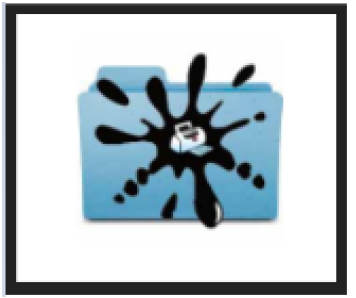
PYTHON

Mini Guida (Seconda Parte)

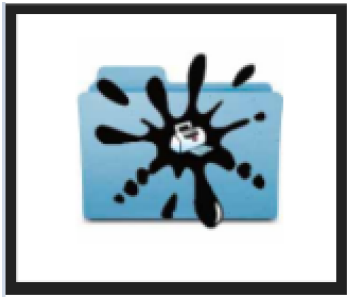
Il ciclo while



- Il ciclo while si differenzia dal ciclo for precedentemente descritto più a livello sintattico che funzionale, in generale è comunque possibile affermare che mentre nel ciclo for le iterazioni proseguono fino al verificarsi di una determinata condizione (ad esempio il raggiungimento di un dato incremento di valore a carico di una variabile), nel ciclo while le iterazioni continuano invece finché la condizione passata come argomento rimane vera.

- Anche in questo caso un semplice esempio potrebbe essere di aiuto per una migliore comprensione del costrutto in discussione, l'applicazione seguente permette di stampare tutti i valori inferiori al parametro di condizione ("20") partendo dal valore associato ad una variabile precedentemente definita che verrà incrementata di 2 unità ad ogni iterazione del ciclo.

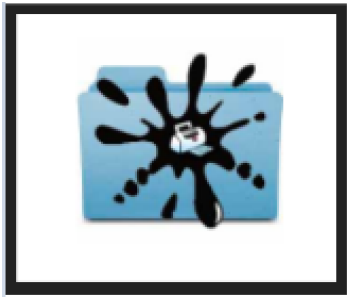
Il ciclo while



```
# Utilizzo del ciclo while
# per la stampa dei valori compresi tra 10 e 20
# vincolati all'incremento di una variabile

val = 10
while val < 20:
    print (val)
    val+=2
```

- L'esecuzione del codice porterà alla stampa della sequenza composta dai valori "10 (valore di base della variabile sottoposta ad incremento), 12, 14, 16 e 18", non verrà invece restituito in output il valore "20" che, rappresentando la condizione di terminazione dell'incremento ed essendo superiore al più alto valore consentito, verrà giustamente considerato al di fuori dell'intervallo di valori ciclato.

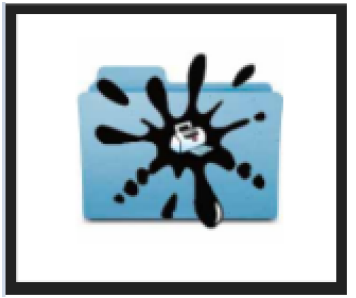
Il ciclo while



- Esattamente come nel caso di for, anche while può essere integrato tramite i blocchi condizionali, il codice dell'applicazione seguente introdurrà però una modalità inedita nella gestione degli output prodotti dai cicli e sottoposti ad una condizione introdotta tramite if:

```
# Utilizzo del ciclo while
# per la stampa dei valori compresi tra 10 e 20
# vincolati all'incremento di una variabile
# fino al raggiungimento del valore di confronto introdotto da if

val = 10
while val < 20:
    print (val)
    val+=2
    if val == 16:
        break
```

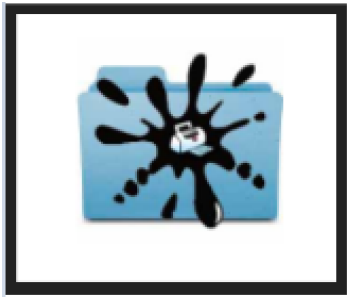
Il ciclo while



- Analizzando il sorgente proposto si noterà come la prima parte, quella relativa al blocco del ciclo while, sia del tutto identica al codice dell'esempio proposto in precedenza; questa volta però è stata implementata una seconda parte contenente una condizione per la quale l'incremento della variabile di base ("val") fino ad un valore pari a "16" porterà all'arresto del ciclo che non produrrà ulteriori iterazioni.

- Si otterrà quindi come risultato la sequenza "10, 12 e 14", la presenza dell'istruzione break alla fine del codice consentirà di terminare il ciclo che, altrimenti, continuerebbe nel tentativo di incrementare la variabile fino all'ultimo valore inferiore a "20" e, non riuscendoci a causa della condizione dovuta ad if, porterebbe alla generazione di un loop infinito, cioè di un ciclo che potrebbe essere fermato soltanto forzando l'arresto dell'applicazione.

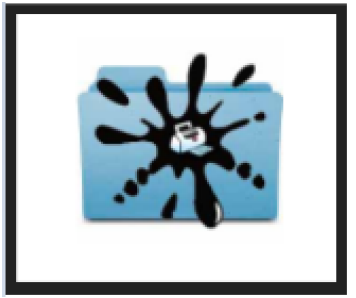
Il ciclo while



- Anche se il discorso è stato affrontato parzialmente nel capitolo precedente, alla luce delle nuove features descritte fino ad ora è possibile sottolineare come un altro punto in comune tra for e while sia rappresentato dalla possibilità di utilizzare un blocco condizionale quando richiesto dalla logica dell'applicazione sviluppata.
- A tal proposito, si analizzi quindi un primo esempio basato sul ciclo for che sarà anche riassuntivo per molti dei costrutti analizzati fino ad ora (liste, break..):

```
# Applicazione interattiva basata su un ciclo for
# destinato a verificare la presenza di un valore digitato dall'utente
# all'interno di una lista precedentemente definita

# definizione della lista
fibonacci = [1,1,2,3,5,8,13,21,34,55,89,144]

# valorizzazione di una variabile
# tramite l'input digitato dall'utente
val = int(input("Digita un valore compreso nella sequenza di Fibonacci: "))

# verifica del valore inviato in input
for x in fibonacci:
    if val == x:
        print(str(val) + " è un valore corretto.")
        break
else:
    print(str(val) + " non è un valore corretto.")
```

Il ciclo while

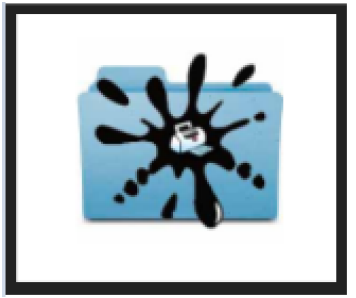


- Per quanto riguarda invece il ciclo while è possibile presentare un caso forse ancora più interessante, cioè quello che prevede l'introduzione di un blocco else senza la necessità di una precedente condizione definita tramite if.

- Ciò è possibile perché in while è presente una condizione definita esplicitamente che dovrà essere vera per consentire il proseguimento delle iterazioni:

```
# Uso del ciclo while per l'incremento del valore di una variabile
# fino al superamento di un valore limite gestito tramite else

val = 5
valore_limite = 50
incremento = 10

while val < valore_limite:
    print(str(val) + " è compreso nel valore limite pari a " + str(valore_limite))
    val = val + incremento
else:
    print(str(val) + " è superiore al valore limite pari a " + str(valore_limite))
```

Il ciclo while

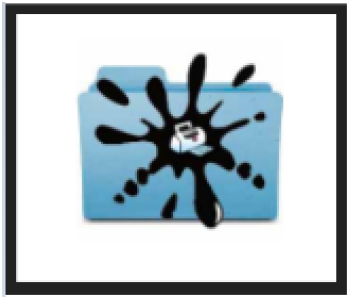


- L'output prodotto dall'esecuzione dell'applicazione proposta sarà il seguente:

```
5 è compreso nel valore limite pari a 50
15 è compreso nel valore limite pari a 50
25 è compreso nel valore limite pari a 50
35 è compreso nel valore limite pari a 50
45 è compreso nel valore limite pari a 50
55 è superiore al valore limite pari a 50
```

- Sostanzialmente il ciclo while dell'applicazione dovrà incrementare una variabile precedentemente definita ("val") fino ad un valore inferiore a quello limite, nel caso in cui quest'ultimo dovesse essere superato l'alternativa introdotta da else si occuperà di lanciare un'apposita segnalazione.

break e continue



- Grazie agli esempi proposti in precedenza è stato possibile introdurre l'istruzione break, quest'ultima, quando impiegata all'interno di un ciclo si occuperà di terminarlo in corrispondenza di un determinato evento, come per esempio il verificarsi di una condizione specifica o dell'eventualità opposta.

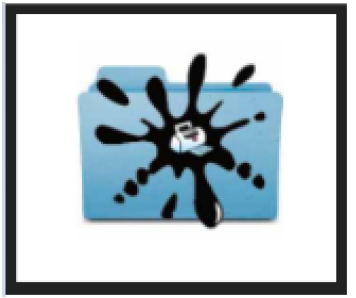
- Sostanzialmente break rappresenta uno strumento attraverso il quale influenzare il flusso di esecuzione di un ciclo e, più in generale, di un'applicazione.

- Per quanto riguarda la sintassi richiesta da Python per i costrutti basati su di esso, sarà necessario posizionare break subito dopo il corpo del loop, sia questo un ciclo for o un ciclo while.

- Nel caso in cui il progetto sviluppato preveda la definizione di cicli annidati (nested loops), sarà possibile inserire break dopo il corpo di un ciclo interno a quello iniziale.

- Data la sua natura, è logico dedurre che break sia stato concepito per essere impiegato in associazione a condizioni introdotte tramite if, nel caso di un ciclo for, per esempio, potremmo utilizzare questa istruzione per arrestare il loop in corrispondenza del verificarsi di una condizione di identità.

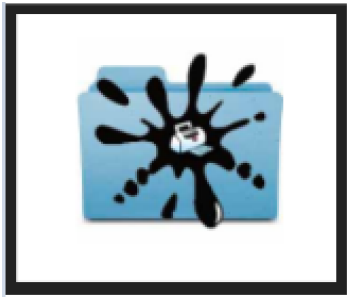
break e continue



- A questo proposito è possibile analizzare l'esempio seguente, esso infatti riassume in poche righe di codice quello che potrebbe essere il ruolo di break nel terminare un flusso di iterazione nel caso in cui la condizione definita tramite if risulti vera:

```
# Uso dell'istruzione break per terminare un ciclo
# al verificarsi di una condizione introdotta con if

for val in "python":
    if val == "h":
        break
    print(val)

print("Basta così.")
```

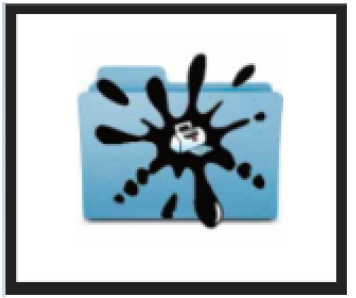
break e continue



- Eseguendo la piccola applicazione appena proposta si otterrà in output un risultato simile a quello mostrato di seguito:

p
y
t
Basta così.

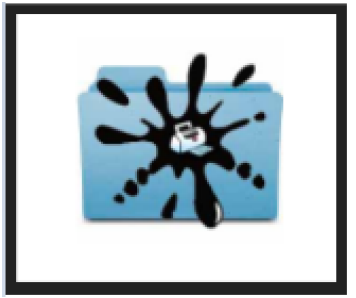
break e continue



- In pratica il for dovrà occuparsi di ciclare la stringa passata come argomento ("python") stampando uno per uno i caratteri che la compongono; teoricamente il loop applicato su tale parametro dovrebbe dare luogo a sei iterazioni, tante quanti sono i caratteri che compongono la stringa, ma la condizione introdotta da if richiede che venga verificata l'identità tra i carattere iterato e "h", se questa condizione dovesse essere soddisfatta allora break bloccherà l'esecuzione del ciclo impedendo ulteriori iterazioni e verrà stampata la notifica prevista per questo caso ("Basta così.").

- Grazie all'istruzione di terminazione inserita in un costrutto condizionale avremo quindi soltanto tre iterazioni più un messaggio (chiaramente opzionale) da parte dell'applicazione.

break e continue

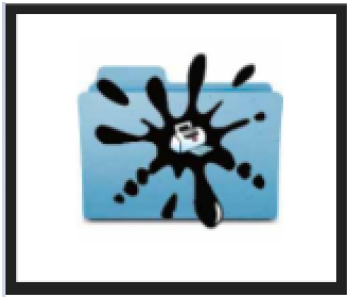


- Come già sottolineato, break risulta particolarmente utile quando si ha l'esigenza di terminare un ciclo in corrispondenza di una condizione precedentemente definita; in alcuni casi però lo sviluppatore potrebbe non necessitare di un arresto completo del loop utilizzato e voler semplicemente evitare che quest'ultimo produca delle specifiche iterazioni.

- A questo scopo è disponibile un'ulteriore istruzione denominata continue, essa in pratica arresta l'iterazione che soddisfa una determinata condizione ma permette il proseguimento del ciclo nella quale viene impiegata.

- Per chiarire meglio quanto detto sarà possibile proporre un semplice esempio realizzato applicando delle piccole modifiche al codice sorgente mostrato in precedenza.

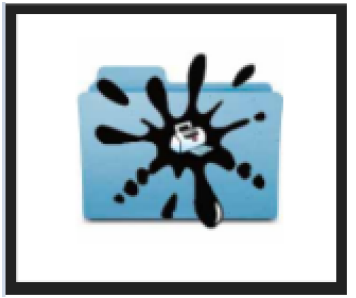
break e continue



```
# Uso dell'istruzione continue per arrestare un'iterazione
# al verificarsi di una condizione introdotta con if

for val in "python":
    if val == "h":
        continue
    print(val)

print("Ciclo completato.")
```

- Una volta eseguita l'applicazione appena digitata si otterrà in output un risultato come quello presentato di seguito.

break e continue

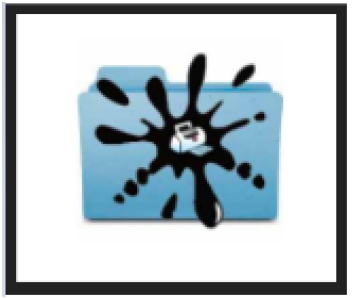


p
y
t
o
n
Ciclo completato.

- Anche in questo caso for avrà il compito di ciclare una stringa stampando in output uno alla volta tutti i caratteri da cui è composta, ciò però non avverrà se non in parte perché if introduce una condizione di identità.

- Nel momento in cui il carattere iterato dovesse essere uguale ad "h" questo verrà ignorato e non parteciperà al risultato del ciclo, il loop però non verrà interrotto e verranno effettuate tutte le iterazioni successive previste.

Le liste



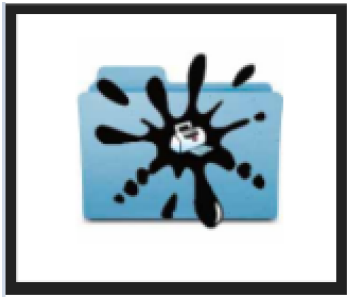
- In Python è possibile definire una lista, alla quale si potrà attribuire un nome, inserendo degli elementi tra parentesi quadre ("[..]"). Chi già lavora con altri linguaggi per la programmazione o lo sviluppo conoscerà sicuramente gli array, o "vettori", che sono delle variabili destinate a contenere ulteriori variabili.

- Sintatticamente e concettualmente le liste possono ricordare gli array, ma il loro funzionamento presenta delle peculiarità che le rendono differenti.

- Di base una lista può esistere ma essere vuota, cioè non presentare alcun elemento al suo interno:

```
# Esempio di lista vuota
nome_lista = []
```

Le liste

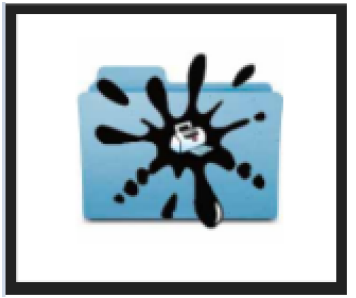


- Nel caso in cui una lista non sia vuota, sarà necessario separare gli elementi che la compongono tramite una virgola.

- E' possibile quindi definire liste che contengano elementi associati ad un solo tipo di dato:

```
# Esempio di lista contenente unicamente valori numerici
nome_lista = [15, 25, 35, 45, 55]
# Esempio di lista contenente unicamente delle stringhe
nome_lista = ['Homer', 'Bart', 'Lisa']
```

Le liste



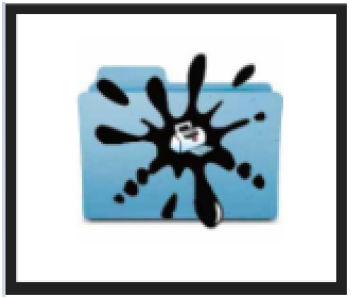
- Così come si potranno creare delle liste che presentino elementi di diversa natura:

```
# Esempio di lista contenente elementi di diverso tipo
# numeri e stringhe
nome_lista = [65, 'Homer', 7.3]
```

- E' infine consentita anche la creazione di liste annidate (nested list), cioè liste che hanno tra i propri elementi altre liste

```
# Esempio di lista annidata
nome_lista = ['Homer', 3.7, 10, [29, 39, 49]].
```

Le liste



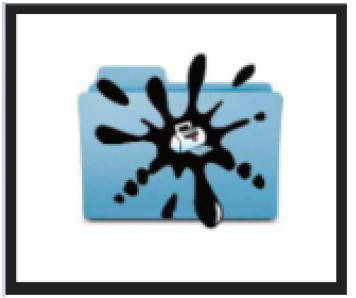
- range() è una funzione nativa di Python (o più propriamente un "immutable sequence type") concepita per generare automaticamente una lista sulla base di un intervallo di valori o di un valore numerico passato come argomento.

- Essa si rivela particolarmente utile quando si devono definire liste formate da una grande quantità di elementi numerici.

- Nel caso di un parametro espresso sotto forma intervallo si avrà che un'istruzione come la seguente:

```
# Utilizzo di range() per generare
# una lista sulla base di un intervallo
range(1,8)
```

Le liste

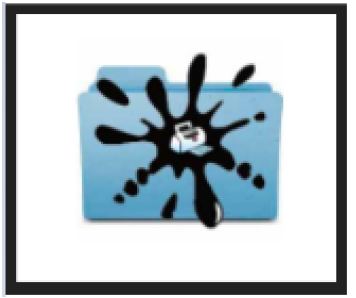


- range() porterà alla generazione di una lista composta da 7 elementi:

  [1, 2, 3, 4, 5, 6, 7]

- Tali elementi saranno il risultato della chiamata alla funzione range() che valuterà i due argomenti dell'intervallo restituendo una lista contenente tutti i valori interi a partire dal primo, incluso nella lista, fino ad arrivare al secondo, escluso invece dalla lista.

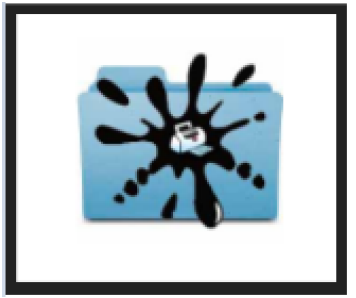
Le liste



- I due argomenti dell'intervallo potranno essere seguiti da un terzo argomento denominato step; esso specifica l'intervallo tra valori successivi, motivo per il quale se, per esempio, si volesse ottenere una lista composta dai soli numeri dispari compresi tra "1" e "8" si potrebbe operare in questo modo:

```
# Utilizzo di range() per generare
# una lista sulla base di un intervallo
# con step pari a 2
range(1,8,2)
# lista generata
# [1, 3, 5, 7]
```

Le liste



- Quando invece si passa alla funzione un intero come nell'esempio proposto di seguito:

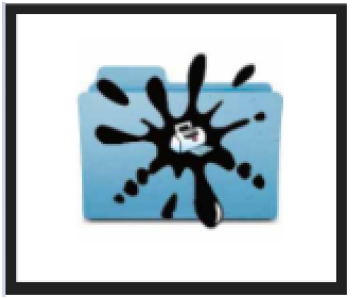
```
# Utilizzo di range() per generare
# una lista sulla base di un intero
range(8)
```

- si avrà come risultato una lista popolata in questo modo:

```
[0, 1, 2, 3, 4, 5, 6, 7]
```

- Gli elementi presenti saranno quindi 8, ponendo lo "0" come elemento e (non) valore iniziale.

Le liste

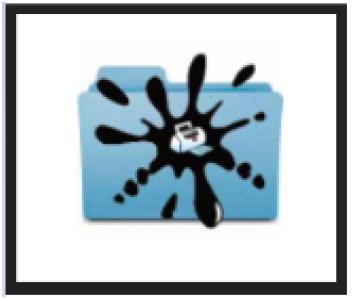


- Di default e liste in Python prevedono che gli elementi interni vengano indicizzati numericamente; l'indicizzazione partirà da "0", per cui una lista composta da tre elementi avrà come indici "0", "1" e "2".

- Ne consegue che un elemento specifico di una lista potrà essere richiamato tramite il suo indice, come nell'esempio seguente:

```
# Accesso ad un elemento di una lista
# attraverso il suo indice

# definizione degli elementi in lista
nome_lista = ['a','b','c','d','e']

# accesso all'elemento con indice "3"
nome_lista[3]
```

Le liste

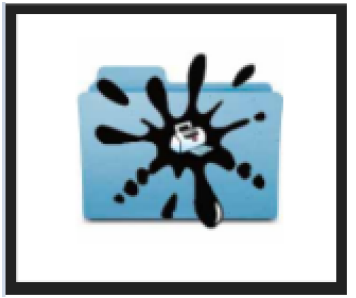


- L'elemento richiamato sarà in questo caso "d", cioè il quarto inserito in lista, questo perché "a" è associato all'indice "0", "b" a "1", "c" a "2" e di conseguenza "d" ha come indice "3".

- L'accesso agli elementi in lista potrebbe avvenire anche tramite indicizzazione negativa, dove l'ultimo elemento avrà come indice "-1", il penultimo "-2" e così via. Una chiamata come la seguente avrà quindi il risultato di consentire l'accesso all'elemento "a".

```
# Accesso ad un elemento di una lista
# attraverso il indice negativo

# definizione degli elementi in lista
nome_lista = ['a','b','c','d','e']

# accesso all'elemento con indice "-5"
nome_lista[-5]
```

Le liste



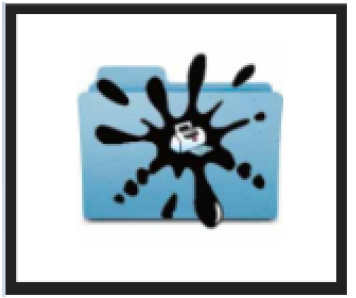
- E' inoltre possibile accedere agli elementi di una lista sulla base di un intervallo di valori.

- Si faccia attenzione al fatto che i due componenti dell'intervallo corrisponderanno ai numeri indice, motivo per il quale un'espressione come questa:

```
# Accesso a più elementi di una lista
# attraverso un intervallo

# definizione degli elementi in lista
nome_lista = ['a','b','c','d','e']

# accesso agli elementi con intervallo "0:2"
nome_lista[0:2]
```

- Consentirà di accedere agli elementi "a", "b", cioè a quelli il cui indice va da "0" a "2" (escluso).

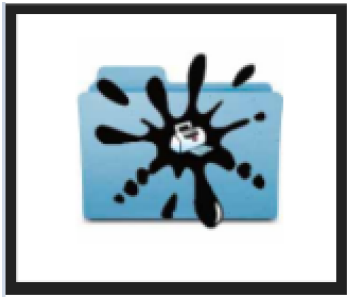
Le liste



- Il simbolo dei due punti (":"), utilizzato in precedenza per la definizione dell'intervallo, prende il nome di slicing operator. Esso potrà essere utilizzato anche per altri scopi, come per esempio definire l'indice dal quale partire per l'accesso ai dati:

```
# Accesso a più elementi di una lista
# a partire da un indice specifico

# definizione degli elementi in lista
nome_lista = ['a','b','c','d','e']

# accesso al primo elemento in lista ('a') tramite indice negativo
# gli ultimi 4 elementi verranno esclusi
nome_lista[:-4]

# accesso a partire dal terzo elemento fino all'ultimo
# 'c', 'd' ed 'e'
nome_lista[2:]

# accesso a tutti gli elementi in lista
nome_lista[:]
```

Le liste

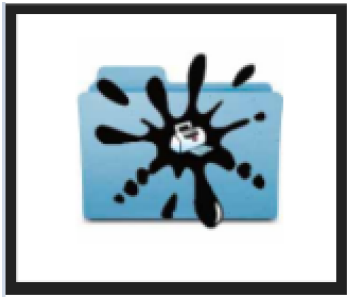


- Gli elementi presenti all'interno di una lista potranno essere modificati dinamicamente; nel prossimo capitolo analizzeremo un costrutto sintattico simile alle liste, le tuple che abbiamo analizzato parzialmente in precedenza; queste ultime funzionano in modo simile alle liste ma hanno la caratteristica di essere immutabili, mentre per quanto riguarda le liste si avrà una maggiore libertà nella manipolazione degli elementi assegnati.

- Per modificare un elemento di una lista è necessario fare ricorso all'operatore "=", cioè lo stesso che consente di assegnare dei valori alle variabili; si ipotizzi per esempio di voler modificare un elemento, il "6", presente nella seguente lista:

```
nome_lista = [10, 9, 8, 6]
```

Le liste



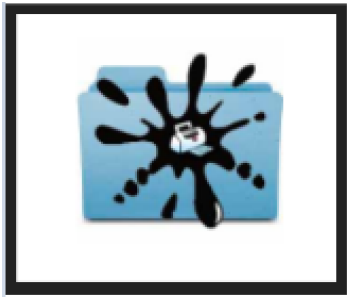
- In questo caso, dato che l'elemento che vogliamo modificare è associato all'indice "3", dovremo operare in questo modo:

  nome_lista[3] = 7

- Fatto ciò dovremmo ottenere una nuova lista formata dai seguenti elementi:

  nome_lista

Le liste



- In sostanza la sintassi per la sostituzione di un elemento in lista prevede l'assegnazione dell'elemento sostitutivo al numero indice della lista da modificare tramite l'operatore "=".

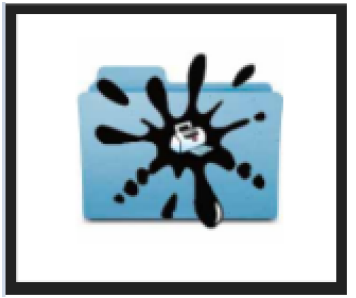
- E' chiaramente possibile anche l'operazione che permette di modificare più elementi di una lista alla volta, per cui un'istruzione come la seguente applicata alla lista ottenuta in precedenza:

  nome_lista[1:4] = [11, 12, 13]

- Determinerà la sostituzione degli elementi che vanno dal secondo fino al quarto tra quelli presenti il lista e la lista risultante sarà la seguente:

  nome_lista

Le liste



- Il metodo append() consentirà di inserire un nuovo elemento in lista accodandolo a quelli già presenti.

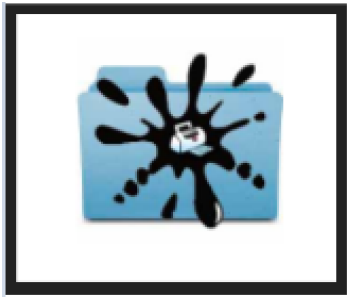
- Data l'ultima lista ottenuta tramite il precedente esempio, un'istruzione come la seguente:

  nome_lista.append.(14)

- Porterebbe alla generazione della lista proposta di seguito:

  nome_lista

Le liste



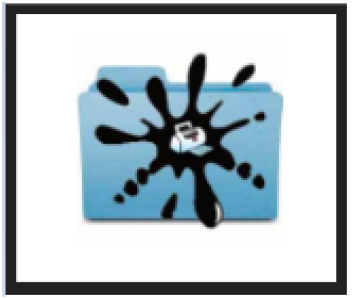
- Nel caso in cui si vogliano invece aggiungere più elementi ad una lista con un'unica istruzione, sarà possibile utilizzare un altro metodo, extend():

```
nome_lista.extend([15, 16, 17, 18])
```

- che nel caso specifico dell'esempio mostrato permetterà di ottenere la lista:

```
nome_lista
```

Le liste



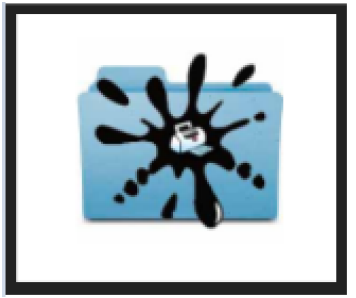
- Un altro caso interessante riguarda la concatenazione delle liste, operazione per la quale si farà ricorso all'apposito operatore di concatenazione simboleggiato com'è noto dal segno "+":

```
nome_lista
nome_lista + [19, 20, 21]
```

- L'operatore "*" consentirà invece di definire liste formate da più elementi identici tra loro ripetuti un numero di volte pari al valore passato come moltiplicatore nell'istruzione:

```
[Homer] * 4
```

Le liste

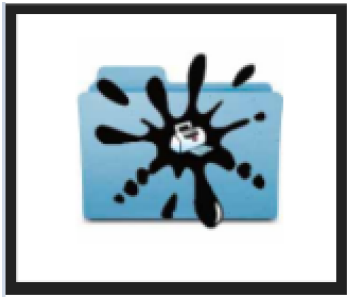


- E' infine da citare il metodo denominato insert() con il quale si potrà aggiungere un elemento ad una lista nella posizione desiderata:

```
nome_lista = [10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21]
nome_lista.insert(0,9)
nome_lista
```

- In sostanza si tratta di un metodo che accetta due argomenti, il primo è il numero indice (nel nostro caso "0") che dovrà essere assegnato al nuovo componente della lista, il secondo ("9" nell'esempio) è l'elemento destinato all'inserimento nella lista.

Le liste

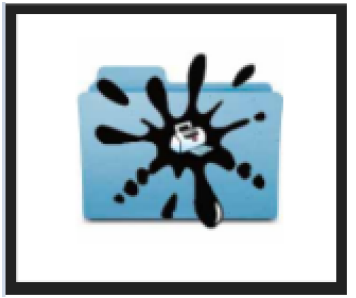


- Uno sviluppatore potrebbe voler verificare che un determinato elemento sia effettivamente presente all'interno di una lista; per questo scopo è disponibile la parola chiave in che in sostanza effettua un un confronto di tipo booleano tra il valore passato come argomento e gli elementi presenti in lista.

- Nel caso in cui il parametro dovesse trovare corrispondenza con un elemento in lista la verifica restituirà TRUE, altrimenti restituirà FALSE:

```
nome_lista = [10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21]
14 in nome_lista
True # viene restituito "True" perché l'elemento è in lista
22 in nome_lista
False # viene restituito "False" perché l'elemento non è in lista
```

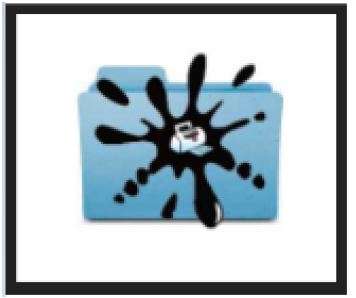
Le liste



- Così come è possibile inserire nuovi elementi in una lista è anche possibile cancellarli utilizzando, in questo caso la parola chiave del; volendo rimuovere un elemento da una lista sarà necessario passare a del l'indice associato ad esso, come nell'esempio seguente dove ad essere eliminato sarà l'elemento con indice "5" (cioè quello con valore "15"):

```
# Cancellazione di un singolo elemento da una lista
nome_lista
del nome_lista[5]
nome_lista
```

Le liste

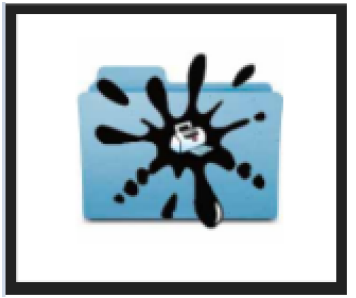


- Si noti come l'interprete di Python interverrà per riordinare gli indici della lista, motivo per il quale l'indice utilizzato come identificatore per la cancellazione non andrà perduto ma verrà riassegnato:

nome_lista

nome_lista[5]

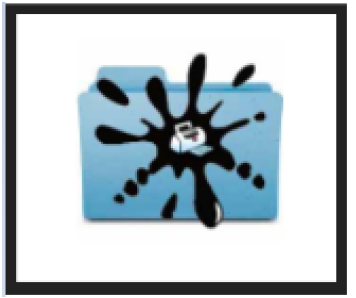
Le liste



- Nel caso in cui si desideri cancellare simultaneamente più elementi sarà necessario specificare l'intervallo all'interno del quale sono compresi, ad esempio:

```
# Cancellazione dei valori compresi in un intervallo
nome_lista
del nome_lista[2:6]
nome_lista
```

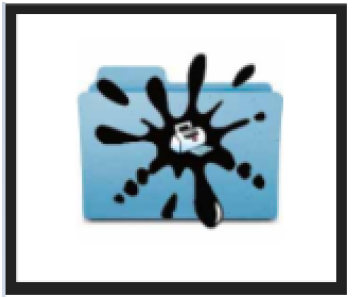
Le liste



- Se si desidera cancellare un elemento specifico senza richiamarlo attraverso il suo indice lo si potrà fare tramite il metodo remove() a cui passare il nome dell'elemento da eliminare:

```
# Rimozione di un elemento specifico da una lista
nome_lista
nome_lista.remove(17)
nome_lista
```

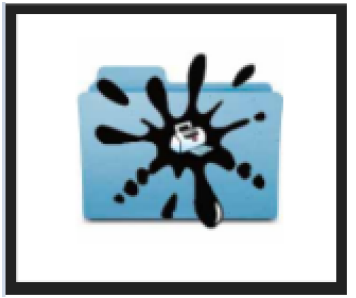
Le liste



- Per ripulire interamente una lista dal suo contenuto è invece disponibile il metodo clear() a cui passare come argomento il nome della lista da "svuotare":

```
# Rimozione simultanea di tutti gli elementi di una lista
nome_lista
nome_lista.clear()
nome_lista
```

Le liste



- Anche dopo aver perso tutti i suoi elementi la lista continuerà ad esistere come lista vuota, per eliminarla del tutto si dovrà fare ricorso nuovamente ad un'istruzione basata su del:

```
# Cancellazione definitiva di una lista
del nome_lista
# la chiamata alla lista produrrà un errore
# in quanto la lista non esiste più
nome_lista

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'nome_lista' is not defined
```