Prontuario Python

(prof. Ivaldi Giuliano)

Python è un linguaggio di programmazione ad alto livello, multi-paradigma, che supporta sia la programmazione procedurale (che fa uso delle funzioni), sia la programmazione ad oggetti. Ha ottenuto un enorme successo nelle comunità dei programmatori grazie al connubio unico tra la semplicità di apprendimento e la potenza offerta dalle sue librerie. E' completamente gratuito ed è possibile usarlo e distribuirlo senza restrizioni di copyright; ha una comunità molto attiva, e riceve costantemente miglioramenti che lo mantengono aggiornato e al passo coi tempi. Python è un linguaggio portabile sviluppato in ANSI C, è possibile usarlo su diverse piattaforme come: Unix, Linux, Windows, DOS, Macintosh, cellulari Android e iOS. Anche se Python è considerato un **linguaggio interpretato**, i programmi vengono automaticamente compilati in un formato chiamato *bytecode* prima di essere eseguiti. Python è stato sviluppato ad Amsterdam alla fine degli anni ottanta del Novecento da Guido van Rossum, che ha dato al linguaggio il nome della propria serie televisiva preferita *"Monty Python's Flying Circus"*.

IDLE di Python

IDLE (**I**ntegrated **D**evelopment and **L**earning **E**nvironment): IDLE offre un'interfaccia visuale per Python, e può essere eseguito in ambiente Windows, Linux o Mac OS. Avviando IDLE, apparirà una finestra con il prompt dei comandi Python (>>>), e delle voci di menù (in alto). L'IDLE di Python è un programma che permette di inserire i comandi di Python una riga alla volta. Una volta terminato di lavorare con IDLE si potrà chiudere la sessione di Python scrivendo il comando quit() o exit() e poi premendo INVIO,

es. >>>quit()

Esistono comunque dei comandi che permettono di chiudere sia la console che i programmi in esecuzione in maniera brusca.

CTRL+Z per interrompere l'esecuzione della console

CTRL+C per interrompere l'esecuzione di un programma in Python, molto utile nei casi in cui un programma in Python vada in loop o impieghi un tempo troppo elevato per poter attendere il risultato.

Come creare un file Python

Dalla console IDLE di Python cliccare sul menù File -> New File -> Verrà creato un nuovo file

Salvare il file con estensione .py cliccando su File -> Save as -> Inserire il nome del file e cliccare su Salva

Eseguire il programma appena scritto -> Cliccare sul menù Run -> Run Module (il programma verrà eseguito in IDLE)

Commenti

= riga di commento

Indentazione

In Python è necessario indentare (incolonnare) correttamente le istruzioni per livelli, altrimenti viene segnalato un errore (l'indentazione si usa al posto delle parentesi graffe, utilizzate invece ad esempio in C, C++, Java).

Convenzioni,

- usare sempre 4 spazi per livello di indentazione;
- evitare l'uso dei caratteri di tabulazione;
- non mischiare mai l'uso di tabulazioni e spazi.

Python è un linguaggio di programmazione **case sensitive**, cioè fa differenza fra lettere maiuscole e lettere minuscole

Operatori aritmetici

Operatori (=, +, -, *, /, quoziente intero //, resto divisione %, elevamento a potenza **, radice quadrata math.sqrt() -> import math)

Gli spazi inseriti tra gli operatori e gli operandi non sono obbligatori ma se si vuole seguire gli standard di programmazione per Python è bene farlo sempre.

Forme contratte, +=, -=, *=, /=

Es. x = x + 5 si può scrivere -> x += 5 x = x - 2 si può scrivere -> x -= 2

x = x * 3 si può scrivere -> x *= 3 x = x / 4 si può scrivere -> x /= 4

Operatori di confronto

==, uguale !=, diverso >, maggiore <, minore, >=, maggiore o uguale <=, minore o uguale

Operatori booleani

and or not

In Python ogni oggetto è o *vero* (numeri diversi da 0, la costante True, o contenitori che contengono almeno un elemento) o *falso* (ovvero il numero 0, le costanti False e None, contenitori vuoti). È possibile verificare se un oggetto è *vero* o *falso* usando bool(oggetto).

Operatori binari

x << n	esegue uno shift a sinistra di n posizioni dei bit di x
x >> n	esegue uno shift a destra di n posizioni dei bit di x
x & y	esegue un and tra i bit di x e di y
x | y	esegue un or tra i bit di x e di y
x ^ y	esegue un or esclusivo tra i bit di x e di y
~x	inverte i bit di x

Stringhe

Si racchiudono fra apici (' ') o fra virgolette (" "): il delimitatore utilizzato per determinare l'inizio di una stringa deve essere utilizzato anche per indicarne il suo termine (es. stringa = "Ciao", oppure, stringa = 'Ciao').
Se all'interno di una stringa si devono usare ' o ", si può scrivere \' oppure \"
'\n' serve per mandare a capo il contenuto di una stringa.
'+' concatenazione, serve per concatenare più stringhe
' * ' ripetizione, permette di replicare una stringa (es. stringa = "Ciao" * 3, risultato, stringa = "CiaoCiaoCiao")

Variabili

In Python per dichiarare una variabile, basta assegnarle un valore con l'operatore '=', non è necessario specificarne il tipo

Es. nome_variabile = valore
a = 5 stringa = "Ciao"

Ogni nome di variabile deve iniziare con una lettera o con il carattere underscore "_", e può essere seguita da lettere, numeri, o underscore.

Input e Output

Output, print ("messaggio", var1, var2, ..., varN)

Es,
```
a = 3
b = 4
somma = a + b
print("La somma vale: ", somma)
```

Input, variabile = input("messaggio")

Es,
```
nome = input("Inserire il proprio nome: " )
print("Ciao ", nome)

a = int( input("Inserire il primo numero: ") )
b = int( input("Inserire il secondo numero numero: ") )
somma = a + b
print("La somma vale: ", somma)
```

La funzione input() restituisce una stringa, se si desidera inserire un numero intero, bisogna convertire l'input in un valore intero con la funzione **int()**, stessa cosa se si desidera inserire un numero con la virgola, con la funzione **float()**

Selezione

Selezione semplice (if-then-else)

Sintassi,
```
if (condizione):              # se la condizione è vera...
   ...istruzioni allora...
else:
   ...istruzioni altrimenti...
```

Le '...istruzioni allora...' e le '...istruzioni altrimenti...' devono essere indentate (incolonnate) come indicato altrimenti verrà segnalato un errore (vedere paragrafo dedicato all'indentazione)

Es.
```
if (var == 5):
    print("var è uguale a 5")
else:
    print("var è diverso da 5")
```

Selezione multipla (switch-case)

Si può realizzare con **if-elif-else**: nel caso un 'if' sia seguito da più 'else' a loro volta seguiti da altri 'if', si può utilizzare la parola chiave 'elif (condizione):' (elif = else if)

Sintassi:
```
if (condizione1):
   # gruppo di istruzioni eseguite se la condizione1 è vera
elif (condizione2):
   # gruppo di istruzioni eseguite se la condizione2 è vera
elif (condizioneN):
   # gruppo di istruzioni eseguite se la condizioneN è vera
else:
   # gruppo di istruzioni eseguite se tutte le condizioni sono false
```

Es.
```
if (n == 0):
   print('zero')
elif (n == 1 or n == 2):
   print('uno o due')
elif (n == 3):
   print('tre')
else:
   print('numero diverso da 0, 1, 2, 3')
```

Si ricorda di rispettare l'indentazione (incolonnamento) delle istruzioni se non si vuole incorrere in un errore

Dalla versione Python 3.10 in poi è stata introdotta una struttura apposita "**match-case**" (simile allo switch-case dei linguaggi C-like).

Sintassi:

```
match nome_variabile:
   case valore1:
      # istruzioni caso1
   case valore2:
      # istruzioni caso2
   ...
   case valoreN:
      # istruzioni casoN
   case default:
      # istruzioni caso default
```

dove nome_variabile è una variabile che va a confrontarsi con i diversi casi (case) e valore1, valore2, valoreN sono i valori di confronto con nome_variabile. "case default" è un'opzione che si attiva se il valore di nome_variabile non compare fra i vari "case".

Es

```
# Programma che realizza un semplice menù con la selezione multipla

# inizio programma

print("\nMenu'")
print("-----")
print("1) Scelta 1")
print("2) Scelta 2")
print("3) Scelta 3")
print("4) Scelta 4")
print("5) Scelta 5")
scelta = input("Eseguire una scelta --> ")

match scelta:
   case '1':     # se scelta è stata dichiarata come una variabile intera, i valori delle condizioni del 'case' non vanno fra apici
      print("\nE' stata eseguita la Scelta 1")
   case '2':
      print("\nE' stata eseguita la Scelta 2")
   case '3':
      print("\nE' stata eseguita la Scelta 3")
   case '4':
      print("\nE' stata eseguita la Scelta 4")
   case '5':
      print("\nE' stata eseguita la Scelta 5")
   case default:
      print("\nScelta errata. Operazione non consentita")

# fine programma


# Si ricorda di rispettare l'indentazione (incolonnamento) delle istruzioni se non si vuole incorrere in un errore
```

Cicli

Ciclo while

```
while (condizione):
   ...istruzioni ciclo while...
```

Es.
```
contatore = 0
while (contatore<5):
   print("N = ",contatore)
   contatore = contatore + 1
```
visualizza i numeri da 0 a 4, 5 = valore finale, è escluso

Ciclo for

```
for nome_variabile in range(valore_iniziale, valore_finale, incremento/decremento):
   ...istruzioni ciclo for...
```

Es.
```
for contatore in range(0,5,1):
   print("N = ",contatore)
```
visualizza i numeri da 0 a 4, 5 = valore finale, è escluso

```
for contatore in range(5,0,-2):
   print("N = ",contatore)
```
visualizza i numeri 5, 3, 1, perché il decremento è -2, 0 = valore finale, è escluso

Si ricorda di rispettare l'indentazione (incolonnamento) delle istruzioni se non si vuole incorrere in un errore

Funzioni e Procedure

Sintassi,
```
# definizione funzione
def nome_funzione(nome_parametro1, nome_parametro2, ..., nome_parametroN):
   ...istruzioni funzione...
   return valore1, valore2, ..., valoreN

# chiamata funzione
variabile1, variabile2, ..., variabileN = nome_funzione(valore_parametro1, valore_parametro2, ..., valore_parametroN)
```

Es1.
```
# funzione che calcola la somma di due numeri presi in input
def somma_due_numeri(num1, num2):
   somma = num1 + num2
   return somma

a = int(input("Inserire il primo numero: "))
b = int(input("Inserire il secondo numero: "))
risultato = somma_due_numeri(a, b)
print("Il risultato é: ", risultato)
```

Es2.
```
# funzione che calcola il punto medio fra due punti
def punto_medio(x1, y1, x2, y2):
   """Ritorna il punto medio fra (x1; y1) e (x2; y2)."""
   xm = (x1 + x2) / 2
   ym = (y1 + y2) / 2
   return xm, ym

x, y = punto_medio(2, 4, 8, 12)
print("x_medio = ", x)
print("y_medio = ", y)
```

Il valore ritornato dalla funzione è sempre uno: quella che si chiama, una singola tupla di 2 elementi; Python poi supporta un'operazione chiamata *unpacking*, che permette di assegnare contemporaneamente diversi valori a più variabili, permettendo quindi operazioni come la seguente: x, y = punto_medio(2, 4, 8, 12)
In tal modo, è possibile assegnare il primo valore della tupla a x e il secondo a y.

Le tre virgolette (""" = docstrings) prima e dopo il messaggio scritto dopo il comando 'def', servono per descrivere la funzione e permettono di richiamare il commento esplicativo della funzione stessa con il comando help(nome_funzione)

Es. se digito -> help(punto_medio) nell'IDLE otterrò...
```
Help on function punto_medio in module __main__:
    punto_medio(x1, y1, x2, y2)
        Ritorna il punto medio fra (x1; y1) e (x2; y2).
```

Sintassi, # definizione procedura
la sintassi per la creazione di una procedura è la stessa di quella di una funzione, basta omettere la parola chiave 'return', è la funzione non restituirà alcun valore (in realtà viene restituito il valore 'None')

Si ricorda di rispettare l'indentazione (incolonnamento) delle istruzioni se non si vuole incorrere in un errore

Liste (Vettori)

In Python le liste sono un tipo di oggetto; così come le stringhe e le tuple, anche le liste sono un tipo di *sequenza*, e supportano quindi le operazioni comuni a tutte le sequenze, come indexing, slicing, contenimento, concatenazione (+), e ripetizione (*) (vedere paragrafo sulle stringhe)

sintassi,

```
nome_vettore = []                  dichiarazione di una lista vuota
nome_vettore[indice]               leggere un elemento della lista,   l'indice delle liste parte da zero
print (nome_vettore)               visualizza l'intero contenuto della lista
```

Es1.

```
vettore = []                                dichiara un vettore vuoto
vettore = [0] * 100                         dichiara un vettore di 100 elementi inizializzati a [0]
lista = [1, 1, 2, 3, 5, 8, 13, 21]          dichiara un vettore dandogli dei valori iniziali
a = lista[2]                                si riferisce alla elemento di valore 2 dell'esempio precedente
lista[2] = 0                                assegna 0 all'elemento di posizione 2 dell'esempio precedente

matrice = [[1,2,3],[4,5,6],[7,8,9]]         per definire una matrice si definisce una lista di liste
matrice[1][2]    ->    6                    per accedere ad un elemento della matrice, il primo indice è riferito alla
                                            lista generale, mentre il secondo è riferito alla lista interna
```

Es2. Cercare un elemento in una lista

```
a = int(input("Inserire un valore intero: "))
lista = [0,1,2,3,4]
if a in lista:
   print("Elemento trovato")
else:
   print("Elemento non trovato")
```

Le liste supportano anche funzioni e metodi comuni alle altre sequenze: len() per contare gli elementi, min() e max() per trovare l'elemento più piccolo/grande (a patto che i tipi degli elementi siano comparabili), nome_lista.index() per trovare l'indice di un elemento, e nome_lista.count() per contare quante volte un elemento è presente nella lista:

Es.

```
lettere = ['a', 'b', 'c', 'b', 'a']
lunghezza = len(lettere)    -> 5        # numero di elementi
minore = min(lettere)       -> 'a'      # elemento più piccolo (alfabeticamente nel caso di stringhe)
maggiore = max(lettere)     -> 'c'      # elemento più grande
indice = lettere.index('c') -> 2        # indice dell'elemento 'c', nel caso di più 'c', restituisce il primo trovato
occorrenze1 = lettere.count('c')  -> 1  # numero di occorrenze di 'c'
occorrenze2 = lettere.count('b')  -> 2  # numero di occorrenze di 'b'
```

A differenza di tuple e stringhe che sono immutabili, le liste possono essere mutate. È quindi possibile assegnare un nuovo valore agli elementi, rimuovere elementi usando la parola chiave 'del', o cambiare gli elementi usando uno dei metodi aggiuntivi delle liste:

- lista.append(elem) aggiunge elem alla fine della lista;
- lista.extend(seq) estende la lista aggiungendo alla fine gli elementi di seq;
- lista.insert(indice, elem) aggiunge elem alla lista in posizione indice, spostando a destra tutti gli elementi successivi;
- lista.pop() rimuove e restituisce l'ultimo elemento della lista;
- lista.remove(elem) trova e rimuove elem dalla lista;
- lista.sort() ordina gli elementi della lista dal più piccolo al più grande;
- lista.reverse() inverte l'ordine degli elementi della lista;
- lista.copy() crea e restituisce una copia della lista;
- lista.clear() rimuove tutti gli elementi della lista;

Es.

```
lettere = ['a', 'b', 'c']
lettere.append('d')          # aggiunge 'd' alla fine                                  ->  lettere = ['a', 'b', 'c', 'd']
lettere.extend(['e', 'f'])   # aggiunge 'e' e 'f' alla fine                           ->  lettere = ['a', 'b', 'c', 'd', 'e', 'f']
lettere.append(['e', 'f'])   # aggiunge la lista come elemento alla fine               ->  lettere = ['a', 'b', 'c', 'd', 'e', 'f', ['e', 'f']]
lettere.pop()                # rimuove e ritorna l'ultimo elemento (la lista)          ->  ['e', 'f']
lettere.pop()                # rimuove e ritorna l'ultimo elemento ('f')               ->  'f'
lettere.pop(0)               # rimuove e ritorna l'elemento in posizione 0 ('a')       ->  'a'
lettere.remove('d')          # rimuove l'elemento 'd'                                  ->  lettere = ['b', 'c', 'e']
lettere.reverse()            # inverte l'ordine "sul posto" e non ritorna niente       ->  lettere = ['e', 'c', 'b']
lettere[1] = 'x'             # sostituisce l'elemento in posizione 1 ('c') con 'x'     ->  lettere = ['e', 'x', 'b']
del lettere[1]               # rimuove l'elemento in posizione 1 ('x')                 ->  lettere = ['e', 'b']
lettere.clear()              # rimuove tutti gli elementi rimasti                      ->  lettere = []
```

Stringhe

Una stringa si può considerare come una lista di caratteri immutabile, non si può modificare il contenuto di una stringa: i metodi che operano sulle stringhe, sono caratterizzati dal fatto di non modificare la stringa su cui vengono applicati, ma di ritornarne una nuova.

nome_stringa[indice] questa istruzione restituirà l'elemento in posizione "indice", l'indice parte da zero

indici negativi partono dalla fine della stringa (l'ultimo elemento ha indice -1, il penultimo -2, ecc.)

Questa operazione è chiamata **indexing**.

```
Es.  stringa1 = "Python"
     stringa2 = stringa1[0]      # elemento in posizione 0 (il primo)          ->  'P'
     stringa2 = stringa1[5]      # elemento in posizione 5 (il sesto)          ->  'n'
     stringa2 = stringa1[-1]     # elemento in posizione -1 (l'ultimo)         ->  'n'
     stringa2 = stringa1[-4]     # elemento in posizione -4 (il quartultimo)   ->  't'
```

La sintassi nome_stringa[inizio:fine] permette di ottenere una nuova stringa che include tutti gli elementi partendo dall'indice inizio (incluso) all'indice fine (escluso). Se inizio è omesso, gli elementi verranno presi dall'inizio, se fine è omesso, gli elementi verranno presi fino alla fine. Questa operazione è chiamata **slicing** (letteralmente *"affettare"*).

```
Es.  stringa1 = "Python"
     stringa2 = stringa1[0:2]    # sottostringa con elementi da 0 (incluso) a 2 (escluso) ->  "Py"
     stringa2 = stringa1[:2]     # dall'inizio all'elemento con indice 2 (escluso)     ->  "Py"
     stringa2 = stringa1[3:5]    # dall'elemento con indice 3 (incluso) a 5 (escluso)  ->  "ho"
     stringa2 = stringa1[4:]     # dall'elemento con indice 4 (incluso) alla fine      ->  "on"
     stringa2 = stringa1[-2:]    # dall'elemento con indice -2 (incluso) alla fine     ->  "on"
```

Contenimento

Gli operatori **in** e **not in** possono essere usati per verificare se un elemento fa parte di una sequenza o no. Nel caso delle stringhe, è anche possibile verificare se una sottostringa è contenuta in una stringa:

```
Es.  stringa1 = "Python"
     variabile = 'P' in stringa1       # controlla se il carattere 'P' è contenuto nella stringa s      ->  True
     variabile = 'x' in stringa1       # il carattere 'x' non è in s, quindi ritorna False              ->  False
     variabile = 'x' not in stringa1   # "not in" esegue l'operazione inversa                           ->  True
     variabile = 'Py' in stringa1      # controlla se la sottostringa 'Py' è contenuto nella stringa s  ->  True
     variabile = 'py' in stringa1      # il controllo è case-sensitive, quindi ritorna False            ->  False
```

Concatenazione

'+' serve per concatenare più stringhe

```
Es.  stringa1 = "Ciao"
     stringa2 = stringa1 + ", come stai?"      -> risultato, stringa2 = "Ciao, come stai?"
```

Ripetizione

' * ' permette di replicare una stringa (es. stringa = "Ciao" * 3, risultato, stringa = "CiaoCiaoCiao")

Lunghezza di una stringa (len)

Per ottenere la lunghezza di una stringa si può utilizzare la funzione **len(nome_stringa)**

```
Es.  stringa = "prova"
     lunghezza = len(stringa)      # lunghezza = 5
```

Convertire una stringa in maiuscolo o in minuscolo

```
Es.  stringa = 'Python'
     stringa2 = stringa.upper()   # il metodo upper ritorna una nuova stringa tutta uppercase  ->  stringa2 = 'PYTHON'
     stringa2 = stringa.lower()   # il metodo lower ritorna una nuova stringa tutta lowercase  ->  stringa2 = 'python'
```

Tuple (equivalenti alle 'struct' del C)

Rappresentano una sequenza immutabile di oggetti, in genere eterogenei.

Le tuple sono un tipo di *sequenza* (come le liste e le stringhe), e supportano le operazioni comuni a tutte le sequenze, come indexing, slicing, contenimento, concatenazione (+), ripetizione (*), len() per contare gli elementi, min() e max() per trovare l'elemento più piccolo/grande (a patto che i tipi degli elementi siano comparabili), nome_tupla.index() per trovare l'indice di un elemento, e nome_tupla.count() per contare quante volte un elemento è presente nella tupla.

Sintassi, nome_tupla = (elemento1, elemento2, ..., elementoN)

```
Es.  tupla = ( )                    # crea una tupla vuota
     tupla = ('abc', 123, 45.67)    # crea una tupla di 3 elementi
     a = tupla[0]                   # legge l'elemento 0 della tupla dell'esempio precedente,
                                    # l'indice delle tuple parte da zero
```

E' possibile creare liste di tuple (o come si direbbe in C, array di struct, vettori di strutture)

Es.

Nome	Cognome	Età
John	Smith	20
Jane	Johnson	30
Jack	Williams	28

```
lista = [('John', 'Smith', 20),
         ('Jane', 'Johnson', 30),
         ('Jack', 'Williams', 28)]
```

Dizionari (equivalenti ai 'vettori associativi' in PHP)

Sono un tipo mutabile e non ordinato che contiene elementi (*items*) formati da una chiave (*key*) e un valore (*value*). Una volta che il dizionario è creato e valorizzato con un insieme di coppie <chiave, valore>, si può usare la chiave (che deve essere univoca) per ottenere il valore corrispondente.

Sintassi,
```
nome_dizionario = { }                # crea un dizionario vuoto
nome_dizionario = { "chiave0": valore0, "Chiave1": valore1, ..., "ChiaveN": valoreN }    # definizione di un dizionario
nome_dizionario["chiave1"]           # accede all'elemento con chiave "chiave1"
```

Es.
```
prezzo = { "chiave": 12, "porta": 24,  "serratura": 18 }
print(prezzo["chiave"])              # visualizza 12
```

Le chiavi definibili all'interno di un dizionario sono immutabili, cioè una volta definite non possono cambiare di nome, mentre il loro valore corrispondente può variare liberamente.

Moduli (Librerie)

Importare un modulo (libreria) -> **import nome_modulo**

Richiamare una funzione di un modulo -> **nome_modulo.nome_funzione()**

Es1. calcolo della radice quadrata di un numero
```
import math
a = float(input("Inserire un numero: "))
radice_quadrata = math.sqrt(a)
print("La radice quadrata di ", a, " è uguale a: ", radice_quadrata)
```

Es2. generare 10 numeri casuali che vanno da 1 a 100
```
import random
for i in range(0,10,1):
    print(random.randint(1,100))
```

A volte si necessita solo di una determinata funzione di un modulo, è quindi possibile importarla usando la clausola **from**

from modulo import nomefunzione

in questo caso la chiamata della funzione non deve essere preceduta dal nome del modulo -> **nome_funzione()**

Es. nel caso del generatore di numeri casuali avremmo potuto scrivere così
```
from random import randint
for i in range(0,10,1):
   print(randint(1,100))
```
Come si può notare, in questo caso, non c'è più bisogno di specificare il nome del modulo durante la chiamata alla funzione.

Inoltre è possibile importare più funzioni (o oggetti) dallo stesso modulo, basta metterle in sequenza separate da una virgola

from modulo import nomefunzione1, nomefunzione2, nomefunzione3

Se il nome del modulo è lungo si può utilizzare un'abbreviazione usando la clausola **as**

Es. *import nome_modulo_lungo as nome_abbreviato*

e richiamando la funzione del modulo nel seguente modulo -> *nome_abbreviato.nome_funzione()*

I file con estensione '.py' possono essere trattati sia come programmi a se stanti sia come moduli da importare; all'inizio del programma principale basta scrivere import seguito dal nome del file 'libreria' senza l'estensione '.py', mentre le funzioni si possono richiamare con la sintassi, *nome_modulo.nome_funzione()*

I moduli '.py' vengono *compilati in bytecode*, un formato più efficiente che viene salvato in file con estensione '.pyc' che a loro volta vengono salvati in una cartella chiamata *__pycache__*; quando viene eseguito un programma che ha bisogno di importare un modulo, se modulo.pyc esiste già ed è aggiornato, allora Python importerà quello invece di ricompilare il modulo in bytecode ogni volta.

Gestire le 'eccezioni'

Sintassi,
```
try:
   ...istruzioni di cui si vuole gestire l'eccezione...
except tipo_eccezione:
   ...istruzioni per gestire l'eccezione...
else:
   ...istruzioni nel caso non ci siano errori
```

Es.
```
x = input("Inserire un numero: ")
try:
   n = int(x)                        # prova a convertire x in intero
except ValueError:                   # eseguito in caso di ValueError
   print('Numero non valido!')
except TypeError:                    # eseguito in caso di TypeError
   print('Tipo non valido!')
else:                                # eseguito se non ci sono errori
   print('Numero valido!')
```

La programmazione ad oggetti

Sintassi, # definizione di una classe

```
class nome_classe:
   def __init__(self, proprietà1, proprietà2, ..., proprietàN):
      self.proprietà1 = proprietà1
      self.proprietà2 = proprietà2

# istanziare un oggetto
nome_oggetto = nome_classe(valore_proprietà1, valore_proprietà2, ..., valore_proprietàN)
```

Le proprietà/attributi di un oggetto si possono richiamare con la sintassi, nome_oggetto.proprietà
I metodi di un oggetto si possono richiamare con la sintassi, nome_oggetto.metodo()

```
Es.  class Auto:                                  # definizione classe
        def __init__(self, colore, prezzo):      # metodo costruttore
           self.colore = colore                   # definizione delle proprietà
           self.prezzo = prezzo

        def compra(self):                         # metodo della classe, bisogna esplicitare il parametro 'self'
           print("Hai comprato l'auto ", self.colore)

     #programma principale
     bmw = Auto("blu",40000)       # istazia un oggetto bmw della classe Auto()
     fiat500 = Auto("bianco",20000)          # istanzia un oggetto fiat500 della classe Auto()
     print(bmw.colore)             # visualizza 'blu'
     print(fiat500.prezzo)         # visualizza '20000'
     bmw.compra()                  # il passaggio del parametro 'self' è implicito,          visualizza 'Hai comprato l'auto blu'
```

Nel caso precedente abbiamo visto che nella definizione di una classe, abbiamo definito un metodo chiamato **__init__()**, questo metodo è il più importante metodo all'interno di una classe e viene chiamato automaticamente al momento della creazione dell'oggetto (istanza della classe): gli argomenti passati durante la creazione dell'istanza vengono ricevuti da __init__
Il metodo __init__ , viene paragonato spesso al metodo **costruttore** di altri linguaggi, ma se ne differenzia perché non crea l'istanza, ma la inizializza solamente; accetta l'argomento self, e poi tutti le proprietà della classe.

Tutti i metodi all'interno di una classe devono avere **self** come primo argomento, self è un argomento che si riferisce all'istanza: al momento della chiamata del metodo, poi questo parametro non verrà passato esplicitamente, ma lo farà Python implicitamente.

Attributi di classe

Gli attributi (proprietà) si possono raggruppare in due categorie:

- attributi di istanza
- attributi di classe

Come si può intuire, gli attributi di istanza sono legati a un'istanza (oggetto) specifica e sono accessibili solo da essa, mentre gli attributi di classe sono legati alla classe e sono accessibili da tutti gli oggetti (istanze) di quella classe.

```
Es.  # definiamo una classe Dog con un attributo di classe
     class Dog:
        scientific_name = 'Canis lupus familiaris'      # definiamo un attributo di classe
        def __init__(self, name):                       # definiamo un __init__ che accetta un nome
           self.name = name                             # creiamo un attributo di istanza per il nome

     # creiamo due istanze di Dog
     rex = Dog('Rex')
     fido = Dog('Fido')
     # verifichiamo che ogni istanza ha un nome diverso
     print(rex.name)                    # visualizza 'Rex'
     print(fido.name)                   # visualizza 'Fido'
     # accediamo all'attributo di classe da Dog
     print(Dog.scientific_name)         # visualizza 'Canis lupus familiaris'
     # accediamo all'attributo di classe dalle istanze
     print(rex.scientific_name)         # visualizza 'Canis lupus familiaris'
     print(fido.scientific_name)        # visualizza 'Canis lupus familiaris'
     # modifichiamo il valore dell'attributo di classe
     Dog.scientific_name = 'Canis lupus lupus'
     # verifichiamo il cambiamento dall'istanza
     print(rex.scientific_name)         # visualizza 'Canis lupus lupus'
```

Come si può notare l'attributo di classe 'scientific_name' è visibile da tutti gli oggetti e dalla classe stessa.
Gli attributi di classe sono utili quando il valore che contengono vale per tutti gli oggetti appartenenti a quella classe.

Ereditarietà

Sintassi, class nome_sottoclasse(nome_superclasse):
mettendo il nome_superclasse fra parentesi, si indica che nome_sottoclasse eredita da nome_superclasse.

per realizzare l'*overriding (riscrittura)* basta riscrivere nella sottoclasse, il metodo della superclasse da sostituire.

Es.
```
import math

# definizione superclasse/classe madre Poligono
class Poligono:
    def __init__(self, num_lati, lato):
        self.num_lati = num_lati
        self.lato = lato

    # metodi 'perimetro' e 'area' della classe genitore 'Poligono'
    def Calcolo_perimetro(self):
        perimetro = self.num_lati * self.lato
        return perimetro

    def Calcolo_area(self):
        area = 0
        return area
# chiusura superclasse/classe madre Poligono

# definizione sottoclasse/classe figlia TriangoloEquilatero
#     per ereditare da Poligono, il nome della superclasse viene messo fra parentesi
class TriangoloEquilatero(Poligono):
    # metodo 'Calcolo_area' della classe figlia 'TriangoloEquilatero'
    # il metodo 'Calcolo_area' della classe 'Poligono' viene sostituito con questo (overriding/riscrittura)
    def Calcolo_area(self):
        # applico Pitagora
        altezza = float(math.sqrt(self.lato * self.lato - (self.lato / 2) * (self.lato / 2)))
        area = (self.lato * altezza) / 2;
        return area
# chiusura sottoclasse/classe figlia TriangoloEquilatero

# programma principale
# istanzia un oggetto 'figura' prima con la superclasse Poligono
#   poi con la sottoclasse TriangoloEquilatero per vedere le differenze
#figura = Poligono(0, 0)
figura = TriangoloEquilatero(3, 0)

# controllo sugli input delle proprietà
while ((figura.num_lati < 3) or (figura.lato <= 0)):
    figura.num_lati = int(input("Inserire il numero dei lati della figura: "))
    figura.lato = float(input("Inserire la misura del lato della figura: "))

# calcolo richiamando i metodi
perimetro = figura.Calcolo_perimetro()      # Metodo ereditato dalla classe Poligono (lasciato inalterato)
area = figura.Calcolo_area()                # Metodo sostituito nella classe Triangolo_equilatero (ereditato da Poligono)

# visualizzazione dei risultati
print("\nIl perimetro è uguale a ", perimetro)
print("L'area è uguale a ", area)
```

Se nella classe figlia TriangoloEquilatero si vogliono aggiungere nuove proprietà nel metodo __init__ , si può modificare il codice come segue

Es.
```
class TriangoloEquilatero(Poligono):
    # definizione di un nuovo metodo __init__ che accetta le proprietà num_lati, lato, e aggiunge nome
    def __init__(self, num_lati, lato, nome):
        # con super() richiama il metodo __init__ della superclasse Poligono, che assegna 'num_lati' e 'lato' all'istanza
        super().__init__(num_lati, lato)
        # assegna la proprietà 'nome' all'istanza
        self.nome = nome

    # ...segue il metodo 'Calcolo_area' della classe figlia 'TriangoloEquilatero'...
```

Il metodo **super()** serve per riferirsi ad una proprietà o metodo di una superclasse: nell'esempio, con super().__init__(num_lati, lato), ci si riferisce al metodo __init__ della superclasse Poligono.

Bibliografia

https://www.meccanismocomplesso.org/lezioni-di-python-il-corso-per-imparare-a-programmare-in-python/
https://www.html.it/guide/guida-python/
https://www.freecodecamp.org/news/python-switch-statement-switch-case-example/