A.A. 2022-2023

Laboratorio di Programmazione

Introduzione a Python

Docente: Aniello Minutolo
E-mail: aniello.minutolo@unina.it

Laboratorio di Programmazione

INTRODUZIONE A PYTHON

Python IDEs

- **Anaconda Navigator**
  https://www.anaconda.com/products/individual
  - notebook Jupiter
  - spyder
- **Visual Studio Code**
  https://code.visualstudio.com/
- **PyCharm**
  https://www.jetbrains.com/pycharm/download/
- **Terminale …**

Conda

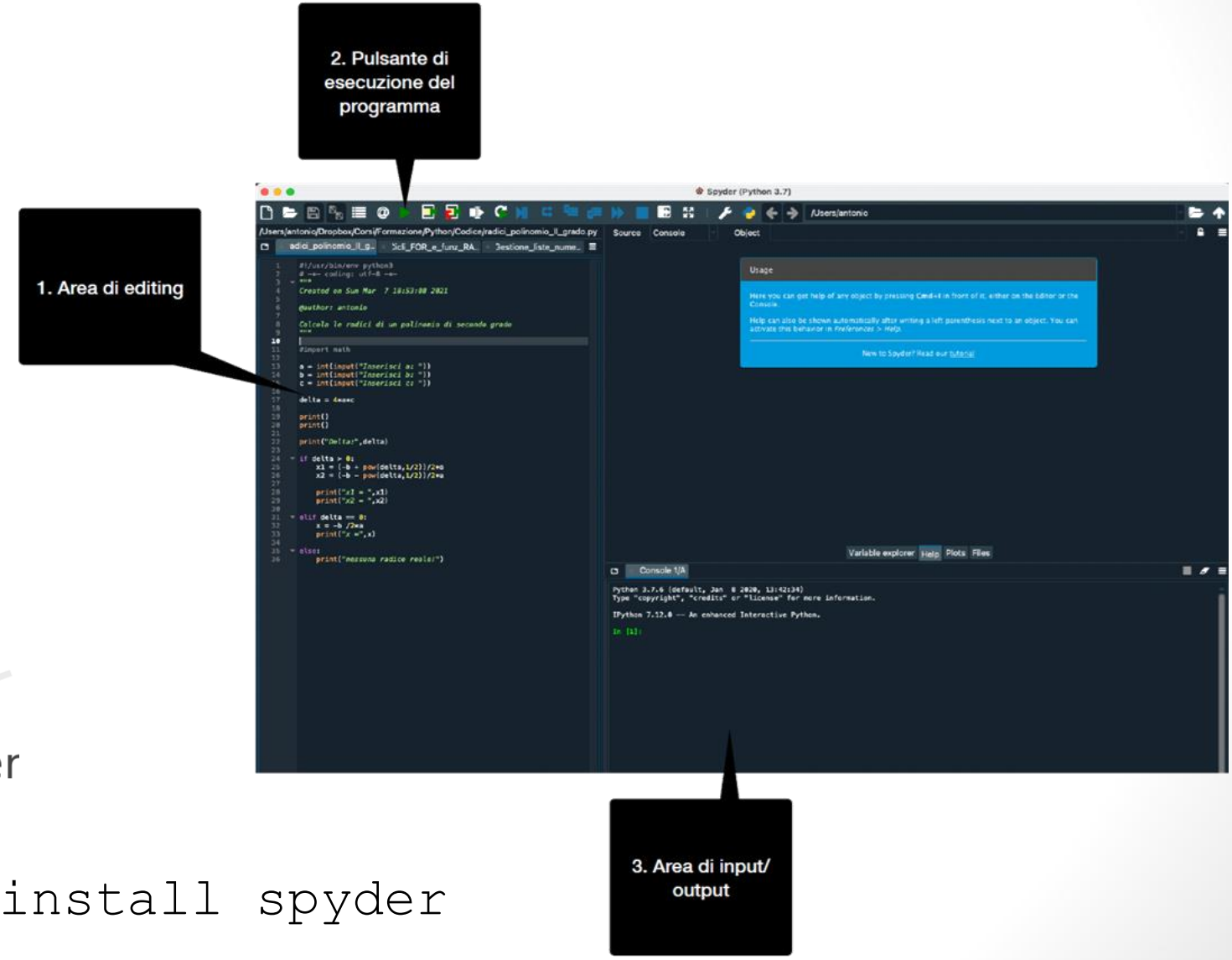
Conda è un environment e package manager open source, creato per Python ma può essere usato anche per altri linguaggi di programmazione

- consente di installare, eseguire e aggiornare rapidamente pacchetti softwaree le relative dipendenze
- consente di creare e gestire ambienti di sviluppo separati

Alcuni comandi utili

- `conda info –envs` **oppure** `conda env list`
- `conda create -n ENVIRONMENT python=3.7`
- `conda activate ENVIRONMENT`
- `conda deactivate`
- `conda env remove --name ENVIRONMENT`

Python IDEs: spyder



spyder

```
pip install spyder
```

Cosa è Python ?

E’ un linguaggio di programmazione interpretato

- interpreta in automatico le istruzioni traducendole in linguaggio macchina

E’ un linguaggio general-purpose

- non crea molti “limiti” al programmatore in termini di sviluppo di codici e di architettura
- elevata disponibilità di codice e librerie disponibili

Invocato a linea di comando è pronto ad interpretare

- il prompt è caratterizzato da “>>> ”
- gli script Python e le funzioni che si possono “importare”, generalmente con estensione “.py”

Consente in maniera “selettiva” di importare parti di librerie e codici

Programmi Python

Un **programma Python** è una sequenza di definizioni e comandi

- le definizioni sono **valutate**
- i comandi sono **eseguiti** dall'interprete Python in una shell (terminal)

I comandi (statements) istruiscono l'interprete a fare qualcosa

I comandi possono essere digitati direttamente nella **shell** oppure immagazzinati in un **file** che viene letto dalla shell per essere valutato

Ogni istruzione è delimitata dal fine riga

- volendo si può anche usare (come in C/C++) " ; " per separare istruzioni sulla stessa riga ma non è consigliato
- per far continuare un'istruzione anche sulla linea successiva è necessario inserire un "\" a fine riga

Oggetti Python

I programmi Python manipolano cosiddetti **data objects**

Gli oggetti hanno un **tipo** che definisce il genere di cose che il programma può fare su di essi

- Mario è una persona che può camminare, parlare in italiano, etc.
- Chewbacca è un extraterrestre che può camminare, dire, “mwaaarhrhh”, etc. ☺

Gli oggetti possono essere

- ***scalari*** (non possono essere divisi)
- ***non-scalari*** (hanno una struttura interna a cui si può accedere)

Oggetti scalari

`int` – rappresenta **interi**, e.g., `5`

`float` – rappresenta **numeri reali**, e.g., `3.27`

`bool` – rappresenta i valori **booleani** `True` e `False`

`NoneType` – tipo **speciale** e assume un solo valore, `None`

Possiamo usare `type()` per vedere il tipo di un oggetto

```
>>> type(5)
int
>>> type(3.0)
float
```

what you write into the Python shell

what shows after hitting enter

Conversione di tipo (*type casting*)

Possiamo **convertire un oggetto di un tipo in un oggetto di altro tipo**

`float(3)` converte l'intero `3` in un float `3.0`

`int(3.9)` tronca il float `3.9` nell'intero `3`

Stampa verso la console

Per poter mostrare l'output dal codice verso l'utente, possiamo usare il comando `print`

```
In [11]: 3+2
Out[11]: 5

In [12]: print(3+2)
5
```

"Out" tells you it's an interaction within the shell only

No "Out" means it is actually shown to a user, apparent when you edit/run files

Espressioni

Combinare oggetti e operatori per ottenere espressioni

Un'espressione ha un **valore**, che ha un tipo

La sintassi per un'espressione semplice è la seguente

```
<object> <operator> <object>
```

Esempi di espressioni e tipi

```
>>> type (4*7.0)
      <type 'float '>
>>> type (2*8)
      <type 'int '>
>>> 7.3%2
      1.2999999999999998
>>> 7%2
      1
>>> 3.0 and 2.4
      2.4
>>> not 3.9
      False
```

Operazioni per interi e float

`i+j` → **somma**

`i-j` → **differenza**

`i*j` → **prodotto**

`i/j` → **divisione**

Se entrambi sono interi, il risultato sarà un intero

Se entrambi sono float, il risultato sarà un float

Il risultato è un float

`i%j` → operazione **modulo** quando `i` viene diviso per `j`

`i**j` → **elevamento a potenza** di `i` a `j`

Semplici operazioni

Le parentesi sono utilizzate per istruire l'interprete Python ad eseguire certe operazioni prima di altre (**operatore di precedenza**)

Senza le parentesi, **l'operatore di precedenza** segue

1. **
2. *
3. /
4. + e – sono eseguite dalla sinistra verso destra, come appaiono nell'espressione

Binding per le variabili e valori

Una variabile **è un'associazione** fra un identificatore e un valore;

- è una "scatola" che contiene un valore

In Python ogni variabile è associata ad un **tipo** di dati

Differentemente da altri linguaggi di programmazione in Python non bisogna dichiarare il tipo di una variabile che si vuole utilizzare

- questo permette di **cambiare il tipo più volte il tipo** di valore memorizzato in una variabile durante l'esecuzione del programma
- è comunque necessario **inizializzare la variabile**

```
x = "Hello world!"
print('x: ', x)  # x: Hello world!

...

y = 5
x = 3 * y
print('x: ', x)  # x: 15
```

Binding per le variabili e valori

Nell'esempio riportato

- si immagini di aver creato una scatola nella memoria del computer, di averla chiamata `x`, e di averci depositato la stringa "`Hello world!`«
- fino a che il contenuto della scatola non viene alterato, si potrà riaccedere al valore contenuto tramite il nome assegnato alla scatola, e quindi alla variabile
- è possibile cambiare più volte il contenuto nella scatola, assegnando un nuovo valore alla variabile, che può essere dello stesso tipo, o diverso

```
x = "Hello world!"
print('x: ', x)  # x: Hello world!

...

y = 5
x = 3 * y
print('x: ', x)  # x: 15
```

Binding per le variabili e valori

Il simbolo di uguale indica un **assegnazione** di un valore ad un nome di variabile

variable value

```
pi = 3.14159
```

```
pi_approx = 22/7
```

- il valore è memorizzato nella memoria principale del calcolatore
- un'assegnazione effettua un collegamento (**bind**) tra nome e valore
- è possibile recuperare il valore associato ad un nome o una variabile semplicemente invocando il suo nome, e.g., digitando `pi`

Astrarre le espressioni

Perché **dare nomi** a valori di espressioni?

- per **riusare nomi** al posto di valori

Questo faciliterà il cambiamento del codice nel futuro

E.g.:

```
pi = 3.14159
radius = 2.2
area = pi*(radius**2)
```

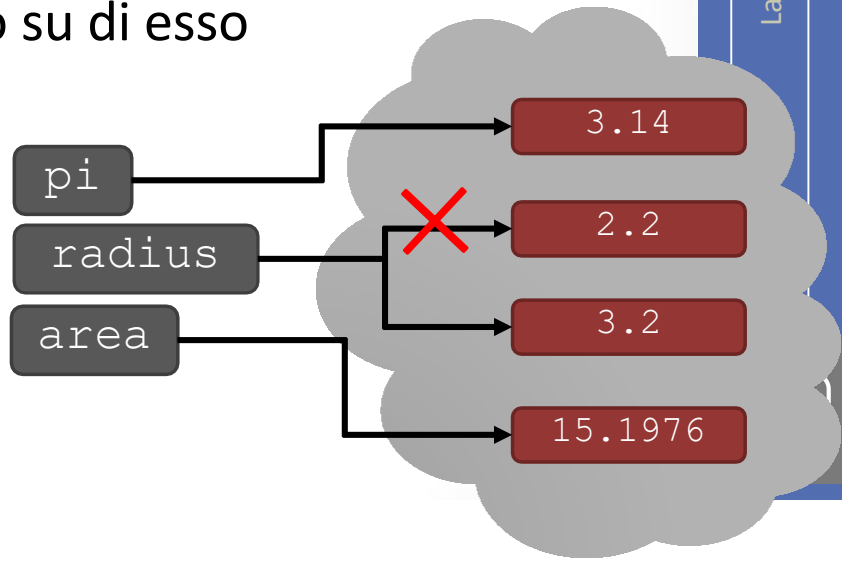
Modificare i *bindings*

E' possibile effettuare un **re-bind** di nomi di variabile effettuando semplicemente una nuova assegnazione

Il valore precedente potrebbe essere memorizzato in memoria principale ma sarà perso il suo riferimento (*handle*)

Il valore per quella specifica area di memoria non cambierà finché non effettueremo un nuovo calcolo su di esso

```
pi = 3.14
radius = 2.2
area = pi*(radius**2)
radius = radius+1
```



Tipi di dati

I tipi di dati messi a disposizione da **Python** sono i seguenti:

Semplici:

- interi (int),
- interi lunghi (long),
- numeri in virgola mobile (float)
- numeri complessi (complex)
- valori booleani (bool)
- stringhe (str)

```
spam=10
print(type(spam))
    <class 'int'>
```

```
spam=3.14
print(type(spam))
    <class 'float'>
```

```
spam='lettera'
type(spam)
    <class 'str'>
```

```
spam=[1,2,3,4]
type(spam)
    <class 'list'>
```

```
spam=(1,2,3,4)
type(spam)
    <class 'tuple'>
```

Tipi di dati

I tipi di dati messi a disposizione da **Python** sono i seguenti:

Tipi Semplici:

- interi (int)
- interi lunghi (long)
- numeri in virgola mobile (float)
- numeri complessi (complex)
- valori booleani (bool)
- stringhe (str)

```
age = 24
testo = "La mia età è..."
print(testo + '   ' + age)
```

TypeError: can only concatenate str (not "int") to str

CASTING is required

Tipi di dati

I tipi di dati messi a disposizione da **Python** sono i seguenti:

Tipi Strutturati

- le ***liste*** (list), sequenze ordinate di oggetti
  - es. `[10, 'hello', 200.3]`
  - ammettono duplicati
- le **tuple** (tup), sequenze ordinate e immutabili di oggetti
  - es. `(10, 'hello', 200.3)`
  - ammettono duplicati
- i **set** (set), collezioni non ordinate e immutabili di oggetti unici
  - es. `{"a", "b"}`
  - non ammettono duplicati
- i **dizionari** (dict), collezioni non ordinate di coppie `"key":"value"`
  - es. `{"name": "Mario", "surname": "Rossi"}`
  - non ammettono chiavi duplicate

Stringhe

Possono contenere lettere, caratteri speciali, spazi, numeri, etc.

Sono definite utilizzando **doppio apice (*quotation marks*) o singolo apice (*single quotes*)**

- `hi = "hello there"`
- `hi2 = 'hello there again'`

E' possibile Concatenare stringhe

- `name = "gigi"`
- `greet = hi + name`
- `greeting = hi + " " + name`

Eseguire **operazioni** su stringhe come definito nella documentazione Python

- `silly = hi + " " + name * 3` → `hi there MarioMarioMario`

Stringhe

Pensare alle stringhe come una **sequenza** case-sensitive di caratteri

Possiamo comparare stringhe attraverso gli operatori ==, >, < etc.

`len()` è una funzione utilizzata per ottenere la **lunghezza** di una stringa passata come primo argomento

```
s = "abc"
len(s)  →  evaluates to 3
```

Stringhe

Le parentesi quadre sono usate per **indizzare** una stringa e accedere ad un suo valore ad una certa posizione/indice

```
s = "abc"
```

index: 0 1 2 ← indexing always starts at 0
index: -3 -2 -1 ← last element always at index -1

```
s[0]   → evaluates to "a"
s[1]   → evaluates to "b"
s[2]   → evaluates to "c"
s[3]   → trying to index out of bounds, error
s[-1]  → evaluates to "c"
s[-2]  → evaluates to "b"
s[-3]  → evaluates to "a"
```

Stringhe

Possiamo affettare (**slice**) le stringhe utilizzando la notazione `[start:stop:step]`

- se `step` viene omesso, [`start:stop`], vale 1 di default

Possiamo anche utilizzare solo la notazione ::

```
s = "abcdefgh"
```

`s[3:6]` → evaluates to `"def"`, same as `s[3:6:1]`

`s[3:6:2]` → evaluates to `"df"`

If unsure what some command does, try it out in your console!

`s[::]` → evaluates to `"abcdefgh"`, same as `s[0:len(s):1]`

`s[::-1]` → evaluates to `"hgfedbca"`, same as `s[-1:-(len(s)+1):-1]`

`s[4:1:-2]` → evaluates to `"ec"`

Stringhe

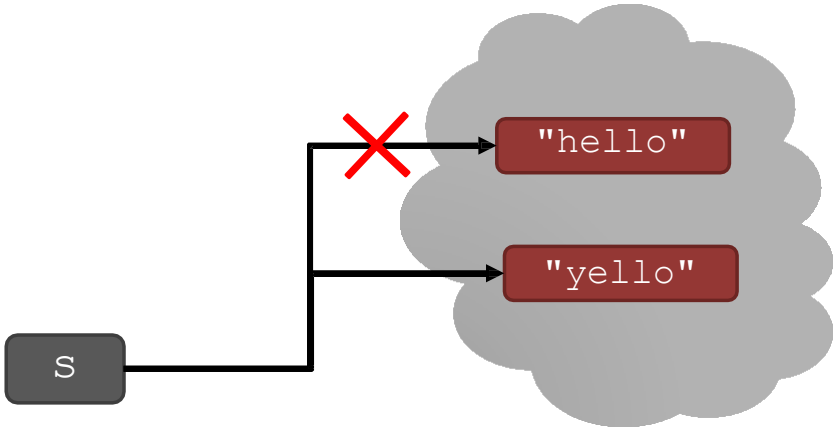
Le stringhe sono "**immutabili**" – non possono essere modificate

```
s = "hello"

s[0] = 'y'
s = 'y'+s[1:len(s)]
```

→ gives an error

→ is allowed,
s bound to new object



INPUT/OUTPUT: `print`

Utilizzato per effettuare **output** di cose sulla console

La parola chiave è `print`

```
x = 1
print(x)
x_str = str(x)
print("my fav num is", x, ".", "x =", x)
print("my fav num is " + x_str + ". " + "x = " + x_str)
```

INPUT/OUTPUT: `input("")`

Il comando **`input`** effettua la stampa di qualunque cosa ci sia tra doppi apici e attende che l'utente inserisce qualcosa da tastiera.

Una volta che digita INVIO, il *bind* avviene tra il valore inserito da tastiera e la variabile che viene assegnata tramite **`input`**

```
text = input("Type anything... ")
print(5*text)
```

`input` **restituisce una stringa**, quindi è necessario il cast se dobbiamo operare per esempio con numeri

```
num = int(input("Type a number... "))
print(5*num)
```

Operatori di comparazione tra `int`, `float`, `string`

Assumiamo `i` e `j` come nomi di variabile

Le comparazioni di seguito restituiscono un booleano (True, False)

`i > j`

`i >= j`

`i < j`

`i <= j`

`i == j` → **equality** test, `True` se il valore di `i` è lo stesso di `j`

`i != j` → **inequality** test, `True` se il valore di `i` non è lo stesso di `j`

Operatori logici tra booleani

Assumiamo `a` e `b` nomi di variabili booleane

`not a` → `True` se `a` è `False` `False` se `a` è `True`

`a and b` → `True` se entrambi sono `True`

`a or b` → `True` se una delle due è `True`

A	B	A and B	A or B
True	True	True	True
True	False	False	True
False	True	False	True
False	False	False	False

Esempi di comparazione

```
pset_time = 15
sleep_time = 8
print(sleep_time > pset_time)

derive = True
drink = False
both = drink and derive
print(both)
```

Flusso di controllo - Branching

```
if <condition>:
    <expression>
    <expression>
    ...
```

```
if <condition>:
    <expression>
    <expression>
    ...
else:
    <expression>
    <expression>
    ...
```

```
if <condition>:
    <expression>
    <expression>
    ...
elif <condition>:
    <expression>
    <expression>
    ...
else:
    <expression>
    <expression>
    ...
```

- `<condition>` può essere `True` o `False`
- Si valuta l'espressione nel blocco se `<condition>` è `True`

Indentazione

- **L'indentazione in Python conta!**
- Viene utilizzata per definire un blocco di istruzioni

```
x = float(input("Enter a number for x: "))
y = float(input("Enter a number for y: "))
if x == y:
    print("x and y are equal")
    if y != 0:
        print("therefore, x / y is", x/y)
elif x < y:
    print("x is smaller")
else:
    print("y is smaller")
print("thanks!")
```

Operatore = vs Operatore ==

```
x = float(input("Enter a number for x: "))
y = float(input("Enter a number for y: "))
if x == y:
    print("x and y are equal")
    if y != 0:
        print("therefore, x / y is", x/y)
elif x < y:
    print("x is smaller")
else:
    print("y is smaller")
print("thanks!")
```

What if x = y here?

get a SyntaxError

- Legend of Zelda – Lost Woods
- keep going right, takes you back to this same screen, stuck in a loop

```
if <exit right>:
    <set background to woods_background
    if <exit right>
        <set background to woods_background
         if <exit right>
             set background to woods_background
             nd so on and on and on..
         else
             set background to exit_background
     else:
        <set background to exit_background
else:
    <set background to exit_background
```


35

- Legend of Zelda – Lost Woods
- keep going right, takes you back to this same screen, stuck in a loop


```
while <exit right>:
    <set background to woods_background
<set background to exit_background>
```