Laboratorio di programmazione

Python

A.A. 2020-2021

Informazioni

Docenti

- Erik Romelli erik.romelli@inaf.it
- Daniele Tavagnacco daniele.tavagnacco@inaf.it

Informazioni

Docenti

- Erik Romelli erik.romelli@inaf.it
- Daniele Tavagnacco daniele.tavagnacco@inaf.it

Ricevimento

Lavoriamo presso INAF-Osservatorio Astronomico Trieste quindi non abbiamo un ufficio in Università

- appuntamenti da concordare tramite email
- “ricevimento” tramite email o Teams

Lezioni

Gruppo 1 (martedì ore 16-18) → Erik Romelli

Gruppo 2 (giovedì ore 16-18) → Daniele Tavagnacco

Le lezioni sono tenute in remoto tramite Teams

Lezioni

Gruppo 1 (martedì ore 16-18) → Erik Romelli

Gruppo 2 (giovedì ore 16-18) → Daniele Tavagnacco

Le lezioni sono tenute in remoto tramite Teams

Frequenza

E' necessario frequentare almeno il 75% delle lezioni

Lezioni

Gruppo 1 (martedì ore 16-18) → Erik Romelli

Gruppo 2 (giovedì ore 16-18) → Daniele Tavagnacco

Le lezioni sono tenute in remoto tramite Teams

Frequenza

E' necessario frequentare almeno il 75% delle lezioni

Esame

L'esame "scritto" consiste in:

- un questionario da svolgere al computer
- la presentazione e discussione di un piccolo progetto software

---

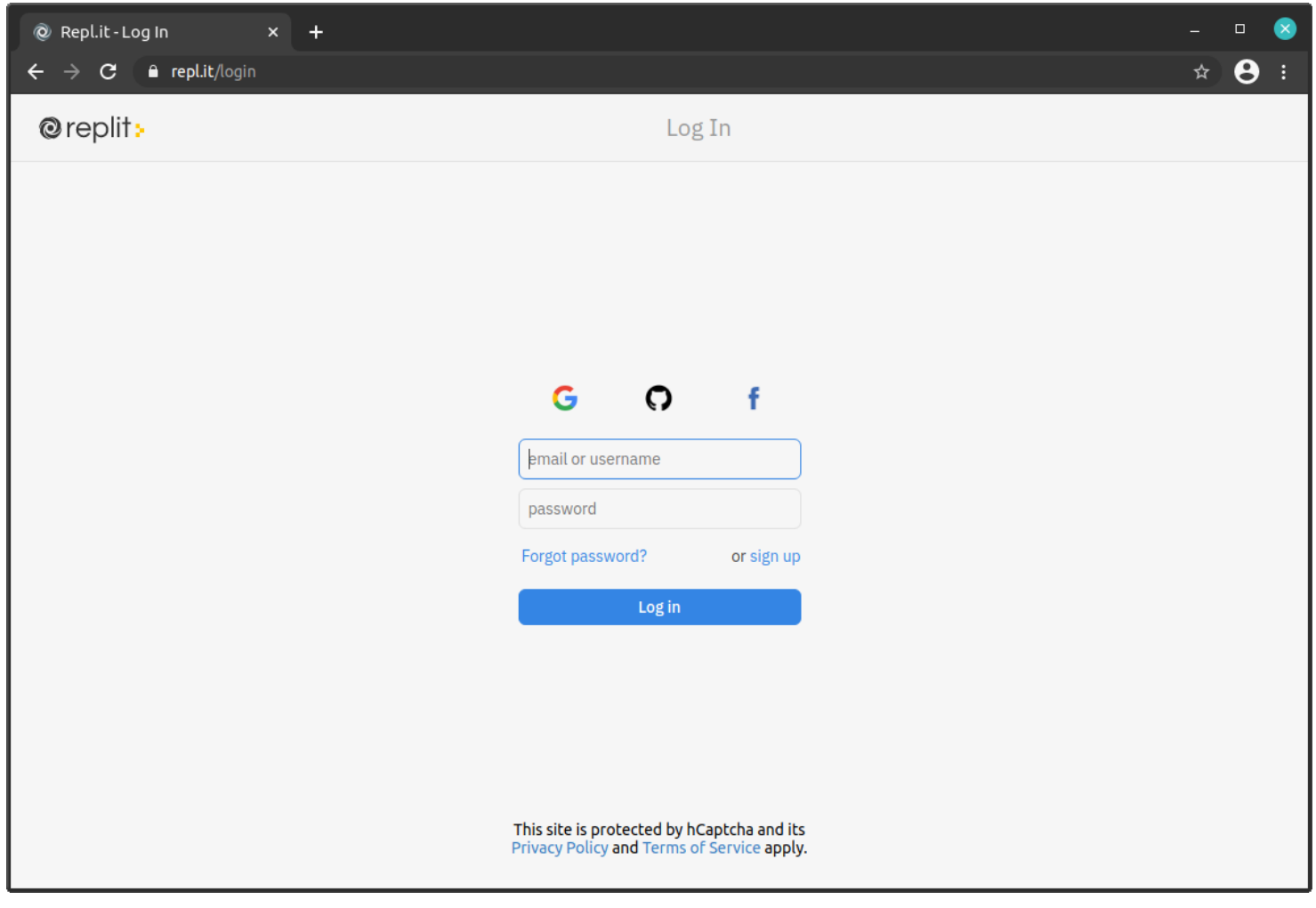
Strumenti

Per il corso è necessario:

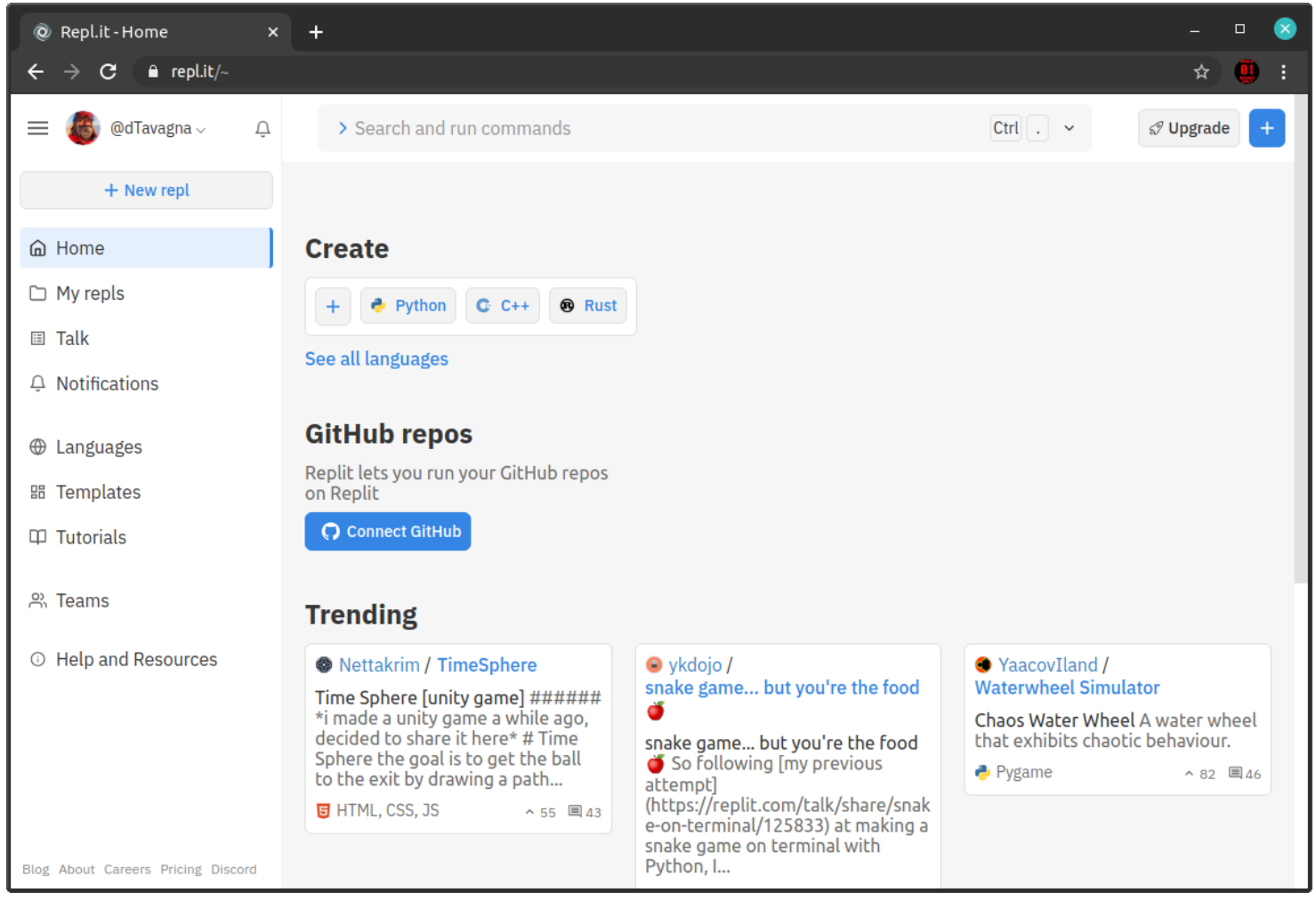
- un account UniTs
- un account sulla piattaforma https://repl.it
- piattaforma MS Teams (se in remoto)

Il laboratorio utilizzerà la piattaforma online *Repl*. Non è necessaria l'installazione di alcun software per frequentare il laboratorio.

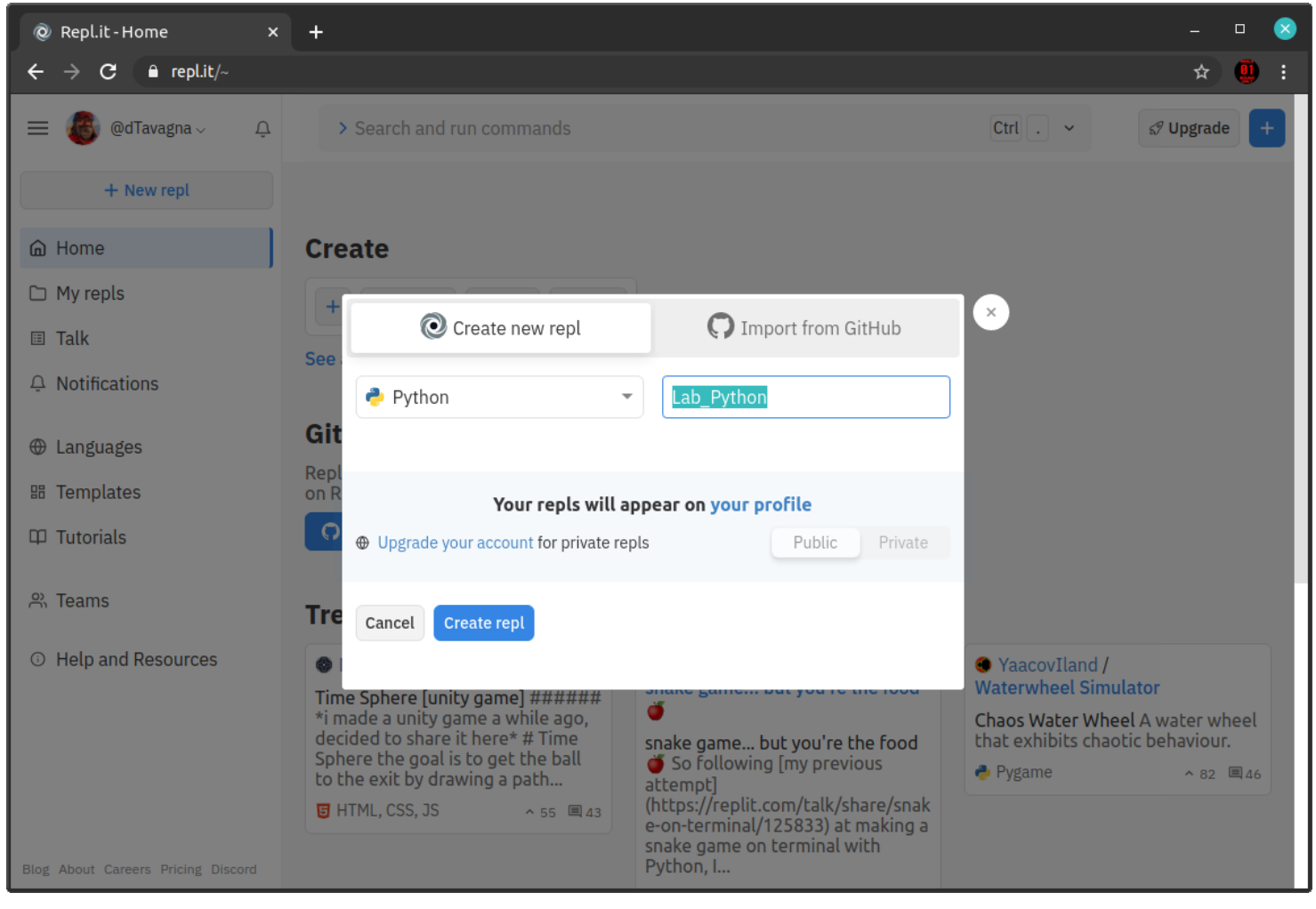
www.repl.it


Repl.it - Log In
repl.it/login
replit
Log In
email or username
password
Forgot password?
or sign up
Log in
This site is protected by hCaptcha and its
Privacy Policy and Terms of Service apply.

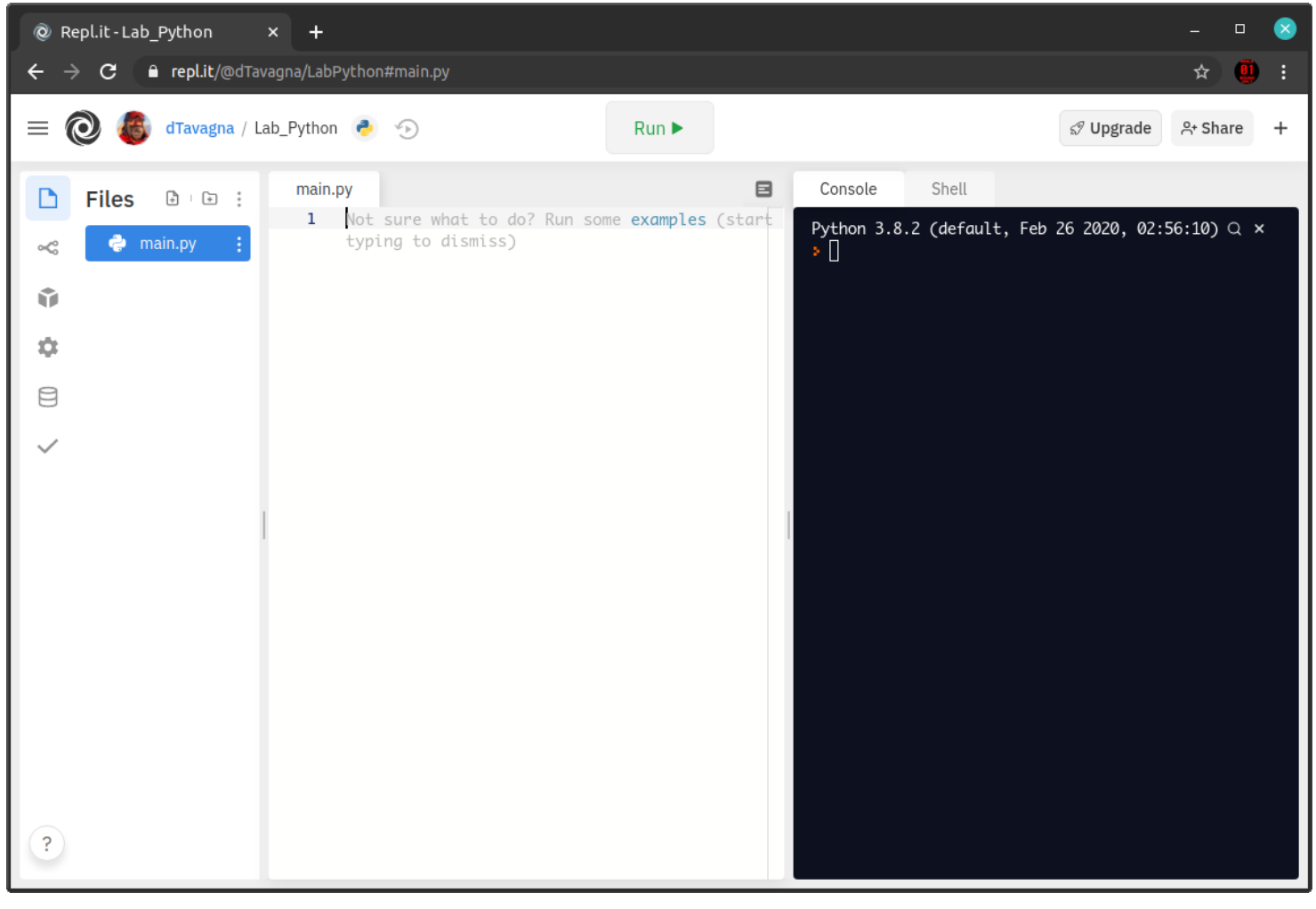
repl home


Repl.it - Home
repl.it/~
@dTavagna
Search and run commands
Ctrl .
Upgrade
+ New repl
Home
My repls
Talk
Notifications
Languages
Templates
Tutorials
Teams
Help and Resources
Blog About Careers Pricing Discord
Create
Python
C++
Rust
See all languages
GitHub repos
Replit lets you run your GitHub repos on Replit
Connect GitHub
Trending
Nettakrim / TimeSphere
Time Sphere [unity game] ######
i made a unity game a while ago, decided to share it here # Time Sphere the goal is to get the ball to the exit by drawing a path...
HTML, CSS, JS
55
43
ykdojo /
snake game... but you're the food
snake game... but you're the food So following [my previous attempt](https://replit.com/talk/share/snake-on-terminal/125833) at making a snake game on terminal with Python, I...
YaacovIland /
Waterwheel Simulator
Chaos Water Wheel A water wheel that exhibits chaotic behaviour.
Pygame
82
46

New Python Project



Python Project



Se volete approfondire: Documentazione Repl

Altri *link* di approfondimento:

- Python
- Think Python 2nd Edition

Cosa è Python

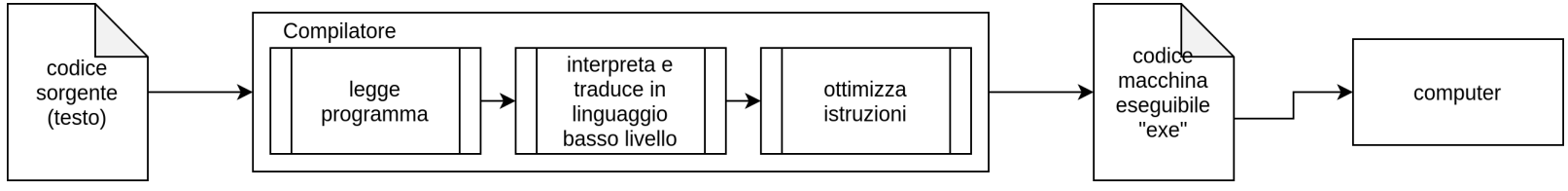
Partiamo da qualche definizione:

> *Python è un **linguaggio** di programmazione ad **alto livello**,... [...]È un linguaggio **multi-paradigma** che ha tra i principali obiettivi: dinamicità, semplicità e flessibilità. Supporta il **paradigma object oriented**, la **programmazione strutturata** e molte caratteristiche di **programmazione funzionale**... (Wikipedia)*
>
> *[...]Python è un linguaggio di programmazione **interpretato**, **interattivo**, orientato agli oggetti. Include moduli, eccezioni, tipizzazione dinamica... (Wikipedia)*

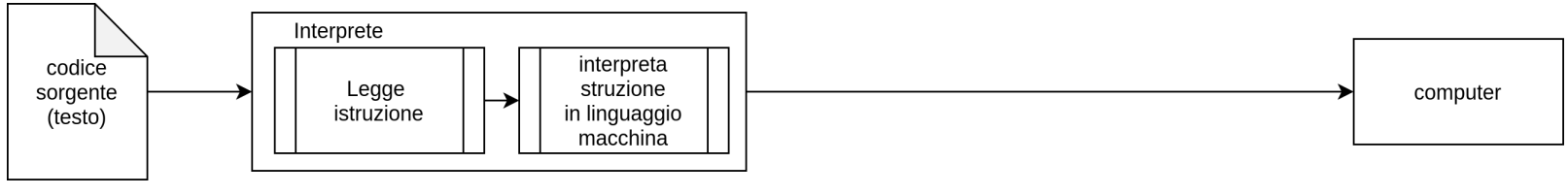
Linguaggio di alto livello

- I linguaggi di **alto livello** sono vicini al ragionamento umano, sono astratti rispetto al **linguaggio macchina** e rispetto ai dettagli del funzionamento di un computer
- **Semplificano** attività del programmatore, **automatizzano dettagli** come allocazione della memoria e rendono disponibili i costrutti per la **programmazione strutturata** come elementi di base della sintassi.
- Per eseguire un programma scritto con questi linguaggi è necessario un **compilatore** oppure un **interprete**: un programma apposito per tradurre il linguaggio di programmazione in linguaggio macchina.

Linguaggio compilato



Linguaggio interpretato



Linguaggio

E' un **sistema di comunicazione strutturato**, composto da un **sistema** definito di suoni o gesti o **simboli** di significato comune ad uno specifico ambiente di interazione: **un codice**

Linguaggio

E' un **sistema di comunicazione strutturato**, composto da un **sistema** definito di suoni o gesti o **simboli** di significato comune ad uno specifico ambiente di interazione: **un codice**

Linguaggio Naturale

Forma di comunicazione usata tra persone che parlano la stessa lingua.
Evolve naturalmente, è ambiguo e significato di una frase dipende dal contesto.
Per questo è verboso e ridondante.
Ha regole precise ma non severe per la sintassi: **mesaggio si capisce anche se struttura non corretta**

Linguaggio

E' un **sistema di comunicazione strutturato**, composto da un **sistema** definito di suoni o gesti o **simboli** di significato comune ad uno specifico ambiente di interazione: **un codice**

Linguaggio Naturale

Forma di comunicazione usata tra persone che parlano la stessa lingua.
Evolve naturalmente, è ambiguo e significato di una frase dipende dal contesto.
Per questo è verboso e ridondante.
Ha regole precise ma non severe per la sintassi: **mesaggio si capisce anche se struttura non corretta**

Linguaggio Formale

E' linguaggio progettato per svolgere un ruolo preciso (es. matematica).
E' composto da un set di parole che definiscono il suo alfabeto.
Raramente è ambiguo, è conciso e letterale: il significato è indipendente dal contesto.
Le regole di sintassi sono precise e molto severe: **se struttura non corretta, il messaggio non può essere compreso**

Un **linguaggio di programmazione** è un linguaggio **formale** progettato per **descrivere delle operazioni di calcolo**

La sintassi

E' la struttura delle istruzioni e l'insieme delle regole (grammatica) con cui le istruzioni vanno costruite.

L'italiano ha regole precise per la struttura di una frase e la sua grammatica:

- Si iniziano le frasi con la maiuscola e si terminano col punto.
- Non si mette la virgola tra soggetto e predicato.
- Per non parlare dei congiuntivi...

Queste frasi contengono tutte errori di sintassi:

1. questa inizia con la minuscola.
2. Questa non ha il punto
3. Questa frase, è un insulto alla Lingua Italiana.
4. Ah, se avrei un euro per ogni errore che vedremo nel corso.

Nel caso di un linguaggio naturale, come l'Italiano, gli errori di sintassi non ~~sono un problema per~~ impediscono di comunicare.

La sintassi

Un linguaggio di programmazione è **estremamente rigido!**
Per essere eseguita, l'istruzione (**statement**) deve essere interpretata (**parsing**) e compresa.
Per comprendere l'istruzione, la struttura **deve** essere corretta e contenere solo parole (**token**) che fanno parte dell'alfabeto del linguaggio.

La sintassi

Un linguaggio di programmazione è **estremamente rigido!**
Per essere eseguita, l'istruzione (**statement**) deve essere interpretata (**parsing**) e compresa.
Per comprendere l'istruzione, la struttura **deve** essere corretta e contenere solo parole (**token**) che fanno parte dell'alfabeto del linguaggio.

In matematica il comando

```
3 + 2
```

ha **struttura corretta** e **token corretti**

La sintassi

Un linguaggio di programmazione è **estremamente rigido!**
Per essere eseguita, l'istruzione (**statement**) deve essere interpretata (**parsing**) e compresa.
Per comprendere l'istruzione, la struttura **deve** essere corretta e contenere solo parole (**token**) che fanno parte dell'alfabeto del linguaggio.

In matematica il comando

```
3 + 2
```

ha **struttura corretta** e **token corretti**

---

mentre il comando

```
3 + ?2&
```

anche se ha **struttura corretta**, contiene degli elementi **token NON corretti**

La sintassi

Un linguaggio di programmazione è **estremamente rigido!**
Per essere eseguita, l'istruzione (**statement**) deve essere interpretata (**parsing**) e compresa.
Per comprendere l'istruzione, la struttura **deve** essere corretta e contenere solo parole (**token**) che fanno parte dell'alfabeto del linguaggio.

In matematica il comando

```
3 + 2
```

ha **struttura corretta** e **token corretti**

---

mentre il comando

```
3 + ?2&
```

anche se ha **struttura corretta**, contiene degli elementi **token NON corretti**

---

oppure nel comando

```
+ 3  2
```

la **struttura NON corretta** impedisce la comprensione del messaggio fatto di **token corretti**

La semantica

La semantica è il **significato** di una istruzione la cui sintassi è corretta.

Nel linguaggio naturale il significato di una frase è legato al contesto.
Se dico "acqua in bocca", cosa significa?

Nel **linguaggio formale il significato è indipendende dal contesto**.
Analizziamo questa battuta da Nerd (del resto imparare a programmare serve anche per capire meglio le battute da programmatore):

La semantica

La semantica è il **significato** di una istruzione la cui sintassi è corretta.

Nel linguaggio naturale il significato di una frase è legato al contesto.
Se dico "acqua in bocca", cosa significa?

Nel **linguaggio formale il significato è indipendende dal contesto**.
Analizziamo questa battuta da Nerd (del resto imparare a programmare serve anche per capire meglio le battute da programmatore):

> *Mamma: Pierino, vai a prendere 1 bottiglia di latte. Se hanno le uova, prendi**ne** 6.*
>
> *Pierino torna con 6 bottiglie di latte*
>
> *Mamma: Pierino! Perché cacchio di divolo hai preso 6 bottiglie di latte?*
>
> *Pierino: Perché avevano le uova!*

Quello che avrei voluto dire:

```
compro 1 Latte
se hanno Uova:
    compro 6 Uova
```

Quello che avrei voluto dire:

```
compro 1 Latte
se hanno Uova:
    compro 6 Uova
```

Quello che ho detto con la sintassi scelta nell'istruzione:

```
(implicito) se NON hanno Uova:
    compro 1 Latte
se hanno Uova:
    compro 6 Latte
```

Esercizi

1. Scrivete una frase con una semantica corretta ma una sintassi errata.

2. Scrivete una frase con una sintassi corretta ma una semantica errata.

Python, le basi

Python è un linguaggio di scripring interpretato e può funzionare in 2 modi:

- interattivamente, scrivendo direttamente le istruzioni all'interprete
- passando una lista di istruzioni (*script*) in formato file di testo all'interprete

Esistono diverse versioni di Python che introducono migliorie al linguaggio.

Le versioni sono tutte compatibili, tranne il passaggio tra versione 2.7 e 3.x.

Sebbene la versione 2.7 di Python sia ancora molto diffusa, è ormai obsoleta.

Per il corso utilizzeremo la **versione 3.8** dell'interprete

---

Avviare l'interprete

In *Repl*, nel progetto Python toviamo un file di testo al centro e 2 *tabs* lateriali.

Nella tab **Console**, trovate già l'interprete avviato

```
Python 3.8.2 (default, Feb 26 2020, 02:56:10)
>
```

Il *prompt* è caratterizzato dal simbolo `>`

Nella tab **Shell** possiamo avviare a mano l'interptete con il comando **python** (o **python3**).

In questo caso otterrete la classica risposta di avvio di Python:

```
Python 3.8.7 (default, Dec 21 2020, 20:10:35)
     [GCC 7.5.0] on linux
     Type "help", "copyright", "credits" or "license" for more information.
     >>>
```

Il *prompt* dell'interprete è caratterizzato dal simbolo `>>>`

Esercizi

1. Nella *Console*, scrivete all'interprete Python le seguenti espressioni:

```
3 + 2
```

```
3 + ?2&
```

```
+ 3   2
```

Errori

Python ha diversi tipi di errore, ma tutti fanno parte di 3 categorie:

- Errori di sintassi (*Syntax Error*)
- Errori durante l'esecuzione (*Runtime Error*)
- Errori semantici (*Semantic Error*)

Syntax errors

Python esegue l'istruzione **solo se la sintassi è corretta**, altrimenti torna un messaggio di errore.

Syntax errors

Python esegue l'istruzione **solo se la sintassi è corretta**, altrimenti torna un messaggio di errore.

In [1]:
```
3 + ?4£
```

```
  File "<ipython-input-1-f1b35881b638>", line 1
    3 + ?4£
        ^
SyntaxError: invalid syntax
```

Runtime errors

Errori che avvengono **durante l'esecuzione dell'istruzione**, sebbene la sintassi sia corretta.

Sono chiamati anche **exceptions**: significa che si sta verificando qualcosa di "eccezionale" (e molto probabilmente sbagliato!)

Hanno diversi nomi a seconda del *tipo di eccezione* e possono anche essere gestiti dal programma senza impedirne il funzionamento

Runtime errors

Errori che avvengono **durante l'esecuzione dell'istruzione**, sebbene la sintassi sia corretta.

Sono chiamati anche **exceptions**: significa che si sta verificando qualcosa di "eccezionale" (e molto probabilmente sbagliato!)

Hanno diversi nomi a seconda del *tipo di eccezione* e possono anche essere gestiti dal programma senza impedirne il funzionamento

In [7]:
```
5 / 0
```

```
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
<ipython-input-7-adafc2937013> in <module>
----> 1 5 / 0

ZeroDivisionError: division by zero
```

Semantic errors

Gli errori semantici sono i più subdoli perché **non impediscono l'esecuzione del programma**:

- non essendo errori di sintassi, il programma verrà correttamente interpretato
- non ci saranno eccezioni durante il runtime

Noteremo solo che il risultato non è quello atteso.

Quello che sta succedendo è che il programma esegue **esattamente** quello che gli è stato detto:
il problema è che **il flusso logico delle istruzioni scritte non corrisponde a quello che PENSATE che il programma debba fare**.

In pratica: *vorreste che il vostro programma comprasse 6 uova e vi ritrovate con 6 cartoni di latte*.

Cosa è un programma?

Un programma **è un insieme di istruzioni ordinate** che specifica come fare un calcolo

I tipi di istruzioni, in qualunque tipo di programma o di linguaggio di programmazione sono:

- **input** prendere dati da una sorgente (tastiera, file, device, ...)
- **calcolo** effettuare operazione matematiche base sui dati
- **controllo condizione** controllare il verificarsi di una condizione e scegliere quale istruzione eseguire
- **ripetizione** eseguire più volte istruzioni con o senza modifiche
- **output** consegnare il risultato ad una destinazione (schermo, file, device,..)

Ogni programma che avete mai usato o userete e scriverete, non importa quanto sia complesso, è e sarà fatto da queste istruzioni

Statements, Commenti, Keywords, Variabili, Espressioni

Statement

Sono le istruzioni che l'interprete Python è in grado di eseguire.

Quando si scrive uno statement sulla linea di comando, Python lo esegue:

- lo statement è scomposto in token durante il parsing
- lo statement è eseguito dall'interprete

Gli statement non producono alcun risultato

Ad esempio si può fare uno statement di assegnazione con il token `=`

Statement

Sono le istruzioni che l'interprete Python è in grado di eseguire.

Quando si scrive uno statement sulla linea di comando, Python lo esegue:

- lo statement è scomposto in token durante il parsing
- lo statement è eseguito dall'interprete

Gli statement non producono alcun risultato

Ad esempio si può fare uno statement di assegnazione con il token `=`

In [1]:
```
a = 3 + 2
```

Statement multilinea

La fine di uno statement è segnalata del carattere *newline*, cioè quando si va a capo e la linea finisce.

E' possibile estendere uno statement su più linee con il carattere di continuazione di linea `\` in questo modo

Statement multilinea

La fine di uno statement è segnalata del carattere *newline*, cioè quando si va a capo e la linea finisce.

E' possibile estendere uno statement su più linee con il carattere di continuazione di linea `\` in questo modo

In [2]:
```
a = 1 + 2 + \
    3 + 4 + \
    5 + 6
```

In questo caso la continuazione di linea è *esplicita*, ma esistono anche delle continuazioni di linea *implicite* date ad esempio dai token parentesi `( )`, `[]`, `{ }`

In questo caso la continuazione di linea è *esplicita*, ma esistono anche delle continuazioni di linea *implicite* date ad esempio dai token parentesi `( )`, `[]`, `{ }`

In [3]:
```
a = ( 1 + 2 +
      3 + 4 +
      5 + 6 )
```

In questo caso la continuazione di linea è *esplicita*, ma esistono anche delle continuazioni di linea *implicite* date ad esempio dai token parentesi `( )`, `[]`, `{ }`

In [3]:
```
a = ( 1 + 2 +
      3 + 4 +
      5 + 6 )
```

E' anche possibile avere più *statements* nella stessa riga separandoli con il carattere `;`
...Ma è meglio **non farlo**

In questo caso la continuazione di linea è *esplicita*, ma esistono anche delle continuazioni di linea *implicite* date ad esempio dai token parentesi `( )`, `[]`, `{ }`

In [3]:
```
a = ( 1 + 2 +
      3 + 4 +
      5 + 6 )
```

E' anche possibile avere più *statements* nella stessa riga separandoli con il carattere `;`
...Ma è meglio **non farlo**

In [4]:
```
a = 3; b = 4
```

Esistono altri statement oltre all'assegnazione (`=`) , come `while` , `for` , `if` , `import` ,...

Vedremo molti di questi statement più avanti, ma possiamo cominciare a familiarizzare con `import`

Import

`import` è lo statement che permette di **importare** nel progamma le **istruzioni** scritte in un altro file

Import

`import` è lo statement che permette di **importare** nel progamma le **istruzioni** scritte in un altro file

Come primo **import statement**, scriviamo nel prompt dell'interprete Python il comando

```
import this
```

..che è un simpatico *Easter Egg* che ci ricorda come andrebbe scritto *un buon programma Python*

In [1]:

```
import this
```

```
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

Commenti

I commenti **sono una parte** molto importante della scrittura **del programma**.

Hanno lo scopo di **descrivere quello che sta succedendo** a chi legge il codice sorgente.

Quella persona sarete voi tra 1 mese: sicuramente avrete dimenticato i passaggi chiave e i ragionamenti che vi hanno portato a scrivere quelle istruzioni.

Capire cosa fa(ceva) il vostro programma senza l'aiuto dei commenti sarà complicatissimo.

Utilizzare del tempo per scrivere buoni commenti che spiegano i concetti e cosa sta succedendo è *sempre* tempo ben speso

In Python i commenti cominciano sempre con il carattere *hash* `#` e terminano con *newline*.

L'interprete Python ignora i commenti quando li incontra nel flusso del programma

In Python i commenti cominciano sempre con il carattere *hash* `#` e terminano con *newline*.

L'interprete Python ignora i commenti quando li incontra nel flusso del programma

In [6]:
```
# sommo 5 e 2
5+2
```

Out[6]:
```
7
```

Commento multilinea

Non esiste un commento multilinea in Python.

Si può usare il carattere `#` all'inizio di ogni linea per ottenere lo stesso effetto.

Un altro modo è quello di usare i tripli apici (**triple quote**) nella loro forma singola `' ' '` oppure doppia `" " "` che servono a creare una *stringa* su più linee ma possono anche essere riutilizzati per i commenti.

I **commenti non generano codice** e sono sempre ignorati dall'interprete, a meno che non siano *docstrings*, ma lo vedremo più avanti

Commento multilinea

Non esiste un commento multilinea in Python.

Si può usare il carattere `#` all'inizio di ogni linea per ottenere lo stesso effetto.

Un altro modo è quello di usare i tripli apici (**triple quote**) nella loro forma singola `' ' '` oppure doppia `" " "` che servono a creare una *stringa* su più linee ma possono anche essere riutilizzati per i commenti.

I **commenti non generano codice** e sono sempre ignorati dall'interprete, a meno che non siano *docstrings*, ma lo vedremo più avanti

In [5]:
```
# commento
# su più linee
4*2
```

Out[5]:
```
8
```

Commento multilinea

Non esiste un commento multilinea in Python.

Si può usare il carattere `#` all'inizio di ogni linea per ottenere lo stesso effetto.

Un altro modo è quello di usare i tripli apici (**triple quote**) nella loro forma singola `' ' '` oppure doppia `" " "` che servono a creare una *stringa* su più linee ma possono anche essere riutilizzati per i commenti.

I **commenti non generano codice** e sono sempre ignorati dall'interprete, a meno che non siano *docstrings*, ma lo vedremo più avanti

In [5]:
```
# commento
# su più linee
4*2
```

Out[5]: 8

In [1]:
```
'''
commento
su più linee
'''
4*2
```

Out[1]: 8

Keywords

Le **keywords** sono **parole riservate** in Python che **servono a definire la sintassi e la struttura del linguaggio Python**

- Non è possibile utilizzare keywords per altri scopi (nomi di variabili)
- Le keyword sono **case sensitive** cioè maiuscolo o minuscolo è diverso
- In Python 3.8 ci sono 35 keyword
- Tranne `True`, `False` e `None` sono tutte scritte in *minuscolo*

Keywords

Le **keywords** sono **parole riservate** in Python che **servono a definire la sintassi e la struttura del linguaggio Python**

- Non è possibile utilizzare keywords per altri scopi (nomi di variabili)
- Le keyword sono **case sensitive** cioè maiuscolo o minuscolo è diverso
- In Python 3.8 ci sono 35 keyword
- Tranne `True`, `False` e `None` sono tutte scritte in *minuscolo*

In [6]:
```
True = 7 # assegno un valore ad una keyword
```

```
  File "<ipython-input-6-7e9ed01ddf2d>", line 1
    True = 7 # assegno un valore ad una keyword
    ^
SyntaxError: cannot assign to True
```

Potete controllare in qualunque momento la lista di keyword della versione di Python che state utilizzando:

Potete controllare in qualunque momento la lista di keyword della versione di Python che state utilizzando:

In [2]:
```
import keyword
print(keyword.kwlist)
```

```
['False', 'None', 'True', '__peg_parser__', 'and', 'as', 'assert', 'async', 'await', 'break', 'class', 'continue', 'def', 'del', 'elif', 'else', 'except', 'finally', 'for', 'from', 'global', 'if', 'import', 'in', 'is', 'lambda', 'nonlocal', 'not', 'or', 'pass', 'raise', 'return', 'try', 'while', 'with', 'yield']
```

Potete controllare in qualunque momento la lista di keyword della versione di Python che state utilizzando:

In [2]:
```
import keyword
print(keyword.kwlist)
```

```
['False', 'None', 'True', '__peg_parser__', 'and', 'as', 'assert', 'async', 'awai
t', 'break', 'class', 'continue', 'def', 'del', 'elif', 'else', 'except', 'finall
y', 'for', 'from', 'global', 'if', 'import', 'in', 'is', 'lambda', 'nonlocal', 'n
ot', 'or', 'pass', 'raise', 'return', 'try', 'while', 'with', 'yield']
```

> `print()` *è una* ***funzione*** *built-in di Python per visualizzare su schermo*

Valori

Un **valore** è il dato che il programma usa nelle operazioni.

Ad esempio il numero **4** o l'insieme di caratteri **'ciao'** sono valori.

I *valori* hanno **tipo** diverso e si possono raggruppare in **classi**.

Per sapere il tipo di un valore, Python ha la funzione *built-in*: `type()`

Valori

Un **valore** è il dato che il programma usa nelle operazioni.

Ad esempio il numero **4** o l'insieme di caratteri **'ciao'** sono valori.

I *valori* hanno **tipo** diverso e si possono raggruppare in **classi**.

Per sapere il tipo di un valore, Python ha la funzione *built-in*: `type()`

In [8]:
```
type(4)
```

Out[8]:
```
int
```

Valori

Un **valore** è il dato che il programma usa nelle operazioni.

Ad esempio il numero **4** o l'insieme di caratteri **'ciao'** sono valori.

I *valori* hanno **tipo** diverso e si possono raggruppare in **classi**.

Per sapere il tipo di un valore, Python ha la funzione *built-in*: `type()`

In [8]:
```
type(4)
```

Out[8]:
```
int
```

In [9]:
```
type('ciao')
```

Out[9]:
```
str
```

Valori

Un **valore** è il dato che il programma usa nelle operazioni.

Ad esempio il numero **4** o l'insieme di caratteri **'ciao'** sono valori.

I *valori* hanno **tipo** diverso e si possono raggruppare in **classi**.

Per sapere il tipo di un valore, Python ha la funzione *built-in*: `type()`

In [8]:
```
type(4)
```

Out[8]:
```
int
```

In [9]:
```
type('ciao')
```

Out[9]:
```
str
```

Nell'esempio:

- `4` è un numero *intero* (`int` eger)
- `'ciao'` è una *stringa* di caratteri (`str` ing).

Possiamo facilmente identificare le stringhe di caratteri (sia noi che l'interprete Python) perché sono racchiuse degli apici (**quote**)

Variabili

Le **variabili** sono i **nomi** assegnati ai valori salvati nella memoria.

I programmi operano con le variabili per ottenere dei risultati.

Lo **statement di assegnazione** effettuato tramite il token `=` collega un valore alla sua variabile

Variabili

Le **variabili** sono i **nomi** assegnati ai valori salvati nella memoria.

I programmi operano con le variabili per ottenere dei risultati.

Lo **statement di assegnazione** effettuato tramite il token `=` collega un valore alla sua variabile

In [10]:
```
number  = 10
website = 'www.units.it'
number  = 3.14
```

Variabili

Le **variabili** sono i **nomi** assegnati ai valori salvati nella memoria.

I programmi operano con le variabili per ottenere dei risultati.

Lo **statement di assegnazione** effettuato tramite il token `=` collega un valore alla sua variabile

In [10]:
```
number  = 10
website = 'www.units.it'
number  = 3.14
```

Alla variabile `number` è assegnato il *valore* 10, mentre alla variabile `website` il *valore* 'www.units.it'.

Poi alla **variabile** `number` è assegnato un nuovo *valore* 3.14, cioè è stata **riutilizzata**.

Da questo momento, se voglio usare il **valore 3.14** posso fare riferimento al nome `number`.

Valutare le variabili

Se chiediamo all'interprete Python di valutare le variabili, ci dirà il loro valore

Valutare le variabili

Se chiediamo all'interprete Python di valutare le variabili, ci dirà il loro valore

In [11]:
```
number
```

Out[11]:
```
3.14
```

Valutare le variabili

Se chiediamo all'interprete Python di valutare le variabili, ci dirà il loro valore

In [11]:
```
number
```

Out[11]:
```
3.14
```

L'assegnazione `=` crea una connessione tra la variabile nella parte sinistra dell'operatore `=` (**lvalue**) ed il valore nella parte destra dell'operatore (**rvalue**)

Valutare le variabili

Se chiediamo all'interprete Python di valutare le variabili, ci dirà il loro valore

In [11]:
```
number
```

Out[11]:
```
3.14
```

L'assegnazione `=` crea una connessione tra la variabile nella parte sinistra dell'operatore `=` (**lvalue**) ed il valore nella parte destra dell'operatore (**rvalue**)

In [12]:
```
10 = number  # assegno lvalue a rvalue: ERRORE
```

```
  File "<ipython-input-12-7acbdac05375>", line 1
    10 = number  # assegno lvalue a rvalue: ERRORE
    ^
SyntaxError: cannot assign to literal
```

Valutare le variabili

Se chiediamo all'interprete Python di valutare le variabili, ci dirà il loro valore

In [11]:
```
number
```

Out[11]:
```
3.14
```

L'assegnazione `=` crea una connessione tra la variabile nella parte sinistra dell'operatore `=` (**lvalue**) ed il valore nella parte destra dell'operatore (**rvalue**)

In [12]:
```
10 = number  # assegno lvalue a rvalue: ERRORE
```

```
  File "<ipython-input-12-7acbdac05375>", line 1
    10 = number  # assegno lvalue a rvalue: ERRORE
    ^
SyntaxError: cannot assign to literal
```

> *Il token di assegnazione `=` non va confuso con il simbolo di uguagliaza, che in Python si indica con il token `==`.*
> *Non si può invertire lo statement di assegnazione*

Regole e convenzioni per i nomi delle variabili

Espressioni

Una espressione è una combinazione di valori, variabili, operatori e *chiamate a funzioni*. Scrivendo una espressione, l'interprete Python la **valuta** e visualizza il risultato.

Espressioni

Una espressione è una combinazione di valori, variabili, operatori e *chiamate a funzioni*. Scrivendo una espressione, l'interprete Python la **valuta** e visualizza il risultato.

In [14]:
```
1 + 1
```

Out[14]:
```
2
```

Espressioni

Una espressione è una combinazione di valori, variabili, operatori e *chiamate a funzioni*. Scrivendo una espressione, l'interprete Python la **valuta** e visualizza il risultato.

In [14]:
```
1 + 1
```

Out[14]:
```
2
```

In [15]:
```
len('ciao')
```

Out[15]:
```
4
```

Espressioni

Una espressione è una combinazione di valori, variabili, operatori e *chiamate a funzioni*. Scrivendo una espressione, l'interprete Python la **valuta** e visualizza il risultato.

In [14]:
```
1 + 1
```

Out[14]:
```
2
```

In [15]:
```
len('ciao')
```

Out[15]:
```
4
```

In questo caso la funzione *built-in* di Python `len()` ritorna il numero di caratteri in una stringa (la sua lunghezza)
come poco fa la funzione `type()` ritornava il tipo di un valore, o la funzione `print()` visualizza sullo schermo

Espressioni

Una espressione è una combinazione di valori, variabili, operatori e *chiamate a funzioni*. Scrivendo una espressione, l'interprete Python la **valuta** e visualizza il risultato.

In [14]:
```
1 + 1
```

Out[14]:
```
2
```

In [15]:
```
len('ciao')
```

Out[15]:
```
4
```

In questo caso la funzione *built-in* di Python `len()` ritorna il numero di caratteri in una stringa (la sua lunghezza)
come poco fa la funzione `type()` ritornava il tipo di un valore, o la funzione `print()` visualizza sullo schermo

In [16]:
```
a = 3 + 2
print(a)
```

```
5
```

Operatori

Gli operatori sono dei token speciali che indicano delle operazioni matematematiche come:

- addizione: `+`
- moltiplicazione: `*`
- divisione: `/`
- sottrazione: `-`

La combinazione di operatori e parentesi in Python ha lo stesso significato che in matematica e segue le stesse regole.
L'elevemento a potenza è indicato dal token `**` (due asterischi)

Operatori

Gli operatori sono dei token speciali che indicano delle operazioni matematematiche come:

- addizione: `+`
- moltiplicazione: `*`
- divisione: `/`
- sottrazione: `-`

La combinazione di operatori e parentesi in Python ha lo stesso significato che in matematica e segue le stesse regole.
L'elevemento a potenza è indicato dal token `**` (due asterischi)

In [17]:
```
5**2 + 2
```

Out[17]:
```
27
```

Operatori

Gli operatori sono dei token speciali che indicano delle operazioni matematematiche come:

- addizione: `+`
- moltiplicazione: `*`
- divisione: `/`
- sottrazione: `-`

La combinazione di operatori e parentesi in Python ha lo stesso significato che in matematica e segue le stesse regole.
L'elevemento a potenza è indicato dal token `**` (due asterischi)

In [17]:
```
5**2 + 2
```

Out[17]:
```
27
```

La divisione tra interi è un caso speciale: Python3 assume sempre che il risultato sia un numero reale.
Per imporre una divisione tra interi si usa il token `//` (doppio slash)

Operatori

Gli operatori sono dei token speciali che indicano delle operazioni matematematiche come:

- addizione: `+`
- moltiplicazione: `*`
- divisione: `/`
- sottrazione: `-`

La combinazione di operatori e parentesi in Python ha lo stesso significato che in matematica e segue le stesse regole.
L'elevemento a potenza è indicato dal token `**` (due asterischi)

In [17]:
```
5**2 + 2
```

Out[17]: 27

La divisione tra interi è un caso speciale: Python3 assume sempre che il risultato sia un numero reale.
Per imporre una divisione tra interi si usa il token `//` (doppio slash)

In [18]:
```
7//3
```

Out[18]: 2

Esercizi

1. Commentate il risultato di queste espressioni quando sono eseguite dall'interprete Python:

   ```
   5 % 2,
   35 % 32,
   32 % 35,
   8 % 7,
   7 % 8,
   9 % 0,
   0 % 9
   ```

   ... cos'è l'operatore `%`?

Operazioni e tipi

Python può fare operazioni solo tra *variabili* (*valori*) dello stesso **tipo** (*classe*).

Se i **tipi** sono **compatibili**, come il caso degli interi che sono un sottoinsieme dei numeri reali, Python fa automaticamente il **cast** per noi al tipo più alto

Operazioni e tipi

Python può fare operazioni solo tra *variabili* (*valori*) dello stesso **tipo** (*classe*).

Se i **tipi** sono **compatibili**, come il caso degli interi che sono un sottoinsieme dei numeri reali, Python fa automaticamente il **cast** per noi al tipo più alto

In [4]:
```
# Python converte l'intero 5 in float e fa l'operazione
3.14 + 5
```

Out[4]:
```
8.14
```

Operazioni e tipi

Python può fare operazioni solo tra *variabili* (*valori*) dello stesso **tipo** (*classe*).

Se i **tipi** sono **compatibili**, come il caso degli interi che sono un sottoinsieme dei numeri reali, Python fa automaticamente il **cast** per noi al tipo più alto

In [4]:
```
# Python converte l'intero 5 in float e fa l'operazione
3.14 + 5
```

Out[4]:
```
8.14
```

Se i tipi **non** sono compatibili, nella conversione l'interprete genererà un errore `TypeError`

Operazioni e tipi

Python può fare operazioni solo tra *variabili* (*valori*) dello stesso **tipo** (*classe*).

Se i **tipi** sono **compatibili**, come il caso degli interi che sono un sottoinsieme dei numeri reali, Python fa automaticamente il **cast** per noi al tipo più alto

In [4]:
```
# Python converte l'intero 5 in float e fa l'operazione
3.14 + 5
```

Out[4]:
```
8.14
```

Se i tipi **non** sono compatibili, nella conversione l'interprete genererà un errore `TypeError`

In [13]:
```
4 + 'ciao'
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-13-4f57a681e1c7> in <module>
----> 1 4 + 'ciao'

TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Esercizi

1. Utilizzando l'interprete Python, scrivete il comando

   ```
   oggi + 4
   ```

   e fate in modo che il risultato sia 10.

2. Utilizzando l'interprete Python, create la variabile `valore` e la variabile `percentuale` per calcolare il 7% di 1372

Operatori con stringhe

Gli operatori applicati alle stringhe hanno un comportamento particolare.

- L'operatore addizione (`+`) diventa **concatenazione**
- L'operatore moltiplicazione (`*`) diventa **ripetizione** e si può usare solo con interi e stringhe

Operatori con stringhe

Gli operatori applicati alle stringhe hanno un comportamento particolare.

- L'operatore addizione (`+`) diventa **concatenazione**
- L'operatore moltiplicazione (`*`) diventa **ripetizione** e si può usare solo con interi e stringhe

In [26]:
```
# concatenazione
'ciao' + 'a' + 'tutti'
```

Out[26]:
```
'ciaoatutti'
```

Operatori con stringhe

Gli operatori applicati alle stringhe hanno un comportamento particolare.

- L'operatore addizione (`+`) diventa **concatenazione**
- L'operatore moltiplicazione (`*`) diventa **ripetizione** e si può usare solo con interi e stringhe

In [26]:
```
# concatenazione
'ciao' + 'a' + 'tutti'
```

Out[26]:
```
'ciaoatutti'
```

In [27]:
```
# ripeto 3 volte 'ciao'
3 * 'ciao'
```

Out[27]:
```
'ciaociaociao'
```

Esercizi

1. Usando l'interprete Python, preparate una variabile per ognuna delle parole nella frase

   ```
   "Questo corso è bello bello bello in modo assurdo"
   ```

   e visualizzate la frase in una linea sola usando gli operatori

Il tipo *None*

Molti linguaggi di programmazione supportano il concetto di ***null***, un valore speciale che significa "*not-a-number*" per indicare che **il valore** esiste ma è "vuoto", cioè **non è ancora definito**.
In Python si indica con la keyword `None` che è del tipo `NoneType`

Assegnare il *valore* `None` ad una variabile non la cancella: lo spazio viene riservato ma è riempito con il valore *None*.

Il tipo *None*

Molti linguaggi di programmazione supportano il concetto di ***null***, un valore speciale che significa "*not-a-number*" per indicare che **il valore** esiste ma è "vuoto", cioè **non è ancora definito**.
In Python si indica con la keyword `None` che è del tipo `NoneType`

Assegnare il *valore* `None` ad una variabile non la cancella: lo spazio viene riservato ma è riempito con il valore *None*.

In [2]:
```
a = None
type(a)
```

Out[2]:
```
NoneType
```

Casting

Se vogliamo fare una conversione esplicita, sono disponibili delle funzioni di **casting** che si chiamano come la classe del tipo in cui voggliamo trasformare il valore:

- `int()` fa il casting del valore verso la classe del tipo integer
- `str()` fa il casting del valore verso la classe del tipo string
- `float()` fa il casting del valore verso la classe del tipo floating point

Ricordate che il cambiamento del tipo di un valore può introdurre degli errori di precisione dati all'arrotondamento o dal troncamento dei valori.
Se il casting avviene tra tipi **non** compatibili, Python produrrà un errore `ValueError`

In [21]:
```
int(3.14) # float --> int
```

Out[21]:
```
3
```

In [21]:
```
int(3.14) # float --> int
```

Out[21]:
```
3
```

In [22]:
```
str(3.14) # float --> str
```

Out[22]:
```
'3.14'
```

In [21]:
```
int(3.14) # float --> int
```

Out[21]:
```
3
```

In [22]:
```
str(3.14) # float --> str
```

Out[22]:
```
'3.14'
```

In [23]:
```
int('ciao') # str --> int
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-23-fac6597e93cf> in <module>
----> 1 int('ciao') # str --> int

ValueError: invalid literal for int() with base 10: 'ciao'
```

In [21]:
```
int(3.14) # float --> int
```

Out[21]:
```
3
```

In [22]:
```
str(3.14) # float --> str
```

Out[22]:
```
'3.14'
```

In [23]:
```
int('ciao') # str --> int
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-23-fac6597e93cf> in <module>
----> 1 int('ciao') # str --> int

ValueError: invalid literal for int() with base 10: 'ciao'
```

In [24]:
```
int(9.9999) # valore troncato!
```

Out[24]:
```
9
```

In [21]:
```
int(3.14) # float --> int
```

Out[21]:
```
3
```

In [22]:
```
str(3.14) # float --> str
```

Out[22]:
```
'3.14'
```

In [23]:
```
int('ciao') # str --> int
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-23-fac6597e93cf> in <module>
----> 1 int('ciao') # str --> int

ValueError: invalid literal for int() with base 10: 'ciao'
```

In [24]:
```
int(9.9999) # valore troncato!
```

Out[24]:
```
9
```

In [25]:
```
int(-9.9999) # valore troncato!
```

Out[25]:
```
-9
```

Controllare se due variabili hanno lo stesso tipo

Abbiamo visto che con `type()` possiamo sapere il tipo di un valore.

Per verificare se due variabili sono dello stesso tipo, bisogna sempre usare `type()` e non (soltanto) l'operatore `==`

Controllare se due variabili hanno lo stesso tipo

Abbiamo visto che con `type()` possiamo sapere il tipo di un valore.

Per verificare se due variabili sono dello stesso tipo, bisogna sempre usare `type()` e non (soltanto) l'operatore `==`

In [12]:
```
a = 1     # int
b = 1.0   # float
a == b    # hanno lo stesso valore?
```

Out[12]:
```
True
```

Controllare se due variabili hanno lo stesso tipo

Abbiamo visto che con `type()` possiamo sapere il tipo di un valore.

Per verificare se due variabili sono dello stesso tipo, bisogna sempre usare `type()` e non (soltanto) l'operatore `==`

In [12]:
```
a = 1     # int
b = 1.0   # float
a == b    # hanno lo stesso valore?
```

Out[12]:
```
True
```

In [15]:
```
type(a) == type(b)  # hanno lo stesso TIPO di valore?
```

Out[15]:
```
False
```

Considerazioni

Alle **variabili** si assegnano i **valori** e quando si chiede di valutare una variabile all'interprete Python si vede il valore che *contiene*.
Se scrivo:

Considerazioni

Alle **variabili** si assegnano i **valori** e quando si chiede di valutare una variabile all'interprete Python si vede il valore che *contiene*.
Se scrivo:

In [10]:
```
a = b = 4
```

Considerazioni

Alle **variabili** si assegnano i **valori** e quando si chiede di valutare una variabile all'interprete Python si vede il valore che *contiene*.
Se scrivo:

In [10]:
```
a = b = 4
```

quante variabili e quanti valori ho nel programma?

Considerazioni

Alle **variabili** si assegnano i **valori** e quando si chiede di valutare una variabile all'interprete Python si vede il valore che *contiene*.
Se scrivo:

In [10]:
```
a = b = 4
```

quante variabili e quanti valori ho nel programma?

In [11]:
```
print(a)  # visualizza valore a
```

```
4
```

Considerazioni

Alle **variabili** si assegnano i **valori** e quando si chiede di valutare una variabile all'interprete Python si vede il valore che *contiene*.
Se scrivo:

In [10]:
```
a = b = 4
```

quante variabili e quanti valori ho nel programma?

In [11]:
```
print(a)  # visualizza valore a
```

```
4
```

In [12]:
```
print(b) # visualizza valore b
```

```
4
```

Considerazioni

Alle **variabili** si assegnano i **valori** e quando si chiede di valutare una variabile all'interprete Python si vede il valore che *contiene*.
Se scrivo:

In [10]:
```
a = b = 4
```

quante variabili e quanti valori ho nel programma?

In [11]:
```
print(a)  # visualizza valore a
```

```
4
```

In [12]:
```
print(b) # visualizza valore b
```

```
4
```

per rispondere bisogna capire le relazioni tra variabili, valori e memoria in Python

[...]Python è un linguaggio ... orientato agli oggetti

In Python una variabile è un riferimento (o *puntatore*) ad un oggetto che si trova in memoria.

Ogni oggetto ha un **identificativo univoco** (la sua *posizione in memoria*).

Quando si assegna un oggetto ad una variabile, si può usare il nome della variabile per fare riferimento all'oggetto. Ma il dato è contenuto nell'oggetto.

Ad esempio lo statement

[...]Python è un linguaggio ... orientato agli oggetti

In Python una variabile è un riferimento (o *puntatore*) ad un oggetto che si trova in memoria.

Ogni oggetto ha un **identificativo univoco** (la sua *posizione in memoria*).

Quando si assegna un oggetto ad una variabile, si può usare il nome della variabile per fare riferimento all'oggetto. Ma il dato è contenuto nell'oggetto.

Ad esempio lo statement

In [13]:
```
a = 4
```

[...]Python è un linguaggio ... orientato agli oggetti

In Python una variabile è un riferimento (o *puntatore*) ad un oggetto che si trova in memoria.

Ogni oggetto ha un **identificativo univoco** (la sua *posizione in memoria*).

Quando si assegna un oggetto ad una variabile, si può usare il nome della variabile per fare riferimento all'oggetto. Ma il dato è contenuto nell'oggetto.

Ad esempio lo statement

In [13]:
```
a = 4
```

crea un oggetto di tipo `int`eger in una posizione della memoria e lo assegna alla variabile `a` che diventa il suo *puntatore*

> *Si può conoscere l'identificativo univoco di un oggetto con la funzione built-in* `id()`

[...]Python è un linguaggio ... orientato agli oggetti

In Python una variabile è un riferimento (o *puntatore*) ad un oggetto che si trova in memoria.

Ogni oggetto ha un **identificativo univoco** (la sua *posizione in memoria*).

Quando si assegna un oggetto ad una variabile, si può usare il nome della variabile per fare riferimento all'oggetto. Ma il dato è contenuto nell'oggetto.

Ad esempio lo statement

In [13]:
```
a = 4
```

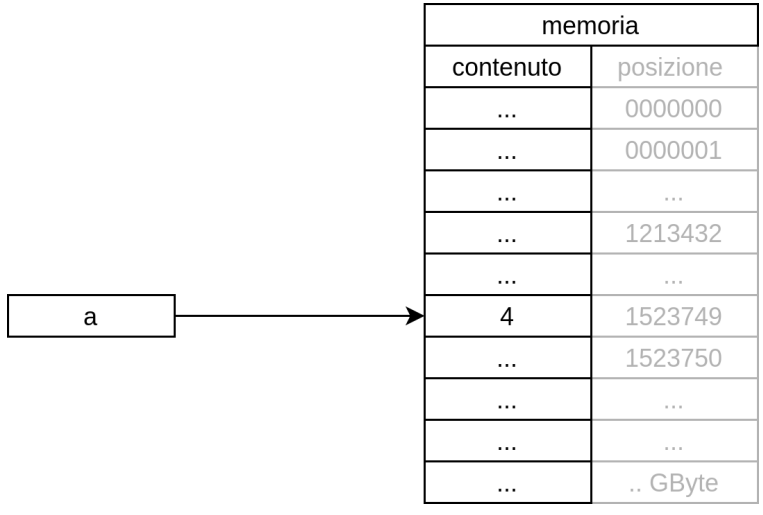
crea un oggetto di tipo `int`eger in una posizione della memoria e lo assegna alla variabile `a` che diventa il suo *puntatore*

> *Si può conoscere l'identificativo univoco di un oggetto con la funzione built-in `id()`*

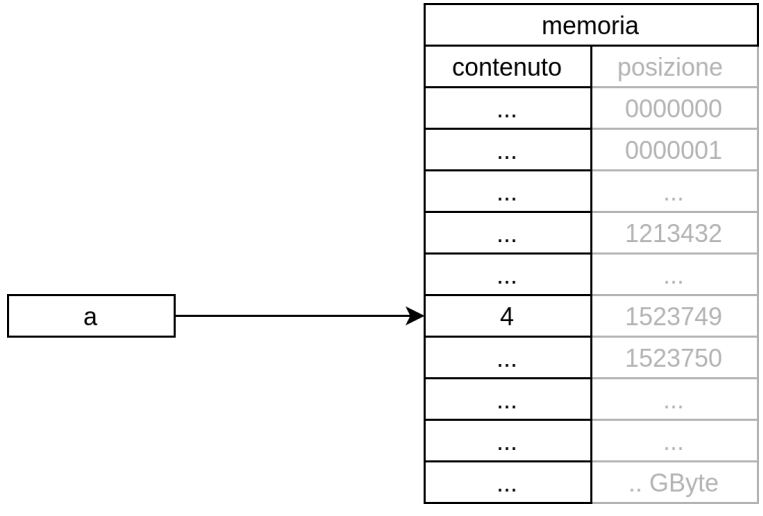
In [14]:
```
id(a)  # identificativo univoco dell'oggetto assegnato ad 'a'
```

Out[14]:
```
9784992
```

con un disegno:



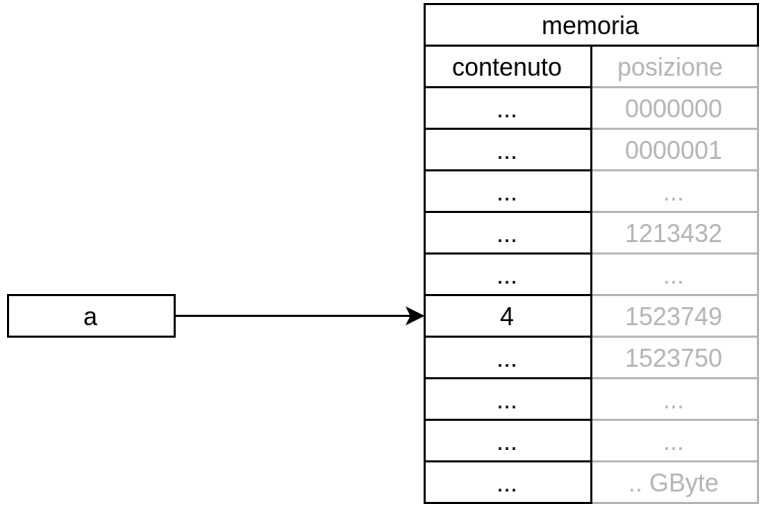
con un disegno:



In [15]:
```
id(a)  # identificativo univoco dell'oggetto assegnato ad 'a'
```

Out[15]:
```
9784992
```

con un disegno:



In [15]:
```
id(a)  # identificativo univoco dell'oggetto assegnato ad 'a'
```

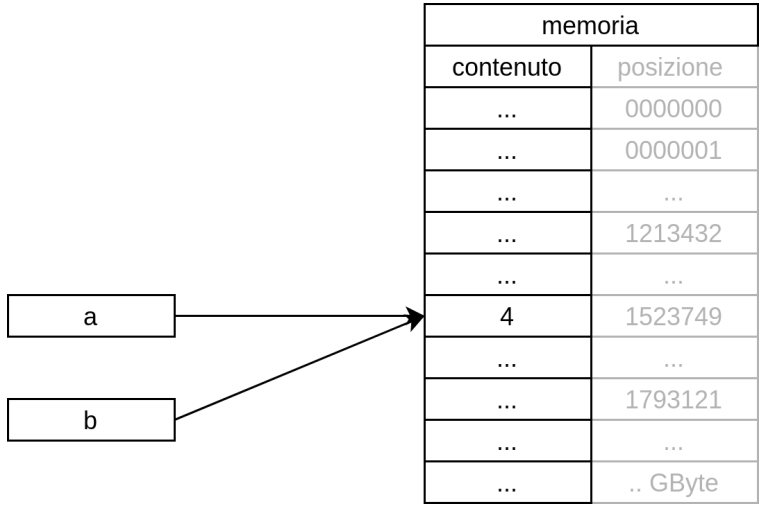
Out[15]: 9784992

In [16]:
```
id(4) # identificativo univoco dell'oggetto di valore '4'
```

Out[16]: 9784992

Se adesso scriviamo `a = b` **Python non crea un altro oggetto ma solo un nuovo link** `b` che punta allo stesso di `a`

Se adesso scriviamo `a = b` **Python non crea un altro oggetto ma solo un nuovo link** `b` che punta allo stesso di `a`

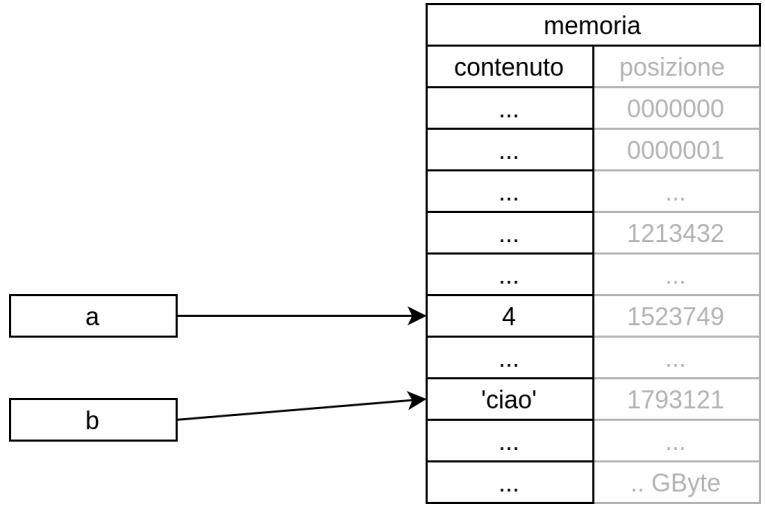


In [17]:
```
a = b
print(id(a))
print(id(b))
```

```
9784992
9784992
```

se ora riutilizzo `b` scrivendo `b = 'ciao'`

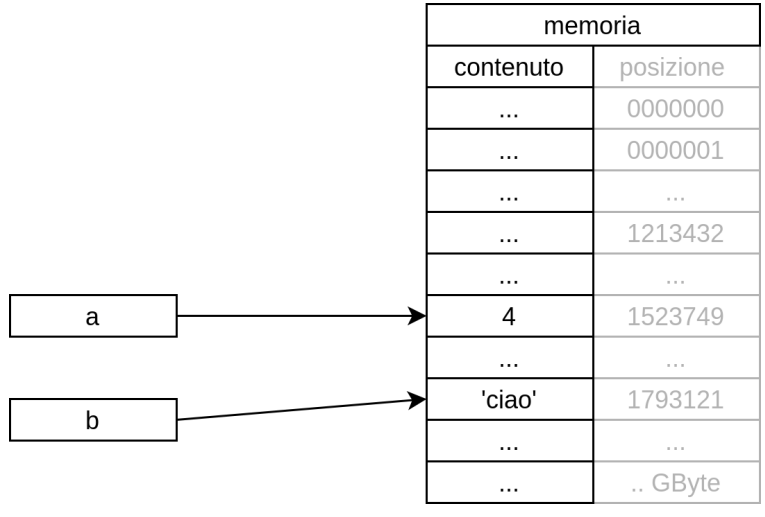
se ora riutilizzo `b` scrivendo `b = 'ciao'`



In [18]:
```
b = 'ciao'
id(b)
```

Out[18]:
```
140277588943792
```

se ora riutilizzo `b` scrivendo `b = 'ciao'`

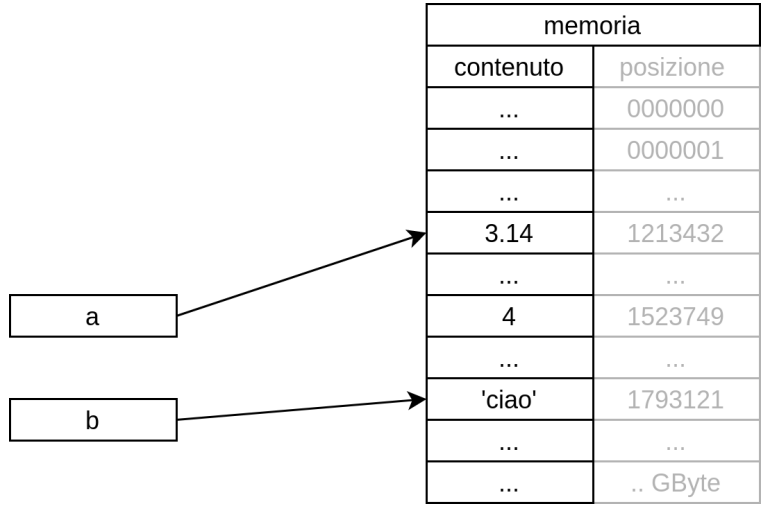


In [18]:
```
b = 'ciao'
id(b)
```

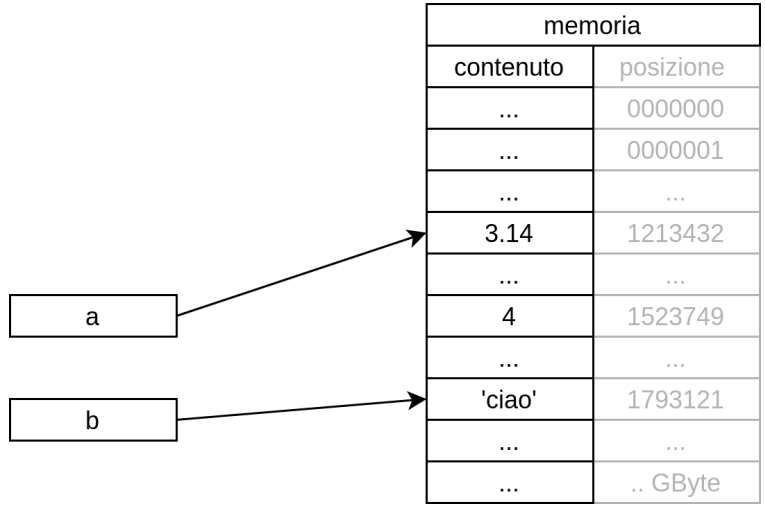
Out[18]:
```
140277588943792
```

La variabile `b` punterà il nuovo oggetto `str` 'ciao', che ha un suo `id()`

e se riutilizzo anche `a` scrivendo `a = 3.14`



e se riutilizzo anche `a` scrivendo `a = 3.14`



adesso `a` punta al nuovo oggetto `3.14` e il valore `4` non è più raggiungibile: ha raggiunto il suo **end-of-life** e la memoria verrà *liberata dal garbage collector*

> *In Python gli oggetti raggiungono l'endo-of-life quando il numero dei loro puntatori è `0`*

a = b = 4

Per sapere se due variabili si riferiscono allo stesso oggetto si usa sempre `id()`

a = b = 4

Per sapere se due variabili si riferiscono allo stesso oggetto si usa sempre `id()`

In [32]:
```
a = b = 4
print(id(4))   # visualizza id univoco di '4'
print(id(a))   # visualizza id univoco di 'a'
print(id(b))   # visualizza id univoco di 'b'
```

```
9784992
9784992
9784992
```

e se faccio una operazione su una variabile che punta allo stesso oggetto?

e se faccio una operazione su una variabile che punta allo stesso oggetto?

In [33]:
```
b = b + 1 # sommo 1 all'oggetto puntato da b, cioè il '4', quanto vale a?
print(a)   # visualizza ivalore di 'a'
print(b)   # visualizza ivalore di 'b'
```

```
4
5
```

In [34]:
```
print(id(a))   # visualizza id univoco di 'a'
print(id(b))   # visualizza id univoco di 'b'
```

```
9784992
9785024
```

e se faccio una operazione su una variabile che punta allo stesso oggetto?

In [33]:
```
b = b + 1 # sommo 1 all'oggetto puntato da b, cioè il '4', quanto vale a?
print(a)   # visualizza ivalore di 'a'
print(b)   # visualizza ivalore di 'b'
```

```
4
5
```

In [34]:
```
print(id(a))   # visualizza id univoco di 'a'
print(id(b))   # visualizza id univoco di 'b'
```

```
9784992
9785024
```

> *quando eseguo l'operazione l'oggetto cambia. Questo è molto importante quando si lavora con oggetti grossi e bisogno fare attenzione alla memoria utilizzata*

Stesso oggetto

Per sapere se due variabili *puntano* lo stesso oggetto si usa la *keyword* `is`

Stesso oggetto

Per sapere se due variabili *puntano* lo stesso oggetto si usa la *keyword* `is`

In [36]:
```
a = b = 4  # assegno il falore 4 alle variabili a e b
a is b     # puntano lo stesso oggetto '4'?
```

Out[36]:
```
True
```

Stesso oggetto

Per sapere se due variabili *puntano* lo stesso oggetto si usa la *keyword* `is`

In [36]:
```
a = b = 4  # assegno il falore 4 alle variabili a e b
a is b     # puntano lo stesso oggetto '4'?
```

Out[36]:
```
True
```

In [35]:
```
a = 'ciao' # assegno la str 'ciao' ad 'a'
b = 7      # assegno l'int '7' a 'b'
a is b     # puntano allo stesso oggetto?
```

Out[35]:
```
False
```

Stesso oggetto

Per sapere se due variabili *puntano* lo stesso oggetto si usa la *keyword* `is`

In [36]:
```
a = b = 4  # assegno il falore 4 alle variabili a e b
a is b     # puntano lo stesso oggetto '4'?
```

Out[36]:
```
True
```

In [35]:
```
a = 'ciao' # assegno la str 'ciao' ad 'a'
b = 7      # assegno l'int '7' a 'b'
a is b     # puntano allo stesso oggetto?
```

Out[35]:
```
False
```

> ***Il modo corretto per testare se variabili `None` è usare `is`***

Stesso oggetto

Per sapere se due variabili *puntano* lo stesso oggetto si usa la *keyword* `is`

In [36]:
```
a = b = 4  # assegno il falore 4 alle variabili a e b
a is b     # puntano lo stesso oggetto '4'?
```

Out[36]:
```
True
```

In [35]:
```
a = 'ciao' # assegno la str 'ciao' ad 'a'
b = 7      # assegno l'int '7' a 'b'
a is b     # puntano allo stesso oggetto?
```

Out[35]:
```
False
```

> ***Il modo corretto per testare se variabili `None` è usare `is`***

In [2]:
```
a = None    # assegno 'None' and 'a'
a is None   # testo se 'a' è un oggetto vuoto
```

Out[2]:
```
True
```

Stesso oggetto

Per sapere se due variabili *puntano* lo stesso oggetto si usa la *keyword* `is`

In [36]:
```
a = b = 4  # assegno il falore 4 alle variabili a e b
a is b     # puntano lo stesso oggetto '4'?
```

Out[36]:
```
True
```

In [35]:
```
a = 'ciao' # assegno la str 'ciao' ad 'a'
b = 7      # assegno l'int '7' a 'b'
a is b     # puntano allo stesso oggetto?
```

Out[35]:
```
False
```

> ***Il modo corretto per testare se variabili `None` è usare `is`***

In [2]:
```
a = None    # assegno 'None' and 'a'
a is None   # testo se 'a' è un oggetto vuoto
```

Out[2]:
```
True
```

In [5]:
```
a == None   #..funziona anche con l'operatore == ma non è del tutto corretto (ricordate 1 == 1.0)?
```

Out[5]:
```
True
```

Esercizi

1. Definite la variabile:

   ```
   a = 1
   ```

   e valutate i seguenti statement:

   ```
   a is int
   type(a) is int
   ```

   infine dimostrate i risultati ottenuti con l'ausilio della funzione `id()`

Input

Per acquisire informazioni in modo interattivo dall'utente è possibile usare la funzione *built-in* `input( )`

La funzione accetta come parametro una stringa che diventerà il testo visualizzato.

> *un parametro è l'argomento di una funzione*

Input

Per acquisire informazioni in modo interattivo dall'utente è possibile usare la funzione *built-in* `input( )`

La funzione accetta come parametro una stringa che diventerà il testo visualizzato.

> *un parametro è l'argomento di una funzione*

In [ ]:
```
# chiedo un numero all'utente
a = input('Scrivi un numero')
```

Input

Per acquisire informazioni in modo interattivo dall'utente è possibile usare la funzione *built-in* `input( )`

La funzione accetta come parametro una stringa che diventerà il testo visualizzato.

> *un parametro è l'argomento di una funzione*

In [ ]:
```
# chiedo un numero all'utente
a = input('Scrivi un numero')
```

In questo esempio la scritta ***Scrivi un numero*** sarà visualizzata assieme ad una finestra per immettere del testo.

Il numero immesso sarà poi assegnato alla variabile **a**.

> *Il numero immesso è letto come tipo **string** dalla funzione `input()`, e dovrà quindi essere convertito (**cast**) per poterlo utilizzare in operazioni matematiche*

Esercizi

1. Utilizzando la funzione `input()`, assegnate il nome dell'utente a una variabile `nome` e visualizzate la scritta `Ciao 'nome'` usando la funzione `print()`

2. Risolvete, tramite l'interprete Python, questo problema: se ora sono le 17, che ore saranno tra 67 ore?

3. Utilizzando la funzione `input()`, chiedete all'utente:

    - l'ora attuale (in ore)
    - tra quante ore mettere la sveglia

    e visualizza su schermo l'ora che segnerà l'orologio quando suonerà la sveglia

Scrivere un programma

Un programma Python è un file di testo che contiene le istruzioni eseguibili da Python.

Si riconosce perchè ha estensione `.py`.

Intestazione

Ogni **buon programma** comincia sempre con una intestazione (**header**).
L'header è un commento multilinea che contiene *almeno* i campi:

- NOME del programma
- AUTORE del programma
- DATA di creazione/ultima modifica del programma
- VERSIONE del programma
- DESCRIZIONE del programma

Intestazione

Ogni **buon programma** comincia sempre con una intestazione (**header**).
L'header è un commento multilinea che contiene *almeno* i campi:

- NOME del programma
- AUTORE del programma
- DATA di creazione/ultima modifica del programma
- VERSIONE del programma
- DESCRIZIONE del programma

In [29]:
```
#
# File: Programma.py
#
# Author: E.Romelli, D.Tavagnacco
#
# Date: 2021/03/08
#
# Version: 1.0
#
# Description: My First Project Program to print "Hello, World!".
#
```

Corpo del programma

Il corpo del programma è l'insieme delle istruzioni per l'interprete Python.

Python **costringe** i programmatori a scrivere le istruzioni in modo ordinato

- ogni istruzione va su una nuova linea
- le linee devono avere la stessa indentazione

Corpo del programma

Il corpo del programma è l'insieme delle istruzioni per l'interprete Python.

Python **costringe** i programmatori a scrivere le istruzioni in modo ordinato

- ogni istruzione va su una nuova linea
- le linee devono avere la stessa indentazione

In [30]:
```
a = 3
  b = 5
```

```
  File "<ipython-input-30-543ee60811ab>", line 2
    b = 5
    ^
IndentationError: unexpected indent
```

Corpo del programma

Il corpo del programma è l'insieme delle istruzioni per l'interprete Python.

Python **costringe** i programmatori a scrivere le istruzioni in modo ordinato

- ogni istruzione va su una nuova linea
- le linee devono avere la stessa indentazione

In [30]:
```
a = 3
  b = 5
```

```
  File "<ipython-input-30-543ee60811ab>", line 2
    b = 5
    ^
IndentationError: unexpected indent
```

Le **indentazioni** servono ad indicare l'inizio e la fine di **blocchi di istruzioni**.
Nelle prossime lezioni vedremo come

Eseguire un programma

Per eseguire un programma *myProgram.py* si avvia l'interprete Python dalla *Shell* **passando** il nome del file come **primo argomento**

```
python3 myProgram.py
```

Quando il programma termina, la *shell* torna disponibile per accettare altri comandi.

> *si può interrompere un programma Python in ogni momento con la combinazione Ctrl+c*

Hello, World!

Scriviamo un programma che visualizza su schermo la scritta "Hello, World!"

Hello, World!

Scriviamo un programma che visualizza su schermo la scritta "Hello, World!"

In [31]:
```
#
# File: HelloWorld.py
#
# Author: E.Romelli, D.Tavagnacco
#
# Date: 2021/03/08
#
# Version: 1.0
#
# Description: My First Project Program to print "Hello, World!".
#

print("Hello, World!")
```

```
Hello, World!
```

Esercizi

1. Create un file `Hello_World.py` contenente le istruzioni per scrivere "Hello, World!" e provate ad eseguirlo con il comando

   ```
   python3 Hello_World.py
   ```

2. Modificate il programma `Hello_World.py` per fargli scrivere anche il vostro nome sotto a "Hello, World!"

3. Modificate il programma `Hello_World.py` utilizzando la funzione `input()` per chiedere il nome dell'utente, ad esempio "Luca" e fargli scrivere "Hello, Luca!"