UNIVERSI E COERCIZIONE

Universi dei numeri interi, razionali, reali e complessi e uso dell'operatore **in**

```
sage: ZZ
Integer Ring
sage: QQ
Rational Field
sage: RR
Real Field with 53 bits of precision
sage: CC
Complex Field with 53 bits of precision
```

```
sage: 1 in ZZ
True
sage: 1/2 in ZZ
False
sage: 1/2 in QQ
True
sage: sqrt(2) in QQ
False
sage: sqrt(2) in RR True
```

NB: **i** (o anche **I**) rappresenta il numero immaginario i (radice quadrata di -1): **i** in CC rende True

Per capire a quale universo appartiene un oggetto si usa la funzione parent

```
sage: parent(1)
Integer Ring
sage: parent(1/2)
Rational Field
sage: parent(5.7)
Real Field with 53 bits of precision
sage: parent(pi.n())
Real Field with 53 bits of precision
```

parent rende sempre l'universo di appartenenza più semplice per ogni numero

Universo dei simboli

```
sage: parent(sqrt(2))
Symbolic Ring
sage: parent(I)
Symbolic Ring
sage: parent(pi)
Symbolic Ring
```

NB: all'universo dei simboli appartengono anche le variabili simboliche

```
sage: parent(1 + 2)
Integer Ring
sage: parent(1/2 + 2)
Rational Field
sage: parent(1/2 + 2.0)
Real Field with 53 bits of precision
```

Conversioni automatiche tra numeri appartenenti ad universi diversi

```
sage: exp(1.)*pi
2.71828182845905*pi
sage: parent(exp(1.)*pi)
Symbolic Ring
```

Sage assegna un'espressione all'universo (compatibile) che presenta la precisione più elevata

Coercizione

```
sage: QQ(.5)
1/2
sage: parent(QQ(.5))
Rational Field
sage: RR(sqrt(2))
1.41421356237310
sage: parent(RR(sqrt(2)))
Real Field with 53 bits of precision
```

Per convertire un numero ad un universo (compatibile) diverso si «applica» l'universo (come fosse una funzione).

Se la conversione non è ammissibile si ottiene un messaggio di errore (ad esempio `QQ(I)`)

Universo dei booleani

```
sage: parent(True)
<class 'bool'>

sage: not True
False
sage: not False
True
```

Due soli valori: True e False

Operatori: not, and, or, ^^ (corrisponde all'xor o or esclusivo)

Operatori di confronto: ==, != (o <>), <, >, <=, >= (rendono True o False)

VARIABILI

Vanno distinti due tipi di variabili;

- Variabili di memoria: aree di memoria dove sono conservati dei valori
- Variabili simboliche: indicano valori indeterminati

Le prime sono le classiche variabili della programmazione, le seconde corrispondono a quelle della matematica e appartengono all'universo dei simboli.

- ❑ Non è necessario dichiarare espressamente le prime, mentre vanno dichiarate le seconde.
- ❑ SageMath definisce automaticamente la variabile simbolica x

Variabile simbolica

```
sage: var("y")
y
sage: 3*y - y
2*y
sage: e*e^y
e^(y + 1)
```

Variabile di memoria

```
sage: m=2^19-1
sage: m
524287
sage: (m+1).factor()
2^19
```

```
sage: a,b=1,2
sage:a
1
sage:b
2
sage: c,d,e=2,3,5
sage: c,d,e
(2, 3, 5)
```

Assegnazione multipla

Visualizzazione multipla

```
sage:
a,_,c=1,2,3
sage: a
1
sage:c
3
sage: a = b = 1
sage: a
1
sage: b
1
```

Assegnazione con «salto» di variabile

Assegnazione multipla del medesimo valore

```
sage: x = 1
sage: a = 2
sage: restore('x')
sage: restore('a')
sage: x
x
sage: a
NameError    Traceback (most recent call last)
....
name 'a' is not defined
```

restore() resetta le variabili globali predefinite al loro valore di default

```
sage: a = 1
sage: b = 2
sage: c = 5
sage: x = 56
sage: reset()
sage: a
NameError    Traceback (most recent call last)
.....
NameError: name 'a' is not defined
sage: x
x
```

reset() cancella tutte le variabili definite dall'utente e resetta tutte le variabili globali al loro valore di default

del opera sulla singola variabile indicata (anche reset() passando il nome della variabile)

```
sage: a = [2, 3,4 ,5 ]
sage: del a
sage: a
NameError          Traceback   (most
recent call last)
.....
NameError: name 'a' is not defined
```

LISTE

Le liste sono raccolte ordinate di oggetti. Sono indicizzate a partire dall'indice zero.

```
sage: [6,28,496,8128]
[6, 28, 496, 8128]
sage: L = [2,3,5,7,11,13,17,2]
sage: L[0]
2
sage: L[5]
13
sage: L[6]
17
```

```
sage: len(L)
8
sage: len([2,3,5,7,11])
5
```

len rende la lunghezza di una lista

```
sage: M = [ 'apple', 'pear']
sage: parent(M[1])
 <type 'str'>
```

Esempio di lista di stringhe

```
sage: M = [[1,2],[1,3],[1,4]]
sage: M[2]
[1,4]
sage: len(M)
3
sage: M[2][1]
4
```

Si possono avere liste di liste

Significa: considera l'elemento di indice 2 della lista (cioè [1,4]) e accedi al suo elemento di indice 1 (cioè 4)

In questo caso M[1][2] non funzionerebbe

Indicizzazione e «slicing»

```
sage: M = [1, 2, 0, 3, 4, 0, 4, 5]
sage: M
[1, 2, 0, 3, 4, 0, 4, 5]
sage: M[2:5]
[0, 3, 4]
sage: M[2:]
[0, 3, 4, 0, 4, 5]
sage: M[:5]
[1, 2, 0, 3, 4]
sage: M[:]
[1, 2, 0, 3, 4, 0, 4, 5]
```

Notazione “slice”

Nell’esempio, vengono estratti 3 elementi (3=5-2) a partire da quello di indice 2, arrivando fino all’elemento di indice 4, cioè quello che precede l’ elemento di indice 5 (4=5-1).

Se un indice manca, si arriva all’ultimo elemento o si inizia dal primo.

Se mancano entrambi gli indici si ottiene tutta la lista.

Se un indice è negativo, si pensi alla lista come se fosse ciclica.

```
sage: M[:-2]
[1, 2, 0, 3, 4, 0]
```

Qui si termina all'elemento precedente quello situato due posizioni prima dell'elemento in posizione zero (il primo), cioè il terzultimo elemento.

```
sage: M[-2:]
[4,5]
```

Qui si inizia dall'elemento posizionato due posizioni prima dell'elemento in posizione zero (il primo), quindi il penultimo elemento.

```
sage: M[-4:-2]
[4, 0]
```

Qui si inizia dal quartultimo elemento e si termina al terzultimo elemento.

```
sage: M[-2:-2]
[]
```

Qui si inizia al penultimo elemento e si termina al terzultimo, quindi la lista risultante è vuota.

```
sage: M = [2, 3, 3, 3, 2, 1, 8, 6, 3]
sage: M.index(3)
1
sage: M.index(14)
ValueError: list.index(x): x not in list
sage: M.count(3)
4
```

index restituisce l'indice del primo elemento corrispondente al valore passato come parametro

count restituisce il numero di volte che l'elemento passato come parametro compare nella lista

Creazione di liste

```
sage: [1..7]
[1, 2, 3, 4, 5, 6, 7]
sage: [4..9]
[4, 5, 6, 7, 8, 9]
sage: [2,4..10]
[2, 4, 6, 8, 10]
```

```
sage: [1,4..13]
[1, 4, 7, 10, 13]
sage: [1,11..31]
[1,11, 21,31]
sage: [1,11..35]
[1,11, 21,31]
```

Uso di .. per creare liste con elementi equidistanti.

```
sage: [pi,4*pi..32]
[pi, 4*pi, 7*pi, 10*pi]
```

Funziona anche con alcuni elementi simbolici come pi.

Modifica di liste

```
sage: M = [2,3,3,3,2,1,8,6,3]

sage: M.sort(); M

[1, 2, 2, 3, 3, 3, 3, 6, 8]

sage: M.index(2)

1

sage: N = M[:]

sage: N.sort()

sage: N

[1, 2, 2, 3, 3, 3, 3, 6, 8]

sage: M

[2, 3, 3, 3, 2, 1,8, 6, 3]
```

sort() consente di ordire una lista «in place» (cioè modifica la lista stessa, non rende una copia)

NB: il ; permette di inserire due comandi sulla stessa linea

Se non si vuole modificare direttamente la lista, la si copi e si ordini la copia

Le liste sono «mutabili», cioè i singoli valori possono essere modificati

```
sage: L = [1,2,3,4]
sage: L[0]=-1
sage: L
[-1, 2, 3, 4]
```

Per togliere un elemento: remove

```
sage: M = [1,2,3,0,3,4,4,0,4,5]
sage: M.remove(3)
sage:M
[1, 2, 0, 3, 4, 4, 0, 4, 5]
```

```
sage: L = [1,2,3]
sage: L.append(4)
sage: L
[1, 2, 3, 4]
```

Per aggiungere un elemento: append

```
sage: L=[1,2]
sage: L.extend([10,11,12])
sage: L
[1,2, 10, 11, 12]
```

Per concatenare due liste extend o l'operatore +

```
sage: [1,3,5]+[2,4,6]+[100] [1,3, 5, 2, 4, 6, 100]
sage: [2,4,6]+[1,3,5]+[100] [2, 4, 6, 1,3, 5, 100]
```

Operazioni su liste

```
sage: sum([1,2,3])
6
```
Somma di elementi di una lista

```
sage: prod([1..4])
24
```
Prodotto di elementi di una lista

```
sage: zip([1,2,3,4],['a','b','c','d'] )
[(1, 'a'), (2, 'b'), (3, 'c'), (4, 'd')]
```
Accoppiamento degli elementi di due liste

```
sage: zip([1,2,3,4],['a','b','c'] )
[(1, 'a'), (2, 'b'), (3, 'c')]
sage: zip([1],['a','b','c'] )
[(1, 'a')]
```
zip opera fino ad esaurire una delle due liste

INSIEMI

Gli insiemi sono simili alle liste, ma con due importanti differenze:

- Gli elementi non sono ordinati (non si può accedere a un elemento tramite un indice)
- Ogni elemento può apparire una volta sola nell'insieme

```
sage: y = [2,3,3,3,2,1,8,6,3]
sage: A = Set(y); A
```

Conversione di una lista in un insieme

```
{8, 1, 2, 3, 6}
sage: A.cardinality()
```

cardinality() rende il numero di elementi dell'insieme

(in alternativa len(A), come per le liste)

```
5
sage: 8 in A
```

in per verificare l'appartenenza di un elemento

```
True
```

Operazioni tra insiemi

```
sage: y = [2,3,3,3,2,1,8,6,3]
sage: A = Set(y)
sage: A
{8, 1, 2, 3, 6}
sage: B = Set([8,6,17,-4,20, -2 ])
sage: B
{17, 20, 6, 8, -4, -2}
```

```
sage: A.union(B)
{1,2, 3, 6, 8, 17, 20, -4, -2}
sage: A.intersection(B)
{8, 6}
sage: A.difference(B)
{1, 2, 3}
sage: B.difference(A)
{17, 20, -4, -2}
sage: A.symmetric_difference(B)
{17, 2, 3, 20, 1, -4, -2}
```

```
sage: A = Set([1,2,3]); A
{1, 2, 3}
sage: powA = A.subsets(); powA
Subsets of {1,2,3}
sage: pairsA = A.subsets(2); pairsA
Subsets of {1,2,3} of size 2
sage: powA.list()
[{}, {1}, {2}, {3}, {1,2}, {1,3}, {2, 3}, {1,2, 3}]
sage: pairsA.list()
[{1,2}, {1,3}, {2, 3}]
```

subsets() consente di costruire sottoinsiemi

list() per visualizzare la lista dei sottoinsiemi

MATRICI – un esempio

```
sage: A1 = matrix(QQ, 3, [1,2,3,-1,2,5,2,3,1]); A1

[ 1 2 3] [-1 2 5] [ 2 3 1]

sage: b=vector(QQ, [1,5,2]); b
(1, 5, 2)

sage: A1.solve_right(b)
(-5/2, 5/2, -1/2)

sage: A1\b
(-5/2, 5/2, -1/2)
```

In questo esempio vengono creati una matrice e un vettore di numeri razionali e viene risolto il corrispondente sistema con due modalità diverse (funzione solve_right e operatore matriciale \ )

STRINGHE

In Sage le costanti di tipo stringa possono essere costruite con le virgolette (singole o doppie).
L'accesso ai singoli caratteri avviene per mezzo di un indice.

```
sage: s='I am a string'
sage: s
'I am a string‘
sage: print s
I am a string
```

```
sage: a='mathematics‘
sage: a[0]
'm'
sage: a[4]
'e'
```

```
sage: b='Gauss‘
sage: len(b)
5
```

Lunghezza di una stringa

```
sage: b + " is " + a
'Gauss is mathematics'
```

Concatenazione di stringhe

```
sage: s.split()
['I', 'am', 'a', 'string']
```

Suddivisione di una stringa

```
sage: vals = "18,spam,eggs,28,70,287,cats"
sage: vals.split(',')
['18','spam', 'eggs','28','70','287','cats']
```

Suddivisione di una stringa in base ad uno specifico separatore

```
sage: list(map(Integer, data.split(',')))
[17, 18, 20, 19, 18, 20]

sage: data.split(',')
['17', '18', '20', '19', '18', '20']

sage: L = ['Learning', 'Sage', 'is', 'easy.']
sage: '' ''.join(L)
'Learning Sage is easy.'
```

Uso di map per trasformare una stringa contenente numeri interi in una lista di interi (qui data è la stringa ‘17,18,20,19,18,20’)

Il contrario di split è join (chiamato nell’esempio su una stringa contenente uno spazio)

ISTRUZIONE CONDIZIONALE if

```
sage: n=44
sage: if n%2 == 0:
          print(n/2)
```

SageMath usa l'indentazione per indicare quali sono le istruzioni da eseguire nella struttura if e in generale anche nelle istruzioni di ciclo

```
sage: m=31
sage: if m%3==0:
          print(m/3)
      elif m%3==1:
          print((m-1)/3)
```

elif corrisponde a «else if» e nei vari rami **if** e **elif** possono essere inserite anche più istruzioni (rispettando l'indentazione)

```
if condizione 1:
        istruzione a1
        istruzione a2
        ……
elif condizione 2:
        istruzione b1
        istruzione b2
        ……
else:
        istruzione c1
        istruzione c2
        ……
```

SCHEMA GENERALE

NB: possono esserci più rami **elif** e né i rami **elif** né il ramo **else** sono obbligatori

ISTRUZIONE DI CICLO while

```
sage: i=0
sage: while i < 5:
          print i^2
          i=i+1
0
1
4
9
16
```

SCHEMA GENERALE

```
sage: while condizione:
          istruzione 1
          istruzione 2
          ..............
```

ISTRUZIONE DI CICLO for

```
sage: for i in [0..4]:
          print i^2
```

```
0
1
4
9
16
```

```
sage: for indice in [a..b]:
          istruzione 1
          istruzione 2
          ..............
```

Esistono i comandi **break** e **continue**. Il primo interrompe il ciclo, il secondo lo fa saltare direttamente all'iterazione successiva

Il ciclo **for** si può impiegare su qualunque tipo di lista, per esempio con liste di stringhe

```
sage: for str in ["apple","banana","coconut","dates"]:
          print str.capitalize()

Apple
Banana
Coconut
Dates
```

```
sage: for char in "Leonhard Euler":
          print char.swapcase()

I
E
O
N
H
A
R
D

...
```

Il ciclo **for** si può impiegare anche
per scorrere i caratteri di una stringa

LIST COMPREHENSIONS (CICLI NELLE LISTE)

Notazione matematica per l'insieme dei numeri pari tra 0 e 20: {2 • *k* | *k e* Z, 0<=k<=10}

"Equivalente" in SageMath:

```
sage: [ 2*k for k in [0..10] ]
```

[0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20]

```
sage: [pi/4,pi/2..2*pi]
[1/4*pi, 1/2*pi, 3/4*pi, pi, 5/4*pi, 3/2*pi, 7/4*pi,
2*pi]
sage: [ cos(x) for x in [pi/4, pi/2..2*pi]]
[1/2*sqrt(2), 0, -1/2*sqrt(2), -1, -1/2*sqrt(2), 0,
1/2*sqrt(2), 1]
```

Si potrebbe ottenere lo stesso risultato anche con **map**:

```
list(map(cos(x),[pi/4, pi/2..2*pi]))
```

```
sage: [ k for k in [1..19] if gcd(k,20) == 1 ]
[1,3, 7, 9, 11, 13, 17, 19]
```

Si può inserire anche una condizione con **if**

Corrisponde a $\{k \in \mathbb{N} \mid 0<k<20 \text{ and } \gcd(k,20) = 1\}$

Prodotto cartesiano con la list comprehension

```
sage: U = [ k for k in [1..19] if gcd(k,20) == 1]
sage: [ (a,b) for a in U for b in U ]
```

[(1, 1), (1,3), (1,7), (1,9), (1, 11), (1, 13), (1, 17), (1, 19), (3, 1), (3, 3), (3, 7), (3, 9), (3, 11), (3, 13), (3, 17), (3, 19), (7, 1), …]

Corrisponde a $Ax\,B = \{(a, b) \mid a \in A, b \in B\}$

Altri esempi:

```
sage: [ a*b for a in U for b in U ]
sage: [ a + b for a in U for b in U ]
sage: [ gcd(a,b) for a in U for b in U ]
```

Nella list comprehension si può usare anche **if/else**

```
sage: U
[1,3, 7, 9, 11, 13, 17, 19]
sage: [ 'prime' if x.is_prime() else 'not prime' for x in U]
['not prime', 'prime', 'prime', 'not prime', 'prime', 'prime', 'prime', 'prime']
```

FUNZIONE sum

La funzione **sum** consente di calcolare agevolmente somme, anche simboliche e anche di serie

```
sage: k, n = var('k, n')
sage: sum(k, k, 1, n).factor()
```
½ (n + 1)n

```
sage: k, n = var('k, n')
sage: sum(1/(n*(n+1)), n, 1, infinity)
```
1

```
sage: n, k, y = var('n, k, y')
sage: sum(binomial(n,k) * x^k * y^(n-k), k, 0, n)
```
$(x + y)^n$

Iterations functions of the `..range` form for `a`, `b`, `c` integers	
`for k in [a..b]: ...`	constructs the list of Sage integers $a \leq k \leq b$
`for k in srange (a, b): ...`	constructs the list of Sage integers $a \leq k < b$
`for k in range (a, b): ...`	constructs a list of Python integers (`int`)
`for k in xrange (a, b): ...`	enumerates Python integers (`int`) without explicitly constructing the corresponding list
`for k in sxrange (a, b): ...`	enumerates Sage integers without constructing a list
`[a,a+c..b]`, `[a..b, step=c]`	Sage integers $a, a+c, a+2c, \ldots$ as long as $a+kc \leq b$
`..range (b)`	equivalent to `..range (0, b)`
`..range (a, b, c)`	sets the iteration increment to c instead of 1

TABLE 3.3 – The different enumeration loops.

FUNZIONI

Algoritmo di Euclide per il massimo comun divisore

```
sage: def euclid(a,b):
        r = a%b
        while r != 0:
            a=b; b=r
            r = a%b
        return b
```

Anche nelle funzioni si usa l'indentazione

```
sage: euclid(75,21)
3
```

Chiamata della funzione euclid

SCHEMA GENERALE

NB: parametri e return sono opzionali

```
sage: def nomefunzione(parametro1, parametro2, …):
        istruzione 1
        istruzione 2
        ……..
        return …
```

```
sage: def h():
          return 1/2
```

Esempio di funzione senza parametri

```
sage: h()
1/2
```

Esempio di funzione senza return.

Tutti i comandi di Sage rendono qualcosa.
Se non viene specificato return, allora viene restituito automaticamente l'oggetto None.

Buona norma è in ogni caso restituire il valore a cui si è interessati per mezzo di return.

```
sage: def lazy(x):
          print x^2
sage: lazy(sqrt(3))
3
sage: a = lazy(sqrt(3))
3
sage: a
None
```

```
sage: def s(x):
          return x^2, x^3
sage: s(1)
(1, 1)
sage: s(2)
(4, 8)
sage: a, b=s(3)
sage: a
9
sage: b
27
```

Si possono restituire più valori, separandoli con virgole dopo return

Le variabili nelle quali si desidera che i valori restituiti vengano salvati vanno posizionate prima del simbolo di assegnazione separate dalla virgola

Strumenti di programmazione funzionale in SageMath

Sage contiene alcuni costrutti che costituiscono degli strumenti di programmazione funzionale.

La programmazione funzionale è un paradigma di programmazione che fa uso della valutazione di funzioni. L'obiettivo è quello di evitare effetti collaterali, come la modifiche delle variabili all'esterno dell'ambito delle funzioni stesse o la modifica dei loro argomenti.

Questo rende più agevole la verifica della correttezza e della mancanza di difetti del programma.

Funzione map

map ha come argomenti una funzione e una lista e restituisce un oggetto map (da convertire eventualmente in lista) con elementi i valori della funzione applicata ai singoli valori della lista di partenza

```
sage: a=map( cos, [0, pi/4, pi/2, 3*pi/4, pi] ); list(a)
[1, 1/2*sqrt(2), 0, -1/2*sqrt(2), -1]
sage: map_threaded( cos, [0, pi/4, pi/2, 3*pi/4, pi] )
[1, 1/2*sqrt(2), 0, -1/2*sqrt(2), -1]
sage: list(map(factorial,[1,2,3,4,5]))
[1,2, 6, 24, 120]
sage: sum(map(exp,[1,2,3,4,5]))
e + e^2 + e^3 + e^4 + e^5
```

Conversione del risultato di map in lista

map_threaded rende direttamente una lista

Funzione filter

filter costruisce una oggetto (da convertire eventualmente in lista) filtrando gli elementi di una lista partenza in base ad una condizione

```
sage: list( filter (is_prime, [1..55]))
```

```
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43,
47, 53]
```

che risulta equivalente a

```
sage: [n for n in [1..55] if is_prime(n)]
```

```
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43,
47, 53]
```

Espressione lambda

L'espressione lambda è un modo particolare di definire una funzione

```
funz_esempio = lambda x: risultato1 if condizione else risultato2
```

```
def funz_esempio (x):
if condizione:
    return risultato1
else:
    return risultato2
```

NB: la parte "if condizione else risultato2" è opzionale

```
sage: list(map(lambda n: 2*n , [1..5]))
[2, 4, 6, 8, 10]

sage: list(map(lambda n: n if n>3 else 2*n , [1..5]))
[2, 4, 6, 4, 5]
```

Dizionari

I dizionari associano dei valori a delle chiavi

```
sage: D={}; D['one']=1; D['two']=2; D['three']=3; ...; D['ten']=10

sage: D['two'] + D['three']
5
sage: D={'one':1, 'two':2, 'three':3, ..., 'ten':10}
```

Notazione alternativa

L'operatore in verifica se una chiave compare nel dizionario.

del D['one'] o D.pop('one') cancellano il corrispondente elemento dal dizionario.

I metodi keys() e values() rendono rispettivamente le chiavi e i valori del dizionario (da trasformare in insieme o in lista).

I dizionari possono essere impiegati per la memoizzazione, cioè la tecnica di salvare i valori di una funzione per riutilizzarli in seguito.

FILE DI COMANDI E SESSIONI

```
#    Begin pythag.sage

a=3

b=4

c=sqrt(a^2+b^2)

print(c)

#    End
```

Esempio di un file di comandi (chiamato qui pythag.sage che può essere caricato in Sage con il comando load.
Il simbolo # indica un commento.

```
sage: load pythag.sage
```

Caricamento ed esecuzione del file

```
5
```

```
a,b,c
```

Dopo il caricamento del file, tutte le variabili inizializzate risultano esistere nella sessione Sage

```
(3, 4, 5)
```

Salvataggio e recupero di sessioni

Salvataggio di una sessione in un file (tipicamente sage_session.sobj)

```
sage: a=101

sage: b=103

sage: save_session()

sage: exit

Exiting SAGE (CPU time 0m0.06s,

Wall time 0m31.27s).
```

Recupero della sessione dal file

```
sage: load_session()

sage: a

01

sage: b

103
```

Le variabili vengono recuperate

```
sage: T=1729
sage: save_session('nomesessione')
sage: exit
Exiting SAGE (CPU time 0m0.06s, Wall time
0m16.57s).
```

Si può specificare il nome della sessione

```
sage: load_session('nomesessione')
sage:T
1729
```

Si usa il nome per recuperare la sessione

Usando Jupyter si può salvare il proprio notebook attraverso il menu e allo stesso modo caricarne uno