenza(b,e-1)` (cioè il valore di b^{e-1})

PER CAPIRE LA RICORSIONE, BISOGNA PRIMA CAPIRE LA RICORSIONE

Vogliamo scrivere una funzione **ricorsiva** `potenza(b,e)` che presi due numeri b ed e calcoli b^e (senza usare l'operatore potenza `**`)

- Identificare il caso base: per definizione, $b^0 = 1$

```
    if e == 0:
        return 1
```

- **Supporre** di avere, come "`per magia`", già la soluzione al problema ma in un caso "più semplice". Supponiamo dunque di avere già `potenza(b,e-1)` (cioè il valore di b^{e-1})
- Usiamo la soluzione del problema più semplice, in aggiunta a qualche operazione, per risolvere il problema generale.

Vogliamo determinare se una stringa è palindroma (esattamente, cioè se s==s[::-1]) in modo ricorsivo.

- Identificare il caso base: per definizione, se `len(s)==0` o `len(s)==1` la stringa è palindroma.

PER CAPIRE LA RICORSIONE, BISOGNA PRIMA CAPIRE LA RICORSIONE

Vogliamo determinare se una stringa è palindroma (esattamente, cioè se s==s[::-1]) in modo ricorsivo.

- Identificare il caso base: per definizione, se `len(s)==0` o `len(s)==1` la stringa è palindroma.
- **Supporre** di avere, come "`per magia`", già la soluzione al problema ma in un caso "più semplice". Supponiamo dunque di avere già una funzione che ci dice se `s[1:-1]` (cioè ho tolto il primo e l'ultimo carattere) è palindroma o no (cioè supponiamo di sapere già se `palindroma(s[1:-1])`)

Vogliamo determinare se una stringa è palindroma (esattamente, cioè se s==s[::-1]) in modo ricorsivo.

- Identificare il caso base: per definizione, se `len(s)==0` o `len(s)==1` la stringa è palindroma.
- `Supporre` di avere, come "`per magia`", già la soluzione al problema ma in un caso "più semplice". Supponiamo dunque di avere già una funzione che ci dice se `s[1:-1]` (cioè ho tolto il primo e l'ultimo carattere) è palindroma o no (cioè supponiamo di sapere già se `palindroma(s[1:-1])`)
- Usiamo la soluzione del problema più semplice, in aggiunta a qualche operazione, per risolvere il problema generale: se $s[1:-1]$ è palindroma, allora s è palindroma solo se cioè anche il primo e l'ultimo carattere sono uguali)

ESERCIZIO

Scrivere una funzione **ricorsiva** che, presa come parametro una stringa, ne restituisca l'inversa (senza ovviamente usare usare il **doppio** operatore di slicing [::-1])

Scrivere una funzione **ricorsiva** che, presa come parametro una stringa, ne restituisca l'inversa (senza ovviamente usare usare il **doppio** operatore di slicing [::-1])

- **Caso base**:
- **Ipotesi induttiva**:

- **Soluzione generale** usando l'ipotesi:

ESERCIZIO

Scrivere una funzione **ricorsiva** che, presa come parametro una stringa, ne restituisca l'inversa (senza ovviamente usare usare il **doppio** operatore di slicing [::-1])

- **Caso base**: L'inversa della stringa vuota è una stringa vuota.
- **Ipotesi induttiva**: conosco l'inversa della stringa dal secondo carattere in poi.
- **Soluzione generale** usando l'ipotesi: l'inversa della stringa originale è l'inversa della stringa dal secondo carattere in poi a cui aggiungo, alla fine, il primo carattere della stringa originale

ESERCIZIO

Scrivere una funzione **ricorsiva** che, presa come parametro una stringa, ne restituisca l'inversa (senza ovviamente usare usare il **doppio** operatore di slicing [::-1])

- **Caso base**: L'inversa della stringa vuota è una stringa vuota.
- **Ipotesi induttiva**: conosco l'inversa della stringa dal secondo carattere in poi.
- **Soluzione generale** usando l'ipotesi: l'inversa della stringa originale è l'inversa della stringa dal secondo carattere in poi a cui aggiungo, alla fine, il primo carattere della stringa originale

```python
def inversa(s):
    if len(s)==0:
        return ''
    return inversa(s[1:])+s[0]
    #oppure
    #return s[-1]+inversa(s[:-1])
```

Scrivere una funzione **ricorsiva** che prende come parametro una qualsiasi sequenza (tupla, stringa,...) e restituisce l'elemento minimo della sequenza. *Non usare* `for`, `while`, `min`.

Scrivere una funzione **ricorsiva** che prende come parametro una qualsiasi sequenza (tupla, stringa,...) e restituisce l'elemento minimo della sequenza. *Non usare* `for`, `while`, `min`.

- **Casi base**:
- **Ipotesi induttiva**:
- **Soluzione generale** usando l'ipotesi:

Scrivere una funzione **ricorsiva** che prende come parametro una qualsiasi sequenza (tupla, stringa,...) e restituisce l'elemento minimo della sequenza. *Non usare* `for`, `while`, `min`.

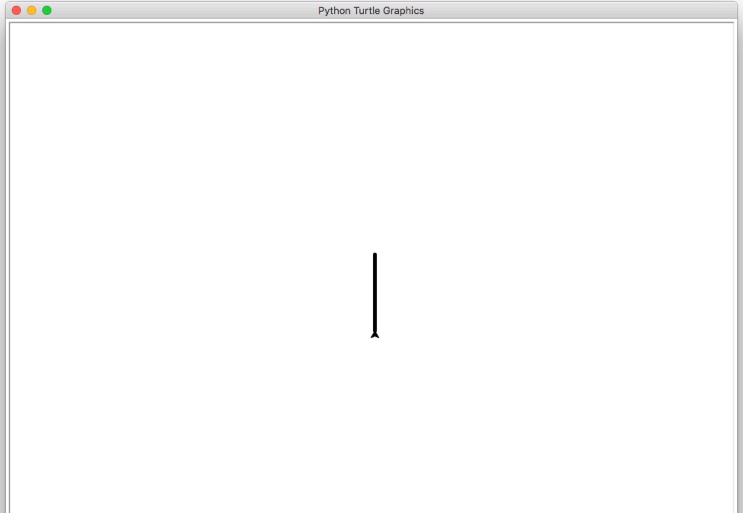
- **Casi base**: la sequenza vuota non ha minimo, la sequenza lunga 1 ha il suo unico elemento come minimo.
- **Ipotesi induttiva**: Conosco il minimo della sequenza che va dal secondo elemento alla fine, `m = minimoRic(s[1:])`
- **Soluzione generale** usando l'ipotesi: il minimo è dato dal più piccolo elemento tra il primo elemento (`s[0]`) e `m = minimoRic(s[1:])`

Scrivere una funzione **ricorsiva** che prende come parametro una qualsiasi sequenza (tupla, stringa,...) e restituisce l'elemento minimo della sequenza. *Non usare* `for`, `while`, `min`.

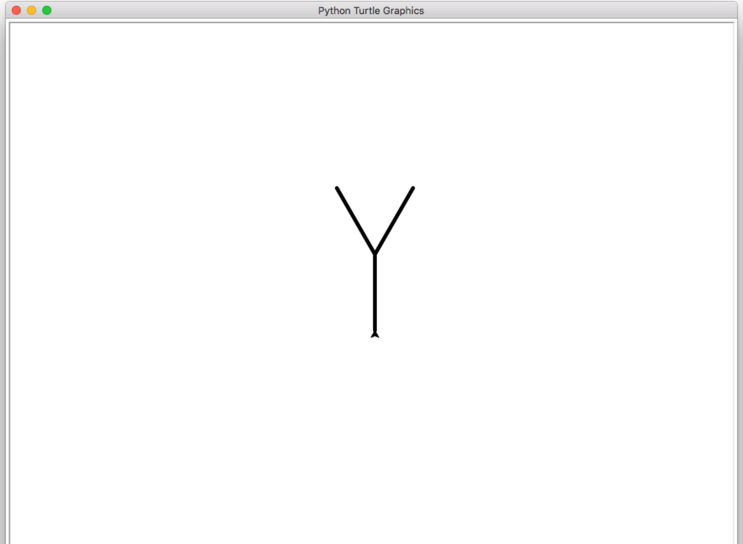
- **Casi base**: la sequenza vuota non ha minimo, la sequenza lunga 1 ha il suo unico elemento come minimo.
- **Ipotesi induttiva**: Conosco il minimo della sequenza che va dal secondo elemento alla fine, `m = minimoRic(s[1:])`
- **Soluzione generale** usando l'ipotesi: il minimo è dato dal più piccolo elemento tra il primo elemento (`s[0]`) e `m = minimoRic(s[1:])`

```
def minimoRic(s):
    if len(s)==0:
        return None
    if len(s)==1:
        return s[0]
    m = minimoRic(s[1:])
    if s[0] < m: return s[0]
    else: return m
```

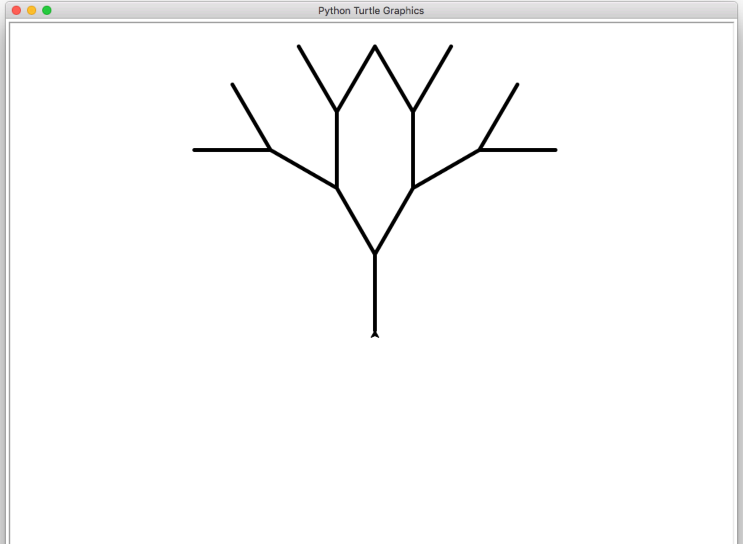
Un albero di livello 1 è un segmento rivolto verso l'alto. Notare che dopo aver disegnato il segmento, la turtle ritorna nella sua posizione iniziale.



Un albero di livello 2 è un segmento, sovrastato da due alberi di livello 1 inclinati rispettivamente di $-30°$ e di $+30°$



Un albero di livello 4 è un segmento, sovrastato da due alberi di livello 3 inclinati rispettivamente di $-30°$ e di $+30°$



Un albero di livello n è un segmento, sovrastato da due alberi di livello n-1.

- **Caso base**: Un albero di livello 1 è un segmento rivolto verso l'alto.
- **Ipotesi induttiva**: So disegnare un albero di livello n-1.
- **Soluzione generale**: disegno un segmento e poi invoco il disegno dei due alberi di n-1, opportunamente orientati.

```
import turtle

def albero(n):
    if n < 1:
        return
    if n == 1: #caso base
        turtle.forward(100)
        turtle.backward(100)
    else: #caso ricorsivo
        ...

#Programma di prova
#guardo in su
turtle.setheading(90)
#vado velocissimo
turtle.speed(0)
turtle.pensize(5)
albero(4) #provo
```

SOLUZIONE I

```python
import turtle

def albero(n):
    if n < 1:
        return
    if n == 1: #caso base
        turtle.forward(100)
        turtle.backward(100)
    else: #caso ricorsivo
        turtle.forward(100) #disegno il tronco
        turtle.left(30)     #mi giro a sinistra
        albero(n-1)   #disegno l'albero a sx
        turtle.right(30)    #mi rimetto al centro
        turtle.right(30)    #mi giro a destra
        albero(n-1)   #disegno l'albero a dx
        turtle.left(30)     #mi rimetto al centro
```

```
        turtle.backward(100)#tronco indietro

#Programma di prova
turtle.setheading(90) #guardo in su
turtle.speed(0) #velocissimo
turtle.pensize(5)
albero(4) #esempio di chiamata della funzione
```

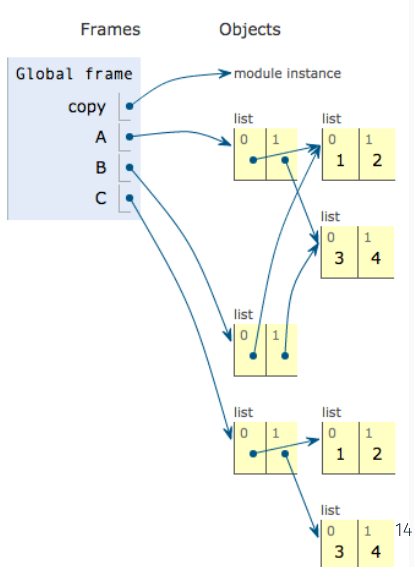
Se una lista ha al suo interno liste annidate, non basta usare `.copy()`.

```
import copy

A = [[1,2],[3,4]]

#copy() crea una copia della
    lista, ma non delle
    sottoliste
#di cui vengono copiati solo i
    riferimenti
B = A.copy()

#deepcopy permette invece di
    creare una copia
    indipendente
#andando in profondita' nelle
    sottoliste
C = copy.deepcopy(A)
```



Scrivere una funzione `copiaprofonda(L)` che crea una deep copy della lista `L` di elementi qualsiasi a qualsiasi livello di annidamento (es. `[1, [[2,4.5],[3,“ciao”]]]`) senza usare la funzione `deepcopy`.

Idea con ricorsione e iterazione: scorrere ogni elemento della lista e decidere se va inserito direttamente nella lista risultato (è un valore) oppure nella lista risultato ne va inserita una deepcopy (è una lista).

Idea con sola ricorsione: suppongo di avere già la deepcopy dal secondo elemento in poi. Decido in modo analogo al precedente se prendere il primo elemento (è un valore) o una sua deepcopy (è una lista).

DUE POSSIBILI SOLUZIONI

```python
def copiaprofonda(L):
    newL = []
    for e in L:
        if type(e)==list:
            newL.append(copiaprofonda(e))
        else:
            newL.append(e)
    return newL

def copiaprofonda2(L):
    if L==[]:
        return []
    if type(L[0])!=list: #se trovo valore, copio
        primo = [L[0]]
    else: #se trovo una lista, vado ricorsivamente
        primo = [copiaprofonda2(L[0])]
    return primo+copiaprofonda2(L[1:])
```

Sia

$$Fibonacci(n) = \begin{cases} 0 & \text{se } n = 0 \\ 1 & \text{se } n = 1 \\ Fibonacci(n-1) + Fibonacci(n-2) & \text{se } n > 1 \end{cases}$$

Scrivere una funzione **ricorsiva** `ListaFibonacci(n)` che restituisce una lista con la sequenza da $Fibonacci(0)$ a $Fibonacci(n)$, compreso. Non usare liste ausiliarie o funzioni ausiliarie.

Es.: `ListaFibonacci(10)` restituirà `[0, 1, 1, 2, 3, 5, 8, 13, 21, 34]`.

Suggerimenti: fare attenzione alla definizione corretta dei due casi base; fare uso della lista che si sta costruendo come strumento per "ricordare" i valori precedenti già calcolati.

```python
def ListaFibonacci(n):
    if n<0:
        return []
    if n==0:
        return [0]
    if n==1:
        return [0,1]
    l = ListaFibonacci(n-1)
    l.append(l[-1]+l[-2]) #non posso fare ora return
    return l
```

Esercizi per casa

1. Scrivere una funzione **ricorsiva** che calcoli la somma dei primi n numeri naturali dispari. **Non usare for o while**. (NB: Ad esempio se $n = 5$, i primi 5 numeri dispari sono $1, 3, 5, 7, 9$)
2. Scrivere una funzione **ricorsiva** che prende come parametri una tupla `t` e un numero **n**, e restituisce una nuova tupla in cui ogni elemento è stato moltiplicato per n. Esempio: $t = (4, 2, 5, 3)$, $n = 2$, la funzione restituisce $(8, 4, 10, 6)$. **Non usare for o while**.
3. Scrivere una funzione **ricorsiva** che presa come argomento una lista (semplice, non annidata) e un elemento, restituisce una **nuova** lista identica in cui sono stati eliminati gli elementi uguali a quello passato come parametro. Non usare **while** o **for**.