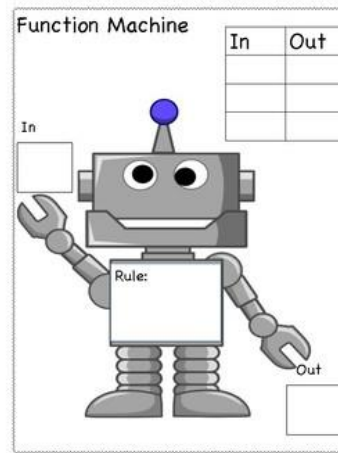




UNIVERSITÀ
DI PARMA

funzioni

informatica e laboratorio di programmazione



- *operatore*, applicato a *operandi*, per ottenere un *risultato*
- **def** per *definire* una funzione
- **return** per terminare e restituire un *risultato*

```
def hypotenuse(a, b):  
    c = (a ** 2 + b ** 2) ** 0.5  
    return c
```



- **def** definisce una funzione, ma non la esegue!
- per far *eseguire* una funzione è necessario «*chiamarla*»
 - la funzione, quando viene eseguita, crea nuovo spazio di nomi
 - i parametri e le variabili hanno ambito locale
 - non sono visibili nel resto del programma
 - nomi uguali, definiti in ambiti diversi, restano distinti

```
side1 = float(input('1st side? '))  
side2 = float(input('2nd side? '))  
side3 = hypotenuse(side1, side2)  
print('3rd side:', side3)
```

- è spesso preferibile creare una *funzione principale (main)*
- in questo modo si limitano le variabili globali

```
# def hypotenuse ...

def main():
    side1 = float(input("1st side? "))
    side2 = float(input("2nd side? "))
    side3 = hypotenuse(side1, side2)
    print("3rd side:", side3)

main()  ## remove, if importing the module elsewhere
```

- la definizione della funzione opera sui *parametri formali*
- al momento della chiamata si definiscono i *parametri attuali*
- le variabili definite nella funzione rimangono locali a questa

```
def dummy(f1, f2):  
    loc = f1 ** f2  
    f1 = f1 * 2  
    return loc  
  
a1 = float(input("fist value: "))  
a2 = float(input("secondt value: "))  
print(dummy(a1,a2))  
print(loc)      # NameError: name 'loc' is not defined  
print(a1)       # print ???
```

- *call-by-object*
- parametri passati «per oggetto»
 - se il parametro è una *variabile* le modifiche non si ripercuotono all'esterno
 - se il parametro è una *lista* o un *oggetto* le modifiche si ripercuotono

```
def inc(f):  
    f = f + 1  
    print(f) # 11  
  
a = 10  
inc(a)  
print(a)    # 10
```

```
def inc(f):  
    for i in range(0, len(f)):  
        f[i] = f[i] + 1  
    print(f) # [3, 4, 6]  
  
a = [2, 3, 5]  
inc(a)  
print(a) # [3, 4, 6]
```

- si possono restituire più valori, come tupla

```
def min_max(f):  
    '''  
    restituisce valore minimo e massimo della lista f  
    '''  
    minimo = massimo = f[0]  
    for i in range(1, len(f)):  
        if f[i] < minimo:  
            minimo = f[i]  
        if f[i] > massimo:  
            massimo = f[i]  
    return minimo, massimo  
  
def main():  
    a = [2, 13, 5, -3, 8]  
    x, y = min_max(a)  
    print("minimo: ", x, " massimo: ", y)  
  
main()  ## remove if importing the module elsewhere
```

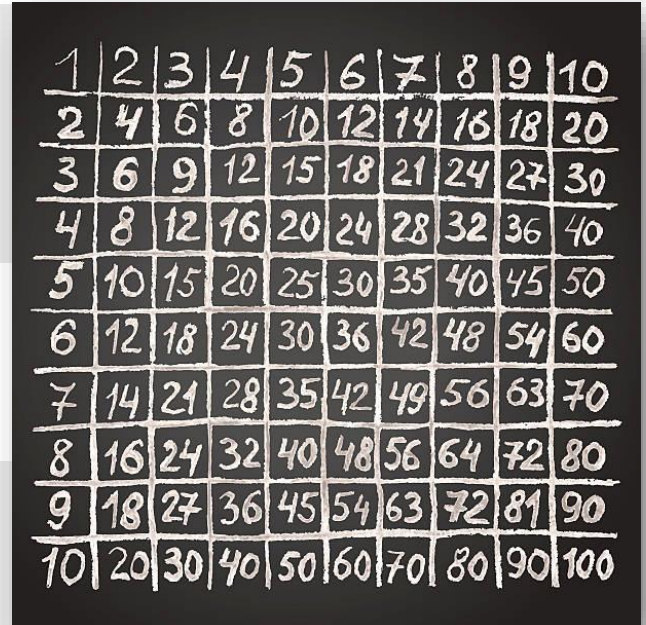
- ***annotazioni***: utili per documentare il tipo dei parametri e il tipo del valore di ritorno (*ma non c'è verifica!*)
- ***docstring***: descrizione testuale di una funzione
- ***help***: funzione per visualizzare la documentazione

```
def hypotenuse(cathetus1: float, cathetus2: float) -> float:
    """
    Return the hypotenuse of a right triangle, given both its legs (catheti).
    """
    return (cathetus1 ** 2 + cathetus2 ** 2) ** 0.5
```


- la stringa di documentazione, posta all'inizio di una funzione, ne *illustra l'interfaccia*
- per convenzione, la **docstring** è racchiusa tra triple virgolette, che le consentono di essere divisibile su più righe
- è breve, ma contiene le informazioni essenziali per usare la funzione
 - spiega in modo conciso **cosa fa** la funzione (non come lo fa)
 - spiega il significato di ciascun parametro e il suo tipo
- è una parte importante della progettazione dell'interfaccia
 - un'interfaccia deve essere **semplice** da spiegare

```
size = 10
y = int(input("Insert a value: "))
for x in range(1, size + 1):
    print(x * y, end=" ") # ends with blank no newline
```

```
size = 10
for y in range(1, size + 1):
    for x in range(1, size + 1):
        val = x * y print(f"{val:3}", end=" ") # val represented as text
                                                    # with at least 3 chars
    print()
```



1	2	3	4	5	6	7	8	9	10
2	4	6	8	10	12	14	16	18	20
3	6	9	12	15	18	21	24	27	30
4	8	12	16	20	24	28	32	36	40
5	10	15	20	25	30	35	40	45	50
6	12	18	24	30	36	42	48	54	60
7	14	21	28	35	42	49	56	63	70
8	16	24	32	40	48	56	64	72	80
9	18	27	36	45	54	63	72	81	90
10	20	30	40	50	60	70	80	90	100

- funzione *senza return*
 - non restituisce valori
 - solo I/O ed effetti collaterali
- *astrazione*, per riuso e leggibilità
- riduce i livelli di annidamento

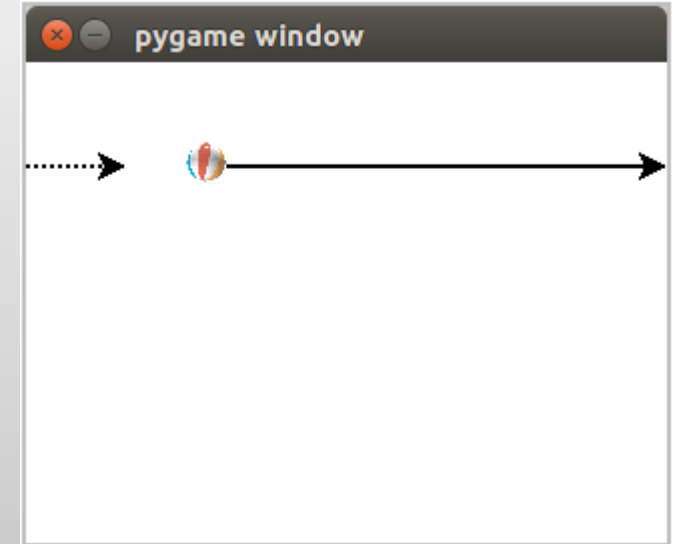
```
def print_row(y: int, size: int):  
    for x in range(1, size + 1):  
        val = x * y  
        print(f"{val:3}", end=" ")  
    print()  
  
def print_table(size: int):  
    for y in range(1, size + 1):  
        print_row(y, size)  
  
def main():  
    print_table(10)
```

```
import g2d

def update():
    global x
    g2d.fill_canvas((255, 255, 255)) # Draw background
    g2d.draw_image(image, (x, 50))   # Draw foreground
    x = (x + 5) % 320                 # Update ball's position

g2d.init_canvas((320, 240))
image = g2d.load_image("ball.png")
x = 50

g2d.main_loop(update, 1000 // 30)    # Call update 30 times/second
```



```
import g2d

def keydown(code: str):
    print("Key pressed: ", code)

def keyup(code: str):
    print("Key released: ", code)

g2d.handle_keyboard(keydown, keyup)
```



- nome del modulo in esecuzione: `__name__`
 - è il nome del file, senza estensione
- il modulo di avvio dell'app ha nome speciale
 - in CPython, nome `"__main__"`
 - *in Brython nome "script..."; il "trucco" non funziona*

```
# def hypotenuse ...
def main():
    side1 = float(input("1st side? "))
    side2 = float(input("2nd side? "))
    print("3rd side:", hypotenuse(side1, side2))

# if this module is imported, main is not executed
if __name__ == "__main__":
    main()
```

- la funzione può modificare oggetti passati come parametri o variabili globali o effettuare operazioni di lettura/scrittura...
 - effetti collaterali annullano la *trasparenza referenziale*
 - impossibile semplificare, sostituendo una chiamata a funzione col suo valore di ritorno (es. presenti operazioni di I/O)
 - effetti collaterali rendono la funzione *non idempotente*
 - chiamata più volte, con gli stessi parametri, la funzione può restituire risultati diversi
 - → difficile fare verifiche matematiche
- ```
z = f(sqrt(2), sqrt(2))
s = sqrt(2)
z = f(s, s)
```

- esempio di semplificazione

$p = f(x) + f(y) * (f(x) - f(x))$

$p = f(x) + f(y) * (0)$

$p = f(x) + 0$

$p = f(x)$

- ma se  $f$  ha effetti collaterali non è corretto

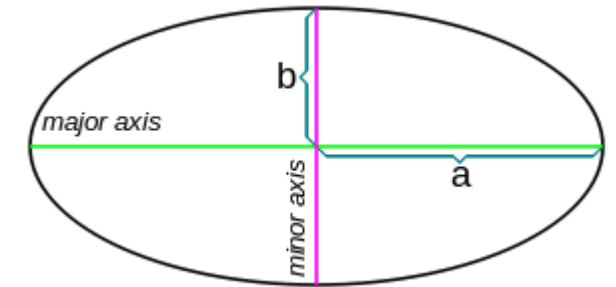
```
base_value = 0 # global variable

def f(x: int) -> int:
 global base_value
 base_value += 1
 return x + base_value
```



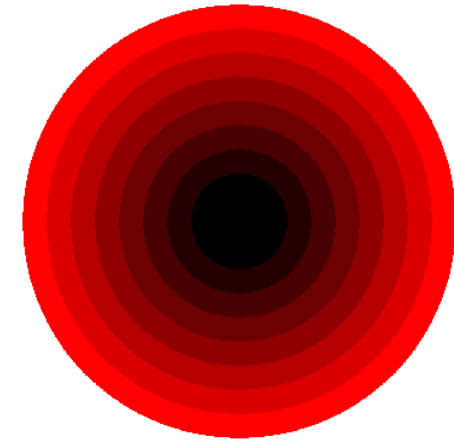
## 3.1 area di un'ellisse

- definire una funzione ***ellipse\_area*** che:
  - riceve come parametri i ***semiassi*** di una ellisse: ***a***, ***b***
  - restituisce come risultato l'area dell'ellisse:  $\pi \cdot a \cdot b$
- definire una funzione ***main*** che:
  - chiede all'utente due valori
  - invoca la funzione `ellipse_area` con questi parametri
  - stampa il risultato ottenuto



### 3.2 cerchi concentrici

- chiedere all'utente il **numero** di cerchi da disegnare
- **disegnare** i cerchi con **raggio decrescente**, ma tutti con lo **stesso centro**
- far variare il **colore** dei cerchi
  - dal **rosso** del livello più **esterno**
  - fino al **nero** del livello più **interno**

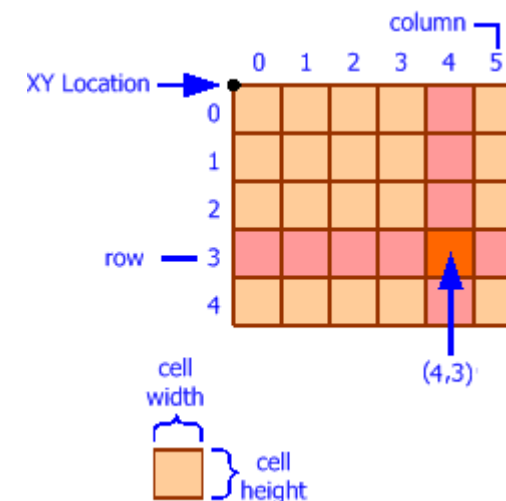
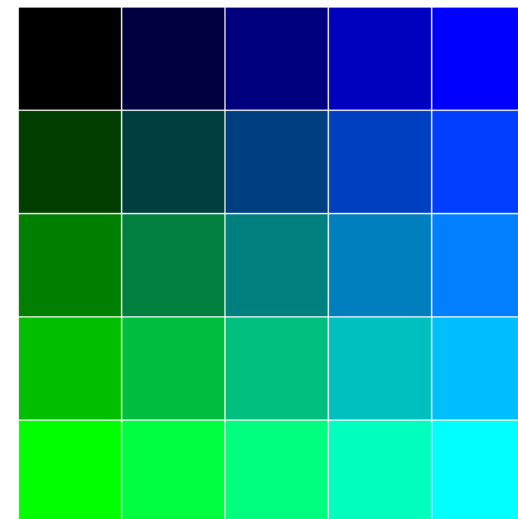


*cominciare a disegnare un grosso cerchio rosso  
poi, inserire l'operazione di disegno in un ciclo,  
togliendo ad ogni passo 10 (p.es.) al raggio e al livello di rosso  
infine, determinare automaticamente, prima del ciclo, le variazioni migliori per raggio e colore*

## 3.3 griglia di colori

- chiedere all'utente dei valori per *rows* e *cols*
- mostrare una griglia di *rettangoli* di dimensione *rows*×*cols*
- partire da un rettangolo nero in *alto a sinistra*
- in *orizzontale* aumentare gradatamente la componente di *blu*
- in *verticale* aumentare gradatamente la componente di *verde*

*cominciare a creare una griglia di riquadri tutti neri  
con due cicli for annidati  
lasciare tra i riquadri un piccolo margine*

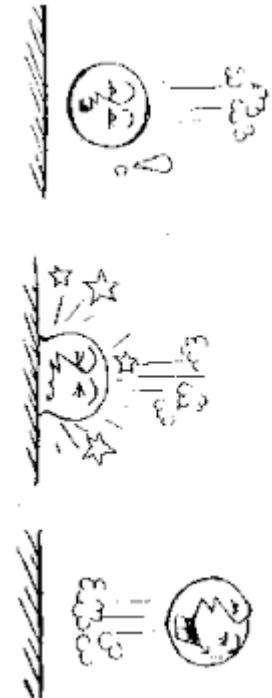


### 3.4 movimento orizzontale

- mostrare una *pallina* che si muove in *orizzontale*
- la pallina *rimbalza* sui bordi

*memorizzare in una variabile  $dx$  lo spostamento orizzontale  
da effettuare ad ogni ciclo*

*cambiare segno a  $dx$  quando  $x < 0$  oppure  $x + w > screen\_width$*



### 3.5 movimento a serpentina

- mostrare una *pallina* che si muove a *serpentina*
- partire dall'esercizio precedente
- al momento del **rimbalzo**, imporre un spostamento **verticale**
- fare in modo che, in ogni frame, lo spostamento sia *solo orizzontale*, o *solo verticale*, ma *non diagonale*

