Statements

Descrizione

- Uno *statement* è un blocco di istruzioni
- *statements semplici* consistono di una singola riga. I tipi visti finora sono:

  assegnazione per creare riferimenti ad oggetti

  ```
  >>> a = [3,4,1,2]
  ```

  chiamata per eseguire funzioni, metodi.

  ```
  >>> a.sort()
  >>> print(a)
  [2, 3, 3, 4]
  ```

  import per poter usare i metodi di un modulo

  ```
  >>> import pickle
  ```

Statements semplici: eccezioni

Continuazione su più righe

- Uno statement semplice più spaziare più righe se particolarmente lungo.
- Se lo statement è racchiuso tra parentesi (di qualsiasi tipo), l'interprete includerà tutte le righe fino alla parentesi di chiusura

```
l = [0 , 1 , 2,
     3, 4,
     5 ,6]

"".join([ r for r in "AHHDCCEDGGTA"
          if r in "HC"])
```

- L'indentazione delle righe successive non conta in questo caso, ma conviene farla in modo da rendere chiaro il significato complessivo

Statements semplici: eccezioni

Continuazione su più righe

- Una espressione può sempre essere racchiusa tra parentesi tonde

```
if (x > 0 and
    y > x and
    z):
    print("True")
```

- Una stringa può spaziare più righe se racchiusa tra triplici apici:

```
s = """questa deve essere una
       stringa molto lunga"""
```

Statements composti

Descrizione

- *statements composti* contengono al loro interno degli statements *annidati*, ed occupano più righe
- un programma Python è composto combinando tali statements (programmazione *strutturata*)
- Uno statement composto in Python consta di
  1. una riga di intestazione terminata da ":"
  2. seguita da un blocco di istruzioni (una o più righe) *indentate* rispetto all'intestazione

```
if "U" in s.upper():
    print("RNA")
```

Statements composti
Indentazione

- L'*indentazione* consiste nell'inserire un certo numero di spazi (in genere un `tab` per velocità) prima del codice
- Tutte le istruzioni del blocco relativo ad uno statement composto devono essere indentate dello stesso numero di spazi
- Python infatti individua i blocchi come istruzioni con la stessa indentazione.
- Una istruzione con indentazione uguale a quella dell'intestazione indica la fine del blocco (e dello statement composto) e l'inizio di un'istruzione al di fuori di esso

```
if y < x:
    z = x
    x = y
    y = z
print(x,y)
```

Statements composti
Indentazione

- E' possibile annidare arbitrariamente statements uno dentro l'altro, aumentando l'indentazione ad ogni annidamento:

```
if len(s1) < len(s2):
    print("Searching %s in %s" % (s1,s2))
    if s1 in s2:
       print("Found %s in %s" % (s1,s2))
print("Done")
```

- Il blocco relativo al primo `if` contiene tutte le istruzioni fino a `print("Done")` escluso
- Al suo interno è annidato uno statement composto (secondo `if`)

Lo statement `if`

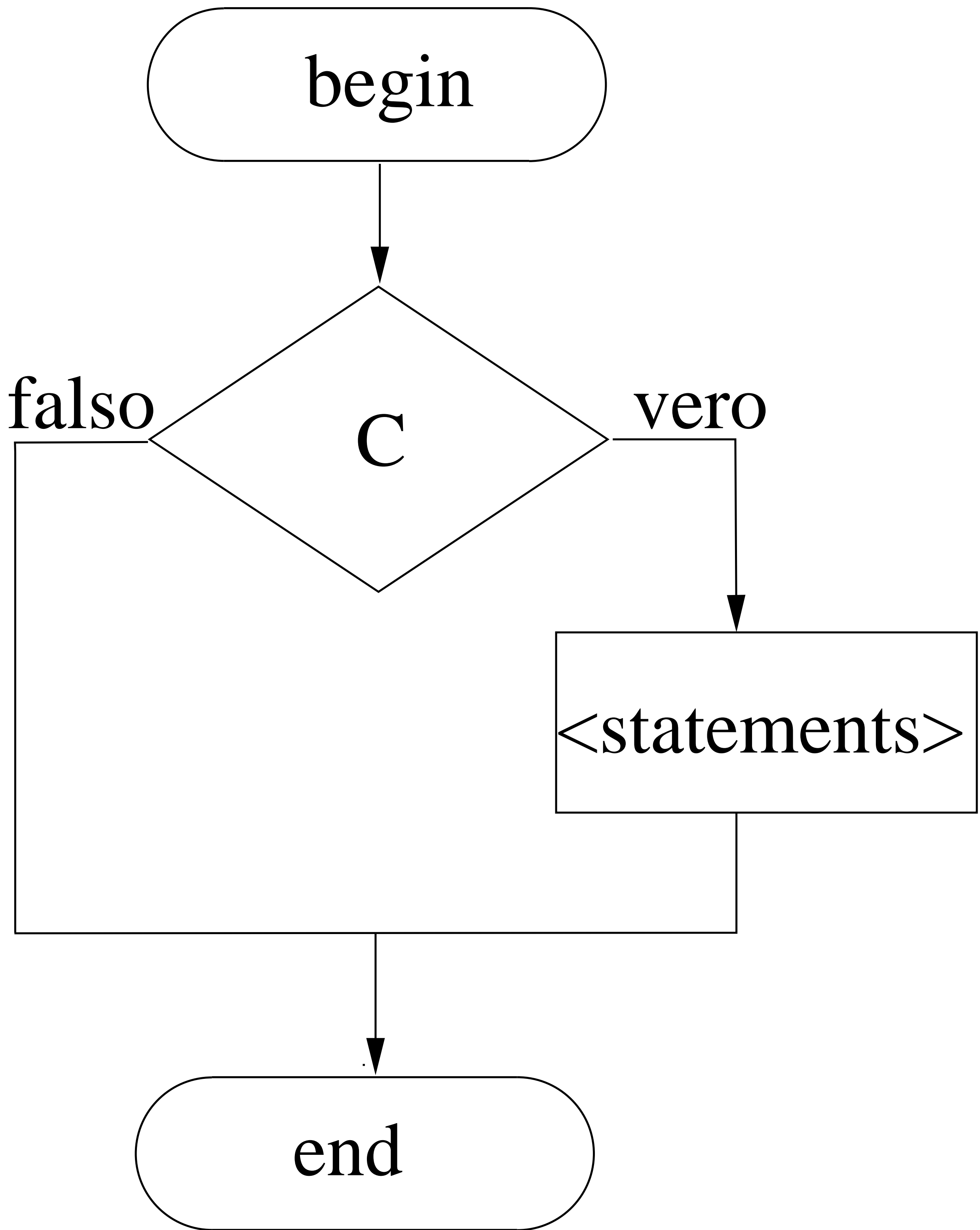
Descrizione

- Il formato più semplice esegue un blocco di istruzioni se una certa condizione è soddisfatta:

```
if <C>:
    <statements>
```

- Ad esempio:

```
if "U" in s.upper():
    print("RNA")
```



Lo statement `if`

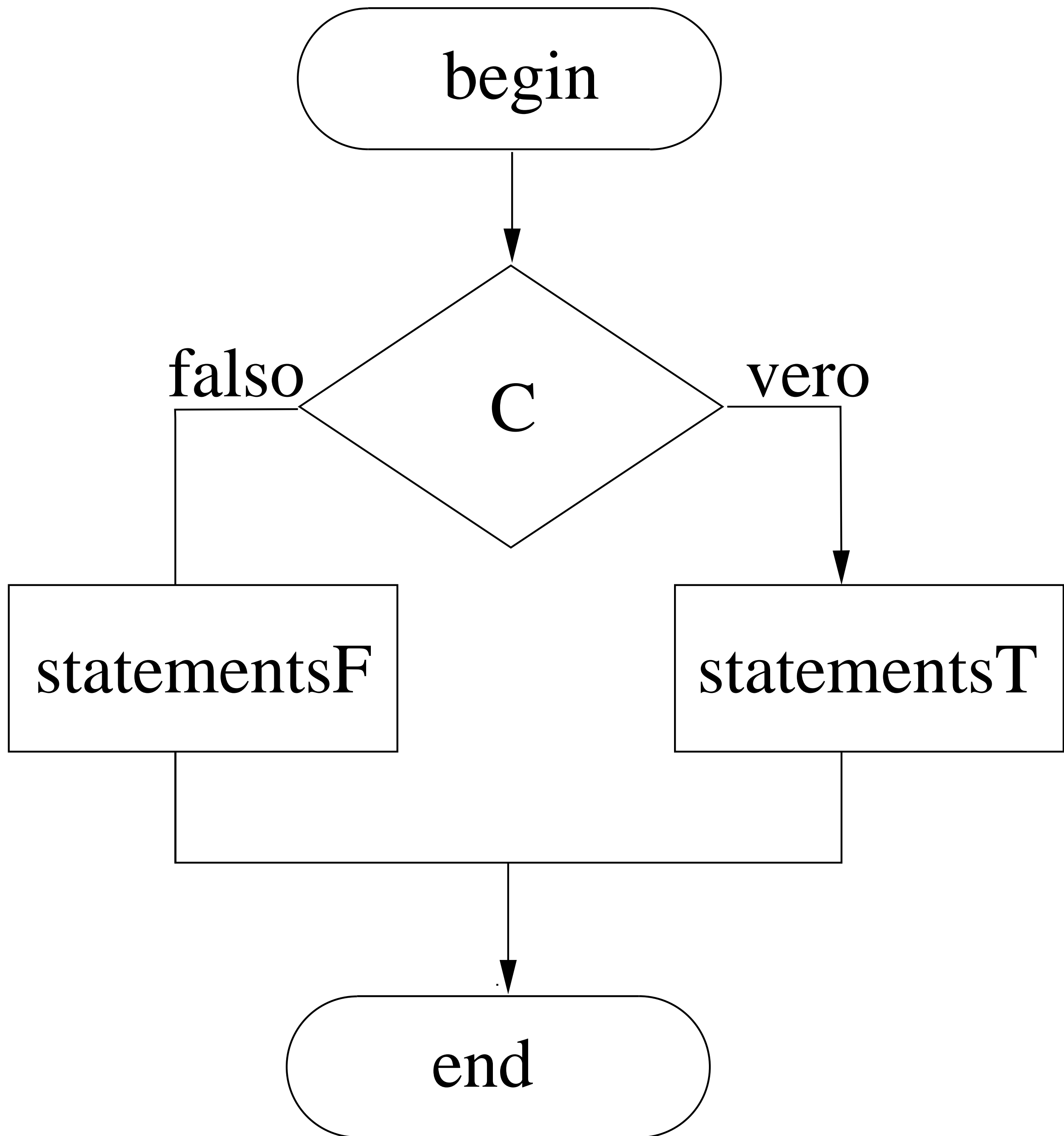
Descrizione

- la coppia `if` `else` permette di specificare istruzioni sia per condizione vera che falsa

```
if <C>:
    <statementsT>
else
    <statementsF>
```

- Ad esempio:

```
if "U" in s.upper():
    print("RNA")
else:
    print("DNA")
```



Lo statement `if`
Descrizione

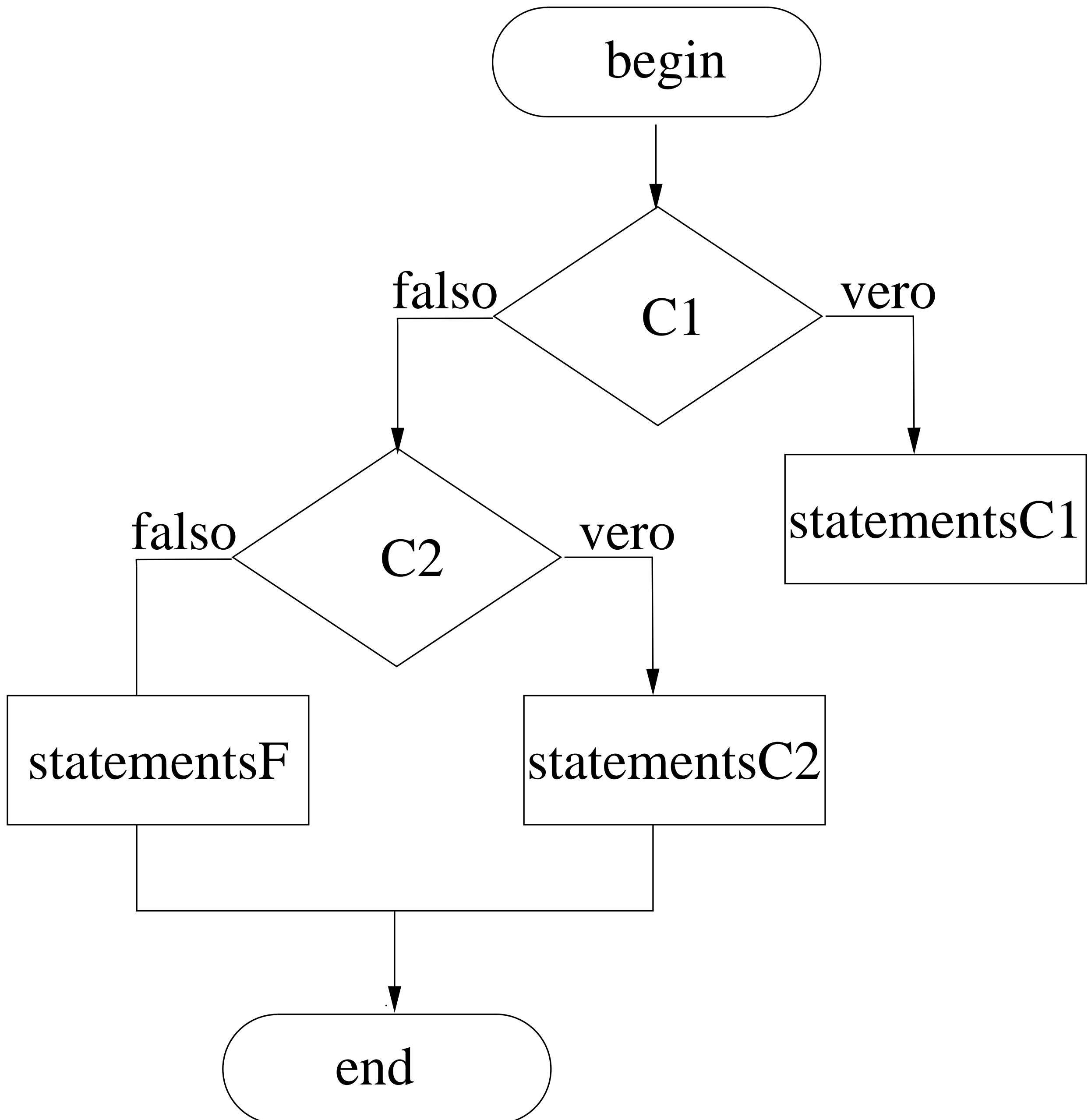
- E’ inoltre possibile specificare condizioni alternative se la prima non è soddisfatta, tramite `elif`

```
if <C1>:
    <statementsC1>
elif <C2>:
    <statementsC2>
else:
```

```
    <statementsF>
```

- Ad esempio:

```
if "U" in s.upper():
    print("RNA")
elif "T" in s.upper():
    print("DNA")
else:
    print("Not a nucleotide sequence")
```

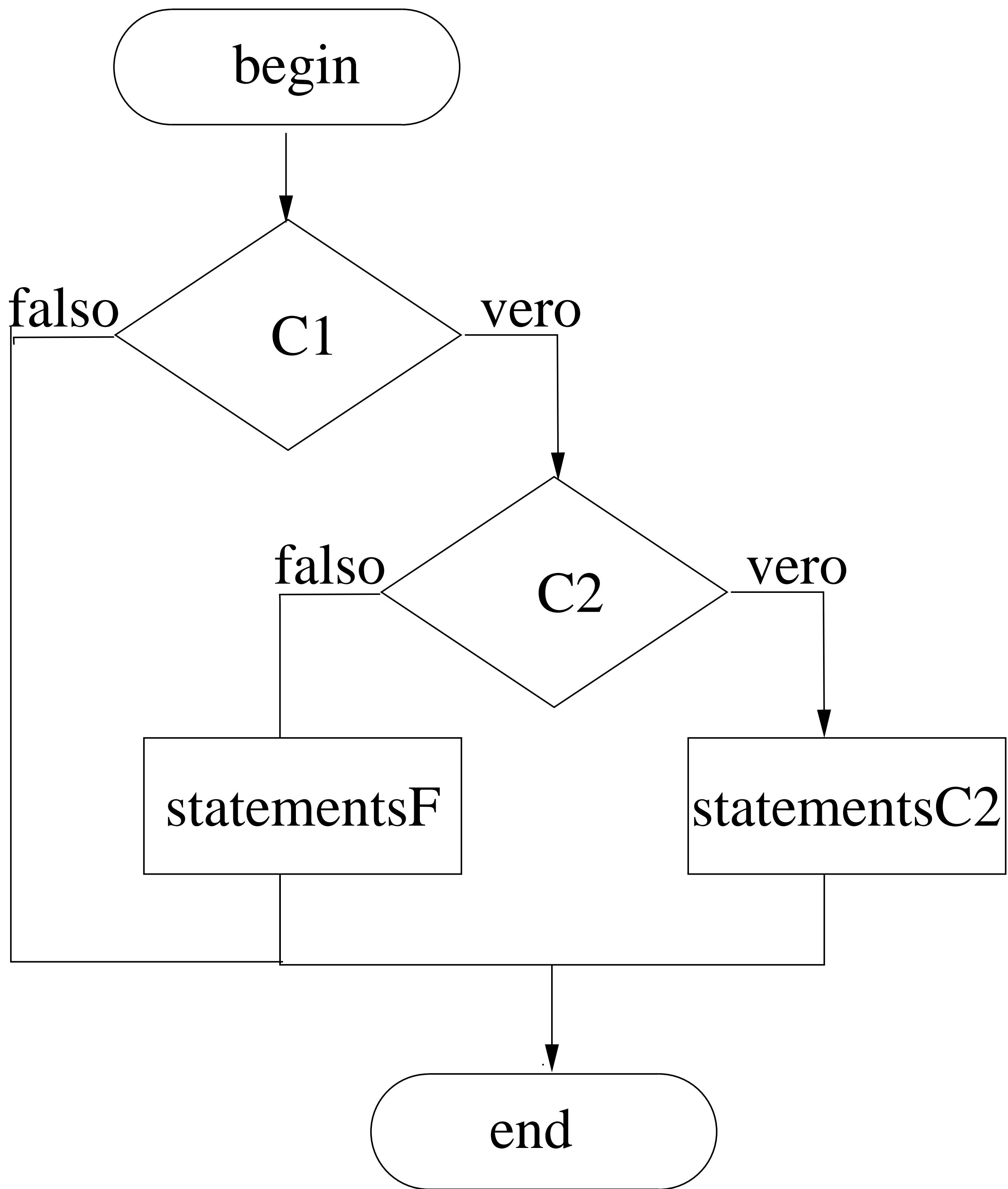


Lo statement `if`

Descrizione

- Il livello di indentazione stabilisce di quale blocco una certa istruzione fa parte
- Nel caso di condizioni annidate, il livello di indentazione stabilisce quindi a quale `if` un certo `elif` o `else` si riferisce

```
if <C1>:
    if <C2>:
        <statementsC2>
    else:
        <statementsF>
```

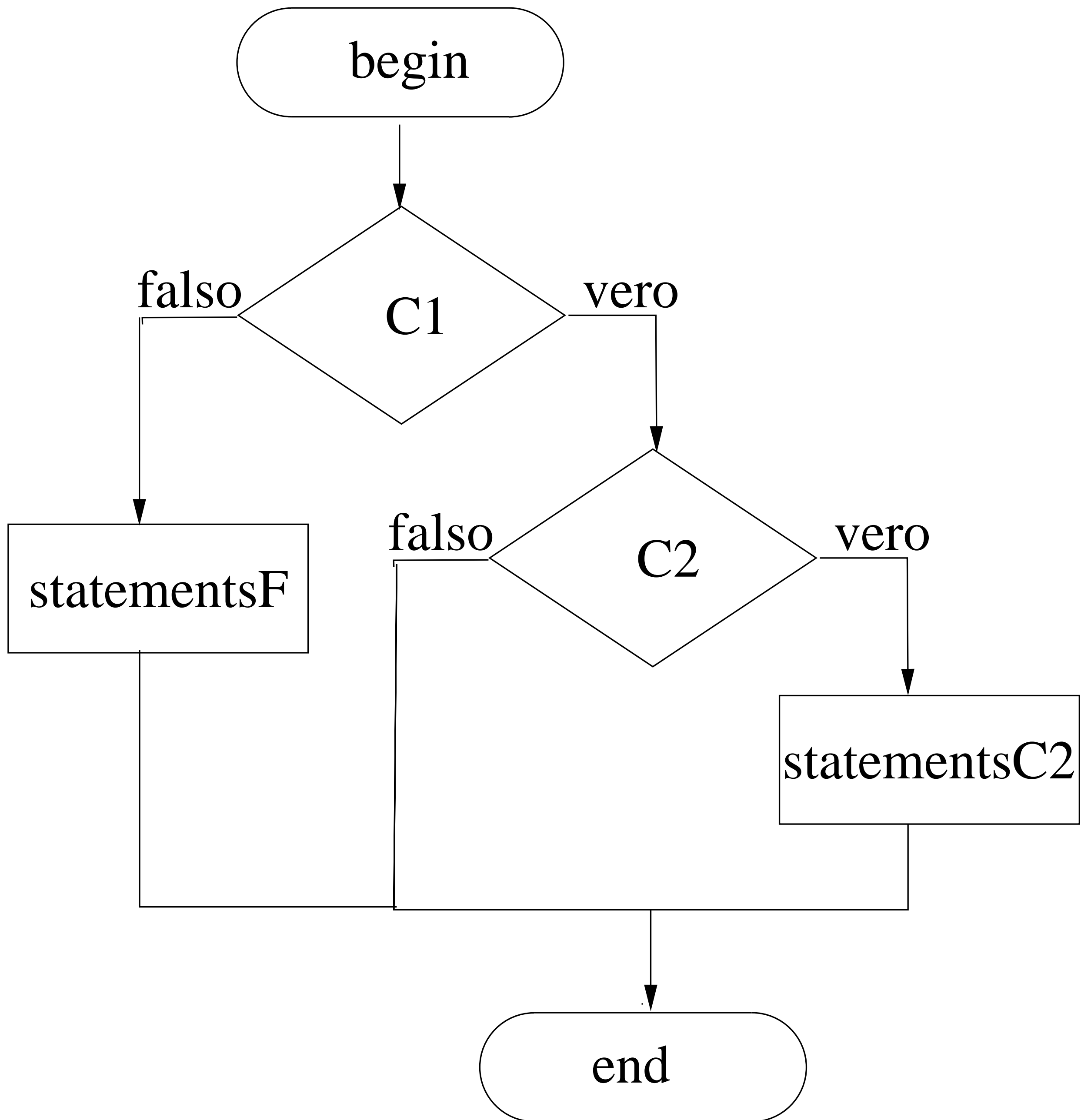


Lo statement `if`

Descrizione

```
if <C1>:
```

```
    if <C2>:
        <statementsC2>
else:
    <statementsF>
```



Test di verità

cosa è vero e cosa è falso

- Un numero diverso da zero è vero, un numero uguale a zero (e.g. 0, 0.0) è falso
- Un oggetto non vuoto è vero, un oggetto vuoto (e.g. `""`, `()`, `[]`, `{}`) è falso

- `None` è falso

Connettivi logici

- I connettivi logici `or` ed `and` possono essere utilizzati per combinare espressioni logiche:

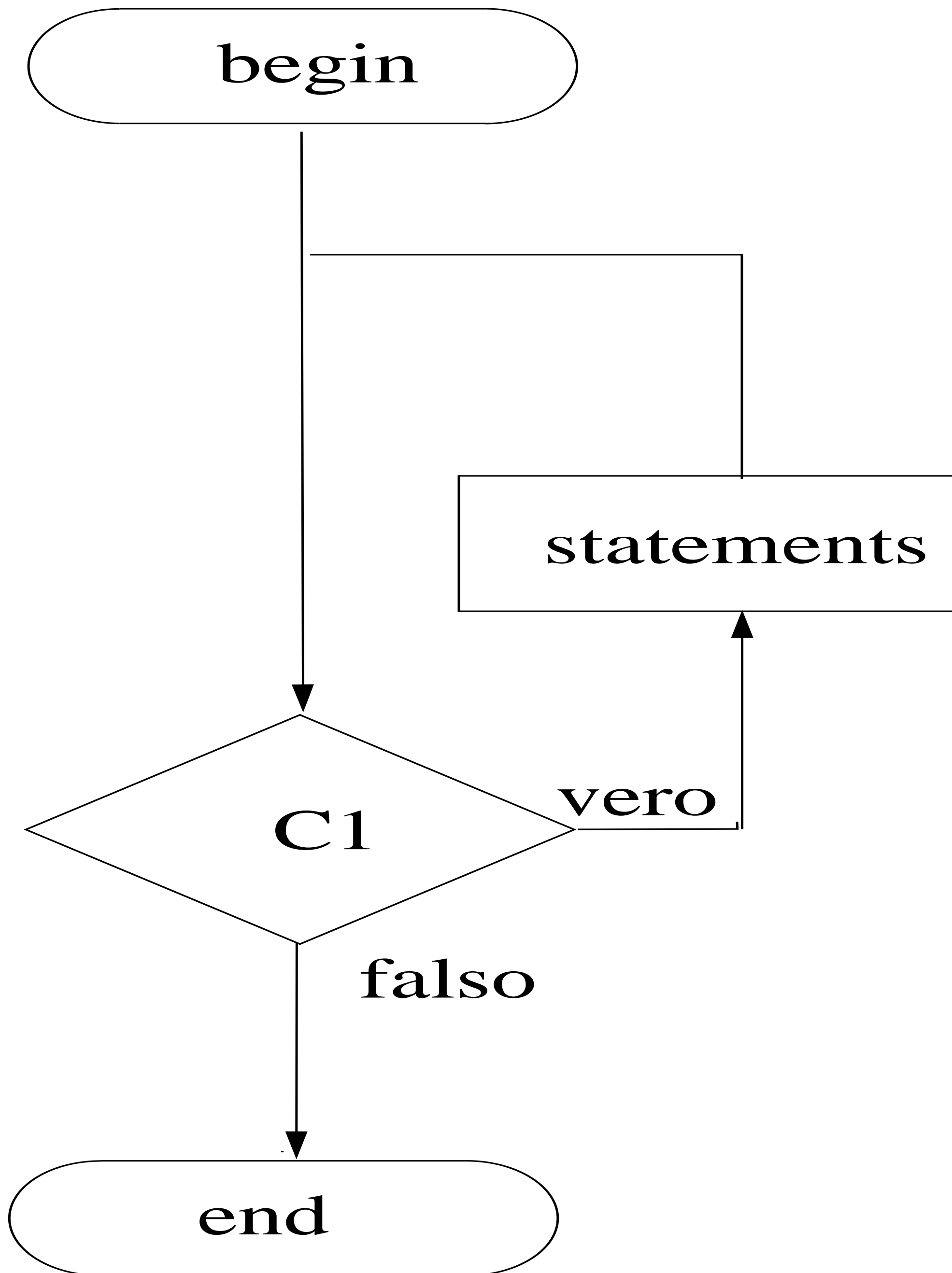
```
if "U" in s1.upper() and "U" in s2.upper():
    print("RNA sequences")
```

Lo statement **`while`**

Descrizione

- Permette di codificare *cicli*
- E' uno schema di iterazione per *vero* (si itera finché la condizione rimane soddisfatta)
- E' uno schema di iterazione con *controllo in testa* (la condizione viene verificata prima delle istruzioni del ciclo)

```
while <C>:
    <statements>
```



Lo statement `while`

Esempi

- Si usa principalmente per cicli *indefiniti*, di cui non si conosce a priori il numero di iterazioni

```
while s != "indovinami":
   s = input("Prossimo tentativo?")
```

Lo statement **`while`**

statement **`break`**

- Lo statement `break` permette di uscire subito dal ciclo corrente senza terminare la sequenza di istruzioni
- Viene in genere usato quando si verifica una certa condizione che richiede l'uscita

```
while s != "indovinami":
   s = input("""Prossimo tentativo?
(stop per terminare)""")
   if s == "stop":
      print("alla prossima")
      break
```

Lo statement **`while`**

statement **`continue`**

- Lo statement `continue` permette di saltare all'intestazione del ciclo corrente senza terminare la sequenza di istruzioni
- Viene in genere usato quando si verifica una certa condizione che fa sì che non si debba (o sia inutile) eseguire le istruzioni successive in quella particolare iterazione

```
l=[]
while len(l) < max_len:
   s = f.readline()
   if s[0] == '>' # riga da ignorare
     continue
   l.append(s.strip())
print("read %d sequences" %len(l))
```

Lo statement **`while`**

final **`else`**

- Spesso al termine di un ciclo è utile sapere se si è usciti "normalmente" per il fallimento della condizione, o a causa di un `break`
- Ad esempio, il ciclo più essere interrotto prematuramente se un dato oggetto è stato trovato, o terminare se non ci sono più oggetti da cercare (o viceversa)
- Tale situazione può essere gestita in Python aggiungendo un `else` finale dopo il blocco annidato, che verrà eseguito solo se si esce per il fallimento della condizione:

```
while s != "keyword":
   s = f.readline()
   if not s:  # fine file
     break
else
  print("keyword found!")
```

lo statement `for`

Descrizione

- Permette di codificare cicli *enumerativi*, in cui si esegue un numero predefinito di iterazioni
- In Python, il ciclo for prende ad ogni iterazione l'elemento successivo di una certo oggetto, ed esegue delle istruzioni (il blocco annidato)

```
for <elemento> in <oggetto>:
    <statements>
```

- L'oggetto deve supportare l'operazione di `next`, ad esempio un oggetto sequenza (stringhe, liste, tuple) o una vista (`d.keys()`)

```
for (k,v) in d.items():
    if v > 1:
       s.add(k)
```

lo statement `for`

Nota

lo statement `for` è in genere molto più rapido di una corrispettiva versione con `while`, per cui va preferito dove possibile

statements opzionali

- Come il ciclo `while`, anche il ciclo `for` può contenere `break`, `continue` ed `else` finale.

```
for s in f.readlines():
    if s[0] == '>':
       continue
    if pattern in s:
       patseq = s
       break
else
   print("Pattern not found")
```

lo statement `for`

Assegnazione di tupla

- Come già visto per le list comprehension, il `for` permette di assegnare l'elemento successivo ad una tupla della dimensione corretta

```
dict = {"1a43" : "aaasdsdafa",
        "1b24" : "gfdgehh"}
for (k,v) in dict.items():
    if v.find("hh") > 0:
        print k
        break
```

lo statement `for`

cicli annidati

- Spesso sono necessari nei programmi più cicli annidati, ad esempio per eseguire un'operazione su tutte le coppie di elementi di due liste

```
for (name,s1) in d.items():
    print("%s\t%s" %(name,s1))
    for s2 in l:
        if s2 in s1:
           print(s2)
```

lo statement `for`

Modifica dell'oggetto sequenza

- Il ciclo `for` prende successivamente elementi dell'oggetto sequenza
- Modifiche a tali elementi sono possibili solo se essi sono mutabili
- Altrimenti, modificare un elemento implica assegnare alla variabile del ciclo l'oggetto modificato, ma la sequenza mantiene l'oggetto originario

```
l = [0, 1, 2, 3]
for x in l:
    x += 1
print l
```

produce

```
[0, 1, 2, 3]
```

lo statement `for`
Modifica dell'oggetto sequenza

- Per poter sostituire un elemento dell'oggetto sequenza con una versione modificata (o qualsiasi altra cosa), è necessario che il ciclo `for` fornisca degli *indici* e non gli elementi stessi
- La funzione `range` permette di generare un *iteratore* su una lista di interi specificando il valore successivo all'ultimo
- Tali interi possono essere usati come indici per modificare il contenuto delle corrispettive posizioni nell'oggetto

```
l = [5, 3, 0, -1]
for i in range(len(l)):
   l[i] += 1
print l
```

produce

```
[6, 4, 1, 0]
```

La funzione `range`

Descrizione

- La funzione `range` può prendere fino a tre argomenti:
  - Con un solo argomento, il primo estremo vale 0, il secondo il valore dell'argomento - 1.

    ```
    >>> list(range(5))
    [0, 1, 2, 3, 4]
    ```

  - Con due argomenti, il primo estremo ha il valore del primo argomento, il secondo il valore del secondo argomento - 1.

    ```
    >>> list(range(3,9))
    [3, 4, 5, 6, 7, 8]
    ```

  - Il terzo argomento specifica di quanto modificare un elemento per calcolare il successivo (può anche essere negativo)

    ```
    >>> list(range(0,9,3))
    [0, 3, 6]
    >>> list(range(3,-3,-1))
    [3, 2, 1, 0, -1, -2]
    ```

lo statement `for`

Usi di `range`

- La funzione `range` può anche essere usata per iterare solo su alcuni elementi dell'oggetto

  ```
  s = "abcdefg"
  for i in range(0,len(s),2):
     print s[i],
  ```

  produce

  ```
  a c e g
  ```

lo statement `for`

Attraversamenti paralleli

- Spesso ci si trova nella necessità di visitare più sequenze in `parallelo`, magari per combinarne gli elementi in posizioni corrispondenti
- La funzione `zip` permette di iterare su un'unica lista a partire da due o più liste
  - ogni elemento della lista finale è una tupla contenente gli elementi corrispondenti delle liste ricevute come argomento
  - se le liste hanno dimensioni diverse, la funzione tronca il risultato alla più corta

  ```
  >>> list(range(4))
  [0, 1, 2, 3]
  >>> list(range(6))
  [0, 1, 2, 3, 4, 5]
  >>> list(zip(range(4),range(6)))
  [(0, 0), (1, 1), (2, 2), (3, 3)]
  ```

Iteratori

Descrizione

- Un *iteratore* è un oggetto che permette di scorrere sugli elementi di un oggetto iterabile (e.g. una lista, una stringa)
- Un *iteratore* si crea tramite la funzione `i = iter(o)` dove `o` è un oggetto iterabile (e.g. una lista)
- Una volta creato l'iteratore, la funzione `next(i)` si ottiene il prossimo elemento dell'oggetto su cui si itera
- `next(i)` genera un'eccezione quando si è arrivati alla fine dell'oggetto

Nota
`for` rende tali operazioni trasparenti al programmatore

Iteratori

Esempio

```
>>> l = [0,1,2,3]
>>> i = iter(l)
>>> next(i)
0
>>> next(i)
1
>>> next(i)
2
>>> next(i)
3
>>> next(i)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

Iteratori

Iteratori con `for`

- Lo statement `for` permette di iterare su oggetti iterabili rendendo trasparente l'uso dell'iteratore
- Ad inizio ciclo crea l'iteratore
- Ad ogni iterazione assegna il prossimo oggetto alla variabile specificata
- Quando next solleva un'eccezione esce dal ciclo

Iteratori
Iteratori e viste

- Una *vista* è un oggetto usato per iterare in maniera efficiente su di un oggetto iterabile
- Le funzioni `d.keys(),d.values(),d.items()` restituiscono tutte viste
- In python ≤ 2.7 queste funzioni restituivano *liste*, copiando contenuti in memoria nel crearle.
- Da python 3, le funzioni restituiscono viste, che non copiano contenuti in fase di creazione, ma generano singoli oggetti quando ci si itera sopra
- Le funzioni `range,zip` funzionano in maniera simile, generando al volo i prossimi elementi

Nota
E' sempre possibile ottenere una lista da una vista con il metodo `list` (come visto per `range`)

Iteratori

Iteratori su file

- Qualsiasi oggetto iterabile può essere attraversato con un `for`
- In particolare, un file è un oggetto iterabile
- L'iteratore del file restituisce una riga per volta, fino alla fine del file
- E' quindi possibile utilizzare un oggetto file dovunque ci sia bisogno di operare su un oggetto tramite un iteratore

```
for line in open("seq.fasta"):
    print(line)
```

List comprehension

Versione compatta di statement `for`

- La list comprehension è un modo compatto (e produce codice più efficiente) di eseguire operazioni che debbano generare una nuova lista modificando con la stessa espressione tutti (o una selezione de) gli elementi di una lista
- Una lista comprehension può sempre essere scritta in modo più prolisso con uno statement `for` (ed inizializzazione)

List comprehension

Descrizione come statement `for`: esempio

- List comprehension

```
l = [ r for r in "AHHDCCEDGGTA" if r in "HC"]
```

- Corrispondente codice tramite `for`

```
l = []
for r in "AHHDCCEDGGTA":
    if r in "HC":
        l.append(r)
```