Esercitazioni di Informatica: Python

Stefano Teso

mailto:stefano.teso@gmail.com

Versione 2015.1

Istruzioni per l'uso

- In questo documento, i riferimenti a *comandi Python* e *percorsi* sono indicati `così`.
- Gli esercizi ed i dati necessari si trovano su `http://disi.unitn.it/~teso` nella sezione "materiale didattico". Qui si trovano anche i dati necessari. I dati sono nella directory `data/`.
- Negli esercizi assumeremo di lavorare *esclusivamente* con file di testo.
- Python non capisce gli accenti nè gli apostrofi (apici, virgolette, etc.). Quando il codice dal PDF in uno script `.py` dovrete sostituire i caratteri "speciali" con caratteri ASCII.
- **TESTATE LE VOSTRE SOLUZIONI CON L'INTERPRETE PYTHON**!!!
- Usate `help` per accedere alla documentazione Python.
- Se volete ripetere un comando eseguito in precedenza, usate il tasto "freccia in alto" ↑.
- Se Python si blocca, potete "ucciderlo" con `Control-c`.

1 Scrivere ed Eseguire Codice

1.1 Interprete

Potete scrivere ed eseguire codice Python interattivamente attraverso l'**interprete**.

Per far partire l'interprete Python, scrivete `python` in un terminale. Vi verrà presentato questo testo:

```
Python 2.7.5 (default, Sep  6 2013, 09:55:21)
[GCC 4.8.1 20130725 (prerelease)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

A questo punto potete scrivere i comandi python nel terminale, e dare invio per eseguirli.

1.2 Moduli

Alternativamente, potete scrivere il codice in un **modulo**: un file di testo[1] con estensione `.py`. L'estensione `.py` è fondamentale!

Per **eseguire** un modulo chiamato `modulo.py` potete scrivere, da linea di comando:

```
python modulo.py
```

Per **importare** un modulo (ed utilizzare le funzioni che contiene) dall'interprete o da un altro modulo, usate la sintassi:

```
import modulo
```

Nota: quando importate un modulo, **omettete** l'estensione `.py`!

[1] Usando un editor di testo qualunque, ad esempio `gedit`.

1.3 Oggetti, Tipi, Valori

Un oggetto è un elemento su cui è possibile operare in qualche modo. Ogni oggetto è definito da:

- Un **tipo**. Specifica che cosa l'oggetto rappresenta.
- Un **valore**.

Il tipo di un oggetto specifica quali **valori** questi può assumere, quali **operazioni** si possono eseguire sull'oggetto, e quali **metodi** l'oggetto mette a disposizione.

I tipi fondamentali sono:

- **Numeri** interi (`int`), interi lunghi (`long`), reali (`float`).
- **Condizioni** (`bool`), che possono essere vere o false. (Tecnicamente sono *numeri*.)
- **Stringhe** (`str`), rappresentano testo.
- **Liste** (`list`) e **tuple** (`tuple`), cioè collezioni di oggetti eterogenei.
- **Dizionari** (`dict`), che sono mappe tra oggetti (es. id-di-proteina $\rightarrow$ sequenza-di-aa.)

Le operazioni fondamentali possono essere combinate attraverso **statement complessi** (`if`, `for`, `while`, *etc.*) e racchiuse in **funzioni**.

1.4 Variabili

Le **variabili** sono "contenitori" di oggetti.

- Gli oggetti possono essere **assegnati** a variabili con `=`.
- Una variabile si **riferisce** all'oggetto a cui è assegnata.
- Il tipo di una variabile è il tipo dell'oggetto a cui si riferisce.
- Per stampare a schermo il **valore** di una variabile `x`:
  - Nell'interprete, basta scrivere `x` e premere invio (**eco** dell'interprete).
  - In un modulo, dovete usare `print x`.
- Per stampare a schermo il **tipo** di una variabile `x`, posso usare la funzione `type`.

Esempio 1. Per assegnare alla variabile `var` un oggetto intero di valore `123`, scrivo:

```
var = 123
```

In nome di `var` è "`var`"; il valore è `123`; il tipo è `int`. Per stampare a schermo valore e tipo di `var`, scrivo:

```
var             # in un modulo invece 'print var'
type(var)       # in un modulo invece 'print type(var)'
```

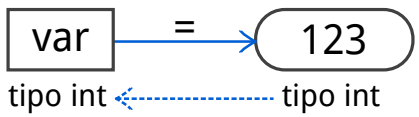


Figure 1: Assegnamento ad una variabile.

Esempio 2. Una variabile può essere assegnata più volte.

```
var = 1
var = "MANLFKLGAENIFLGRKAATKEEAIRF"
var = 3.1415926536
```

Dopo ogni assegnamento, la variabile `var` si riferisce ad un oggetto diverso, ed il suo tipo cambia: prima `int`, poi `str`, poi `float`.

Esempio 3. L'assegnamento funziona anche tra variabili, ad esempio il frammento:

```
a = "testo"
b = a
```

dice a Python di assegnare l'oggetto stringa `"testo"` alla variabile `a`, poi di assegnare alla variabile `b` l'oggetto assegnato ad `a`, cioè sempre la stringa `"testo"`: di conseguenza, `a` e `b` si riferiscono allo **stesso** oggetto! (Caso 1 della Figura 2.)

Consideriamo il seguente codice:

```
a = "testo"
b = "testo"
```

In questo caso `a` e `b` puntano a due **diverse** stringhe (oggetti) che contengono lo stesso testo, `"testo"` (Caso 2 della Figura 2.)

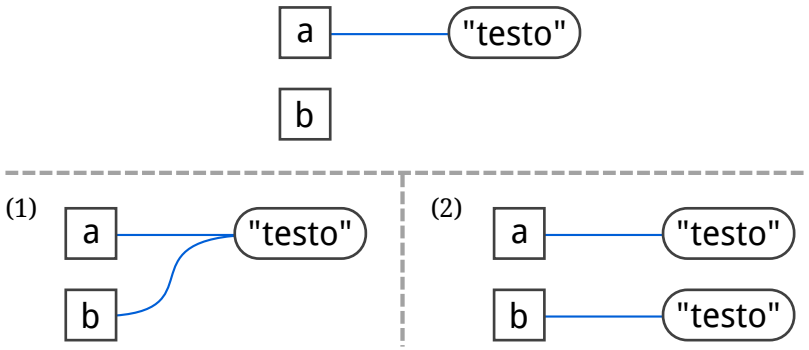


Figure 2: Assegnamento multiplo: due varianti.

1.5 Funzioni e Metodi

Python mette a disposizione un'ampia gamma di **funzioni** e **metodi** predefiniti, come la funzione `print` ed il metodo delle stringhe `upper`.

Funzioni e metodi permettono di eseguire operazioni comuni senza doverle implementare a mano. Ad esempio `print` stampa il valore di una variabile a schermo, mentre `upper` restituisce una stringa in maiuscolo.

- Una **funzione** esegue un'operazione sui suoi **argomenti**, e può ritornare un **risultato**.

  Un **metodo** è una funzione messa a disposizione da un tipo: ci sono i metodi delle stringhe, delle liste, dei dizionari, *etc*.

- La differenza è che le funzioni sono **invocate** sono così:

  $$\texttt{risultato = funzione(argomento}_1\texttt{,...,argomento}_n\texttt{)}$$

  mentre i metodi sono invocati così:

  $$\texttt{risultato = variabile.metodo(argomento}_1\texttt{,...,argomento}_n\texttt{)}$$

  (`variabile.` sotto la dicitura !!!)

  Qui `metodo` è messo a disposizione dal tipo di `variabile`.

- Gli **argomenti** della funzione/metodo (i suoi **input**) qui sono le variabili $\texttt{argomento}_i$. Gli argomenti stanno sempre tra **parentesi**[2].

  Non tutte le funzioni/metodi richiedono un input, nel quale caso possiamo scrivere

  ```
  risultato = funzione()
  ```

- Il **risultato** della funzione (il suo **output**) qui lo assegnamo alla variabile `risultato`. Il tipo ed il valore di `risultato` dipendono da cosa ha ritornato la funzione `funzione`.

  Non tutte le funzioni/metodi restituiscono un risultato, nel quale caso `risultato` sarà assegnato a `None`.

[2] La funzione `print` è l'**unica** che non richiede parentesi attorno agli argomenti!

Esempio 4. Per stampare a schermo un oggetto o una variabile si può usare la funzione `print`:

```
print "Sono una stringa di esempio!"

variabile = "Sono una stringa in una variabile"
print variabile
```

Esempio 5. Se voglio invocare la funzione `somma_gli_argomenti` (me la sono appena inventata) su tre argomenti interi `1`, `2` e `3`, scrivo:

```
risultato = somma_gli_argomenti(1, 2, 3)
```

Se voglio invocare invece un metodo `sono_un_metodo` (ancora, inventato) senza argomenti, scrivo:

```
risultato = variabile_o_oggetto.sono_un_metodo()
```

1.6 Documentazione Interattiva

- Per accedere al **manuale** di un tipo/oggetto/variabile, uso la funzione `help`. Il manuale descrive i `metodi` supportati dal tipo dell'oggetto/variabile. Per uscire dal manuale, premete `q`.
- Per ottenere una lista dei metodi, senza descrizione, potete anche usare la funzione `dir`.

Esempio 6. Per ottenere la lista dei metodi delle stringhe, invochiamo `dir` su una stringa:

```
dir("biotechnology rules!")
["capitalize", "center", "count", "decode", "encode",
"endswith", "expandtabs", "find",
...
"rstrip", "split", "splitlines", "startswith", "strip",
"swapcase", "title", "translate", "upper", "zfill"]
```

(I metodi che cominciano con `"__"` non ci interessano.) Otteniamo lo stesso risultato con **qualunque** stringa:

```
dir("bioinformatics is a-okay!")
dir("python sucks!")
```

Esempio 7. Per leggere il manuale del metodo `split` delle stringhe, scriviamo:

```
help("".split)
Help on built-in function split:

split(...)
    S.split([sep [,maxsplit]]) -> list of strings

    Return a list of the words in the string S, using sep as the
    delimiter string.  If maxsplit is given, at most maxsplit
    splits are done. If sep is not specified or is None, any
    whitespace string is a separator and empty strings are removed
    from the result.
```

Stessa cosa con:

```
help("123".split)

s = "sono una stringa"
help(s.split)
```

1.7 Esercizi

Esercizio 1. Dati il numero intero `1` ed il reale `1.0`, quali sono i metodi messi a disposizione dai numeri interi? Dai numeri reali?

Esercizio 2. Data una stringa `x = "testo"`, aprire il manuale del metodo `replace` delle stringhe.

Esercizio 3. Data una lista `x = []`, aprire il manuale del metodo `remove` delle liste.

Esercizio 4. Dato un dizionario `x = {}`, aprire il manuale del metodo `items` dei dizionari.

Esercizio 5. Aprire il manuale di `dir` ed il manuale di `help`.

Esercizio 6. Scrivete un modulo (usando `gedit` o un altro editor di testo) che stampi a schermo, con la funzione `print`, la stringa `"Hello, world!"`. Il modulo deve chiamarsi `hello.py`. Poi eseguite il modulo da terminale.

Occhio che la funzione `print` non vuole parentesi attorno al suo argomento.

2 Numeri

Si dividono in quattro tipi fondamentali:

- **interi**, di tipo `int`. Possono assumere tutti i valori interi tra -2^{31} e $2^{31}-1$. [3] Esempi: `1`, `2`, `1000`.
- **interi lunghi**, di tipo `long`. Si comportano esattamente come gli interi normali, ma possono assumere valori più piccoli di -2^{31} e più grandi di $2^{31}-1$. Esempi: `1L`, `6136L`. Si noti la `L`.
- **razionali**, di tipo `float` (da *float*ing point). Rappresentano le frazioni $\frac{p}{q}$, dove p e q sono interi o interi lunghi. Esempi: `0.5`, `3.141593`.
- **Booleani**, di tipo `bool`. Assumere solo due valori: `True` (vero) e `False` (falso). Sono usati principalmente per esprimere condizioni (soddisfatte o meno). Occhio alle maiuscole!

I numeri sono oggetti **immutabili**.

2.1 Operatori Aritmetici

Tutti i tipi numerici mettono a disposizione le stesse **operazioni aritmetiche** elementari: somma `+`, differenza `-`, prodotto `*`, quoziente `/`, quoziente intero `//`, modulo (resto della divisione intera) `%`, elevamento a potenza `**`. Le precedenze tra queste operazioni sono le stesse che valgono in aritmetica.

Se `n` e `m` sono numeri e `op` è una delle precedenti operazioni, il tipo di:

```
n op m
```

è il tipo più "complesso" tra quello di `n` e quello di `m`. La scala della complessità è:

$$\texttt{bool} < \texttt{int} < \texttt{long} < \texttt{float}$$

In questo modo, il tipo del risultato dell'operazione riuscirà sempre a "contenere" il valore del risultato.

Esempio 8. Il tipo di `1 * 1L` è intero lungo, mentre il tipo di `1.2345 + 0` è razionale.

[3] I valori precisi non sono importanti.

2.2 Operatori di Comparazione

Gli oggetti di tipo numerico possono essere **comparati** tra loro con le seguenti operazioni: minore di `<`, minore od uguale a `<=`, uguale a `==`, maggiore od uguale a `>=`, maggiore di `>`, diverso `<>` (oppure `!=`).

Se `n` e `m` sono numeri e `comp` è uno dei precedenti operatori di comparazione, il tipo di `n comp m` è sempre `bool`.

Esempio 9. Consideriamo:

```
(percentDiA + percentDiT) > (percentDiC + percentDiG)
```

L'espressione varrà `True` se la condizione è soddisfatta, `False` altrimenti.

2.3 Operatori Booleani

Gli oggetti booleani possono essere combinati usando le operazioni della **logica booleana**: `and`, `or`, `not`.

- `a and b` vale `True` se e solo se sia `a` che `b` sono `True`.
- `a or b` vale `True` se e solo se almeno uno tra `a` e `b` è `True`.
- `not a` vale `True` se e solo se `a` è `False`, altrimenti vale `True`.

Qui `a` e `b` devono essere di tipo `bool`.

Esempio 10. Espressioni come `x > 12` e `x < 34`, dato che hanno tipo booleano, possono essere combinate con `and`, `or` e `not` per ottenere condizioni più complesse, ad esempio:

```
(x > 12) and (x < 34)
```
oppure
```
not (x > 12) or (x < 34)
```

2.4 Esempi

Esempio 11. Per assegnare ad una variabile un numero intero/lungo/razionale/booleano, scrivo:

```
n = 10;  N = 100000000L;  x = 11.2;  cond = True
```

Esempio 12. Per stampare a schermo con `print` più di una variabile alla volta, ad esempio:

```
var1 = 1; var2 = 5; var3 = 6.2
```

posso usare il seguente codice:

```
print var1, var2, var3
```

Se voglio inframezzare al valore delle variabili anche del testo, posso scrivere:

```
print "var1 vale" + str(var1) + "mentre var 2 vale" + str(var2)
```

Per capire perchè questo funziona, date un'occhiata al prossimo capitolo.

Esempio 13. Per eseguire della semplice aritmetica, scrivo:

```
10 // 2
```

Per assegnare il risultato ad una variabile, scrivo:

```
a = 10; b = 2; result = a // b
```

Qualcosa di più interessante, gli zeri di un'equazione quadratica:

```
a = 4.0; b = 4.0; c = 1.0

delta = b*b - 4*a*c
zero1 = (-b + delta**0.5) / (2*a)
zero2 = (-b - delta**0.5) / (2*a)
```

(Per eseguire la radice quadrata di un numero `x`, è sufficiente elevarlo ad "un mezzo", $\frac{1}{2} = 0.5$). Per stampare a schermo il risultato, scrivo:

```
print zero1, zero2
```

Esempio 14. Sono interessato a scoprire qual'è la proporzione di nucleotidi `T` in una sequenza genetica. So che la sequenza è lunga `n = 1521` basi, e che contiene `m = 551` timine. La proporzione `p`, simbolicamente, è

$$\mathtt{p} = \frac{\mathtt{m}}{\mathtt{n}}$$

È forte la tentazione di scrivere:

```
n = 1521; m = 551; p = m / n
```

Purtroppo il risultato, `p = 0`, non torna.

Il problema è che la proporzione, che è circa $p = 0.362$, non può essere espressa con un numero intero: è necessario un numero *razionale*. Però il tipo dell'espressione `m / n` è determinato dai tipi di `n` ed `m`, che sono entrambi interi.

Per ovviare a questo problema, è necessario istruire Python sul fatto che vogliamo un risultato razionale, trasformando almeno uno dei due operandi in un numero razionale[4]:

```
n = 1521.0; m = 551.0
```

Ora il risultato sarà di tipo reale, che può rappresentare il valore 0.362:

```
p = m / n
```

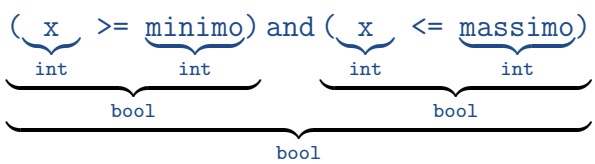
infatti restituisce `p = 0.3622`. Controllate i due casi usando `type(p)`!

Esempio 15. Per comparare due valori ed assegnare il risultato ad una variabile, scrivo:

```
a = 10; b = 15; t = (a > b)
```

Il tipo di `t` è booleano (controllate!). Per combinare più condizioni, ad esempio per controllare che un numero sia incluso tra `10` e `50`, possiamo fare così:

```
minimo = 10; massimo = 50; x = 17
dentro = (x >= minimo) and (x <= massimo)
```



perciò il tipo di `dentro` sarà `bool`.

[4] Alternativamente, posso convertire gli operandi in `float` usando la conversione `p = float(m) / float(n)`.

Esempio 16. Assumendo che `dentroA`, `dentroB`, `dentroC` indichino che un intero `x` è in un intervallo `A`, `B` o `C`, rispettivamente, allora possiamo scrivere le condizioni:

- `x` è in almeno uno dei tre intervalli:

```
dentroAlmenoUno = dentroA or dentroB or dentroC
```

- `x` è sia in `A` che in `B` (possibile se i due intervalli si sovrappongono), ma non in `C`:

```
dentroTuttiTranneC = dentroA and dentroB and not dentroC
```

2.5 Esercizi

Esercizio 7. Creare alcune variabili: `a` e `b` di valore intero `12` e `23`, `c` e `d` di valore intero lungo `34` e `45`, `x` e `y` di valore `21.0` e `14.0`. Creare un'altra variabile `pi` di valore `3.141593`. Di che tipo è?

Esercizio 8. Usando `print`, stampare a schermo la variabile `a`. Stampare `a` e `b`, sulla stessa riga. Stampare `a` e `b` sulla stessa riga, separate da un punto e virgola.

Esercizio 9. Usando `print`, stampare a schermo il prodotto di `a` e `b`. Assegnare ad `r` il risultato del prodotto di `a` e `b`. Usando `print`, stampare a schermo la stringa `Il prodotto di` a `e` b `e'` r, dove a, b, ed r sono opportunamente sostituite dai veri valori.

Esercizio 10. Creare due variabili: `a` che vale `100` e `b` che vale `True`. Usando un numero opportuno di variabili temporanee, scambiare i riferimenti di `a` e `b` (in altre parole, fare in modo che, al termine del vostro programma: (1) `a` si riferisca all'oggetto `True` originariamente riferito da `b`, e che (2) `b` si riferisca all'oggetto `100` originariamente riferito da `a`.) Si può fare con *una sola* variabile ausiliaria?

Esercizio 11. Determinare il valore ed il tipo di:

1. Il prodotto di `a` e `b`
2. La differenza di `c` e `d`
3. Il quoziente di `x` e `y`
4. Il quoziente intero di `a` e `b`
5. Il quoziente intero di `c` e `d`
6. Il quoziente intero di `x` e `y`
7. Il prodotto di `a` e `c`
8. Il prodotto di `b` e `y`
9. Il prodotto di `x` e `d`
10. `2` elevato alla `10`?
11. `2` elevato alla `100`?
12. `2` elevato alla `1.2`?
13. `2` elevato alla `-2`?
14. La radice quadrata di `4`? (Si usi `**`.)

15. La radice quadrata di `2`?

Esercizio 12. Cosa succede (in termini di valore e tipo del risultato) se eseguo:

1. `10 / 12` ; `10 / 12.0` ; `10 // 12` ; `10 // 12.0`.
2. `10 % 3` ; `10 % 3.0`.

Esercizio 13. Due geni si trovano su uno strand di DNA: il primo include i nucleotidi dalla posizione 10 alla posizione 20, il secondo quelli dalla posizione 30 alla 40. Data una posizione arbitraria `pos` (un intero `int`) che può assumere posizioni arbitrarie all'interno dello strand di DNA, scrivere delle condizioni per verificare se:

1. `pos` si trova nel primo gene.
2. `pos` si trova nel secondo gene.
3. `pos` si trova tra l'inizio del primo gene e la fine del secondo.
4. `pos` si trova prima del primo gene o dopo il secondo.
5. `pos` si trova tra l'inizio del primo gene e la fine del secondo, ma in nessuno dei due.
6. `pos` cade in uno dei due geni.
7. `pos` non dista più di 10 dall'inizio del primo gene. (Per capirci, con "distanza" intendo quella "lineare": `1` e `2` distano 1, `10` e `3` distano 7; l'ordine non conta).
8. La funzione `min` ritorna il minore tra due valori, e si invoca così:

   ```
   minore_tra_a_e_b = min(a,b)
   ```

   Usando `min`, calcolare la distanza tra `pos` e l'inzio del primo gene.
9. Controllare la seguente condizione che la distanza tra `pos1` e `pos2` (due posizioni arbitrarie) è minore della somma delle distanze di `pos1` dall'inizio del primo gene e di `pos2` dalla fine del secondo gene.

Esercizio 14. Usando la costate π approssimata `pi = 3.1415926536` e dato `r=2.5`, calcolare:

1. La circonferenza di raggio `r`: $circ = 2\pi r$.
2. L'area di un cerchio di raggio `r`: $area = \pi r^2$.
3. Il volume di una sfera di raggio `r`: $vol = \frac{4}{3}\pi r^3$.

Esercizio 15. Date tre variabili booleane `t`, `u`, `v`, si scrivano delle combinazioni di operazioni booleane che controllino se:

1. `t`, `u` e `v` sono tutte e tre vere.
2. `t` è vera oppure `u` è vera, ma non entrambe.
3. Al più due delle variabili sono vere.
4. Esattamente una delle tre variabili è falsa.
5. Esattamente una delle tre variabili è vera.

3 Stringhe

Le stringhe sono oggetti **immutabili** che rappresentano testo. Sono implementate come **sequenze** di **caratteri**.

È possibile definire una stringa racchiudendono del testo tra apici singoli (`'`) oppure doppi (`"`).

Le sequenti stringhe sono equivalenti:

```
"My name is Bond, James Bond"    'My name is Bond, James Bond'
```

È possibile assegnare una stringa ad una variabile:

```
myString = 'James Bond'
```

Si possono inserire **caratteri speciali** prefissandoli con un backslash (`\`) (questa tecnica è chiamata **escaping**); oppure si può prefissare la stringa con `r` (che sta per **raw**, cioè crudo, non cucinato). Ad esempio:

```
percorso = r"data\fasta"    oppure    percorso = "data\\fasta"
```

Provate a stamparle con `print`!

Nota. La funzione `print` interpreta correttamente i caratteri speciali (es. gli a capo), mentre l'eco dell'interprete no: stampa a schermo *esattamente* quello che c'è nella stringa, inclusi i caratteri speciali, senza interpretarli.

3.1 Stringhe Multi-linea

Per creare una stringa multilinea, si possono utilizzare due metodi.

Il primo prevede di inserire, alla fine di ogni riga un carattere di **a capo** (o **newline**) `\n`. Ad esempio:

```
sadJoke = "Time flies like an arrow.\nFruit flies like a banana."
```

Il secondo metodo permette di scrivere la stringa così come sarà stampata a schermo, senza caratteri di newline, racchiudendola tra **triple** virgolette (singole o doppie poco importa). Ad esempio [5]:

```
lyricsFragment = """
See tomorrow dreamin'
You don't need your freedom

Star A.D.

A little joke that's understood
All over the world
A little joke that's understood
It's all over and over and over and over
"""
```

Si noti che il semplice eco dell'interprete non interpreta [*sic.*] i caratteri `\n`, mentre il comando `print` sì:

```
lyricsFragment; print lyricsFragment
```

[5] Copyright © Faith no More

3.2 Operatori

Le stringhe supportano i seguenti operatori:

- **lunghezza** `len()`. Restituisce la lunghezza della stringa (un intero):

  ```
  len("abc")
  ```

- **concatenazione** `+`. Restituisce una *nuova* stringa che rappresenta la concatenazione degli operandi:

  ```
  "una " + " stringa"
  ```

  Occhio che cose del tipo:

  ```
  "il risultato e' " + 12
  ```

  non funzionano, perchè gli operandi di `+` devono entrambi essere stringhe. Vedremo tra poco come risolvere il problema.

- **ripetizione** `*`. Restituisce una *nuova* stringa che corrisponde ad `n` ripetizioni della stringa originale.

  ```
  "basta python!" * 10
  ```

- `in`. Restituisce `True` se un carattere o una stringa appaiono nella stringa data:

  ```
  s = "abc"; "a" in s; "abc" in s; "x" in s;
  ```

  Restituisce `False` se il controllo fallisce.

- **indicizzazione** o estrazione `[`i`:`j`]`. Restituisce una *nuova* stringa che contiene gli elementi specificati:

  ```
  s = "abcdef"; s[0]; s[2]; s[-1]; s[0:1]; s[1:-1]; s[:6]
  ```

  La stringa originale *non* viene modificata.

Esempio 17. In python si conta a partire da **zero**! Il primo elemento di una stringa `s` sarà dunque `s[0]`, non `s[1]`. Ad esempio:

```
"12345"[0]    "12345"[1]
```

L'ultimo elemento è sempre `s[-1]`, il penultimo `s[-2]`, *etc.* In pratica:

```
"12345"[-1]    "12345"[-2]    etc.
```

Gli elementi dall'`n`esimo all'`m`esimo sono `s[n:m+1]`. Il `+1` qui è necessario perchè in Python:

gli intervalli sono inclusivi rispetto al primo indice

ma esclusivi rispetto al secondo

Ad esempio:

```
"12345"[0:1]    "12345"[0:2]    etc.
```

Per estrarre l'ultimo carattere posso anche usare `len()`:

```
s = "12345"; s[len(s)-1]
```

Il `-1` qui è necessario perchè partiamo a contare da `0`.

Se il primo indice non viene specificato, l'estrazione parte sempre dal primo carattere:

```
"12345"[:3]
```

Se il secondo indice non viene specificato, l'estrazione finisce sempre all'ultimo carattere (incluso):

```
"12345"[3:]
```

Se non specifico alcun indice, estraggo automaticamente *tutti* i caratteri:

```
"12345"[:]
```

3.3 Metodi

- `s.upper()` (`s.lower()`) restituisce una copia della stringa `s` dove i caratteri minuscoli sono sostituiti con caratteri **maiuscoli** (**minuscoli**).

```
"cytoskeleton".upper()    "TRYPTOPHANE".lower()
```

- `s.strip()`/`s.rstrip()`/`s.lstrip()` restituiscono una copia della stringa `s` da cui sono stati rimossi gli spazi a destra ed a sinistra/solo a destra/solo a sinistra:

```
" parola ".strip()
```

  `s.strip(t)` restituisce una copia della stringa `s` da cui sono stati rimossi, a destra ed a sinistra, i caratteri specificati nella stringa `c`.

```
"AAAparolaBBB".strip("AB")
```

  In entrambi i casi il tipo del risultato è `str`.

- `s.startswith(t)`/`s.endswith(t)` restituiscono `True` se la stringa `s` **inizia con**/**finisce con** la stringa `t`, altrimenti `False`.

```
"123456".endswith("456")    "123456".endswith("xyz")
```

  Il tipo del risultato è `bool`.

  L'argomento **deve** essere una stringa! Il seguente esempio dà errore:

```
"123456".endswith(456)
```

- `s.find(t)` **trova** la (prima occorrenza della) stringa `t` all'interno della stringa `s`, e ne restituisce la posizione. Se `s` non contiene `t` come sotto-stringa, allora `find` restituisce `-1`.

```
"123456".find("2")    "123456".find("x")
```

  L'argomento **deve** essere una stringa! Il seguente esempio dà errore:

```
"123456".find(2)
```

- `s.replace(t,u)` restituisce una copia della stringa `s` dove ogni ripetizione della stringa `t` è **rimpiazzata** con la stringa `u`.

```
"se le rose sono rosse allora".replace ("ro", "gro")
```

  Entrambi gli argomenti **devono** essere stringhe.

Esempio 18. Come già spiegato in precedenza, **se** un metodo ritorna un risultato, allora lo possiamo assegnare ad una variabile. Alcuni esempi:

```
risultato = "ascoltami quando parlo!".upper ()
altroRisultato = "  sono circondata da spazi   ".strip()
risposta = "ti piace questo esempio?".endswith("?")
```

Il tipo del risultato è determinato dal metodo chiamato: nei primi due casi sarà `str`, nell'ultimo `bool`.

Esempio 19. Data la seguente stringa di aminoacidi:

```
s = ">MAnlFKLgaENIFLGrKW   "
```

vogliamo prima rimuovere il carattere `>` che sta all'inizio, rimuovere gli spazi che stanno alla fine, poi convertire gli aminoacidi in maiuscolo per uniformità.

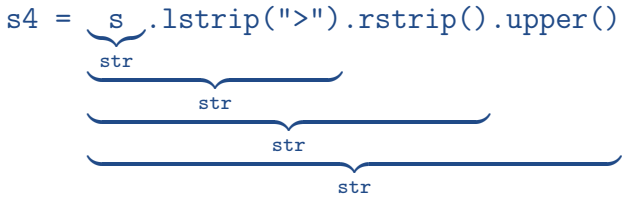
Procedendo per passaggi:

```
s = ">MAnlFKLgaENIFLGrKW   "
s2 = s.lstrip(">")
s3 = s2.rstrip(" ")
s4 = s3.upper()
```

È anche possibile concatenare più metodi assieme senza assegnare il risultato a variabili intermedie:

```
s4 = s.lstrip(">").rstrip().upper()
```

L'anatomia del calcolo è come segue:



Ogni metodo viene applicato al risultato (una stringa) del metodo precedente.

3.4 Conversione Stringa-Numero

È possibile creare una stringa a partire da un oggetto numerico usando la funzione di **conversione tra**
`str()`.

```
n = 10; s = str(10); type(n); type(s)
```

Questo torna utile per concatenare numeri e stringhe:

```
print "Il risultato e':  " + str(12)
```

È anche possibile fare il contrario, e cioè ottenere un numero a partire da una stringa, usando le funzioni di conversione `int()`, `long()`, `float()` — a patto che la stringa rappresenti un numero, naturalmente!

```
s = "10"; s; type(s); n = int(s); n; type(n)
```

Lo stesso vale anche per gli altri tipi numerici:

```
x = float("1.23"); type(x)
```

```
k = long("100000000000L"); type(k)
```

Se la stringa data in pasto a `int()` (`long()`, `float()`) non descrive un numero, la funzione mostrerà un errore.

Alcune stringhe che *non* codificano un numero sono: `"giardinaggio"`, `"1 2 3"` (notate gli spazi), `"quindici"`, `"fifteen"`. La funzione `str` è **stupida**: si aspetta che nella stringa che le passate sia descritto *un solo numero*.

Ad esempio: `x = int("3.14")` non funziona, mentre `x = float("3.14")` sì.

3.5 Esempi

Esempio 20. Creo tre nuove stringhe:

```
a = 'cogito '; b = "ergo"; c = ' sum'
```

Posso concatenare le tre stringhe così:

```
s = a + b + c
```

La lunghezza della stringa finale è `len(s)`. Voglio sapere se la stringa contiene il carattere `'a'`, la sottostringa `'cog'`, le stringhe `b` e `c` concatenate:

```
'a' in s      'cog' in s      (b + c) in s
```

Come per gli operatori di comparazione tra numeri, il risultato di `in` è di tipo **booleano**, e lo posso mettere in una variabile:

```
contieneUnaA = 'a' in s
```

Posso combinare più condizioni così ottenute:

```
contieneUnaVocale = ('a' in s) or ('e' in s) or ('i' in s) or ...
```

Esempio 21. Data una stringa `s = '#A.CC...T.G....'`, posso rimuovere i punti (`.`) ed i "cancelletti" (`#`) **ai bordi della stringa** usando la funzione `strip`:

```
s2 = s.strip ('#.')
```

Si noti che, dato che le stringhe sono immutabili, il metodo `strip` (così come tutte gli altri metodi delle stringhe) non opera direttamente su `s`, ma restituisce una *nuova* stringa, che in questo caso assegnamo ad `s2`. La stringa `s` è rimasta immutata:

```
s1 = "abc"
s2 = s1
s3 = s1.strip ('b')
print s1, s2, s3
```

Le funzioni `rstrip` e `lstrip` funzionano allo stesso modo.

3.6 Esercizi

Esercizio 16. Come posso fare per:

1. Creare una stringa col testo: `sono una stringa`.
2. Creare una stringa col testo: `[il seguente testo]`.
3. Creare una stringa con cinque spazi.
4. Controllare che una stringa contenga cinque caratteri.
5. Controllare che contenga almeno uno spazio.
6. Creare una stringa vuota.
7. Controllare che sia vuota.
8. Creare una stringa che contenga cento ripetizioni di `python e' bello!`.
9. Date le stringhe `"ma biologia"` ed `"e' meglio"`, creare una stringa composita `"ma biologia e'` dunque replicarla mille volte.
10. Creare una stringa che contenga il carattere \.
11. Controllare che contenga il carattere \ all'inizio.
12. Controllare che contenga il carattere \ alla fine.
13. Controllare che contenga il carattere \ all'inizio o alla fine.
14. Controllare che contenga tre caratteri \ all'inizio e alla fine.
15. Controllare che contenga almeno due caratteri \ consecutivi.
16. Controllare che contenga il carattere `x` almeno tre volte sommando le sue occorrenze all'inizio o alla fine della stringa. Ad esempio, `"x...xx"`, `"xx.....x"` e `"....xxx"` ritornano `True`, la somma delle `x` all'inizio e delle `x` alla fine è almeno tre. In `"x...x"` la somma è due, ed il vostro codice dovrebbe stampare `False`.

Esercizio 17. Creare a mano una stringa che contenga, in sequenza, tutti i caratteri dell'alfabeto e metterla nella variabile `alfabeto`.

1. Quanto è lunga?
2. Contiene la sotto-stringa `"pqr"`?

3. Contiene la sotto-stringa `"mon"`?
4. Contiene `abc` (senza virgolette)? Se dà errore, perchè?
5. Estrarre il primo carattere e metterlo nella variabile `primo`.
6. Estrarre l'ultimo carattere e metterlo nella variabile `ultimo`.
7. Estrarre il carattere in mezzo usando `len()` e la divisione intera `//`, metterlo nella variabile `mezzo`.
8. Estrarre tutti i caratteri tranne il primo e l'ultimo, metterli in `nonimporta`.
9. Di che tipo sono `primo`, `ultimo`, `mezzo`? Di che lunghezza?
10. Concatenare le tre variabili in una nuova variabile `prima_mezzo_ultima`.
11. Cosa succede se invoco `x = alfabeto.strip ("zka")`?

Esercizio 18. L'espansione decimale di $\frac{1}{7}$ contiene la cifra `9`? I primi sei decimali sono uguali ai secondi sei?

Suggerimento. Calcolate $\frac{1}{7}$ come `float`, poi convertite a stringa, *etc.*

Esercizio 19. Data la stringa

```
s = "0123456789"
```

cosa succede se scrivo:

1. `s[0:10]`?
2. `s[1000]`?
3. `s[10]`?

Esercizio 20. Data la stringa:

```
"abcd''''"
```

controllare se finisce con quattro singoli apici.

Esercizio 21. Creare una stringa che contenga il seguente testo, inclusi apici e virgolette:

```
urlo':  "non farti piu' vedere!"
```

Esercizio 22. Il comando:

```
dna = open('data/dna-fasta/fasta.1').readlines()[2]
```

legge le sequenza di nucleotidi dal file `data/dna-fasta/fasta.1` (fidatevi) e restituisce una stringa.

1. `dna` è vuota? Quanto è lunga? Contiene degli "a capo"?
2. Contiene la sottostringa `ACCACA`?
3. Controllare se i primi tre caratteri sono uguali agli ultimi tre: il primo col terzultimo, il secondo col penultimo, *etc.*
4. Come sopra, ma in ordine inverso: il primo con l'ultimo, il secondo col penultimo, *etc.*
5. Sostituire `A` con `Ala`, `C` con `Cyt`, etc., mettere in una variabile `dna2`.

Esercizio 23. Data `s = "0 12 23 34 45 12 67 x2"`:

1. Ottenere la posizione della sotto-stringa `"34"`.
2. Ottenere la posizione della sotto-stringa `"99"`.
3. Ottenere la posizione della prima occorrenza della sotto-stringa `"12"`, metterla in `n`. Quindi stampare la posizione della seconda occorrenza usando `n` in qualche modo...

Esercizio 24. Data `s = "0 12 23 34 45 12 67 x2"`, eseguire in sequenza:

1. Rimpiazzare `"67"` con `"99"`, mettere il risultato in `t`. Cosa è successo ad `s`?
2. Rimpiazzare ogni occorrenza di `"12"` con `"11"`.
3. Rimpiazzare ogni spazio con una virgola.
4. Rimpiazzare ogni virgola con un backslash `\`.

Esercizio 25. Date le stringhe:

```
stringa = "a 1 b 2 c 3"; digit="DIGIT"; character="CHAR"
```

rimpiazzare tutte le cifre con il contenuto della stringa `digit`, e tutti i caratteri alfabetici con il contenuto della stringa `character`.