Numeri

Principali tipi numerici

interi normali interi con una gamma prefissata di possibili valori (che dipende dall'architettura, e.g. [-2147483648,2147483647] con parole a 32-bit):

```
3452, -15, 0
```

interi lunghi interi con gamma illimitata (dipende dalla quantità di memoria della macchina):

```
3422252352462462362446,
-134135545444432435
```

reali in virgola mobile numeri reali, rappresentabili sia in notazione scientifica che non:

```
-1.234, 19e5, -2E-6, 45.4E+24
```

booleani assumono solo due valori: `True` (corrispondente a 1) e `False` (corrispondente a 0). Si usano nelle espressioni logiche.

Operazioni su numeri

operatori aritmetici binari: + - * /

- l'interprete esegue operazioni solo su operandi dello stesso tipo.
- Quando gli operandi sono di tipo diverso, l'interprete li converte nel tipo più complesso tra i tipi degli operandi prima di eseguire l'operazione

```
>>> 3 / 2     # operandi interi -> divisione intera
1
>>> 3 + 4.5   # primo operando convertito in reale
7.5
>>> 3.2 / 2   # secondo operando convertito in reale
1.6
>>> 3 / 1.2   # primo operando convertito in reale
2.5
```

Operazioni su numeri

divisione intera: //

- la *divisione intera* restituisce il quoziente della divisione intera a prescindere dal tipo degli operandi numerici
- se di diverso tipo, gli operandi vengono comunque prima convertiti nel tipo più complesso

```
>>> 3 // 2
1
>>> 3.2 // 2
1.0
```

Operazioni su numeri

modulo: %

- l'operazione di *modulo* restituisce il resto della divisione intera
- funziona sia per operandi interi che reali, nel secondo caso il resto sarà un numero reale

```
>>> 3 % 2
1
>>> 4.5 % 2
0.5
```

Operazioni su numeri

elevamento a potenza: **

- eleva il primo operando alla potenza del secondo
- effettua automaticamente la conversione di tipo del risultato se necessario

```
>>> 4 ** 2
16
>>> 4 ** 2.5
32.0
>>> 4 ** -1
0.25
```

Operazioni su numeri

operatori di confronto: < <= > >= == !=

- Confrontano due operandi e restituiscono un valore di verità

```
>>> a = 3
>>> b = 4
>>> a <= b     # a minore o uguale a b
True
>>> -b > a     # -b maggiore di a
False
>>> a != b     # a diverso da b
True
```

Operazioni su numeri

Uguaglianza vs assegnazione

L'operatore di uguaglianza (==) non va confuso con l'istruzione di assegnazione (=)

```
>>> a = 3
>>> b = 4
>>> a == b     # a uguale a b
False
>>> a = b      # a prende il valore di b
>>> print(a)
4
```

Operazioni su numeri

operatori logici: `and or not`

- Permettono di costruire espressioni logiche di cui testare il valore di verità:

```
>>> a == b and a != b
False
>>> a == b or not a == b
True
```

Cosa è vero e cosa è falso

Falso `False`, `None`, lo zero (di qualunque tipo), un oggetto vuoto (stringa, tupla, etc)

Vero tutto il resto

Operatori

Precedenze

- Quando si scrivono espressioni contenenti combinazioni di operatori, l'ordine con cui le operazioni sono eseguite dipende dalla *precedenza* degli operatori
- gli operatori ordinati per precedenza decrescente sono:
  - `* % / //`
  - `+ -`
  - `< <= > >= == <> !=`
  - `not`
  - `and`
  - `or`
- quando due operatori hanno stessa precedenza, l'ordine è da sinistra a destra
- è sempre possibile (e più facile) forzare l'ordine delle operazioni raggruppando tra parentesi tonde le sottoespressioni

Operatori

Precedenze: esempi

```
>>> 3 + 2 * 5
13
>>> (3 + 2) * 5
25
>>> 5 + 4  > 8
True
```

Operatori
Operatori e variabili

- Una variabile viene creata nel momento in cui le si assegna un valore.

```
>>> a = 4 * 2 / 3
```

- Non è possibile utilizzare una variabile prima che venga creata

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

- Una successiva istruzione di assegnazione modificherà il contenuto di una variabile con il risultato dell'espressione a destra dell'assegnazione

```
>>> a = 4 * 5
>>> b = 2
>>> a = a ** b
>>> print(a)
400
```

Stringhe: rappresentazione

Apici singoli e doppi

- Una stringa può essere delimitata da apici *singoli* o *doppi*

```
>>> "abc"
'abc'
>>> 'abc'
'abc'
```

- l'eco dell'interprete (salvo casi particolari) restituisce una stringa delimitandola con apici singoli
- I due tipi di rappresentazione sono utili per scrivere stringhe contenenti l'altro tipo di apice:

```
>>> "l'alba"
"l'alba"
>>> 'tra "apici"'
'tra "apici"'
```

Stringhe: rappresentazione

Sequenze di *escape*

- Le stringhe possono contenere alcuni caratteri speciali di controllo.
- Questi caratteri vengono preceduti dal carattere \ (*backslash*) generando una *sequenza di escape*.
- Sequenze di escape si usano anche per poter scrivere caratteri come apici, e la '\' stessa.
- Le più comuni sequenze di escape sono:

\\ backslash (scrive \)
\' apice singolo (scrive ')
\" apice doppio (scrive ")
n ritorno a capo
t tab orizzontale

Stringhe: rappresentazione

Esempi di sequenze di *escape*

```
>>> print("name\tvalue\n")   # print stampa gia' un ritorno a capo
name    value

>>> "name\tvalue\n"
'name\tvalue\n'
```

eco vs print

- L'eco dell'interprete dei comandi non restituisce lo stesso output della funzione `print`
- La funzione `print` formatta gli oggetti in maniera più fruibile. Per le stringhe, interpreta le sequenze di escape.

Stringhe: rappresentazione

Stringhe *raw*

- In alcuni casi (e.g. percorsi di file in sistemi Windows), la presenza delle sequenze di escape complica le cose
- E' possibile specificare che una stringa deve essere letta così com'è aggiungendo il prefisso `r` (per *raw*):

```
>>> print("C:\system\tmp")
C:\system mp
>>> print(r"C:\system\tmp")
C:\system\tmp
```

Stringhe: rappresentazione

Stringhe con virgolette triple

- Per poter rappresentare una stringa che abbia più righe con le modalità precedenti, è necessario scriverla su un'unica riga inserendo sequenze di escape `\n`.
- E' possibile invece scrivere stringhe su più righe, delimitandole con virgolette triple (usando sia singoli che doppi apici)

```
>>> multiriga = """vai a capo con invio
... l'interprete chiede altro testo
... finisci con tre doppi apici """
>>>
>>>  print(multiriga)
vai a capo con invio
l'interprete chiede altro testo
finisci con tre doppi apici
```

- nelle sessioni interattive, se l'interprete si aspetta del testo su più righe, presenta `...` invece di `>>>`.

Operazioni su stringhe

Operatori

Somma tra stringhe concatena due stringhe e restituisce il risultato

```
>>> "biologia" + "molecolare"
'biologiamolecolare'
```

Prodotto stringa per intero replica la stringa un numero di volte pari all'intero, e restituisce la stringa risultante (e.g. utile per stampare delimitatori)

```
>>> "=" * 20
'===================='
```

Operazioni su stringhe

Operator overloading

- Somma e prodotto producono risultati diversi rispetto alla versione con operandi numerici
- Si parla di *operator overloading*: il risultato di un'operazione dipende dal tipo degli operandi
- Se non è stata definita un'operazione per il tipo degli operandi proposti, si genera un errore

```
 >>> "x" * "="
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can't multiply sequence by
non-int of type 'str'
```

Stringhe come sequenze

Operazioni su sequenze

- Una stringa è una **sequenza** di caratteri
- Una sequenza è una collezione di oggetti (in questo caso caratteri) con ordinamento posizionale (da sinistra a destra)
- Questa caratteristica fa sì che le stringhe supportino una serie di operazioni su sequenza
- Ad esempio si può calcolare la lunghezza di una stringa (numero di caratteri):

```
>>> len("abcd")
4
>>> len("abcd\n ")  # \n vale 1 carattere
6
```

Operazioni su sequenze

Ricerca

- L'operatore di confronto `in` restituisce `True` se un certo carattere (o sottostringa) si trova in una stringa, `False` altrimenti

```
>>> s = "abcd"
>>> "a" in s
True
>>> "bc" in s
True
>>> "bcd" in s
True
>>> "ad" in s
False
```

Operazioni su sequenze

Indicizzazione

- E' possibile recuperare un carattere all'interno di una stringa specificandone la posizione (a partire da zero):

```
>>> s = "abcd"
>>> s[0]
'a'
>>> s[3]
'd'
>>> s[4]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: string index out of range
```

- Superare gli estremi della stringa genera un errore

Operazioni su sequenze
Indicizzazione

- E' possibile specificare posizioni negative, che si contano da destra a sinistra (a partire da -1):

```
>>> s = "abcd"
>>> s[-1]
'd'
>>> s[-3]
'b'
>>> s[-4]
'a'
```

- La posizione può essere specificata tramite un'espressione che restituisca un valore numerico

```
>>> a = -3
>>> b = 1
>>> s[a+b]
'c'
```

Operazioni su sequenze

Sottostringa

- Permette di restituire porzioni di stringa (*slices*)
- `X[i:j]` restituisce la sottostringa che va dalla posizione `i` *inclusa* alla `j` *esclusa*

```
>>> s = "abcd"
>>> s[1:3]
'bc'
```

- `i` vale 0 se non specificato. `j` vale la lunghezza della stringa se non specificato.

```
>>> s[1:len(s)]      >>> s[:3]
'bcd'                'abc'
>>> s[1:]            >>> s[:-1]
'bcd'                'abc'
>>> s[0:3]           >>> s[:] # copia stringa
'abc'                'abcd'
```

Operazioni su sequenze

Operazioni su altri tipi di sequenze

- Le operazioni su sequenze sono definite su tutti i tipi sequenza
- Anche liste e tuple sono sequenze (vedremo).

```
>>> a = [1,2,3,4]
>>> len(a)
4
>>> a[-2]
3
>>> a[:-2]
[1, 2]
>>> a * 2
[1, 2, 3, 4, 1, 2, 3, 4]
>>> a + [5,6]
[1, 2, 3, 4, 5, 6]
```

Immutabilità delle stringhe

Oggetti immutabili

- Le stringhe sono un tipo di oggetti **immutabili**
- Non è possibile modificare una stringa una volta creata
- Le operazioni su stringhe restituiscono sempre un *nuovo* oggetto con il risultato. Il *vecchio* oggetto rimane invariato

```
>>> s = "abcd"
>>> s * 2
'abcdabcd'
>>> s
'abcd'
```

Immutabilità delle stringhe

Oggetti immutabili

- Non è possibile quindi sostituire un carattere all'interno di una stringa

```
>>> s[2] = "h"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item
assignment
```

- E' sempre possibile però assegnare la nuova stringa creata alla variabile che si riferiva alla vecchia stringa

```
>>> s = "abcd"
>>> s = s * 2
>>> s
'abcdabcd'
```

- Non stiamo modificando la vecchia stringa, ma solo cambiando l'oggetto a cui la variabile si riferisce

Conversioni di tipo

Operazioni tra stringhe e numeri

- Abbiamo detto che non è possibile eseguire operazioni tra tipi non compatibili (e.g. sommare una stringa e un numero)
- E' possibile però convertire esplicitamente un oggetto di un tipo in uno di un altro tipo (dove tale conversione è definita)

```
>>> "x=" + str(12)   # str da numeri a stringhe
'x=12'
>>> int("3") + 4     # int da stringhe a interi
7
>>> float("3.2") / 1.2 # float da stringhe a reali
2.666666666666667
```

Formattazione di stringhe

Operatore %

- L'operatore % serve a formattare una stringa a partire da una tupla di argomenti.

```
>>> "nome=%s,lunghezza=%d" % ("7rsa",356)
'nome=7rsa,lunghezza=356'
```

- La stringa a sinistra contiene degli specificatori di conversione (%s), la tupla a destra i valori da sostituirci
- %s indica una stringa, in pratica utilizza la funzione `str` per convertire l'elemento corrispondente della tupla in una stringa

Metodi

Metodi di oggetti

- La maggior parte dei tipi di oggetti Python (a parte i numeri) possiede *metodi*
- Un metodo è semplicemente una funzione che esegue una certa operazione sull'oggetto
- Un metodo può avere degli argomenti e/o restituire un risultato (come abbiamo visto per le funzioni. e.g. `len("abcd")`)
- I metodi di un oggetto si chiamano tramite un riferimento all'oggetto stesso, facendolo seguire da "." e il nome del metodo, eventualmente seguito dai valori degli argomenti tra parentesi

```
>>> s = "abcd"
>>> s.replace("bc","xx")
'axxd'
```

Metodi di stringhe

Ricerche

s.find(sub) Restituisce l'indice della prima occorrenza della sottostringa `sub` (o -1 se `sub` non c'è in `s`). E' possibile specificare una gamma di posizioni in cui cercare più piccola della stringa intera

```
>>> s = "abbcddbbc"
>>> s.find("bc")
2
>>> s.find("bc",4)
7
>>> s.find("bc",4,6)
-1
```

Metodi di stringhe

Ricerche

endswith restuisce vero se la stringa finisce con una certa sottostringa (o una scelta da una tupla), falso altrimenti.

```
>>> s = "abbcddbbc"
>>> s.endswith("bc")
True
>>> s.endswith("xx")
False
>>> s.endswith(("bb","bc"))
True
```

Metodi di stringhe

Modifiche

replace sostituisce tutte le occorrenze (o il numero richiesto) di una certa sottostringa con un'altra

```
>>> s = "  MET CYS ASP CYS\n"
>>> s.replace("CYS","C")
'  MET C ASP C\n'
>>> s.replace("CYS","C",1)
'  MET C ASP CYS\n'
```

upper,lower convertono una stringa in maiuscola o minuscola

```
>>> s.lower()
'  met cys asp cys\n'
```

lstrip,rstrip,strip rimuovono spazi,tab e ritorni a capo da inizio, fine o da inizio e fine stringa

```
>>> s.lstrip()     >>> s.rstrip()
'MET CYS ASP CYS\n'    '  MET CYS ASP CYS'
>>> s.strip()
'MET CYS ASP CYS'
```

Metodi di stringhe

Conversioni con liste

split permette di convertire una stringa in una lista di sottostringhe, specificando il delimitatore da usare (spazi,tab o ritorni a capo di default)

```
>>> s = "spazi o virgole, a scelta, per separare"
>>> s.split()
['spazi', 'o', 'virgole,', 'a', 'scelta, ', 'per',
                                        'separare']
>>> s.split(",")
['spazi o virgole', ' a scelta', ' per separare']
```

Metodi di stringhe

Conversioni con liste

join permette di convertire una lista in una stringa, concatenando gli elementi e usando la stringa di partenza come delimitatore

```
>>> s = ":"
>>> s.join(["a","b","c","d"])
'a:b:c:d'
```

Nota

I metodi si possono applicare anche direttamente agli oggetti, non solo a variabili che siano loro riferimenti

```
>>> "".join(["a","b","c","d"])
'abcd'
```

Metodi di stringhe

Esempio

- Prendere una sequenza proteica e stamparne le sottosequenze separate da C o H, una per riga

```
>>> s = "ACFGEDHTGJDFG"
>>> s1 = s.replace("H","C")
>>> s1
'ACFGEDCTGJDFG'
>>> l = s1.split("C")
>>> l
['A', 'FGED', 'TGJDFG']
>>> s2 = "\n".join(l)
'A\nFGED\nTGJDFG'
>> print(s2)
A
FGED
TGJDFG
```

Metodi di stringhe

Esempio (rivisto)

- Invece di assegnare i risultati delle operazioni intermedie a variabili *temporanee* (e.g. `s1,l,s2`) si possono direttamente *concatenare* le operazioni:

```
>>> s = "ACFGEDHTGJDFG"
>>> s.replace("H","C").split("C")
['A', 'FGED', 'TGJDFG']
>>> print("\n".join(s.replace("H","C").split("C")))
A
FGED
TGJDFG
```

Documentazione

dir

- La funzione `dir` prende come argomento un oggetto (o un suo riferimento), e restituisce la lista dei suoi attributi (vedremo) e metodi.

```
>>> dir("")
['__add__', '__class__', '__contains__', '__delattr__'
, '__doc__', '__eq__', '__ge__', '__getattribute__',
'__getitem__', '__getnewargs__', '__getslice__',
'__gt__', '__hash__', '__init__', '__le__', '__len__',
'__lt__', '__mod__', '__mul__', '__ne__', '__new__',
'__reduce__', '__reduce_ex__', '__repr__', '__rmod__',
'__rmul__', '__setattr__', '__str__', 'capitalize',
'center', 'count', 'decode', 'encode', 'endswith',
'expandtabs', 'find', 'index', 'isalnum', 'isalpha',
'isdigit', 'islower', 'isspace', 'istitle', 'isupper',
'join', 'ljust', 'lower', 'lstrip', 'partition',
'replace', 'rfind', 'rindex', 'rjust', 'rpartition',
'rsplit', 'rstrip', 'split', 'splitlines',
'startswith', 'strip', 'swapcase', 'title',
'translate', 'upper', 'zfill']
```

Documentazione

dir

- Le parole che iniziano per __ indicano attributi e metodi speciali (da ignorare per adesso)
- Le altre parole indicano metodi che possono essere eseguiti sull'oggetto (e.g. `capitalize`, `find`)

Documentazione

help

- La funzione `help` permette di stampare informazioni sull'argomento passato
- Chiamando `help` con il nome di un metodo si ottiene una descrizione di cosa fa il metodo

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

join(...)
    S.join(sequence) -> string

    Return a string which is the concatenation
    of the strings in the sequence.
    The separator between elements is S.
```

- Per uscire dall'help e tornare al prompt dei comandi, premere il tasto `q` (*quit*)