Definizione di Funzioni

Definizione di Funzioni

Abbiamo già parlato di funzioni in Pyhon

...Ed abbiamo detto che, quando vengono chiamate:

- Eseguno un sottoprogramma
- Denotano (o "restituiscono") un valore
  - In casi particolari (e.g. `print`) possono denotare `None`
- Possono riceve dei dati in ingresso (argomenti)

Vedremo ora come definire nuove funzioni

- Sarà nostro compito specificare il sotto programma da eseguire
- ...Così come i dati in ingresso ed il valore da restituire

La funzione potrà quindi essere chiamata normalmente


Definizione di Funzioni

Per definire una funzione si usa l'istruzione seguente:

```
def <nome>(<parametri>):
    <corpo>
```

L'istruzione informa l'interprete dell'esistenza di una nuova funzione:

- `<nome>` è il nome della funzione
  - Segue le regole degli identificatori (e.g. nomi di variabili)
- `<parametri>` è una sequenza di 0 o più nomi di variabile
  - Si usa la virgola come separatore
  - I nomi di variabile sono per uso interno alla funzione
- `<corpo>` è una sequenza di istruzioni
  - Rappresenta il sottoprogramma da eseguire

Definizione di Funzioni

Qualche esempio (per adesso focalizzatevi sulla sintassi)

Elevamento a potenza:

In [1]:
```
def potenza(a, b):
    return a**b
```

- Il nome della funzione è `potenza`
- La funzione riceve in ingresso due argomenti (o ha due parametri)
- I due parametri si chiamano (internamente) `a` e `b`
- Il corpo consiste di una sola istruzione, i.e. `return a**b`


Definizione di Funzioni

Qualche esempio (per adesso focalizzatevi sulla sintassi)

Determinare se un numero sia pari:

In [2]:
```
def pari(num):
    return num % 2 == 0
```

- Il nome della funzione è `pari`
- La funzione riceve ingresso un solo argomento/ha un solo parametro
- Il parametro si chiama (internamente) `num`
- Il corpo consiste dell'istruzione `return num % 2 == 0`


Definizione di Funzioni

Qualche esempio (per adesso focalizzatevi sulla sintassi)

Stampa "ciao mondo":

In [3]:
```
def ciao():
    print('ciao mondo')
```

- Il nome della funzione è `ciao`
- La funzione non ha argomenti/non ha parametri
- Il corpo consiste dell'istruzione `print('ciao mondo')`

Definizione di Funzioni

Qualche esempio (per adesso focalizzatevi sulla sintassi)

Una funzione che non fa nulla

In [4]:
```
def bozza(num):
    pass
```

- Il nome della funzione è `bozza`
- La funzione ha un singolo argomento/parametro (di nome `num`)
- Il corpo consiste dell'istruzione `pass`
- ...Che in Python non fa nulla

`pass` si usa spesso in fase di sviluppo, quando il corpo non è ancora chiaro


Chiamare una Funzione Definita dall'Utente

Una funzione definita può essere chiamata come al solito

I.e. si usa la sintassi:

```
<nome>(<sequenza di espressioni>)
```

Vediamo subito un esempio:

```
In [5]: def potenza(a, b):
            return a**b

        potenza(2, 4)
```

```
Out[5]: 16
```

- Somiglia molto alla sintassi usate per definire una funzione
- ...Ma manca la parola chiave `def`
- ...E c'è una seconda differenza chiave

Parametri Formali ed Attuali

Quando si definisce una funzione

...I parametri specificati indicano dei nomi di variabile

```
In [6]: def potenza(a, b):
            return a**b
```

- Si parla in questo caso di parametri formali

Quando si chiama una funzione

...I parametri specificati indicano delle espressioni

```
In [7]: potenza(3, 6)
```

```
Out[7]: 729
```

- Si parla in questo caso di parametri attuali

Meccanismo di Chiamata a Funzione

Meccanismo di Chiamata

La differenza è dovuta ai dettagli del meccanismo di chiamata

Quando una funzione viene chiamata:

- Viene allocata memoria per l'esecuzione del corpo
- I parametri attuali (espressioni) vengono valutati
- I valori denotati vengono usati per rimpire i parametri formali
- Il corpo della funzione viene eseguito
- Il controllo torna al (codice) chiamante
- L'area di memoria predisposta viene deallocata

In particolare, per quanto riguarda i parametri

- I parametri formali sono le variabili da riempire
- I parametri attuali restituiscono i valori con cui riempirle

Esempi di Chiamata a Funzione

Vediamo qualche esempio (commentato)

Chiamata della funzione `pari`

In [8]:
```
def pari(num):
    return num % 2 == 0

pari(3 * 12 * 6)
```

Out[8]: True

Durante la chiamata a funzione:

- Viene predispone memoria per l'esecuzione del corpo
- Viene valutato il parametro attuale `3 * 12 * 6`
- Il valore denotato viene usato per riempire `num`
- Il corpo della funzione esegue

Esempi di Chiamata a Funzione

Vediamo qualche esempio (commentato)

Chiamata della funzione `pari`

In [9]:
```
def pari(num):
    return num % 2 == 0

pari(3 * 12 * 6)
```

Out[9]: True

L'istruzione `return` ha di norma la sintassi

```
return <espressione>
```

- Termina immediatamente la funzione
- Restituisce il valore dell'espressione al chiamante

In questo modo possiamo specificare cosa la funzione debba restituire

Esempi di Chiamata a Funzione

Vediamo qualche esempio (commentato)

Chiamata della funzione `potenza`

In [10]:
```
def potenza(a, b):
    return a**b

potenza(2, 4)
```

Out[10]: 16

Se la funzione ha più di un parametro

...Il riempimento avviene con un criterio posizionale

- Il primo parametro attuale (i.e. la prima espressione)
- ...È usato per riempire il primo parametro formale (i.e. la prima variabile)

Esempi di Chiamata a Funzione

Vediamo qualche esempio (commentato)

Chiamata della funzione `ciao`

In [11]:
```
def ciao():
    print('ciao mondo')

ciao()
```

```
ciao mondo
```

Se la funzione non ha parametri

- La fase di valutazione dei parametri attuali viene saltata
- ...E così anche la fasi di riempimento dei parametri formali

Esempi di Chiamata a Funzione

Vediamo qualche esempio (commentato)

Chiamata della funzione `ciao`

In [12]:
```
def ciao():
    print('ciao mondo')

x = ciao()
print(x)
```

```
ciao mondo
None
```

Se la funzione non ha una istruzione `return`

- Quando termina l'esecuzione del corpo
- ...Viene automaticamente restituito **`None`**


Esempi di Chiamata a Funzione

Vediamo qualche esempio (commentato)

Chiamata della funzione `ciao`

In [13]:
```
def ciao():
    print('ciao mondo')
    return None

x = ciao()
print(x)
```

```
ciao mondo
None
```

Lo stesso succede se viene eseguita l'istruzione `return None`

Esempi di Chiamata a Funzione

Vediamo qualche esempio (commentato)

Chiamata della funzione `ciao`

In [14]:
```
def ciao():
    print('ciao mondo')
    return

x = ciao()
print(x)
```

```
ciao mondo
None
```

...O se viene eseguita l'istruzione `return` (senza argomenti)

Record di Attivazione

L'area di memoria allocata alla chiamata si chiama `record di attivazione`

Il record di attivazione:

- È associato ad ogni chiamata
- Ospita tutte le variabili definite dalla chiamata a funzione
- ...Inclusi i parametri formali
- Viene deallocato al termine dell'esecuzione

Questo comportamento ha alcune conseguenze importanti

Variabili Locali

Innanzitutto le variabili in una funzione sono locali

...Ossia possono essere utilizzate solo dalla chiamata corrente:

In [15]:
```
def potenza(a, b):
    return a**b

potenza(2, 4)
print(a, b)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[15], line 5
      2     return a**b
      4 potenza(2, 4)
----> 5 print(a, b)

NameError: name 'a' is not defined
```

- `a` e `b` vengono allocate solo all'inizio della chiamata
- ...E cessano di esistere al suo termine

Variabili Locali

Ogni record di attivazione è associato ad una chiamata a funzione

...Quindi anche le variabili sono locali alla chiamata

In [16]:
```
def potenza(a, b):
    print(f'a: {a}, b: {b}')
    return a**b

print('ris. prima chiamata:', potenza(2, 4))
print('ris. prima chiamata:', potenza(3, 3))
```

```
a: 2, b: 4
ris. prima chiamata: 16
a: 3, b: 3
ris. prima chiamata: 27
```

- Nella prima chiamata abbiamo `a = 2` e `b = 4`
- Nella seconda chiamata abbiamo `a = 3` e `b = 3`


Variabili Locali vs Globali

Per contro da una funzione si può accedere ad una variabile globale

In [17]:
```
y = 3

def potenza_y(b):
    return y**b

potenza_y(4)
```

Out[17]: 81

La variabile `y` è accessibile dall'interno della chiamata

- ...Perché `y` è definita nell'ambiente globale
- ...E quindi è la sua esistenza è garantita
- ...Al momento dell'esecuzione di `potenza_y`

Ambienti e Visibilità

In generale, se una funzione è definita in un certo "contesto"

- ...Ogni altra variabile o funzione definita nello stesso contesto
- ...È garantito che esista al momento della chiamata

Lo stesso ragionamento si applica ad ogni identificatore (e.g. funzioni o variabili)

Questi contesti si chiamano ambienti

In Python un ambiente è creato da determinate istruzioni:

- L'esecuzione dell'interprete Python è associata ad un ambiente globale
- Ogni definizione di funzione introduce un nuovo ambiente

La regola generale è che:

- Un identificatore definito in un determinato ambiente
- È accessibile nell'ambiente stesso ed in ogni ambiente "interno"

Ambienti e Visibilità

Torniamo all'esempio precedente

In [18]:
```
y = 3

def potenza_y(b):
    return y**b
```

- `y` è nell'ambiente globale
- `potenza_y` introduce un nuovo ambiente, all'interno di quello globale
- ...Quindi `y` è accessibile da `potenza_y`

Ambienti e Visibilità

Un esempio con due funzioni

In [19]:
```
def get_e():
    return 2.71828

def esponenziale(x):
    return get_e()**x

print(esponenziale(2))
print(get_e())
```

```
7.3890461584
2.71828
```

- `get_e` è nell'ambiente globale
- l'ambiente di `esponenziale` è interno a quello globale
- ...Quindi `get_e` è accessibile da `esponenziale`


Ambienti e Visibilità

Un esempio estremo

In [20]:
```
def esponenziale(x):
    def get_e():
        return 2.71828
    return get_e()**x

print(esponenziale(2))
```

```
7.3890461584
```

- `get_e` è definita nell'ambiente di `esponenziale`
  - Sì, si può definire una funzione dentro un'altra funzione!
- ...Quindi `get_e` è accessibile da `esponenziale`


Risoluzione dei Nomi

Ci possono essere due identificatori (variabili/funzioni) uguali

...Ma solo se sono definite in ambienti diversi

In [21]:
```
y = 3 # variabile y, nell'ambiente globale

def potenza(y, x): # variabile y, nell'ambiente di "potenza"
    return y**x

potenza(4, 2)
```

Out[21]: 16

- Ci sono due variabili `y`, ma in ambienti diversi
  - Una variabile è globale
  - L'altra è locale alla funzione `potenza`
- Se fossero nello stesso ambiente, otterremmo un errore


Risoluzione dei Nomi

Ci possono essere due identificatori (variabili/funzioni) uguali

...Ma solo se sono definite in ambienti diversi

In [22]:
```
y = 3 # variabile y, nell'ambiente globale

def potenza(y, x): # variabile y, nell'ambiente di "potenza"
    return y**x

potenza(4, 2)
```

Out[22]: 16

La variabile `y` globale è visibile in entrambi gli ambienti

- Se si utilizza il simbolo `y` nella funzione `potenza`
- ...Non è chiaro se questo si riferisca ad `y` locale o globale

In questo caso l'ambiguità è risolta dalla regola di risoluzione dei nomi


Risoluzione dei Nomi

Ci possono essere due identificatori (variabili/funzioni) uguali

...Ma solo se sono definite in ambienti diversi

In [23]:
```
y = 3 # variabile y, nell'ambiente globale

def potenza(y, x): # variabile y, nell'ambiente di "potenza"
    return y**x



potenza(4, 2)
```

Out[23]: 16

Gli identificatori vengono individuati a partire dall'ambiente più interno

- In queso caso, l'interprete cerca `y` nell'ambiente locale, poi in quello globale
- Conseguenza: la `y` locale "oscura" quella globale
- Per questa ragione il risultato è 16 e non 9

Qualche Extra

Valori di Default

È possibile specificare valori di default per determinati parametri

Vediamo un esempio:

```
In [24]: def root(x, y=0.5):
             return x**y
```

- Anziché usare solo un nome di variabile per il parametro
- Si aggiunge il segno `=` ed un valore

Così facendo il valore del parametro può essere omesso nella chiamata

```
In [25]: root(4)

Out[25]: 2.0
```

- Al parametro `y` viene assegnato il valore di default

Argomenti con Nome

È possibile assegnare gli argomenti fuori ordine

...Se ne specifichiamo esplicitamente il nome nella chiamata:

In [26]:
```
def potenza(a, b):
    print(f'a: {a}, b: {b}')
    return a**b

potenza(b=2, a=4)
```

```
a: 4, b: 2
```

Out[26]:
```
16
```

- Si usa la sintassi `<parametro>=<espressione>` nella chiamata
- In questo modo specifichiamo manualmente come assegnare i valori

È una notazione verbose, ma può migliorare di molto la leggibilità

Argomenti con Nome

Se si usa sia il passaggio per posizione che con nome

...I parametri con nome devono seguire quelli posizionali

In [27]:
```
def potenza(a, b):
    print(f'a: {a}, b: {b}')
    return a**b

potenza(4, b=2)
```

```
a: 4, b: 2
```

Out[27]: 16

...Altrimenti non ci sono abbastanza informazioni per effettuare gli assegnamenti

In [28]:
```
potenza(a=4, 2)
```

```
  Cell In[28], line 1
    potenza(a=4, 2)
                 ^
SyntaxError: positional argument follows keyword argument
```

Restituzione di Valori Multipli

È possibile restituire "più di un valore"

Non si tratta in realtà di una nuova caratteristica di Python:

- Semplicemente, basta restituire una collezione
- ...E quindi usare tuple unpacking per recuperarne gli elementi ad uno ad uno

Vediamo un esempio:

In [29]:
```
def divisione_con_resto(a, b):
    return a // b, a % b

q, r = divisione_con_resto(10, 3)
print(f'quoziente: {q}, resto: {r}')
```

```
quoziente: 3, resto: 1
```

- L'argomento di `return` è una tupla (senza parentesi)
- La tupla restituita viene immediatamente "spacchettata" dal chiamante


Motivazioni per l'Uso delle Funzioni

La definizione di funzioni è un meccanismo molto potente

...Perché permette di nascondere la complessità di un algoritmo

- Per esempio, se introduciamo una funzione:

In [30]:
```
def fattoriale(n):
    res = 1
    for i in range(1, n+1):
        res *= i
    return res
```

- ...Possiamo calcolare un fattoriale semplicemente chiamando la funzione

In [31]:
```
print(fattoriale(4))
print(fattoriale(32))
```

```
24
263130836933693530167218012160000000
```


Motivazioni per l'Uso delle Funzioni

Questo offre alcuni grossi vantaggi:

Il codice diventa più facile da usare

- Per usare l'algoritmo, non serve sapere come funziona
- ...Basta sapere come eseguirlo!

Il codice diventa più facile da mantenere

- Non è più necessario copiare/incollare codice (con il rischio di errori)
- Una modifica fatta alla funzione, ha effetto su tutti i punti in cui è chiamata

Inoltre se evitiamo di usare variabili globali in una funzione

- ...La funzione comunica con l'esterno solo con il passaggio di parametri
- ...Ed evitiamo di usare inavvertitamente variabili introdotte in altre celle