Funzioni

- Una funzione è una sequenza di istruzioni a cui viene dato un nome, con lo scopo di riutilizzare tale sequenza di istruzioni in più parti del programma senza doverla scrivere ogni volta.
- Una funzione può prendere in ingresso uno o più parametri e può restituire in uscita un risultato.
- Sintassi:

```
def <nome_funzione>(arg1,arg2,...,argN):
        <istruzioni>
```

- Esempio di definizione di una funzione:

```
def ask_ok(prompt, retries=4, complaint='Sì o no, grazie!'):
  while True:
        ok = raw_input(prompt)
        if ok in ('s', 'si', 'yes'): return True
        if ok in ('n', 'no'): return False
        retries = retries - 1
        if retries < 0: raise IOError, 'utente indisciplinato'
        print complaint
```

- Questa funzione può essere chiamata così:

```
        ask_ok('Vuoi davvero uscire?')
```

o così:

```
        ask_ok('Devo sovrascrivere il file?', 2)
```

Funzioni

- Una funzione può restituire un oggetto in uscita tramite l'istruzione `return`

```
def max(l):
    max = l[0]
    for e in l[1:]:
        if e > max:
            max = e
    return max
```

- Le funzioni che non hanno `return` (o che terminano senza averne incontrato nessuno) restituiscono `None`

Funzioni

- E' possibile restituire un numero arbitrario di oggetti, mettendoli in una tupla

```
def max_min(l):
        max = min = l[0]
        for e in l[1:]:
                if e > max:
                        max = e
                elif e < min:
                        min = e
        return max,min
```

- Il risultato può essere assegnato ad una tupla o direttamente alle sue variabili:

```
l = [1,5,3,9,12]
t = max_min(l)
max, min = max_min(l)
print t
print max,min
```

Output:

```
(6, 1)
61
```

Funzioni

- Se un argomento passato ad una funzione è mutabile, le modifiche ad esso fatte all'interno della funzione si ripercuoteranno all'esterno

```
def elimina_valore(d, val):
    for (k,v) in d.items():
        if v == val:
            del d[k]

d = {"a" : 1, "b": 2, "c" : 2, "d" : 3}
elimina_valore(d,2)
print d
```

Output:

```
{'a': 1, 'd': 3}
```

Funzioni

- Le funzioni possono essere chiamate anche usando argomenti a parola chiave nella forma "parolachiave = valore". Per esempio la funzione seguente:

```
def parrot(voltage, state='a stiff', action='voom', type='Norwegian
  Blue'):
  print "-- This parrot wouldn't", action,
  print "if you put", voltage, "Volts through it."
  print "-- Lovely plumage, the", type
  print "-- It's", state, "!"
```

- Potrebbe essere chiamata in uno qualsiasi dei seguenti modi:

```
parrot(1000)
parrot(action = 'VOOOOOM', voltage = 1000000)
parrot('a thousand', state = 'pushing up the daisies')
parrot('a million', 'bereft of life', 'jump')
parrot() # ERRORE manca un argomento necessario
parrot(voltage=5.0, 'dead')
# ERR argomento non a parola chiave seguito da una parola chiave
```

Funzioni

- Tutte le variabili definite all'interno di una funzione sono locali ad essa.
- Una variabile locale è visibile solo all'interno della funzione (scope della variabile).
- Una variabile non può essere usata al di fuori del suo spazio di visibilità.
- Esiste uno scope globale che è esterno alle funzioni, e copre tutto il file in cui le funzioni si trovano.
- Le variabili definite al di fuori delle funzioni, sono visibili anche al loro interno

```
def globali():
        print x,y

x,y = 1,2
globali()
```

- Output:

```
1  2
```

- Se una variabile locale ha lo stesso nome di una variabile globale esistente, la prima nasconde la seconda.

Funzioni mutuate dalla programmazione funzionale

filter (funzione, sequenza) restituisce una sequenza (dello stesso tipo ove possibile) composta dagli elementi della sequenza originali per il quale è vera funzione (elemento)

Per esempio per calcolare alcuni numeri primi:

```
>>> def f(x): return x % 2 != 0 and x % 3 != 0
>>> filter(f, range(2, 25))
[5, 7, 11, 13, 17, 19, 23]
```

map(funzione, sequenza) invoca funzione (elemento) per ciascuno degli elementi della sequenza e restituisce una lista dei valori ottenuti.

Per esempio, per calcolare i cubi di alcuni numeri

```
>>> def cube(x): return x*x*x
>>> map(cube, range(1, 11))
[1, 8, 27, 64, 125, 216, 343, 512, 729, 1000]
```

Funzioni mutuate dalla programmazione funzionale

reduce(funzione, sequenza) restituisce un singolo valore ottenuto invocando la funzione a due argomenti sui primi due elementi della sequenza quindi sul risultato dell'operazione sull'elemento successivo, e così via.

Ad esempio, per calcolare la somma dei numeri da 1 a 10:

```
>>> def add(x,y): return x+y
>>> reduce(add, range(1, 11))
55
```

Lambda

- Il lambda calcolo è un sistema di riscrittura definito formalmente dal matematico Alonzo Church. È stato sviluppato per analizzare formalmente le definizioni di funzioni
- La definizione insiemistica di una funzione non fornisce effettivamente informazioni a proposito di come si possa passare dagli argomenti ricevuti al risultato ad essi associato.
- In altre parole, non si descrive come calcolare il risultato della funzione stessa. Il lambda calcolo, è invece un formalismo che consente di definire il lato meccanico delle funzioni, ovvero proprio quelle procedure che consentono di produrre dei valori in uscita a partire da certi valori in ingresso (per questo è chiamato “calcolo”)

Lambda

- In Python sono presenti alcune funzionalità dei linguaggi funzionali e Lisp.
- Con la parola chiave `lambda` possono essere create piccole funzioni senza nome. Le forme lambda possono essere usate ovunque siano richiesti oggetti
- Esse sono sintatticamente ristrette ad una singola espressione. Dal punto di vista semantico sono solo un surrogato di una normale definizione di funzione:

```
>>> def make_incrementor(n):
      return lambda x: x + n
>>> f = make_incrementor(42)
>>> f(4)
46
>>>f1=make_incrementor(38)
>>>f1(4)
42
>>> print make_incrementor(13)(7)
20
```

- Le forme lambda permettono la creazione di funzioni personalizzate

Lambda

- Altri esempi di utilizzo delle funzioni lambda:

```
>>> foo = [2, 18, 9, 22, 17, 24, 8, 12, 27]
>>> print filter(lambda x: x % 3 == 0, foo)
[18, 9, 24, 12, 27]
>>> print map(lambda x: x * 2 + 10, foo)
[14, 46, 28, 54, 44, 58, 26, 34, 64]
>>> print reduce(lambda x, y: x + y, foo)
139
>>> nums = range(2, 50)
>>> for i in range(2, 8):
        nums = filter(lambda x: x == i or x % i, nums)
>>> print nums
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47]
```

Lambda

```
>>> sentence = 'It is raining cats and dogs'
>>> words = sentence.split()
>>> print words
['It', 'is', 'raining', 'cats', 'and', 'dogs']
>>> lengths = map(lambda word: len(word), words)
>>> print lengths
[2, 2, 7, 4, 3, 4]
```

- E' possibile concentrare il programma in un'unica riga:

```
>>> print map(lambda w: len(w), 'It is raining cats and
 dogs'.split())
[2, 2, 7, 4, 3, 4]
```

Moduli

- Nella sua forma più semplice, un modulo è un file di testo contenente codice Python, salvato con estensione “.py”.
- Un modulo contiene tipicamente funzioni o altri oggetti di utilità che possano essere usati dai programmi.
- Esistono una grande quantità di moduli.
- Esiste un set di moduli standard che è disponibile di default nella maggior parte delle installazioni di Python (esempio: pickle, re)
- Per poter essere utilizzato in un programma, un modulo deve essere importato. Lo si fa mediante l’istruzione `import`

```
>>> import pickle
>>> s = “Benvenuti al corso di Python!”
>>> f = open("tmp","w")
>>> pickle.dump(s,f)
```

Moduli

- Il file contenente il modulo pickle è stato individuato da Python perché la directory che lo contiene è tra quelle standard.
- Le librerie di moduli che installerete, se necessario, aggiorneranno questo spazio di ricerca, ed i moduli potranno essere caricati allo stesso modo.
- Moduli creati dall'utente potranno essere facilmente importati allo stesso modo se presenti nella stessa directory del file che li importa.

Moduli

- Una volta importato è possibile chiamare le funzioni presenti nel modulo.
- La chiamata di una funzione deve essere preceduta dal nome del modulo e seguita da punto:

```
>>> pickle.dump(s,f)
```

- In un programma di dimensioni ragionevoli, esiste un file principale da cui parte l'esecuzione ed una serie di moduli che forniscono le funzionalità necessarie.
- A volte risulta pesante dover sempre precedere una funzione con il nome del modulo che la contiene.
- Per poter chiamare una funzione direttamente, basta importarla con lo statement `from`

```
>>> from <nomemodulo> import <funzione> ["as" nome_locale]
```

- Se `<funzione>` viene sostituito con *, tutte le funzioni del modulo possono essere chiamate direttamente

Spazio dei nomi

- Lo spazio dei nomi è una mappa che collega i nomi agli oggetti built-in. Esempi di spazi dei nomi sono: l'insieme dei nomi built-in, i (funzioni come abs() ed i nomi delle eccezioni bulti-in) i nomi globali in un modulo e i nomi locali in una chiamata di funzione.
- Gli spazi dei nomi vengono creati in momenti diversi ed hanno tempi di sopravvivenza diversi:
  - Lo spazio dei nomi che contiene i nomi built-in, chiamato __builtin__, viene creato all'avvio dell'interprete Python e non viene mai cancellato.
  - Le istruzioni eseguite dall'invocazione a livello più alto dell'interprete, lette da un file di script o interattivamente, vengono considerate parte di un modulo chiamato __main__.
  - Lo spazio dei nomi globale di un modulo viene creato quando viene letta la definizione del modulo; normalmente anche lo spazio dei nomi del modulo dura fino al termine la sessione

Spazio dei nomi

- Uno scope è una regione del codice di un programma Python dove uno spazio dei nomi è accessibile direttamente.
- “Direttamente accessibile” qui significa che un riferimento non completamente qualificato ad un nome cerca di trovare tale nome nello spazio dei nomi.
- Sebbene gli scope siano determinati staticamente, essi sono usati dinamicamente. In qualunque momento durante l'esecuzione sono in uso esattamente tre scope annidati (cioè, esattamente tre spazi dei nomi sono direttamente accessibili):
  - lo scope più interno, in cui viene effettuata per prima la ricerca, contiene i nomi locali;
  - lo scope mediano, esaminato successivamente, contiene i nomi globali del modulo corrente;
  - lo scope più esterno, esaminato per ultimo, è lo spazio dei nomi che contiene i nomi built-in.

I/O

- `open()` torna un oggetto **file** è ha la seguente sintassi:

  `open(`*name*[, *mode*[, *buffering*]])¶

```
>>> f=open('/tmp/workfile', 'w')
>>> print f
<open file '/tmp/workfile', mode 'w' at 80a0960>
```

- Il primo argomento è una stringa contenente il nome del file.
- mode specifica la modalità di apertura del file. È
  - E' `'r'` ('read') quando il file verrà solamente letto, `'w'` per la sola scrittura, in caso esista già un file con lo stesso nome esso verrà cancellato, `'a'` aprirà il file in append: qualsiasi dato scritto sarà automaticamente aggiunto alla fine dello stesso. L'argomento modo è facoltativo; in caso di omissione verrà assunto essere `'r'`.
  - `'b'` aggiunto al modo apre il file in modo binario, per cui esistono `'rb', 'wb' e 'ab'`.
  - `'r+', 'w+' e 'a+'` consentono di aprire il file in lettura e scrittura.
- buffering specifica la dimensione desiderata del buffer del file:
  - 0 significa unbuffering;
  - 1 significa buffering di un alinea;
  - un valore >1 significa buffering di (approssimativamente) quel numero di byte;
  - un valore negativo indica l'utilizzo dei valori di default che, generalmente, sono line buffering per il tty e full buffering per il file.

I/O

- `f.read(lunghezza)`
  - legge e restituisce al più (come stringa) un numero di byte pari a lunghezza. Se lunghezza è omesso o negativo, verrà letto e restituito l'intero contenuto del file. In caso di EOF, f.read() restituirà una stringa vuota ("").

```
>>> f.read()
'Questo è l'intero file.\n'
>>> f.read()
''
```

- `f.readline()`
  - legge una singola riga dal file, un carattere di fine riga (\n) viene lasciato alla fine della stringa, e viene omesso solo nell'ultima riga del file nel caso non finisca con un fine riga. Ciò rende il valore restituito non ambiguo: se f.readline() restituisce una stringa vuota, è stata raggiunta la fine del file, mentre una riga vuota è rappresentata da '\n', stringa che contiene solo un singolo carattere di fine riga.

```
>>> f.readline()
'Questa è la prima riga del file.\n'
>>> f.readline()
'Seconda ed ultima riga del file\n'
>>> f.readline()
''
```

I/O

- `f.readlines()`
  - restituisce una lista contenente tutte le righe di dati presenti nel file.

```
>>> f.readlines()
['Questa è la prima riga del file.\n',
'Seconda riga  del file\n']
```

- `f.write(stringa)`
  - scrive il contenuto di stringa nel file, restituendo None.

```
>>> f.write('Questo è un test\n')
```

I/O

- `f.tell()`
  - restituisce un intero che fornisce la posizione nel file, misurata in byte dall'inizio del file.
- `f.seek(offset, da_cosa)`
  - Serve per variare la posizione all'interno del file. La posizione viene calcolata aggiungendo ad offset l'argomento da_cosa. Un valore pari a 0 effettua la misura dall'inizio del file (default), 1 utilizza la posizione attuale, 2 usa la fine del file.

```
>>> f=open('/tmp/workfile', 'r+')
>>> f.write('0123456789abcdef')
>>> f.seek(5) # Va al sesto byte nel file
>>> f.read(1)
'5'
>>> f.seek(-3, 2) # Va al terzo byte prima della fine del file
>>> f.read(1)
'd'
```