INTRODUZIONE ED ELEMENTI DI PROGRAMMAZIONE

Chi sono

- Umberto Ferrero
  - Ingegnere gestionale
  - 5 anni in consulenza ICT
  - Studente di ingegneria informatica
  - Appassionato di nuove tecnologie
  - Curioso
- Formatore - DiaSofia:
  - Siti Web: diasofia.com
  - Instagram: @diasofiadiasofia

Introduzione

- Python è uno dei linguaggi di programmazione più «semplici» e allo stesso tempo più utilizzati, prestandosi bene come punto di partenza per la programmazione
- Tramite Python verrà effettuato un approccio verso la programmazione: tipi di dato, condizioni, cicli, oggetti, funzioni… con tutto quanto necessario per comprenderli appieno

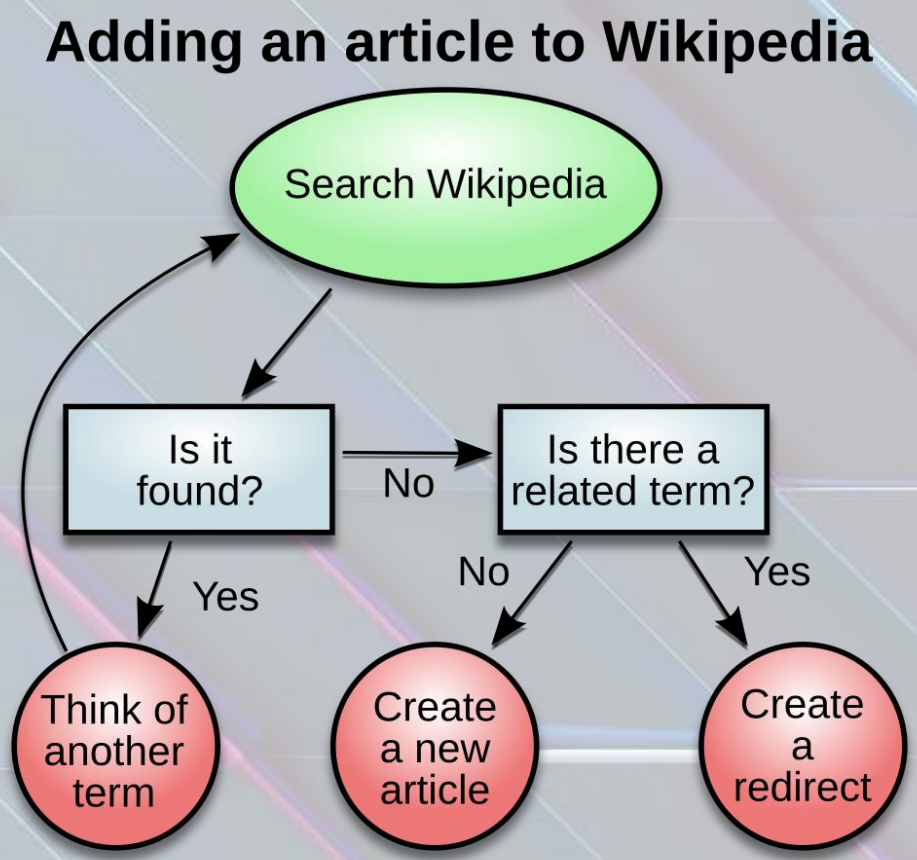
ALCUNE BASI DI LOGICA E PROGRAMMAZIONE

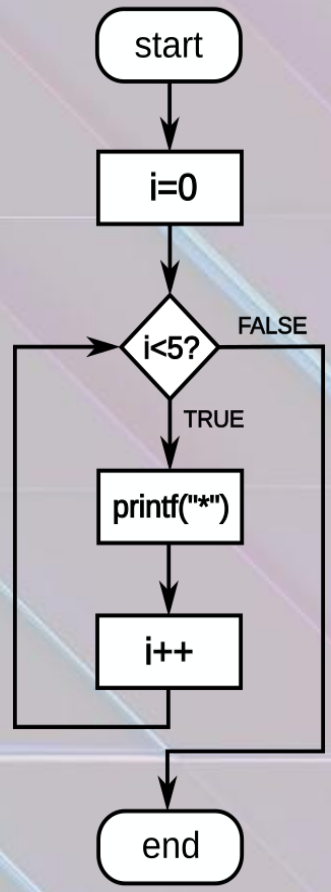
Algoritmo

«Procedimento descrivibile con un numero finito di regole che conduce al risultato dopo un numero finito di operazioni»

- Le regole devono essere chiare e univoche
- Il risultato è deterministico
- Il risultato deve essere raggiungibile
- Si può controllare il risultato per riformulare l'algoritmo e arrivare al risultato atteso

Flow Chart – Diagramma di flusso





Flow Chart – Diagramma di flusso

Importanza della pianificazione:

prima di scrivere del codice bisogna aver chiaro l’obiettivo, come raggiungerlo, con quali strumenti, quali passi saranno necessari.

Solo una volta che si hanno le idee chiare è possibile «tradurre» l’idea in codice.

Algoritmo

È semplice aprire una porta?

Scrivere su un foglio le istruzioni dettagliate per arrivare all'obiettivo

Il punto fondamentale è che tutte le istruzioni devono essere ridotte ad operazioni elementari

Exact Instructions Challenge - Josh Darnit

Connettivi logici

Un connettivo logico o operatore logico (anche «operatore booleano»), è un elemento di collegamento che instaura fra due proposizioni A e B una qualche relazione che dia origine ad una terza proposizione C con un valore vero o falso, in base ai valori delle due proposizioni fattori ed al carattere del connettivo utilizzato.

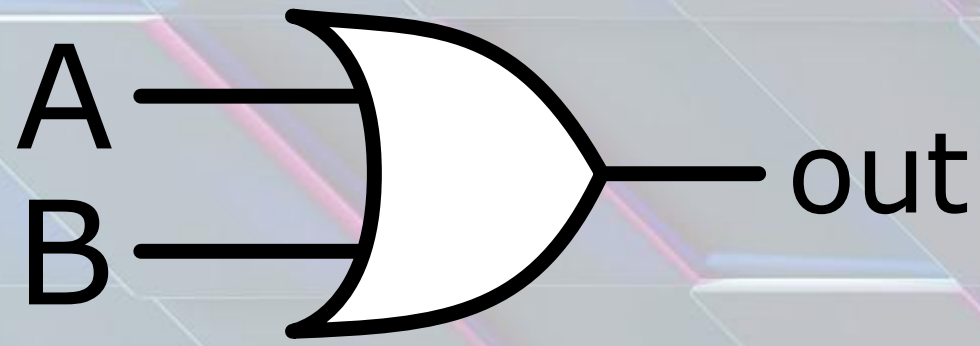
Condizioni e tabelle di verità

Disgiunzione inclusiva «o»

… in latino «vel»

… in logica booleana «**OR**»

… indicata con il simbolo «v»



A	B	A v B
V	V	**V**
V	F	**V**
F	V	**V**
F	F	**F**

Condizioni e tabelle di verità

La congiunzione logica «e»

… in latino «et»

… in logica booleana «**AND**»

… indicata con il simbolo «∧»

A	B	A ∧ B
V	V	**V**
V	F	**F**
F	V	**F**
F	F	**F**

Vero e Falso

In Python ci sono i valori booleani «True» e «False».

Anche per le altre variabili, meglio spiegate nelle successive slide, ci sono dei valori che possono essere letti come «False»:

- Il numero “0” o “0.0”
- Una stringa vuota (“”)
- “{}”, “[]”, “()

Gli altri valori sono considerati «True»

PYTHON

Sintassi, Variabili, I/O

Cos’è Python?

- Un linguaggio di programmazione
- Ad alto livello
  - Semplice
- Orientato agli oggetti (ma flessibile)
- Ottimo per lo scripting
  - Machine Learning

Sintassi e indentazione

- Sintassi, «regole di scrittura»: istruzioni, parole chiave, come scriverle
- Una particolarità di Python è che per sapere come leggere le istruzioni utilizza l'indentazione, a differenza della maggior parte degli altri linguaggi che utilizza parentesi e/o punti e virgola e/o altro
- #commenti
  molto importanti per capirci qualcosa, soprattutto per sé stessi

```
a = 10
b = 5
if b > a:
  print("b è maggiore di a")
elif a == b:
  print("a e b sono uguali")
else:      #altrimenti
  print("a è maggiore di b")
```

Variabili

Dati contenuti in una porzione di memoria (o il riferimento ai dati stessi) (pensiamo alle aule)

Modificabili durante l’esecuzione di un programma

Generalmente identificate da un nome e un tipo

- Numeri: int, float
- Boolean: True/False
- «Testo»: Stringhe, liste
- Oggetti: strutture più complesse
- Costanti «non variabili»? (maiuscolo, underscore)

Variabili - sintassi

- stringaDiTesto = "Hello, World"
- unIntero = 2
- varFloat = 0.1
- laMiaLista = [1, 2, 3]
- veroOFalso = True
- “None” è una variabile con lo stesso ruolo di “NULL” in C, è di un tipo non numerico

Variabili - sintassi

- Per dichiarare una variabile bisogna scriverne il nome composto da lettere, underscore, numeri (ma non per primi), seguito da uguale, seguito dal valore della variabile
- Rientra tra le naming convention («buone maniere») stilizzare il nome delle variabili in camelCase
- L'operatore di assegnazione `=` fa in modo che il nome della variabile sia uguale alla posizione di memoria dove si trova il valore

Variabili - sintassi

- A differenza di altri linguaggi, per Python non è richiesto specificare il tipo (dichiarazione)…
- … ma bisogna sapere cosa si sta facendo, inserire e gestire i valori corretti, sapere come leggere eventuali errori per correggere il codice

Conversioni tra numeri

```
int(x)      #restituisce il numero convertito in intero
float(x)    #restituisce il numero in floating point
type(x)     #restituisce il tipo di una variabile
```

I/O

- input() permette di ricevere un valore in input
- Nella parentesi, tra doppie virgolette, si può inserire il prompt, un testo che rende più chiaro cosa si aspetta il programma il quel momento
- La risposta dell'utente viene restituita come stringa
- Se si voleva un altro tipo di dato, sarò necessario convertirlo, ad esempio con int()
- print() permette di avere in output, sull'interfaccia scelta, l'argomento passato, o più argomenti separati da virgola, che verranno stampati con uno spazio

Un primo esempio

```
mioNumero = input("Inserisci un numero:\n")
x = int(mioNumero)
y=x+2
print("Il risultato è", y, "!")
#oppure print(f'Il risultato è {y}!’)
#quali sono le differenze?
#per le stringhe si può utilizzare sia " " che ‘ ’
#la prima soluzione è consigliata
```

IDE

Dove scrivere codice?

IDE (Integrated Development Environment)

- IDLE
- Pycharm
- Visual Studio Code

IDE «Online»

Consigliato:

- https://www.online-python.com/

Alternative:

- https://www.programiz.com/python-programming/online-compiler/
- https://www.jdoodle.com/python3-programming-online
- https://replit.com/languages/python3

STRINGHE E LISTE

Elaborazioni di stringhe

Si possono usare alcuni operatori visti per i numeri

Ad esempio per concatenazione e ripetizione:
s = "ciao" + ' ' * 5 + " mondo«

str(x) ritorna x convertito in stringa

Elaborazioni di stringhe

```
s = "ciao"
s[1]      # 1° carattere          → i
s[1:3]    # dall'1 al 3 escluso   → ia
s[2:]     # dal 2 alla fine       → ao
s[:3]     # fino al 3 escluso     → cia
s[-1]     # l'ultimo carattere    → o
s[:-2]    # fino al penultimo     → ci
```

c	i	a	o
0	1	2	3
-4	-3	-2	-1

Notato qualcosa di particolare per gli indici?

Elaborazioni di stringhe

- s.split([sep[, max]])
  ritorna una lista contenente le parti di "s" divise usando "sep" come separatore, di default "sep" è " "
- " ".join(iterable)
  ritorna una stringa che unisce gli elementi di "iterable" usando la stringa prima del punto come delimitatore
- s.lower() ritorna "s" tutta in minuscolo
- s.upper() ritorna "s" tutta in maiuscolo

Elaborazioni di stringhe

- s.find(what[, start[, end]])
  ritorna il primo indice di "what" in "s[start, end]", mentre se non trova niente ritorna -1
- s.rfind(what[, start[, end]])
  come "find" ma a partire dal fondo (perché esiste?)
- s.replace(old, new)
  ritorna una copia di "s" sostituendo le occorrenze di "old" con "new"

Liste

- Sono dei vettori, degli array
- Una lista è delimitata da parentesi quadre
  l = []
- Gli elementi della lista sono delimitati da virgole
  [1, 2, 3]
- Possono potenzialmente contenere elementi eterogenei
  [1, 2, "ciao", 10.0]
- for x in l:
  print (type(x))

Liste

Sono utilizzabili molti degli operatori delle stringhe:

```
[1] + [3, 5]          → [1, 3, 5]
[1, 0] * 3            → [1, 0, 1, 0, 1, 0]
[2, 3, 6, 7][1:3]     → [3, 6]
[2, 3, 6, 7][:2]      → [2, 3]
```

A differenza delle stringhe, è possibile mutare un elemento, utilizzando l'indice appropriato:

```
[1, 2, 3][0] = 4      → [4, 2, 3]
```

Liste - Metodi

Alcuni metodi utili per elaborare stringhe:

- l.append(obj)
  aggiunge in fondo la variabile “obj”
- l.extend(list)
  aggiunge in fondo gli elementi di “list”
- l.insert(index, obj)
  aggiunge “obj” prima di “index”
- ATTENZIONE: le funzioni non restituiscono una nuova lista, ma modifica l’originale
- Viceversa l’operatore + crea una nuova lista
  m = l+[4]

Liste - Metodi

Alcuni metodi utili per elaborare stringhe:

- l.pop([index])
  rimuove l'elemento in posizione "index" e lo ritorna, il valore di default di "index" è -1
- l.remove(value)
  cancella la prima occorrenza di "value"
- l.count(x)
  conta quante volte x appare nella lista
- l.index(x)
  restituisce l'indice del valore x
- ATTENZIONE: le funzioni non restituiscono una nuova lista, ma modifica l'originale

Liste - Metodi

Alcuni metodi utili per elaborare stringhe:

- l.reverse()
  inverte l'ordine degli elementi della lista
- l.sort([reverse=True[, key=myFunc])
  ordina la lista «ascending», se con reverse = True in ordine «Z-a», «descending».
  Si può in parallelo definire una funzione myFunc con regole su cui ordinare
- ATTENZIONE: le funzioni non restituiscono una nuova lista, ma modifica l'originale

Un caso particolare

```
x = [0, 1, 2]
y = x
x.append(3)
print (y)                    #[0, 1, 2, 3]
```

Possibile soluzione:

```
y = x.copy()
```

Ce ne sono altre? Pensiamo alle funzioni viste…

len

```
len(seq)          #ritorna il numero di elementi in seq
len("ciao")       #4
len([1, 2, 3])    #3
```

CONDIZIONI E CICLI

Operatori e condizioni

Caratteri, simboli, utilizzati per effettuare operazioni predefinite

- ad esempio >= serve per confrontare se un valore è maggiore o uguale di un altro
- x += 1 equivale a x = x + 1
- and, or, not, == (è uguale?), != (è diverso?), in (per liste o stringhe)
- Si possono effettuare confronti concatenati in un'unica riga (in «and»):
  a < b == c
- Si può assegnare il confronto a una variabile:
  cfr = p1 or p2

Operatori e condizioni

I confronti “And” e “or” sono «cortocircuitati»:

valutati da sinistra a destra, nel caso più confronti siano concatenati, il confronto termina una volta che l’output è chiaro.

Ad esempio:
0 or True or False
al «True» è chiaro che l’output sarà «vero», e quindi termina il confronto

If… Elif… Else…

Permette al codice di valutare diverse condizioni, e procedere con l'alternativa identificata

- Se la condizione è vera si esegue l'istruzione nestata, se è falsa si continua con l'alternativa successiva
- elif, «else if» in altri linguaggi, consente di testare differenti condizioni in un unico blocco, con un'unica condizione di else
- Sia Elif che Else sono facoltativi

```
a = 10
b = 5
if b > a:
  print("b è maggiore di a")
elif a == b:
  print("a e b sono uguali")
else:
  print("a è maggiore di b")
```

For

Il ciclo for esegue le stesse istruzioni per un dato numero di volte, di iterazioni. È usato ad esempio per un elenco, una lista.

- for: keyword del ciclo
- n: una variabile, che può avere qualsiasi nome, che esplicita quante volte iterare, il counter
- in: è un operatore, booleano, che restituisce True se la variabile è in un determinato intervallo, qui definito dal range
- range(start, stop, step) è una funzione molto comoda per i cicli for, ci dice la variabile da quale valore parte, dove si deve fermare (-1) e come deve variare al termine di ogni ciclo
- Anziché range si può utilizzare una sequenza (lista, stringa ecc)

For – Sintassi

```
miaStringa = input("Inserisci un numero: ")
mioNumero = int(miaStringa)
if mioNumero > 5:
    print("Numero troppo grande per l'esempio")
else:
    for n in range(0, mioNumero, 1):
        print(mioNumero)
```

While

Il ciclo while continua ad iterare fino a che una condizione di controllo è rispettata

x è la variabile su cui iterare, ma deve essere definita prima del while

Di fianco al while è posta la variabile booleana, generalmente sotto forma di confronto

While - Sintassi

```
print("Questo è un programma che saluta fino a che non lo si ferma, scrivendo '0'\n")
x=1
while x != 0:
        x=input("Ciao!\n(Inserisci un numero)")
        if x.isdigit():                #Se non è un numero, non lo si può convertire
                x=int(x)
print ("Termine programma")
```

For e While

Quando iniziano e quando si fermano?

Dove differiscono?

Un esempio:

Come leggere una lista di interi da input utente.
Versione sintetica in una sola riga di codice.

I numeri devono essere separati da spazi. Perché?

```
mieiInteri = [int(x) for x in input("inserisci i numeri, separati da spazi").split()]
print (mieiInteri)
```

FUNZIONI E MODULI

Funzioni

- Uno degli obiettivi della programmazione è "riutilizzare il codice", ad esempio se si deve fare qualcosa spesso
- Le funzioni servono a questo: creare un blocco utile, che può essere richiamato in maniera semplice per elaborare le variabili come desiderato
- Ok, anche copiare codice, o intere funzioni, da internet ha lo stesso razionale…
- I linguaggi di programmazione hanno i loro moduli, o librerie, più o meno native

Funzioni - Sintassi

- def: si sta definendo una nuova funzione
  Ricordarsi i due punti al termine della riga!
- miaFunzione(): è il nome della funzione che si vuole creare (e in seguito richiamare). Il nome deve essere chiaro ed evocativo.
- a,b: i parametri della funzione. Non sono obbligatori, come possono essere numerosi. Dall'interno della funzione non sono visibili all'esterno.
- c: argomenti opzionali, se non sono specificati usano il default
- Il «return» statement è una dichiarazione, in teoria opzionale, che restituisce uno o più valori alla chiamata originale alla funzione

Funzioni - Sintassi

```
# Definizione della funzione:

def miaSomma(a, b, c=1):
        return a + b + c

# Chiamata della funzione:

ilRisultato = miaSomma(1,2)
```

Funzioni - Docstring

Le “Docstring” sono stringhe di testo, simili a commenti, che vengono inserite generalmente subito sotto “def”.

Queste stringhe, delimitate da un triplo doppio apice e su può righe, chiariscono meglio cosa fa la funzione e come.

A differenza dei commenti, questo testo è “standardizzato”, ed è ad esempio incluso in automatico nella documentazione del programma, o nella funzione:

```
print(print.__doc__)
print(math.__doc__)
```

Funzioni

- Nella definizione della funzione, tra parentesi ci sono i "parametri".
- Quando la funzione è richiamata, invocata, vengono definiti "argomenti", quindi i valori che assumeranno i parametri.
- Anche print() è una funzione, come tutte quelle viste in precedenza per le stringhe e le liste
- Molte delle funzioni di default possono avere dei parametri opzionali che non sempre vengono esplicitati per semplicità

Funzioni e leggibilità del codice

- Un altro motivo per cui vengono utilizzate le funzioni è per "ordine", "pulizia del codice":
  ad esempio tutte le funzioni di login e possono essere definite nella stessa area, in modo da poter poi richiamare solo «login()» quando serve, in modo molto più compatto

Import file .py

- Per questo possono essere utilizzati più file .py, tenendo un «main» per le logiche e il programma vero e proprio, e gli altri file con funzioni e logiche di supporto
- import fn #se il secondo file è fn.py
  fn.miaFunzione(x,y) #chiamo la funzione

Import moduli (librerie)

I moduli sono insiemi di funzioni «pronte all'uso», per aree tematiche.

È possibile scaricare tutto il modulo o solo alcune funzioni.

- #Es. per importare il modulo delle funzioni matematiche:
  ```
  import math
  a = math.sqrt(2)
  ```

Import moduli (librerie)

- #Oppure
  from math import sqrt
  a = sqrt(2)

- #Si possono anche rinominare le funzioni per comodità
- from math import sqrt
  rad = math.sqrt
  a = rad(2)

Variabili globali e locali

Variabili globali - visibili e modificabili ovunque nel programma

- Pro: sempre utilizzabili
- Contro: possono creare complessità e lentezza

Variabili locali – contenute all’interno di una specifica sezione e visibili solo lì

- Pro: migliore segregazione dei dati

Global

L’assegnamento all’interno della funzione assegna il valore ad una variabile locale, diversa rispetto a quella esterna

```
x = 5
def f():
    x = 42 # x è una nuova variabile locale
```

Viceversa, con “global” si indica all’interprete che la variabile è globale e non locale

```
x = 5
def f():
    global x
    x = 42 # x è la variabile globale, quella definita prima
```

Numero variabile di parametri

Come si fa per gestire un numero variabile di parametri?

```
def sommaLunghezzaVariabile(*args):
    somma = 0
    for arg in args:
        somma += arg
    return somma
```

Differenza con l’utilizzo di una lista come parametro?

PROGRAMMAZIONE A OGGETTI (OOP)

Programmazione procedurale

La «programmazione procedurale» consiste nel creare dei «blocchi di codice», detti anche sottoprogrammi, subroutine, procedure o funzioni.

Semplificando, è il tipo di programmazione che si utilizza per gli «script», dove ci sono step ben definiti che portano dall'input all'output.
Es: Somma di 2 numeri dati dall'utente.

Programmazione a oggetti

Uno dei punti di forza per cui l'OOP è utilizzata così tanto è l'astrazione, la possibilità di modellizzare un concetto, un qualcosa di non lineare, e di farlo a piccoli pezzi, senza dover pensare a tutto in contemporanea, semplificando il lavoro.

Questo è comodo sia per lavorare in team, dove ognuno si occupa di sezioni diverse (oggetti, procedure…), sia per rendere ancora più facile la manutenzione del codice.

Permette anche l'utilizzo di metodi pubblici che possono essere richiamati da più parti del codice, o dall'esterno.

Programmazione a oggetti

Supponiamo di scrivere un programma per giocare a carte.

- Partiremo dalle classi, che potrebbero essere «mazzo», «carta», «giocatore»
- Ci si può concentrare su come funziona ogni classe: «mazzo» avrà dei metodi come «mischia», o «dai una carta», e attributi come numeroDiCarte, carteRimanenti.
- Se alla fine ci si rende conto che può essere utile aggiungere un nuovo metodo, come «tagliaMazzo», lo si potrà aggiungere senza «spaccare» codice precedente, anzi, probabilmente sfruttando funzionalità già esistenti.

Programmazione a oggetti

Consente la modularità e il riutilizzo del codice visto nelle funzioni, espandendo ulteriormente le potenzialità

Similmente una classe «utile» e/o «generale» potrà essere riutilizzata in altri progetti, pensiamo ad esempio a login(user, password), o a librerie dedicate, anche per gli oggetti

Programmazione a oggetti

Un paio di meccanismi propri dei linguaggi a oggetti:

- Astrazione: processo di identificazione degli aspetti essenziali di una forma o concetto, astratti dalle specifiche realizzazioni. Qui significa che i programmatori possono lavorare con concetti ad alto livello senza preoccuparsi dei dettagli.
- Incapsulamento: separazione dell'interfaccia di una classe dalla corrispondente implementazione, in modo che i client di un oggetto di quella classe possano utilizzare la prima, ma non la seconda.

Classe

La classe è un insieme di variabili e funzioni proprie, che definisce un «oggetto»

Class car

Attributes:
model, colour

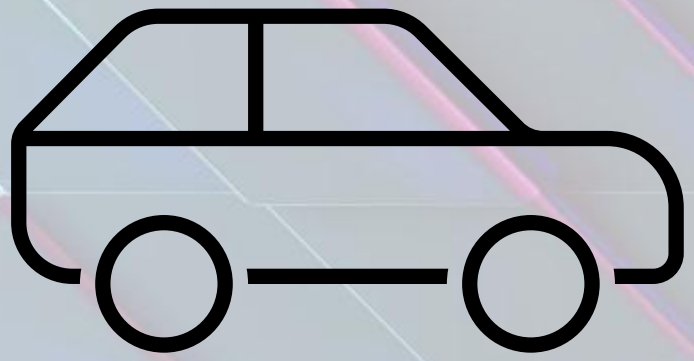
Methods:
getModel(), getColour(), paint(c), getPosition(), repaint(colour)

Oggetto vs Classe

La classe è il «modello astratto», che possiede attributi (variabili) e metodi (funzioni proprie degli oggetti).

Gli oggetti creati a partire dalla classe sono le istanze, i «casi reali», che avranno tutte le proprietà della classe, ma declinate in maniera diversa.

Un’auto può essere di diversi modelli o diversi colori.



```
c1 = car("500", "red")
c2 = car("panda", "white")
```

Metodi

I metodi sono funzioni proprie degli oggetti.

In questo senso, di dichiarano con una normale def, che sia però indentata sotto la classe, con i parametri necessari tra parentesi, e un eventuale return.

I metodi sono applicati all'oggetto riportato prima del punto, con i parametri necessari tra parentesi

Es: def shuffle(self, numberOfplayers)

…

deck1.shuffle(4)

Constructor __init__

Constructor:

def __init__ (self, *attributi*)

__init__ è un particolare metodo, necessario per ogni classe, per creare, inizializzare, istanziare, un nuovo oggetto. Generalmente ha come parametri i propri attributi «base», per poter definire l'oggetto stesso.

Classe - sintassi

```
class car:
    """è una macchina"""                          #docstring
    def __init__ (self, model ="boh", colour = "boh"):
    #inizializzazione, in questo caso con valori di default
        self.model = model                        #attributi
        self.colour = colour                      #attributi


    def ciao(self):                               #metodo della classe
        """restituisce una stringa con gli attributi dell'oggetto"""
        return "ciao!\n" +"Modello: "+ self.model +"\nColore: " \
        + self.colour
```

«Self»

Self serve come parametro «occulto» per «passare se stesso».

Così facendo i metodi conosceranno sempre l’oggetto passato e i relativi attributi.

Viceversa all’interno dei metodi stessi, così come dell’init, self.qualcosa è utilizzato per esplicitare che si sta parlando dell’attributo dell’oggetto, e non del parametro passato nel metodo.

Semplificando: self rappresenta l’oggetto.

Oggetti - sintassi

```
from classi import car
#oppure impor classi, e poi c1=classi.car(…

c1 = car(colour="red")          #creo l’oggetto c1
c2 = car("panda", "white")      #la classe è car

print(c1.ciao())                #richiamo il metodo ciao()
```

Setter e Getter vs variabili pubbliche

Le variabili degli oggetti sono private, ed è meglio lo siano.
Si usano quindi setter e getter per poterle cambiare o leggere

```
def setModel(self, model):      #in class.py, “sotto” class car
        self.model = model

def getModel(self):              #in class.py, “sotto” class car
        return self.model

c1.setModel("500")               #in main.py
print(c1.ciao())
```

Metodi della classe

Altri metodi che si potrebbero usare per la classe car?

GESTIONE FILE

File handling

Su Python, così come in ogni linguaggio di programmazione, è possibile leggere e scrivere file.

Per «partire dalle basi», si gestiscono dei semplici file .txt

Per chi volesse approfondire, esistono librerie dedicate: una molto popolare è pandas, che può essere usata dalla lettura e scrittura di csv e fogli elettronici, fino ad arrivare a data analysis avanzata.

File - open

open(filename, mode)
apre "filename" in modalità "mode" (default: "r") e ritorna un oggetto di tipo "file" (quindi va assegnato a una variabile)

close() chiude il file

Per aprire un file si può usare, ad esempio:

- "r": Read, per leggere il file
- "a": Append, per aggiungere qualcosa al file
- "w": Write, per scrivere sul file, sovrascrivendolo

File – principali funzioni - "r"

Attenzione: per praticamente tutti i metodi seguenti, la lettura parte dal carattere successivo a quanto letto in precedenza.

read() legge (e ritorna) tutto il file

readline() ritorna la riga successiva

readlines() ritorna una lista con le righe rimanenti nel file

close() chiude il file, da fare sempre al termine

File – funzioni - "w", "a"

write("str") scrive "str" sul file.

writelines(list) scrive tutti gli elementi di list sul file

Attenzione: all’apertura del file si è all’inizio. Utilizzare la funzione write sovrascriverà il contenuto presente, se lo si apre in modalità "w".

Aprire un file come "w" o "a" crea il file, se non esisteva in precedenza.

File handling - sintassi

```
f = open("filetoread.txt", "r")
print(f.read())
f.close()
#ad es. con s=f.read() si può poi gestire la stringa s


f = open("filetoread.txt", "r")
for x in f:
  print(x)
```

Tell e Seek

Sono 2 funzioni che aiutano ad «orientarsi» nel file, indicando dove è al momento il cursore (tell) o in quale posizione del file ci si vuole spostare (seek)

```
print(f.tell())          # dov'è il cursore
print(f.seek(5))         # cursore in posizione 5
print(f.seek(0,2))       # ci si posiziona al termine del file
```

Sono possibili degli offset negativi, ma è necessario aprire il file in modalità binaria.

Grazie

Contatti

Siti Web: diasofia.com, corsi.diasofia.com
Instagram: @diasofiadiasofia
Mail: info@diasofia.com

Per approfondire

Link Utili

- Link utili
  - W3schools – Python
- Argomenti in particolare
  - Funzioni