# Informatica per Statistica
# Riassunto della lezione del 22/11/2013

Igor Melatti

## Costrutti con sintassi diversa e uguale semantica

- I due tipi di cicli del Python visti qui (ovvero il `while`, ed il `for`) non hanno lo stesso potere espressivo
- In particolare, il `while` è più potente: si possono scrivere cicli `while` per i quali non c'è un corrispettivo ciclo `for` (almeno, non senza fare altri cicli)
- Invece, è sempre possibile prendere un qualsiasi ciclo `for` e scrivere un ciclo `while` con la stessa semantica (ovvero, che fa la stessa cosa)
  - un generico ciclo `for` sarà scritto così
    `for` $v$ `in` $E$`:` $B$
  - con un po' di semplificazioni e supponendo che la variabile `i` non sia usata nel blocco $B$, un ciclo `while` con la stessa semantica è il seguente:
    ```
    i = 0
    while (i < len(E)):
          v = E[i]
          B
          i = i + 1
    ```
  - infatti, ogni ciclo è condizionato al fatto che la sequenza $E$ non sia ancora esaurita; alla fine del blocco di istruzioni $B$, e prima di ricontrollare nuovamente se manca qualche elemento di $E$, si passa all'elemento successivo
  - **esercizio**: considerare un programma contenente un ciclo `for`, riscriverlo con un ciclo `while` come sopra e controllare che funzioni
- **Esercizio**: scrivere una semplice funzione che mostri la differenza tra un `break` e una `return` (ovvero, se si sostituisce il `break` con una `return`, la semantica della funzione non è più la stessa)
- **Esercizio**: scrivere una semplice funzione che mostri come un `break` e una `return` possano avere lo stesso effetto (ovvero, se si sostituisce il `break` con una `return`, la semantica della funzione è la stessa)

| $p$ | `not`$(p)$ |
|---|---|
| 0 | 1 |
| 1 | 0 |

Table 1: Tabella di verità del `not`

## Un po' di logica

- Si è detto che le espressioni possono avere diversi tipi
- Tra le altre, ci sono le espressioni *booleane*, ovvero il cui valore è o vero o falso
  - ad esempio, `3*x*x + f1(3) < 2` è una espressione booleana
  - ad esempio, `3*x*x + f1(3)` non sarebbe una espressione booleana
  - in realtà, in Python tutto è (anche) booleano: una espressione con valore numerico è vera se è tale valore numerico è diverso da 0, e vera altrimenti
  - un'espressione con valore di tipo stringa è vera se e solo se tale stringa non è vuota (ovvero `""` oppure `''`)
  - un'espressione con valore di tipo lista è vera se e solo se tale lista non è vuota (ovvero `[]`)
  - questo può creare confusione se non si sta ben attenti: l'espressione `[0]` è vera o falsa? l'espressione `"vuota"` è vera o falsa?
- È possibile combinare più espressioni booleane in modo da ottenere nuove espressioni booleane
  - i modi di combinare espressioni booleane possono essere ricondotti a 2 sole operazioni; qui comunque ne si considerano 3, le più usate normalmente, che sono anche le uniche disponibili in Python: `and, or, not` (scritti in questo modo, minuscoli)
  - $E_1$ `and` $E_2$ è vera se e solo se $E_1$ ed $E_2$ sono entrambe vere
  - $E_1$ `and` $E_2$ è falsa se e solo se almeno una tra $E_1$ ed $E_2$ è falsa
  - $E_1$ `or` $E_2$ è vera se e solo se almeno una tra $E_1$ ed $E_2$ è vera
  - $E_1$ `or` $E_2$ è falsa se e solo se $E_1$ ed $E_2$ sono entrambe false
  - `not`$(E)$ è vera se e solo se $E$ è falsa
  - `not`$(E)$ è falsa se e solo se $E$ è vera
  - nelle Tabelle 1 e 2 sono riportate le tabelle di verità: induzione perfetta, per le funzioni booleane si conoscono tutti i possibili casi di input, quindi si possono tabulare facilmente (si provi a tabulare alla stessa maniera la funzione somma tra interi...)

| $p$ | $q$ | $p$ `and` $q$ | $p$ `or` $q$ | $p$ `xor` $q$ | $p \to q$ | $p = q$ |
|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 1 | 1 | 0 | 0 |
| 1 | 1 | 1 | 1 | 0 | 1 | 1 |

Table 2: Tabelle di verità dell'`and` e dell'`or`, e di altre operazioni booleane famose

  - le altre operazioni booleane si possono tutte esprimere usando solo `not`, `and` ed `or`, come ad esempio l'implicazione $E_1 \to E_2 \equiv$ `not` $(E_1)$ `or` $E_2$ e lo XOR (OR esclusivo, mentre l'`or` del Python è inclusivo) $E_1$ `xor` $E_2 \equiv$ (`not` $(E_1)$ `and` $E_2$) `or` (`not` $(E_2)$ `and` $E_1$)

- Valutazione cortocircuitata: figlia del fatto che `False and` $E$ è sempre falsa e `True or` $E$ è sempre vera
  - quando viene valutato un `and` tra due espressioni, se la prima espressione è falsa, allora la valutazione è subito falsa, senza controllare anche la seconda espressione
  - ad esempio, si consideri l'espressione `i < len(A) and A[i] > 4`, e si supponga che `i` valga 10 e `A` sia lungo 9
  - senza valutazione cortocircuitata, Python darebbe errore nel valutare l'espressione di cui sopra: `A[10]` non esiste
  - con la valutazione cortocircuitata, Python valuta semplicemente l'espressione di cui sopra come `False` perché, una volta che risulta falsa `i < len(A)`, non valuta per niente `A[i] < 4`
  - su considerazioni di questo tipo ci si fa affidamento molto spesso: vedere la condizione del `while` dell'*Insertion Sort*
  - conseguenza: nonostante l'`and` (e l'`or`) sia logicamente un'operazione commutativa, non lo è nella versione Python (vale anche per altri linguaggi di programmazione)
  - infatti, `i < len(A) and A[i] > 4` potrebbe ritornare falso quando `A[i] > 4 and i < len(A)` ritorna un errore
  - piccola chiosa: dal momento che in questo corso non sono state trattate le *precedenze* tra operatori, è meglio precisare sempre l'ordine delle operazioni nelle espressioni complesse, quindi è meglio scrivere `(i < len(A)) and (A[i] > 4)`

# Strutture dati e operazioni per pile e code

- Una *pila* (*stack*) è una *struttura dati* che rappresenta un insieme *dinamico*
  - ovvero, a questo insieme possono essere aggiunti o tolti elementi in qualsiasi momento

- La caratteristica della pila è che l'aggiunta e la rimozione di elementi avvengono secondo la politica *LIFO* (*last in first out*, l'ultimo ad essere stato aggiunto è il primo ad essere rimosso)
  - per questo si chiama pila: può essere immaginata come una sequenza di oggetti impilati uno sull'altro
  - per aggiungere un elemento, lo si aggiunge in cima
  - per rimouvere un elemento, lo si rimuove dalla cima
  - dato che nella cima della pila c'è l'elemento inserito più di recente, una rimozione rimuove l'ultimo elemento arrivato (da qui *LIFO*)
  - inoltre, il primo arrivato, che si trova in fondo alla pila, sarà l'ultimo ad essere preso (infatti si può dire anche *FILO*, *first in last out*)
- Assumendo di essere a conoscenza del numero massimo di elementi da memorizzare, si può realizzare uno stack con un array $S$ (da Stack...)
  - negli esempi che seguono, assumeremo che gli elementi da memorizzare siano interi
- Tuttavia, il solo array non basta: occorre un indice che dica qual è l'elemento attualmente in cima
  - infatti, un "normale" array $S$ ha sempre come elementi validi $S[1] \dots S[\text{length}[S]]$
  - quindi sarebbe impossibile dire in quale delle posizioni di $S$ si trovi l'elemento che è attualmente sulla testa dello stack (così da sapere dove effettuare una rimozione o un'aggiunta)
- Allora, oltre ad $S$ c'è anche $\text{top}[S]$ ("cima")
  - contiene l'indice di $S$ in cui è memorizzato l'ultimo elemento inserito
  - per esempio, se in $S$ sono stati inseriti (in quest'ordine) 3, 6, 9, 2, e $\text{length}[S] = 6$, allora $S = \langle 3, 6, 9, 2, x, y \rangle$, dove $x$ e $y$ potrebbero essere qualsiasi
  - se ora si volesse effettuare una rimozione, senza $\text{top}[S]$ non si saprebbe quale elemento prendere ($S[1]$? $S[6]$? $S[3]$?)
  - invece, c'è anche $\text{top}[S] = 4$, quindi se la prossima operazione è una rimozione allora $S[4]$ è l'elemento da rimuovere, mentre se è un'ulteriore aggiunta allora $S[4 + 1]$ è la posizione dove inserire
  - riassumendo, *in una pila $S$ le posizioni valide sono quelle che vanno dall'indice 1 a top$[S]$*
- Nelle pile, l'operazione di aggiunta viene chiamata *Push* ("premi", sottinteso: sulla cima della pila), mentre l'operazione di rimozione viene chiamata *Pop* ("togli")

```
1  InitStack(S)
2    top[S] ← 0
3
4  Push(S, x)
5    top[S] ← top[S] + 1
6    S[top[S]] ← x
7
8  Pop(S)
9    top[S] ← top[S] - 1
10   return S[top[S] + 1]
11
12 StackEmpty(S)
13   return top[S] = 0
14
15 StackFull(S)
16   return top[S] = length[S]
```

Figure 1: Operazioni su una pila

- In Figura 1 vengono riportati gli pseudocodici per le operazioni sulla pila
  - *InitStack* è la prima operazione da chiamare quando si vuole cominciare ad usare lo stack
  - *StackEmpty* ritorna vero se e solo se lo stack è vuoto, ovvero se non contiene alcun elemento
  - *StackFull* ritorna vero se e solo se lo stack è pieno, ovvero se contiene già length[$S$] elementi
  - è sbagliato sia effettuare una push su uno stack pieno (*overflow*), che effettuare una pop su uno stack vuoto (*underflow*)
  - la Figura 2 mostra come *Push* e *Pop* possano tenere conto di questi casi
- Tutte le operazioni di Figura 1 (e Figura 2) eseguono un numero costante di operazioni
  - non c'è nessun ciclo che scorra tutto l'array $S$
  - quindi, se $n$ =length[$S$], tutte queste operazioni sono *indipendenti* da $n$
- Quindi, la complessità delle operazioni sulla pila è $O(1)$
- Una *coda* (*queue*) è anch'essa una struttura dati che rappresenta un insieme dinamico

```
1  Push(S, x)
2    if StackFull(S)
3      return OVERFLOW_ERROR
4    top[S] ← top[S] + 1
5    S[top[S]] ← x
6
7  Pop(S)
8    if StackEmpty(S)
9      return UNDERFLOW_ERROR
10   top[S] ← top[S] - 1
11   return S[top[S] + 1]
```

Figure 2: *Push* e *Pop* con condizioni di errore

- La caratteristica della coda è che l'aggiunta e la rimozione di elementi avvengono secondo la politica *FIFO* (*first in first out*, il primo ad essere stato aggiunto è il primo ad essere rimosso)
  - per questo si chiama coda: può essere immaginata come una sequenza di oggetti infilati l'uno dopo l'altro
  - per aggiungere un elemento, lo si aggiunge in fondo alla fila
  - per rimouvere un elemento, lo si rimuove dall'inizio della fila
  - dato che all'inizio della fila c'è l'elemento inserito meno di recente (ovvero il primo arrivato a suo tempo), una rimozione rimuove il primo elemento arrivato (da qui *FIFO*)
  - inoltre, l'ultimo arrivato, che si trova in fondo alla fila, sarà l'ultimo ad essere preso (infatti si può dire anche *LILO*, *last in last out*)
- Assumendo di essere a conoscenza del numero massimo di elementi da memorizzare, si può realizzare una coda con un array $Q$ (da queue...)
- Tuttavia, il solo array non basta: occorre un indice che dica qual è l'elemento attualmente in fondo alla fila, e un altro che dica qual è l'elemento attualmente all'inizio della fila
  - infatti, un "normale" array $Q$ ha sempre come elementi validi $Q[1] \dots Q[\text{length}[Q]]$
  - quindi sarebbe impossibile dire in quale delle posizioni di $Q$ si trovi l'elemento che è attualmente all'inizio della coda (così da sapere dove effettuare una rimozione), oppure in quale delle posizioni di $Q$ si trovi l'elemento che è attualmente in fondo alla coda (così da sapere dove effettuare un'ulteriore aggiunta)
- Allora, oltre ad $Q$ c'è anche head$[Q]$ ("testa") e tail$[Q]$ ("coda")

- head[$Q$] contiene l'indice di $Q$ in cui è memorizzato il primo elemento inserito
- tail[$Q$] contiene l'indice di $Q$ in cui occorre inserire il prossimo elemento
- per esempio, se in $Q$ sono stati inseriti (in quest'ordine) 3, 6, 9, 2, e length[$Q$] = 6, allora $Q = \langle 3, 6, 9, 2, x, y \rangle$, dove $x$ e $y$ potrebbero essere qualsiasi
- se ora si volesse effettuare una rimozione, senza head[$Q$] non si saprebbe quale elemento prendere ($Q$[1]? $Q$[6]? $Q$[3]?)
- invece, c'è anche head[$Q$] = 1, quindi se la prossima operazione è una rimozione allora $Q$[1] è l'elemento da rimuovere
- inoltre c'è anche tail[$Q$] = 5, quindi se la prossima operazione è un'ulteriore aggiunta allora $Q$[5] è la posizione dove inserire
- riassumendo (ma vedere anche più avanti), *in una coda $Q$ le posizioni valide sono quelle che vanno dall'indice head*[$Q$] *a tail*[$Q$] *- 1*

- Nelle code, l'operazione di aggiunta viene chiamata *Enqueue* ("metti in coda"), mentre l'operazione di rimozione viene chiamata *Dequeue* ("togli dalla coda")

- Differentemente dalla pila, nella coda l'array $Q$ viene usato in modo *circolare*
  - l'idea è che tail[$Q$] viene incrementato ogni volta che c'è da fare una *Enqueue*
  - ma così facendo, si potrebbero fare al massimo $n$ =length[$Q$] operazioni di *Enqueue*
  - quando invece, la presenza di eventuali *Dequeue* libererebbero dello spazio in $Q$ per fare ulteriori *Enqueue*
  - questo spazio si libera all'inizio di $Q$
  - quindi, per trarre vantaggio da ciò, occorre che tail[$Q$] possa "sfondare" oltre length[$Q$] per tornare all'indice 1 (nel caso ovviamente che qualche *Dequeue* abbia liberato dei posti)
  - lo stesso farà ovviamente head[$Q$]
  - questo ha anche l'effetto di dover "sprecare" una posizione all'interno di $Q$
  - ovvero, in $Q$ si potranno memorizzare non length[$Q$] elementi, ma solo length[$Q$] - 1
  - infatti, nel caso della pila, dato che top[$S$] può essere sia incrementato che decrementato, per sapere se la pila è vuota basta vedere se è 0, e per vedere se la pila è piena basta vedere se è andato oltre length[$S$]

- nella coda no, perché è circolare, quindi a 0 non ci si va mai, e non è un errore superare length[$Q$]
- per stabilire se la coda è piena o vuota occorre quindi confrontare tra loro head[$Q$] e tail[$Q$]
- concettualmente, se sono uguali vuol dire che sono state fatte tanti inserimenti quante cancellazioni, quindi la coda è vuota
- non c'è invece modo di dire quando la coda è piena, dato che qualsiasi posizione reciproca tra head[$Q$] e tail[$Q$] sarebbe legittima
  * potrebbe succedere che head[$Q$] sia più grande di tail[$Q$]: ad esempio, se si fanno length[$Q$] + 1 *Enqueue*, con in mezzo una *Dequeue*
  * potrebbe succedere che head[$Q$] sia più piccola di tail[$Q$]: ad esempio, dopo la prima *Enqueue*...
- allora si dice che se head[$Q$] è uguale a tail[$Q$] + 1 la coda è piena: vuol dire che pur potendo inserire un elemento in tail[$Q$] (che attualmente è libero e disponibile), si rinuncia a farlo
- quindi, pur avendo length[$Q$] posizioni, si memorizzano solo length[$Q$] - 1 elementi
- riassumendo, *in una coda $Q$ le posizioni valide sono quelle che vanno dall'indice head[$Q$] a tail[$Q$] - 1 se head[$Q$] è minore di tail[$Q$], altrimenti sono quelle che vanno da head[$Q$] a length[$Q$] e poi da 1 a tail[$Q$] - 1*
- è da notare che, in uno stack $S$, top[$S$] punta all'ultimo elemento inserito, mentre, in una coda $Q$, tail[$Q$] punta al primo elemento libero; quindi $Q$[tail[$Q$]] non contiene un elemento valido, $S$[top[$S$]] sì (a meno che...?)
- invece, $Q$[head[$Q$]] contiene sempre un elemento valido (a meno che...?)

- In Figura 3 vengono riportati gli pseudocodici per le operazioni sulla coda
  - *InitQueue* è la prima operazione da chiamare quando si vuole cominciare ad usare la coda
  - *QueueEmpty* ritorna vero se e solo se la coda è vuoto, ovvero se non contiene alcun elemento
  - *QueueFull* ritorna vero se e solo se la coda è pieno, ovvero se contiene già length[$Q$] - 1 elementi
    * scritta così non considera però tutti i casi: se tail[$Q$] è uguale a length[$Q$], e head[$Q$] è 1, la coda sarebbe piena, ma *QueueFull* ritorna falso
    * **esercizio**: correggere *QueueFull* in modo da considerare il caso di cui sopra

```
1  InitQueue(Q)
2    head[Q] ← 1
3    tail[Q] ← 1
4
5  Enqueue(Q, x)
6    Q[tail[Q]] ← x
7    if tail[Q] = length[Q]
8    then tail[Q] ← 1
9    else tail[Q] ← tail[Q] + 1
10
11 Dequeue(Q)
12   x ← Q[head[Q]]
13   if head[Q] = length[Q]
14   then head[Q] ← 1
15   else head[Q] ← head[Q] + 1
16   return x
17
18 QueueEmpty(Q)
19   return head[Q] = tail[Q]
20
21 QueueFull(Q)
22   return head[Q] = tail[Q] + 1
```

Figure 3: Operazioni su una coda

- è sbagliato sia effettuare una Enqueue su una coda piena (*overflow*), che effettuare una Dequeue su una coda vuota (*underflow*)
- **esercizio**: correggere la *Enqueue* e la *Dequeue* di Figura 3 per tener conto di overflow ed underflow

- Tutte le operazioni di Figura 3 eseguono un numero costante di operazioni
  - non c'è nessun ciclo che scorra tutto l'array $Q$
  - quindi, se $n$ =length$[Q]$, tutte queste operazioni sono *indipendenti* da $n$

- Quindi, la complessità delle operazioni sulla coda è $O(1)$

## Implementazione delle pile in Python

- La Figura 4 mostra un modo di realizzare in Python le pile

```
1  def InitStack():
2    global S
3    global top
4    S = []
5    top = -1
6
7  def Push(x):
8    global S
9    global top
10   top = top + 1
11   if (len(S) > top):
12     S[top] = x
13   else:
14     S = S + [x]
15
16 def Pop():
17   global S
18   global top
19   top = top - 1
20   return S[top + 1]
```

Figure 4: Operazioni su una pila in Python

- realizzati con variabili globali: se si passassero `S` e `top` come parametri a `Push` e `Pop`, le modifiche a `top` non avrebbero alcun effetto
- questo perché verrebbero eseguite sulla copia locale di `top`, e non sul `top` della funzione che chiama `Push` e `Pop`
- analoghi problemi si avrebbero su `S`, alla riga 14

- in realtà sarebbe possibile evitare di avere variabili globali (che su programmi grandi possono generare confusione), ma va al di là degli attuali scopi di questo corso
- dato che ovviamente l'array `S` parte dall'indice 0, `top` va opportunamente scalato di 1 (vedere `InitStack`)
- importante: in questo caso, il `len(S)` del Python non è il `length[`$S$`]` dello pseudocodice
- infatti, `length[`$S$`]` è la massima lunghezza desiderata per lo stack: se ci si mette anche un solo elemento in più, scatta l'errore
- invece, il `len(S)` del Python ha sempre lo stesso significato che ha su qualsiasi altra lista: è il numero di elementi attualmente in `S`; per quanto lo riguarda, è sempre possibile aggiungere un elemento usando ad esempio la concatenazione, come all riga 14
- per introdurre una lunghezza massima anche in Python, occorre farlo e gestirlo esplicitamente (vedere più avanti)
- attenzione, il numero di elementi nella lista `S` non coincide con il numero di elementi "logicamente" presenti nello stack!
- esempio: dopo 5 `Push` e 3 `Pop`, la lista `S` ha comunque 5 elementi, come si può vedere facendosi scrivere `S` sullo schermo; invece, dal punto di vista "logico", nella lista ci sono solo 2 elementi...
- **esercizio**: aggiungere le funzioni mancanti e il controllo di errore su `Push` e `Pop`; per `Push`, immaginare di avere un ulteriore parametro `max_length`
- **esercizio**: aggiungere una funzione che scriva su schermo tutti gli elementi validi di uno stack
- **esercizio**: scrivere un programma che legge da tastiera dapprima un numero $k$ (che dev'essere minore della lunghezza dell'array usato per la pila), poi legge $k$ interi, e infine, sfruttando una pila, scrive i $k$ interi nell'ordine inverso a quello di immissione
- **esercizio**: riscrivere le funzioni della pila in modo tale che lo stack `S` sia non una variabile globale, ma un parametro delle funzioni. Lasciare invece `top` come variabile globale. Provare un programma di esempio: funziona? E se anche `top` viene passata come parametro delle funzioni, il tutto funziona ancora?