Table of Contents

Fondamenti di Programmazione

Andrea Sterbini

lezione 3 - 10 ottobre 2022

0.1 Ancora a proposito di stringhe

Le **stringhe** `str` sono piu complesse di `int` e `float`.
E' possibile pensarle codificate in memoria come `vettori` (sequenze di elementi uguali).

In [1]:
```
1 nome = 'andrea sterbini'
```

carattere	a	n	d	r	e	a		s	t	e	r	b	i	n	i
indice	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

1. Si ottiene la lunghezza della sequenza di caratteri con `len(nome)`

2. E' possibile anche:
   - indicizzare la stringa, ossia accedere **in maniera secca** (`random access`) a ciascun carattere
   - l'indicizzazione avviene come se fosse un vettore
   - per vedere il carattere quinto, non importa che chieda prima secondo, terzo e quarto. Accedo diretto al quinto `nome[4]`

3. Molto importante!
   <u>in informatica si inizia a contare da 0</u>

   - `nome[0]` è il primo elemento

carattere	a	n	d	r	e	a		s	t	e	r	b	i	n	i
indice	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

In [2]:
```
1 nome[0]
```

Out[2]: 'a'

In [3]:
```
1 nome[100]  #attenzione il mio nome e cognome non arriva a 100 caratteri
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
/tmp/ipykernel_9389/2238065423.py in <cell line: 1>()
----> 1 nome[100]  #attenzione il mio nome e cognome non arriva a 100 caratteri

IndexError: string index out of range
```

In [4]:
```
1 len(nome) #infatti i caratteri sono 15
```

Out[4]: 15

In [5]:
```
# ok accediamo all quindicesimo
nome[15]
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
/tmp/ipykernel_9389/3779975648.py in <cell line: 2>()
      1 # ok accediamo all quindicesimo
----> 2 nome[15]

IndexError: string index out of range
```

1 😟

carattere	a	n	d	r	e	a		s	t	e	r	b	i	n	i
indice	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

In [6]:
```
start = nome[0]                    # accedo al primo elemento, lo metto dentro s
end = nome[len(nome)-1]            # calcolo la lunghezza della stringa.
                                   # ultimo elemento è in posizione luneghezza-1
                                   # lo metto dentro end
print(start)  #stampa la prima lettera
print(end)    #stampa l'ultima lettera
```

```
a
i
```

In [7]:
```
# Posso invece contare dalla fine
ultimo = nome[-1]
penultimo = nome[-2]
print(ultimo)
print(penultimo)
```

```
i
n
```

In [8]:
```
# possiamo anche "affettare" la stringa
# per estrarre il mio nome (slicing)
nome[0:6] # da notare MOLTO IMPORTANTE
          # 0 e' compreso, 6 escluso
```

Out[8]: 'andrea'

car.	a	n	d	r	e	a
idx	0	1	2	3	4	5

Assumiamo di volere estrarre `drea`
da `andrea sterbini`.

- Come facciamo?
- Ricordiamoci che il carattere spazio nel mezzo **e'** un carattere (lo spazio si codifica nel computer)

carattere	a	n	d	r	e	a		s	t	e	r	b	i	n	i
indice	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

- Possiamo fare slicing con l'operatore **`:`** da 2 a 6 ossia **`[2:6]`** (prendi la fetta che va dall'indice 2 al 6 escluso)

- Ricordatevi `gli indici di slicing` in python sono sempre [start, end)
- `start` incluso, `end` **ESCLUSO**

In [9]:
```
sub_name = nome[2:6]
print(sub_name)
```

```
drea
```

E se non sappiamo dove inizia `drea` , che facciamo?

- Sappiamo che dobbiamo cercare `drea`
- `drea` è lungo 4 caratteri, ma possiamo usare il metodo `len()` per calcolarsi la lunghezza della stringa, cosi funzionerà con tutte le stringhe, non solo con `drea` .
- Come facciamo a trovare l'indice di inizio di `drea` dentro `nome` ?

> Se ci fate caso ci stiamo piano piano approcciando al **problem solving**

In [10]:
```
1 help(str) # conviene leggersi le funzionalita' del tipo str
```

```
Help on class str in module builtins:

class str(object)
 |  str(object='') -> str
 |  str(bytes_or_buffer[, encoding[, errors]]) -> str
 |
 |  Create a new string object from the given object. If encoding or
 |  errors is specified, then the object must expose a data buffer
 |  that will be decoded using the given encoding and error handler.
 |  Otherwise, returns the result of object.__str__() (if defined)
 |  or repr(object).
 |  encoding defaults to sys.getdefaultencoding().
 |  errors defaults to 'strict'.
 |
 |  Methods defined here:
 |
 |  __add__(self, value, /)
 |      Return self+value.
 |
 |  __contains__(self, key, /)
```

```
|  index(...)
|      S.index(sub[, start[, end]]) -> int
|
|      Return the lowest index in S where substring sub is found,
|      such that sub is contained within S[start:end].  Optional
|      arguments start and end are interpreted as in slice notation.
|
```

S.index(sottostringa) -> int

1. prendo la stringa `S` ci applico la funzionalita' `index` che prende `sottostringa` come argomento
2. Rende un intero `-> int` che indica la posizione trovata (oppure -1 se non c'è)
3. Questa e' documentazione non codice

In [13]:
```
# quindi possiamo fare
# nome vale 'andrea sterbini '
start = nome.index('drea')
start
```

Out[13]: False

In [14]:
```
nome[start:start+len('drea')] # [2,2+4)
```

Out[14]: 'drea'

d	r	e	a
2	3	4	5

In [15]:
```
# mettendo tutto insieme
query = 'drea'                          # input dell'algoritmo
start = nome.index(query)               # trovo il primo indice che contiene il cor
                                        # sulla contenuto di NOME
end = start + len(query)                # mi precalcolo indice di fine stringa
sub_name = nome[start:end]              # faccio slicing della stringa
print(sub_name,start,end,nome)          # stampo cosa ho trovato e dove, a partire
```

```
drea 2 6 andrea sterbini
```

d	r	e	a
2	3	4	5

1.0.1 Di più su slicing

- E' possibile anche **non mettere il valore di start e end**
- E' possibile andare **al contrario**
- E' possibile **saltare ogni K elementi**

In [16]:
```
nome[1:] # tutti i caratteri DOPO il
         # primo (0 escluso, 1 compreso)
         # 0 | 1-----------------
```

Out[16]: 'ndrea sterbini'

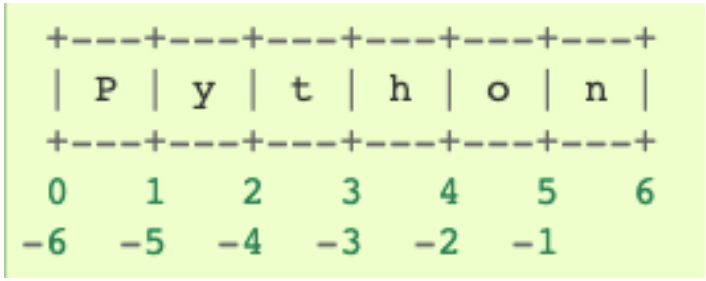
In [17]:
```
nome[:2] # tutti i caratteri PRIMA
         # del indice 2 escluso
         # quindi solo carattere 0-th
         # --0--1--| 2 3 4 5 6 7...
```

Out[17]: 'an'

In [18]:
```
nome[-1]  # al contrario di molti linguaggio a basso livello
          # qui gli indici possono essere negativi
          # il meno inverte la sequenza di lettura
          # sto prendendo da 'andrea sterbini' <--- questa i finale
```

Out[18]: 'i'

- Questa complessità è molto utile e efficace ma ovviamente **va saputa gestire**
- Molto facile ingarbugarsi su `slicing` complessi perchè notazione è macchinosa (imho)



In [19]:
```
nome = 'Python'
vuoto = nome[4:3] # ovviamente se start >= end, abbiamo stringa vuota
type(vuoto),vuoto
```

Out[19]: (str, '')

In [20]:
```
nome = 'Python'
vuoto = nome[-3:-4] # ovviamente se start >= end, abbiamo stringa vuota ''
type(vuoto),vuoto
```

Out[20]: (str, '')

In [22]:
```
nome = 'Python'
nome[-4:-3] == nome[2:3], nome[2:3]
# questo funzione seleziona il carattere 't' (un singolo carattere e' una st
```

Out[22]: (True, 't')

```
+---+---+---+---+---+---+
| P | y | t | h | o | n |
+---+---+---+---+---+---+
 0   1   2   3   4   5   6
-6  -5  -4  -3  -2  -1
```

In [23]:
```
nome = 'Python'
nome[-10:2], nome[3:100]
# se l'inizio è negativo e troppo grande si inizia dal primo carattere
# se la fine della slice è troppo grande si arriva all'ultimo
```

Out[23]: ('Py', 'hon')

In [24]:
```
nome[-1000]
# MA ATTENZIONE: se invece indicizziamo il singolo carattere
# non possiamo chiedere più di -(len(nome)-1)
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
/tmp/ipykernel_9389/137629210.py in <cell line: 1>()
----> 1 nome[-1000]
      2 # MA ATTENZIONE: se invece indicizziamo il singolo carattere
      3 # non possiamo chiedere più di -(len(nome)-1)

IndexError: string index out of range
```

In [25]:
```
# Prendiamo tutti i caratteri in posizione multipla di 3
# il terzo valore indica come incrementare l'indice
nome[::3]
```

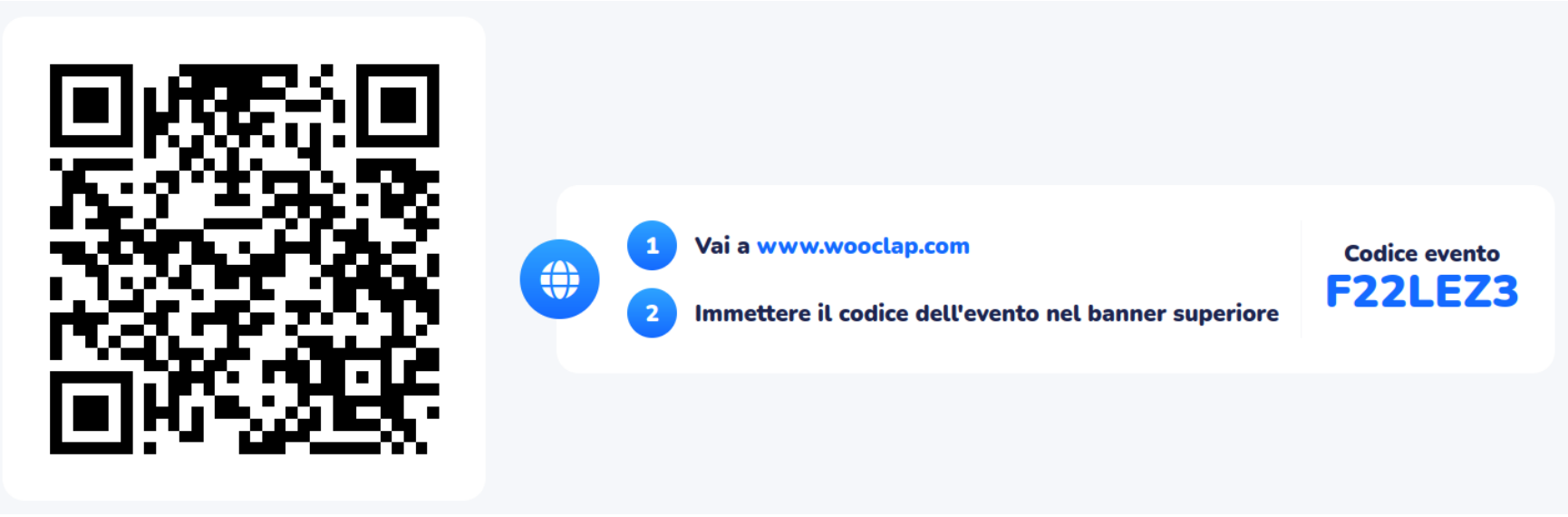
Out[25]: 'Ph'

In [26]:
```
# prendiamoli in direzione opposta (incremento negativo)
nome[::-1]
```

Out[26]:
```
'nohtyP'
```

1.1 Momento Wooclap

1.1.0.1 quanto fa: `'ippopotamo'[2:-2:2]` ?
e invece `'BabbuINO'.lower()[-1:-10:-3]` ?



In [27]:
```
# Risultati
'ippopotamo'[2:-2:2], 'BabbuINO'.lower()[-1:-10:-3]
```

Out[27]:
```
('ppt', 'oua')
```

1.2 Stringhe come Tipo Immutabile

- I tipi **mutabili e immutabili** in python li vediamo meglio nelle prossime lezioni
- Informalmente, un tipo **immutabile** vuol dire che una volta che e' stato creato **NON** può essere più modificato
- Per chi viene dal `C` questa cosa non e' molto chiara perchè in C potete modificare direttamente le stringhe carattere per carattere

In [28]:
```
1 nome = 'Python'
```

assumiamo io voglia modificare la `P` di `Python` in minuscola `python`

In [29]:
```
1 # in C potrei fare
2 nome[0] = 'p'
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
/tmp/ipykernel_9389/2897224970.py in <cell line: 2>()
      1 # in C potrei fare
----> 2 nome[0] = 'p'

TypeError: 'str' object does not support item assignment
```

In realta' non lo posso fare direttamente,
devo per forza creare un'altra stringa

In [30]:
```
1 A = nome.lower()      # trasformo tutto in minnuscole
2 A
```

Out[30]: 'python'

In [31]:
```
1 B = 'p'+nome[1:]      # oppure faccio cut/paste
2 B
```

Out[31]: 'python'

In [32]:
```
# con id(oggetto) posso capire se due cose sono lo stesso oggetto
id(A), id(B), id(A)==id(B)
# le due stringhe sono in posti diversi della memoria ?
```

Out[32]:
```
(139845825377904, 139845815550896, False)
```

In [33]:
```
help(id)
```

```
Help on built-in function id in module builtins:

id(obj, /)
    Return the identity of an object.

    This is guaranteed to be unique among simultaneously existing objects.
    (CPython uses the object's memory address.)
```

1.2.1 String Literals

Se dobbiamo scrivere testi che contengono accapi ('\n') possiamo usare il triplo apice `'''` oppure il triplo doppio apice `"""`

In [36]:
```
print("""
Usage: thingy [OPTIONS]
     -h                        Display this usage message
     -H hostname               Hostname to connect to
""")
```

```
Usage: thingy [OPTIONS]
     -h                        Display this usage message
     -H hostname               Hostname to connect to
```

1.2.2 String interpolation

- precedendo la stringa con la lettera **`f`** (che sta per 'formatted')
- inserendo i valori da interpolare tra parentesi graffe **`{espressione}`**
- se vogliamo, indicando come visualizzare il dato interpolato (vedi documentazione)

In [37]:
```
# definiamo le parti da interpolare nel testo
nome      = 'Paolino'
cognome   = 'Paperino'
indirizzo = 'Via dei Peri 32'
data      = '29 febbraio'
orario    = '20:30'
mittente  = 'Gastone Paperone'
```

In [38]:
```
# e la lettera da spedire (che interpola semplici variabili)
lettera = f'''
Caro {nome} {cognome}
La invito al vernissage che si terrà a {indirizzo}
il giorno {data} alle ore {orario}
Cordialmente
{mittente}
'''
```

In [39]:
```
print(lettera)
```

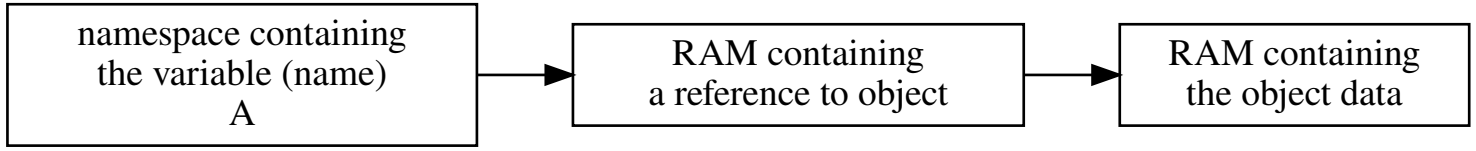
```
Caro Paolino Paperino
La invito al vernissage che si terrà a Via dei Peri 32
il giorno 29 febbraio alle ore 20:30
Cordialmente
Gastone Paperone
```

1.3 Variabili e riferimenti

In [41]:
```
from graphviz import Digraph
# Create Digraph object
figura = Digraph()
figura.body.append('''
    graph [rankdir=LR]
    node [shape=rect]
    "namespace containing\n the variable (name)\nA" -> "RAM containing\n a re
''')
```

In [42]:
```
figura
## Ogni variabile è un nome contenuto in una tabella (namespace)
## che contiene un 'riferimento' all'oggetto in essa contenuto
```

Out[42]:



1.4 Uguaglianza ed identità

Due oggetti possono essere abbastanza simili (se sono dello stesso tipo e contengono le stesse informazioni, sono interscambiabili)

Il confronto **==** torna True (uguaglianza dei contenuti)

Ma possono trovarsi in due aree di memoria diverse e quindi essere oggetti diversi (modificandone uno l'altro resta invariato)

Per distinguerli e sapere in qualche modo dove sono in memoria si usa la funzione **id** e per controllare se sono 'identici' si usa l'operatore **is** (identità)

In [43]:
```
# creiamo una stringa e copiamo la variabile A in B
A = 'Pippo e Pluto sono andati al mare con Minnie e Topolino'
B = A
# Se hanno lo stesso ID, di tratta dello stesso oggetto
print(id(A))
print(id(B))
A is B
```

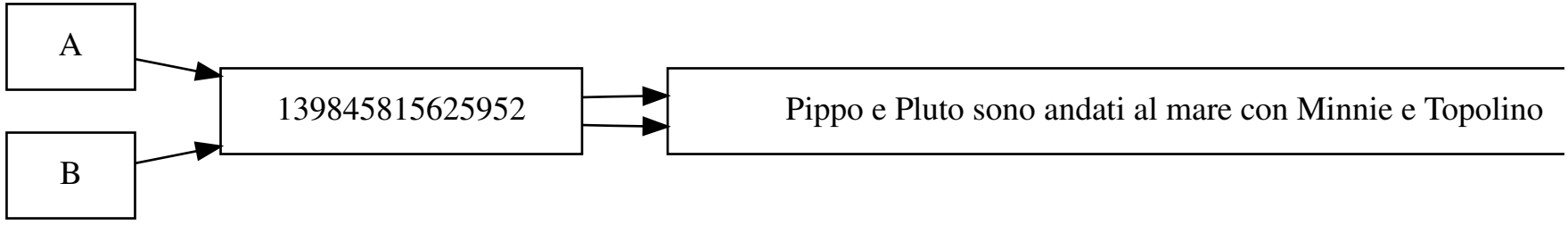
```
139845815625952
139845815625952
```

Out[43]: True

In [45]:
```
from graphviz import Digraph
def show_vars( *variabili ):
    grafo = Digraph()
    grafo.body.append(f'''
graph [rankdir=LR]
node  [shape=rect]''')
    for nome in variabili:
        valore = eval(nome)
        grafo.body.append(f'''
        {nome} -> "{id(valore)}"  -> "{valore}"
        ''')
    return grafo
```

In [46]:
```
show_vars( 'A', 'B' )
```

Out[46]:



1.5 Le variabili sono i nomi che diamo a luoghi nella memoria

e contengono un riferimento (indirizzo) dell'oggetto in memoria

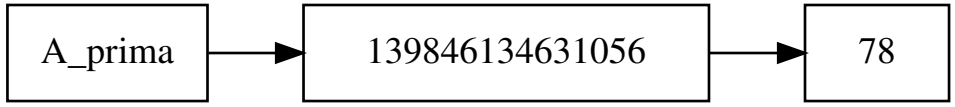
- possiamo **modificare le info contenute in un oggetto** 'riferito' da una variabile
- oppure **sostituire** nella variabile **il riferimento** ad un altro oggetto
  L'operazione di assegnamento CAMBIA IL RIFERIMENTO contenuto nella variabile

In [47]:
```
# esempio: assegnamento che cambia il riferimento
A = 78
A_prima = A
A = 'paperino'
A_dopo  = A
print(id(A_prima))
print(id(A_dopo))
```

```
139846134631056
139845815881648
```

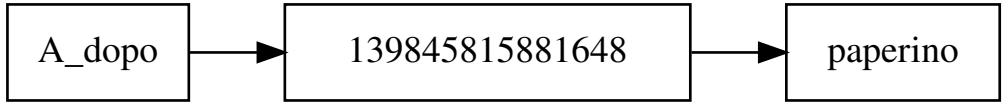
In [48]:
```
show_vars('A_prima')
```

Out[48]:



In [49]:
```
show_vars('A_dopo')
```

Out[49]:

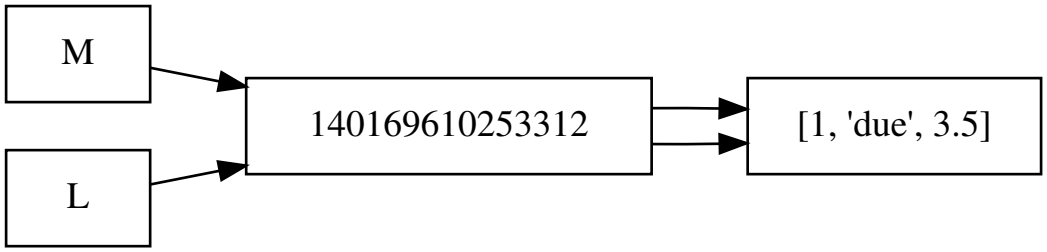


In [50]:
```
# Esempio di un contenitore: una lista di oggetti eterogenei
L = [1, 'due', 3.5]
# ne copio il riferimento in una seconda variabile
M = L
# sono proprio la stessa cosa
print(id(L),id(M), L is M)
```

```
139845815559808 139845815559808 True
```

In [132]:
```
1 show_vars('M', 'L')
```

Out[132]:

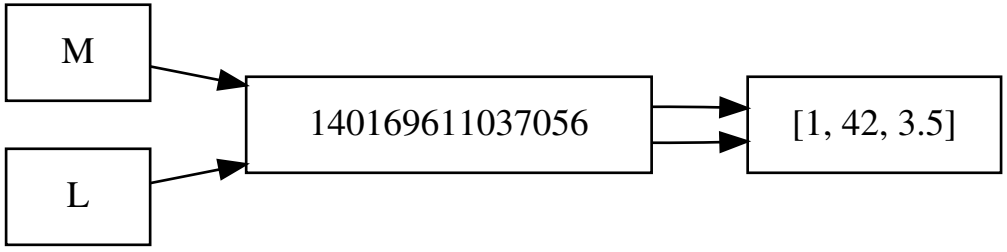


In [51]:
```
1 # modifico il contenuto a posizione 1 (secondo elemento, si conta da 0)
2 L[1] = 42
3 # si tratta sempre dello stesso oggetto sia in L che in M
4 print('identici?', L is M, '\nuguali?', L == M)
5 # ed è stato modificato il secondo elemento
6 print(L, M)
```

```
identici? True
uguali? True
[1, 42, 3.5] [1, 42, 3.5]
```

In [123]:
```
1 show_vars('M', 'L')
```

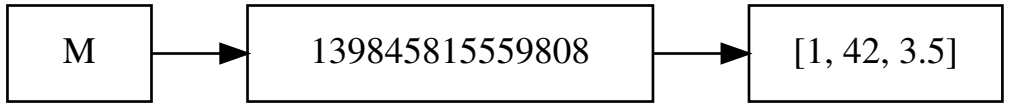
Out[123]:



In [52]:
```
# ma se creo una lista con gli stessi dati
L = [1, 42, 3.5]
print('identici?', L is M, '\nuguali?', L == M)
show_vars('M')
```

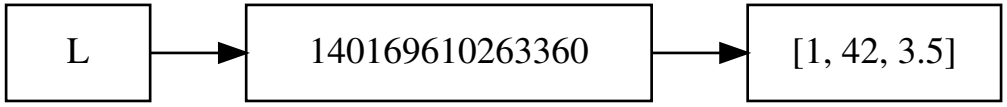
```
identici? False
uguali? True
```

Out[52]:



In [137]:
```
show_vars('L')
```

Out[137]:



2 Tutti i dati in Python sono "oggetti"

- un oggetto è un **gruppo di informazioni coerenti** (`attributi`)
  Es. cane: nome, altezza, razza, peso, colore, ...
  Es. str: lunghezza, sequenza di caratteri, ...

- con tutte le **operazioni che potete eseguire** (`metodi`)
  Es. cane: abbaia, scodinzola, cammina, corre, salta
  Es. str: join, split, upper, lower, startswith, ...

int, str, bool, float, etc... sono oggetti
per scoprirne le caratteristiche usiamo help oppure ctrl-I in Spyder oppure . e l'autocompletamento

In [53]:

```
str                 # testo
int                 # interi                    # abs, mod, %, //
float               # numeri con virgola
complex             # numeri complessi          # conjugate, imag, real, ...
bool                # booleani (True, False)# and, or, not

from fractions import Fraction  # frazioni intere
#help(Fraction)
f = Fraction(124,14)                          # denominator, numerator
f
```

Out[53]:

```
Fraction(62, 7)
```

2.1 I metodi sono le operazioni che possiamo compiere su un oggetto

e che ne possono manipolare il contenuto o crearne di nuovi

Es. le operazioni su str

- join, split, lower, upper, find, ...

2.2 Per eseguire un metodo di un oggetto

si usa la sintassi **`oggetto.nomedelmetodo(argomenti)`**

In [66]:
```
# Esempi
S = 'Paperino andò\t al, mare\n a nuotare'
# split (e varianti)
L = S.split()
# join
'|'.join(L)
S.lower()
# islower, isupper, isalpha, isnumeric
S.islower()
# upper, lower, title
S.upper()
# find
S.find('mare')
```

Out[66]: 19

NOTA: le **stringhe sono IMMUTABILI**, ogni volta che ci fate operazioni sopra ne create una nuova

2.2.1 NOTA: per rendere più efficiente l'uso della memoria

Python, invece che duplicarle, tiene copie uniche dei **numeri piccoli** e delle **stringhe corte**

In [67]:
```
B = 93
C = 92+1
print(id(B), id(C))
B is C
```

```
139846134631536 139846134631536
```

Out[67]: True

In [70]:
```
# per le stringhe l'ottimizzazione è applicata alle 'parole'
# che non contengono spazi (particolarmente importante nella
# ottimizzazione del codice in memoria)
S1 = 'giovanni' * 50
S2 = 'giovanni' * 50
print(id(S1), id(S2))
S1 is S2
```

```
139845825296944 139845825296944
```

Out[70]: True

In [72]:
```
# esempio di ottimizzazione dei piccoli numeri
esponente = 2
print((34**esponente) == ((30+4)**esponente), 'uguaglianza')
print((34**esponente) is ((30+4)**esponente), 'identità')
```

```
True uguaglianza
False identità
```

3 Modello dei dati di Python

3.0.1 DUCK typing: "se cammina come una papera e fa 'quack' come una papera, allora è una papera"

Nel *modello dei dati* del Python vogliamo che due oggetti siano intercambiabili quando 'si comportano allo stesso modo'

'si comportano allo stesso modo' == 'hanno gli stessi metodi'

(MA NOTATE: metodi con nomi uguali potrebbero avere realizzazioni interne diverse)

E' quindi possibile costruire nuovi tipi di oggetti che si comportano in modo simile ad altri

i numeri interi e i float sono un esempio (e anche i razionali e i complessi)

(vedremo in seguito come farlo)

3.1 NOTA: incapsulamento delle informazioni

gli attributi (informazioni interne) degli oggetti in Python NON sono protetti

In altri linguaggi compilati (Java, C++) **è possibile 'nascondere'** le informazioni interne e fare in modo che solo i metodi dell'oggetto le possano manipolare

In Python (interpretato) questo non è possibile (o è stato scelto così)

Ma si è stabilito che per convenzione

TUTTI GLI ATTRIBUTI E METODI CHE INIZIANO CON '_' SONO PRIVATI

e **non vanno letti o modificati o usati direttamente**

(chi li ha inventati potrebbe benissimo modificarli in future versioni del programma)

quindi **NON USATE MAI** i metodi speciali :-)

(vedremo poi come definirli)

Esistono alcuni **metodi speciali** (che iniziano per `__`) che realizzano le funzionalità degli operatori usati nelle espressioni/istruzioni. Per es.

- `__eq__` che implementa l'operatore `==`
- `__add__` che implementa l'operatore `+`
- `__mul__` che implementa l'operatore `*`
- `__len__` che implementa la funzione `len`

Tutti gli oggetti che implementano questi metodi possono essere confrontati, sommati, moltiplicati, contati, ...

Ecco come mai le stringhe possono essere sommate o moltiplicate, hanno un proprio metodo `__add__` e `__mul__`

Questo vi sarà utile quando vedremo come costruire oggetti che possiamo confrontare sommare, o di cui vogliamo calcolare la lunghezza (se contengono elementi)

In [73]:
```
N = 3
N.__add__(5)    # questo è cosa succede veramente se scrivo N + 5
```

Out[73]: 8

In [74]:
```
# i numeri complessi hanno le 4 operazioni come gli altri numeri
# (ma sono implementate diversamente)
X_complesso = 3 + 2j     # si usa j al posto di i=sqrt(-1)
Y_complesso = 7 - 6j
Z_complesso = X_complesso + Y_complesso           # somma complessa
Z_complesso,  Z_complesso.real, Z_complesso.imag
```

Out[74]: ((10-4j), 10.0, -4.0)

4 Funzioni: come definire parti di programma riusabili

- **MAI:** copiare e incollare più volte lo stesso pezzo di codice
- **SEMPRE:** definisci una funzione separata e chiamala quante volte vuoi

4.0.1 Sintassi: (come si scrive)

```
def nome_della_funzione( argomenti ):
    'docstring che descrive cosa fa'
    istruzioni che calcolano il risultato
    return risultato
```

Il corpo della funzione DEVE essere indentato (in genere di 4 spazi)

Gli `argomenti` sono nomi che **indicano le informazioni necessarie** perchè la funzione possa svolgere il suo compito

NOTA: i nomi degli argomenti sono variabili disponibili SOLO nel corpo della funzione (la parte indentata)

4.0.2 ESEMPIO: calcolare l'altezza media di un gruppo

In [75]:
```
# come si definisce la funzione
def media_altezze(altezze):
    'calcolo la media di un gruppo di altezze'    # docstring
    somma_altezze = sum(altezze)
    media_altezze = somma_altezze/len(altezze)
    return media_altezze                          # valore risultante
```

In [76]:
```
## Come la si usa/chiama la funzione
# se abbiamo una lista di altezze
elenco_altezze = [ 170, 190, 165, 155, 186, 172, ]
# e ne calcoliamo la media chiamando la funzione 'media_altezze'
media = media_altezze(elenco_altezze)  # e mettendo il risultato in una varia
# otteniamo
media
```

Out[76]: 173.0

4.0.3 NOTATE CHE

- i nomi che ho usato nella **definizione** della funzione sono `altezze` (**parametri formali**, i dati che la funzione riceve da chi la chiama)
- i nomi che ho usato nel codice che USA la funzione sono `elenco_altezze` (**parametri attuali**, i dati che effettivamente voglio fornire a questa particolare chiamata della funzione)

Stile: è meglio usare nomi diversi per i parametri **formali** e per gli **attuali** per evitare di confondersi

4.0.4 Ancora questioni di stile: identificatori parlanti

- scegliete sempre un **nome di funzione** che vi ricorda bene **cosa fa**

- GOOD: `calcola_altezza_media`
- BAD: `pippo`

- scegliete sempre i **nomi degli argomenti** in modo che sia chiaro **quali informazioni dovete fornire** alla funzione
  - GOOD: `altezze`
  - BAD: `lista`

- scegliete sempre i **nomi delle variabili** in modo che vi sia chiaro **cosa contengono**
  - GOOD: `somma_altezze`
  - BAD: `sa`

4.1 Namespaces: dove stanno i nomi delle variabili e funzioni

- **`globale`** : è il livello più esterno del vostro programma, in cui scrivete le istruzioni direttamente appoggiate al bordo sinistro
  - variabili globali: accessibili dentro a tutte le funzioni e metodi (ALTAMENTE SCONSIGLIATE perchè creano errori difficili da trovare)
  - funzioni globali: accessibili a tutte le altre funzioni del file

- **`modulo`** : è il contenuto di un file importato con `import` , ci trovate le variabili e le funzioni definite in quel file. Le ottenete scrivendo **`modulo.variabile`** oppure **`modulo.funzione(argomenti)`**
- **`locale`** : ogni funzione usa delle variabili proprie, dette variabili **locali**, che sono accessibili SOLO nelle istruzioni del corpo della funzione o di una sua sottofunzione. Gli **argomenti** della funzione sono anche loro variabili locali alla funzione.

ATTENZIONE una variabile locale può *nascondere* una variabile globale

In [77]:

```
variabile = 42                  # definisco una variabile globale

def funzione_di_prova( argomento ):
    variabile = argomento       # qui la ridefinisco come locale
    print(variabile, 'stampo la locale')

print(variabile, 'globale')
funzione_di_prova(92)
print(variabile, 'globale invariata')
funzione_di_prova(667)
```

```
42 globale
92 stampo la locale
42 globale invariata
667 stampo la locale
```

4.2 Chiamata delle funzioni e vita delle variabili locali

Le variabili locali ad una funzione vengono **CREATE** nel momento in cui la funzione viene **ESEGUITA** (e non quando viene definita)

Ogni esecuzione crea un **nuovo gruppo di variabili locali** della funzione

Per cui le funzioni possono richiamare se stesse senza problemi, **ciascuna chiamata userà uno spazio di memoria personale** separato dagli spazi delle altre chiamate

Le **variabili locali** sono **rilasciate quando si esce dalla funzione** e si torna al programma che l'ha chiamata

NOTA: la memoria che viene **rilasciata** è quella dei RIFERIMENTI.
Gli oggetti creati dalla funzione possono sopravviverle

MA QUESTO FA SPRECARE UN SACCO DI MEMORIA ???

NO, esiste un processo in background, il **garbage collector** che esamina la memoria continuamente e libera tutti gli oggetti che **non sono più raggiungibili** dai programmi in esecuzione

In [80]:
```
## Esempio di creazione di un oggetto dentro una funzione
def concatena(prima_parte, seguito):
    # costruisco una nuova stringa
    nuovo_testo = prima_parte + ' ' + seguito
    return nuovo_testo    # e la ritorno
A = 'inizio '
B = 'fine'
C = concatena(A, B)
print(A, B, C, sep='\n')
print(prima_parte)
# errore perchè 'nuovo_testo' non esiste nel namespace globale
```

```
inizio
fine
inizio  fine

---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
/tmp/ipykernel_9389/2666591040.py in <cell line: 10>()
      8 C = concatena(A, B)
      9 print(A, B, C, sep='\n')
---> 10 print(prima_parte)
     11 # errore perchè 'nuovo_testo' non esiste nel namespace globale

NameError: name 'prima_parte' is not defined
```

4.3 Ancora booleani: come scegliere alternative diverse

- gli operatori di confronto sono le espressioni booleane più semplici
- potete combinarli con gli operatori **`and`**, **`or`** e **`not`**
- **`not`** ha priorità su **`and`** che l'ha su **`or`**
- usate le parentesi altrimenti

E' quindi possibile scrivere condizioni complesse

```
piove = not soleggiato or strade_bagnate
porto_l_ombrello = piove or sole_abbacinante
strade_bagnate = ha_piovuto or innaffiamento_automatico_acceso
```

4.4 Condizioni e percorsi alternativi

Per scegliere parti diverse di codice si usa l'istruzione **`if`** e delle condizioni (booleane) che valgono **True** o **False**

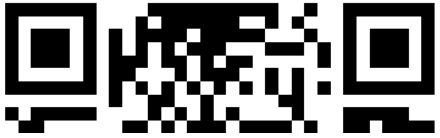
Sintassi:

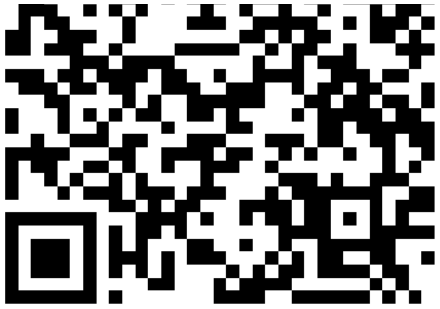
```
if condizione1:
    istruzioni eseguite
    se condizione1 è True
elif condizione2:          # opzionale
    istruzioni eseguite
    se condizione1 è False e condizione2 è True
else:                      # opzionale
    istruzioni eseguite
    se condizione1 e condizione2 sono False
```

4.5 Momento Wooclap

4.5.0.1 Quanto valgono queste espressioni booleane ?

4.5.0.2 Cosa stampa il programma ?





In [81]:
```
## Soluzioni
True and not False or True and False or not True
# è lo stesso che scrivere
(True and (not False) ) or (True and False) or (not True)
# cioè
(True and True) or False or False
```

Out[81]: True

In [ ]:
```
# Visto che True == 1  e che False == 0
True+False/True+True*True-False
# è lo stesso che
1 +  0  /  1 +  1 *  1 -  0
```

In [ ]:
```
# soluzione
is_raining   = True
got_umbrella = False
have_money   = True;
if not is_raining:
    print('go out')
elif not got_umbrella and have_money:
    print('buy umbrella and go out')
elif got_umbrella:
    print('go out since got umbrella')
else:
    print('stay home')
```