Introduction

Simone Campagna

filosofia/1

Python è un linguaggio di programmazione con molti aspetti positivi:

- Grande semplicità d'uso
- Grande semplicità di apprendimento (assomiglia alla pseudocodifica)
- Grande leggibilità (c'è un solo modo per fare qualsiasi cosa)
- Grande portabilità

filosofia/2

- Per favore, non definiamolo un linguaggio di scripting! Anche se può essere utilizzato come tale, è molto, molto di più.
- È un linguaggio di altissimo livello, moderno, completo, con il quale è possibile realizzare software di notevole complessità.
- È un linguaggio interpretato, ma sarebbe più appropriato definirlo “linguaggio dinamico”.

filosofia/3

- Un linguaggio dinamico è un linguaggio di alto livello in cui vengono eseguiti run-time molti dei controlli che altri linguaggi eseguono in compilazione.
- In effetti python “compila” il sorgente in bytecode, che viene eseguito su una virtual machine (come Java).

filosofia/4

È un linguaggio multiparadigma:

- Imperative
- Object-oriented
- Functional
- Structural
- Aspect-oriented
- Design by contract (con una estensione)
- ...

The zen of python

>>> import this
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!

svantaggi

Python è spesso considerato un linguaggio *lento*. In buona misura questo è vero: è più lento di java, ad esempio.

Ma la velocità non è sempre il collo di bottiglia. Spesso la gestione della complessità è un problema più importante della velocità.

Vi sono comunque vari modi per rendere più veloci le parti “critiche” di un programma python.

performance

People are able to code complex algorithms in much less time by using a high-level language like Python (e.g., also C++). There can be a performance penalty in the most pure sense of the term.

optimization

"The best performance improvement is the transition from the nonworking to the working state."

--John Ousterhout

"Premature optimization is the root of all evil."

--Donald Knuth

"You can always optimize it later."

-- Unknown

l'interprete/1

- Python è un linguaggio interpretato
- L'interprete esegue una compilazione del sorgente in bytecode, che viene poi eseguito su una virtual machine, come in Java
- L'interprete è anche un eccellente “calcolatore”, da usare in interattivo!

```
>>> 2**1024
179769313486231590772930519078902473361797697894230657273430081157732675805500963132708477322407536021120113879871393357658789768814416622492847430639474124377767893424865485276302219601246094119453082952085005768838150682342462881473913110540827237163350510684586298239947245938479716304835356329624224137216L
```

l'interprete/2

- Quando usato in interattivo, l'interprete agisce in maniera leggermente diversa.
- Il prompt è “>>>”
- Se una espressione ha un valore, esso viene stampato automaticamente:

```
>>> 34 + 55 – 2

87
```

l'interprete/3

- Qualsiasi errore possa avvenire nel corso dell'esecuzione dell'interprete in interattivo, l'interprete sopravvive, anche in caso di SyntaxError:

```
>>> 5/0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: integer division or modulo by zero
>>> fact(100)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'fact' is not defined
>>> @@@ ausdf9 !?;<j
  File "<stdin>", line 1
    @@@ ausdf9 !?;<j
     ^
SyntaxError: invalid syntax
>>>
```

print

Per stampare si usa *print*.

Può prendere un numero arbitrario di argomenti:

```
>>> print a, b, c
```

Normalmente stampa un newline al termine; non lo fa se la lista degli argomenti termina con una virgola:

```
>>> print a, b, c,
```

numeri interi

- In python <3.0 vi sono due tipi di interi:
  - *int* per interi fino a 2**63-1 (vedi *sys.maxint*)
  - *long* per interi di qualsiasi dimensione!
- Gli int vengono automaticamente trasformati in long quando necessario:

```
>>> print a, type(a)
9223372036854775807 <type 'int'>
>>> a += 1
>>> print a, type(a)
9223372036854775808 <type 'long'>
```

numeri floating point

- I numeri floating point sono rappresentati dal tipo float:

```
>>> 2.0**-1024
5.562684646268003e-309
>>>
```

test *(interactive)*

- Eseguire le seguenti operazioni:
  - 2**1024
  - 100/3
  - 100//3 # floordiv
  - 100.0/3
  - 100.0//3
  - 100%3
  - divmod(100, 3)

numeri complessi

- Esiste il tipo *complex*:

```
>>> z = 3.0 + 4.0j
>>> w = 3.0 - 4.0j
>>> z+w
(6+0j)
>>> z*w
(25+0j)
>>> type(z)
<type 'complex'>
>>> print z.real, z.imag, abs(z)
3.0 4.0 5.0
>>>
```

variabili/1

È possibile assegnare ad un qualsiasi oggetto un nome simbolico, che non necessita di essere dichiarato.

```
>>> a = 5
>>> b = 3.2
>>> c = a
>>> C = b   # non è “c”, è un'altra variabile
>>>
```

variabili/2

Anche se è un po' prematuro spiegarlo ora, in realtà i nomi simbolici *a*, *b, c* e *C* non sono affatto *variabili*.

Inoltre, anche il simbolo “=” non è affatto quello che sembra!

Per ora comunque possiamo proseguire “facendo finta” che questi simboli siano variabili e che “=” esegua un assegnamento o copia, come in C,

C++ o Fortran.

operatori

Sono disponibili i comuni operatori:

- + (somma, concatenazione)
- - (differenza)
- * (prodotto)
- / (divisione)
- // (divisione intera)
- % (modulo)
- ...

operatori binari

Come in C, agli operatori binari sono associati operatori di assegnamento:

- += (incremento)
- -= (decremento)
- …

Dunque,

```
>>> a = 10
>>> a += 4
>>> print a
14
>>>
```

stringhe/1

- Il tipo *str* è comunemente utilizzato per le stringhe. Esiste anche il tipo *unicode*.
- Possono essere create indifferentemente con apici singoli (`'alfa'`) o doppi (`“alfa”`)

```
>>> "alfa" + 'beta'
'alfabeta'
>>>
```

stringhe/2

- Le sequenze di apici tripli “”” o ''' possono essere utilizzate per stringhe che spaziano su più righe, o che contengono apici singoli o doppi (o tripli dell'altro tipo):

```
>>> a = """Questa stringa occupa due righe,
... e contiene apici 'singoli', "doppi" e '''tripli'''"""
>>> print a
Questa stringa occupa due righe,
e contiene apici 'singoli', "doppi" e '''tripli'''
>>>
```

stringhe/3

- I caratteri di escape sono più o meno come in C:

```
>>> print "alfa\nbeta\tgamma"
alfa
beta    gamma
>>>
```

stringhe/4

- È possibile creare stringhe *raw* (senza sostituzione di *escape*) con costanti letterali come:

```
>>> print r"alfa\nbeta\tgamma"
alfa\nbeta\tgamma
>>>
```

- Questo risulta particolarmente utile per definire le *regular expression*

stringhe/5

È possibile compiere varie operazioni sulle stringhe:

```
>>> s = "Ciao, mondo!"
>>> print s.lower()
ciao, mondo!
>>> print s.upper()
CIAO, MONDO!
>>> print s.title()
Ciao, Mondo!
>>> print s.replace('o', 'x')
Ciax, mxndx!
>>>
```

stringhe/6

```
>>> print s.find('nd')
8
>>> print len(s)
12
>>> print “Ciao, mondo!”.upper()
CIAO, MONDO!
>>>
```

stringhe/7

```
>>> print “Ciao, mondo!”.toenglish()
Hello, world!
>>> print “Ciao, mondo!”.tofrench()
Bonjour, tout le monde!
>>>
```

D'accordo, scherzavo... queste non esistono!

stringhe/8

È possibile accedere ai singoli elementi della stringa, o a sottostringhe:

```
>>> hi_folk = "Hi, folk!"
>>> print hi_folk[0]
H
>>> print hi_folk[4]
f
>>> print hi_folk[-1] # ultimo elemento
!
>>> print hi_folk[2:] # dal secondo elemento (incluso) fino alla fine
, folk!
>>> print hi_folk[:3] # fino al terzo elemento (escluso)
Hi,
>>> print hi_folk[2:5] # dal secondo elemento (incluso) al quinto escluso
, f
>>>
```

stringhe/9

“Stranamente”, le stringhe non sono modificabili:

```
>>> hi_folk[1] = 'X'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not
support item assignment
>>>
```

python program file

```
#!/usr/bin/env python


a = 10
print a
```

test *(nome.py)*

- Definire una stringa *nome* contenente il proprio nome, una stringa *cognome* contenente il proprio cognome;
- Stampare la lunghezza delle due stringhe;
- Concatenare le due stringhe formando *nome_cognome;*
- Stampare la lunghezza di nome_cognome.

contenitori

- Uno dei punti di forza di python è nei contenitori disponibili, che sono molto efficienti, comodi da usare, e versatili:
  - *tuple ()*
  - *list []*
  - *dict {}*
  - *set*

tuple/1

```
>>> a = (3, 4, 5)
>>> print a
(3, 4, 5)
>>> print a[1]    # indici da 0 a len-1!
4
>>> b = 2, 3
>>> print b
(2, 3)
```

tuple/2

- Non sono necessariamente omogenee!

```
>>> a = (4, z, "alfa", b)
>>> a
(4, (3+4j), 'alfa', (2, 3))
```

tuple/3

- Possono anche stare alla sinistra dell'uguale:

```
>>> r, i = z.real, z.imag
>>> print z
3.0+4.0j
>>> print r
3.0
>>> print i
4.0
```

tuple/4

Definiscono altre operazioni:

```
>>> a = (1, 2, 3)
>>> b = (4, 5, 6)
>>> print a+b
(1, 2, 3, 4, 5, 6)
>>> print a*3
(1, 2, 3, 1, 2, 3, 1, 2, 3)
>>>
```

tuple/5

La virgola è importante!

```
>>> x = 1
>>> print type(x), x
<type 'int'> 1
>>> y = 1,
>>> print type(y), y
<type 'tuple'> (1,)
>>>
```

tuple/6

Lo slicing consente di accedere a “porzioni” della tupla:

```
>>> a = (0, 1, 2, 3, 4)
>>> print a[1:3]  # dal secondo elemento (incluso)
                  # al quarto (escluso)
(1, 2)
>>> print a[:2]   # dal primo (incluso) al terzo (escluso)
(0, 1)
>>> print a[2:]   # dal terzo (incluso) all'ultimo (incluso)
(2, 3, 4)
>>> print a[2:] + a[:2]
(2, 3, 4, 0, 1)
>>>
```

test *(nome.py)*

Partendo dal file *nome.py*, definire una tupla contenente il proprio nome, il proprio cognome, e l'anno di nascita.

Costruire, a partire da essa, una nuova tupla contenente, nell'ordine, anno di nascita, nome, cognome).

liste/1

Le liste sono “tuple modificabili”:

```
>>> l = [1, 2, 3]
>>> print l
[1, 2, 3]
>>> l.append(4)
>>> print l
[1, 2, 3, 4]
>>> l.insert(2, "XYZ")
>>> print l
[1, 2, 'XYZ', 3, 4]
>>> print len(l)
5
```

liste/2

```
>>> print l[0], l[-1]
1 4
>>> print l[:2]
[1, 2]
>>> print l[-4:]
[2, 'XYZ', 3, 4]
>>> l[1] = 3, 2, 1, 0
>>> print l
[1, (3, 2, 1, 0), 'XYZ', 3, 4]
>>>
```

liste/3

```
>>> l = [3, 5, 7, 9, 11, 13]
>>> l.append(2)
>>> l.remove(9)
>>> print l
[3, 5, 7, 11, 13, 2]
>>> l.sort()
>>> print l
[2, 3, 5, 7, 11, 13]
>>> l.reverse()
>>> print l
[13, 11, 7, 5, 3, 2]
>>>
```

liste/4

Le liste possono essere utilizzate come stack (last-in, first-out):

```
>>> print l
[13, 11, 7, 5, 3, 2]
>>> l.pop()
2
>>> l.pop()
3
>>> l.pop()
5
>>> l.pop()
7
>>> print l
[13, 11]
>>>
```

liste/5

```
>>> l *= 3
>>> print l
[13, 11, 13, 11, 13, 11]
>>> l[1:4] = ['a', 'b', 'c']
>>> print l
[13, 'a', 'b', 'c', 13, 11]
>>> print l.count(13)
2
>>> print l.count(7)
0
>>> print l.index('c')
3
```

liste/6

```
>>> l.extend( (2, 3, 5) )
>>> print l
[13, 'a', 'b', 'c', 13, 11, 2, 3, 5]
>>> del l[:-4]
>>> print l
[11, 2, 3, 5]
>>>
```

range

- Range è una funzione per generare liste di interi:

```
>>> print range(3)
[0, 1, 2]
>>> print range(3, 7)
[3, 4, 5, 6]
>>> print range(3, 10, 2)
[3, 5, 7, 9]
```

test *(books.py)*

Definite una lista contenente alcuni nomi di libri che avete letto.

Aggiungete qualche altro elemento alla lista.

Stampate il numero di elementi della lista.

Riordinate la lista.

extended slices/1

La sintassi dello slicing accetta ora un parametro *stride*, sempre separato da :

```
>>> l = range(20)
>>> l[1:18]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17]
>>> l[1:18:3]
[1, 4, 7, 10, 13, 16]
>>> l[1::3]
[1, 4, 7, 10, 13, 16, 19]
>>> l[:5:3]
[0, 3]
>>> l[::3]
[0, 3, 6, 9, 12, 15, 18]
>>>
```

extended slices/2

Lo *stride* può anche essere negativo

```
>>> l[3:18:3]
[3, 6, 9, 12, 15]
>>> l[18:3:-3]
[18, 15, 12, 9, 6]
>>> l[17:2:-3]
[17, 14, 11, 8, 5]
>>>  l[::-1]
[19, 18, 17, 16, 15, 14, 13, 12, 11, 10, 9, 8, 7,
6, 5, 4, 3, 2, 1, 0]
>>>
```

extended slices/3

Quando si assegna con lo slicing, le extended slices sono meno flessibili delle regular slices, infatti la lunghezza degli operandi deve essere identica:

```
>>> l = range(5)
>>> r = ['a', 'b', 'c', 'd']
>>> r[1:3] = l[:-1]  # len(r[1:3]) != len(l[:-1]), ok
>>> r
['a', 0, 1, 2, 3, 'd']
>>> r[::3] = l[::2]  # len(r[::2]) != len(l[::2]), KO!!!
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: attempt to assign sequence of size 3 to extended slice of size 2
>>> r[::2] = l[::2]  # len(r[::2]) == len(l[::2]), ok
>>> r
[0, 0, 2, 2, 4, 'd']
>>>
```

extended slices/4

Quando si eliminano elementi con lo slicing, non ci sono problemi:

```
>>> l = range(11)
>>> del l[::3]
>>> l
[1, 2, 4, 5, 7, 8, 10]
>>> del l[::-2]
>>> l
[2, 5, 8]
>>>
```

set/1

- I set definiscono insiemi di elementi senza ripetizione. Non sono ordinati (nel senso che sono ordinati automaticamente al fine di rendere veloce la ricerca di elementi).

```
>>> s = set()
>>> s.add(2)
>>> s.add(3)
>>> s.add(2)
>>> s.add(4)
>>> s
set([2, 3, 4])
>>> u = set(['alfa', 2, 3.5])
>>> print u
set([3.5, 2, 'alfa'])
>
```

set/2

```
>>> print s
set([2, 3, 4])
>>> l=[1, 2, 2, 3, 3, 3, 4, 4, 4, 4]
>>> t = set(l)
>>> print t
set([1, 2, 3, 4])
>>> s.intersection(t)
set([2, 3, 4])
>>> s.difference(t)
set([])
>>> t.difference(s)
set([1])
>>>
```

set/3

```
>>> s.symmetric_difference(t)
set([1])
>>> s.union(t)
set([1, 2, 3, 4])
>>> s.discard(3)
>>> print s
set([2, 4])
>>> s.clear()
>>> print s
set([])
>>>
```

frozenset/4

Per analogia con le tuple, che possono essere considerate “liste congelate”, esistono i frozenset:

```
>>> ft = frozenset(t)
>>> ft
frozenset([1, 2, 3, 4])
>>> ft.add(2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'frozenset' object has no attribute
'add'
>>>
```

test *(primes_1.py)*

Avendo a disposizione solo queste strutture dati:

```
>>> all = set(range(1, 20))
>>> primes = set([1, 2, 3, 5, 7, 11, 13, 17, 19])
>>> even = set([2, 4, 6, 8, 10, 12, 14, 16, 18])
```

stampare l'insieme dei numeri dispari, l'insieme dei numeri dispari primi, e l'insieme dei numeri dispari non primi (inferiori a 20).

dizionari/1

- I dizionari implementano array associativi; associano ad una chiave arbitraria un valore arbitrario:

```
>>> atomic_number = {'H': 1, 'He': 2,
'C': 6, 'Fe': 26}
>>> print atomic_number['C']
6
>>> print atomic_number
{'H': 1, 'C': 6, 'Fe': 26, 'He': 2}
>>>
```

dizionari/2

- Non sono necessariamente omogenei:

```
>>> d = {}
>>> d['alfa'] = 3
>>> d[2.5] = 'xyz'
>>> d[3+4j] = [3, 4, 5]
>>> d[(1,2,3)] = { 'x': 2, 'y': 3, 'z': 1 }
>>> print d
{2.5: 'xyz', (1, 2, 3): {'y': 3, 'x': 2, 'z':
1}, 'alfa': 3, (3+4j): [3, 4, 5]}
>>>
```

dizionari/3

```
>>> print d.keys()
[2.5, (1, 2, 3), 'alfa', (3+4j)]
>>> print d.values()
['xyz', {'y': 3, 'x': 2, 'z': 1}, 3, [3, 4, 5]]
>>> print d.items()
[(2.5, 'xyz'), ((1, 2, 3), {'y': 3, 'x': 2, 'z': 1}),
('alfa', 3), ((3+4j), [3, 4, 5])]
>>> d.has_key('alfa')
True
>>> d.has_key('beta')
False
>>>
```

dizionari/4

```
>>> d.get('alfa', -1999)
3
>>> d.get('beta', -1999)
-1999
>>> print d
{2.5: 'xyz', (1, 2, 3): {'y': 3, 'x': 2, 'z': 1}, 'alfa': 3, (3+4j):
[3, 4, 5]}
>>> d.setdefault('alfa', -1999)
3
>>> d.setdefault('beta', -1999)
-1999
>>> print d
{2.5: 'xyz', 'beta': -1999, (1, 2, 3): {'y': 3, 'x': 2, 'z': 1},
'alfa': 3, (3+4j): [3, 4, 5]}
>>>
```

dizionari/5

Possono essere usati come stack:

```
>>> d.pop('alfa', -5)
3
>>> d.pop('gamma', -5)
-5
>>> d.popitem()
(2.5, 'xyz')
>>> d.popitem()
('beta', -1999)
>>> print d
{(1, 2, 3): {'y': 3, 'x': 2, 'z': 1}, (3+4j): [3, 4, 5]}
>>>
```

dizionari/6

```
>>> d1 = dict(a=1, b=2, c=3, d=4)
>>> d2 = {1: 1.0, 2: 2.0}
>>> d1.update(d2)
>>> print d1
{'a': 1, 1: 1.0, 'c': 3, 'b': 2, 'd': 4, 2:
2.0}
>>> d1.clear()
>>> print d1
{}
>>>
```

test *(books.py)*

Definite un dizionario che metta in relazione alcuni dei libri che avete letto con il nome del relativo autore.

Stampare i nomi di tutti gli autori (senza ripetizioni).

costrutti per il controllo del flusso

- Python ha pochi costrutti per il controllo del flusso, secondo la filosofia della massima semplicità.

indentazione

- In python l'indentazione è sintattica, vale a dire, determina l'annidamento degli statement.
- Questo è parte integrante della filosofia di python: siccome indentare è un bene, e siccome un programma senza indentazione è un male, perché non obbligare ad indentare?
- Diventa quindi inutile l'uso delle parentesi graffe per racchiudere blocchi, come in C/C++, o di statement di chiusura, come l'END DO del Fortran.

if-elif-else

```
>>> if a == b:
...     print a
... elif a > b:
...     print b
... else:
...     print a
...
```

for/1

- Il ciclo for consente di iterare su “oggetti iterabili”, come liste, tuple, set, dizionari, …

```
>>> for i in range(3):
...     print i
...
0
1
2
>>>
```

for/2

```
>>> t = ('a', 'b', 10, 5.5)
>>> for i in t:
...     print i
...
a
b
10
5.5
>>>
```

for/3

```
>>> d = {'a': 0, 'b': 1, 'c': 2}
>>> for key in d.keys():
...     print key
...
a
c
b
>>>
```

for/4

```
>>> for val in d.values():
...     print val
...
0
2
1
>>>
```

for/5

```
>>> for key, val in d.items():
...     print key, '=', val
...
a = 0
c = 2
b = 1
>>>
```

for/6

```
>>> for key in d:
...     print key
...
a
c
b
>>>
```

for/7

```
>>> for key in ('a', 'd', 'c'):
...     print d.get(key, None)
...
0
None
2
>>>
```

for/8

```
>>> s = set(range(0, 10, 2)+range(0, 10, 3))
>>> print s
set([0, 2, 3, 4, 6, 8, 9])
>>> for i in s:
...   print i
...
0
2
3
4
6
8
9
>>>
```

test *(books.py)*

Partendo dal dizionario libri : autori, scrivere il codice per stampare, per ciascun autore, il numero di libri presenti nel dizionario.

while/1

- Il ciclo while è il generico ciclo con una condizione:

```
>>> i = 0
>>> while i < 4:
...     print i
...     i += 1
...
0
1
2
3
>>>
```

while/2

- Il ciclo precedente può essere sostituito da questo:

```
>>> for i in range(4):
...     print i
...
0
1
2
3
>>>
```

break

Break permette di uscire dal loop:

```
>>> for i in range(10000):
...     print i
...     if i%3 == 2:
...         break
...
0
1
2
>>>
```

continue

Continue permette di passare all'iterazione successiva:

```
>>> for i in range(4):
...     if i == 1:
...         continue
...     print i
...
0
2
3
>>>
```

loops: else clause

- I loop (for e while) possono avere una clausola *else*, che viene eseguita solo se non si esce dal loop con un *break*; ovvero, se il loop completa naturalmente:

```
>>> for i in range(10):
...   if i > 3: break
...   print i
... else:
...   print "finito!"
...
0
1
2
3
>>>
```

```
>>> for i in range(2):
...   if i > 3: break
...   print i
... else:
...   print "finito!"
...
0
1
finito!
>>>
```

switch

Non esiste un costrutto switch, basta una serie di blocchi *if-elif-else.*

operatori

Vi sono vari operatori:

- Operatori di comparazione: *==, !=, <, <=, >, >=, is, in*
- Operatori logici: *and, or, not*

is

L'espressione *a == b* restituisce *True* se il valore di *a* è identico al valore di *b*.

L'espressione *a is b* restituisce *True* se *a* e *b* si riferiscono allo stesso oggetto fisico

```
>>> l1 = [1, 3, 8]
>>> l2 = [1, 3, 8]
>>> l3 = l1
>>> print l1 == l2, l1 is l2
True False
>>> print l1 == l3, l1 is l3
True True
>>> print l2 == l3, l2 is l3
True False
>>>
```

in

L'espressione *a in lst* restituisce *True* se l'oggetto *a* è contenuto in *lst.*

```
>>> print 2 in l1
False
>>> print 3 in l1
True
>>>
```

None/1

- None è un oggetto particolare del linguaggio, che viene utilizzato per indicare l'assenza di un valore o un valore indefinito

```
>>> a = None
>>> print a
None
>>>
```

bool/1

- Il tipo bool viene utilizzato per i valori logici di verità/falsità
- Un oggetto bool può assumere uno di questi valori:
  - `True`
  - `False`

bool/2

```
>>> a = 1 > 5
>>> print a
False
>>> b = 1 <= 5
>>> print b
True
>>>
```

conversioni a bool/1

- I tipi predefiniti possono essere convertiti a bool, nel senso che possono essere usati in espressioni condizionali.
  - Un *int* uguale a 0 equivale a *False*, altrimenti *True*
  - Un *float* uguale a 0.0 equivale a *False*, altrimenti *True*
  - Una stringa vuota *“”* equivale a *False*, altrimenti *True*
  - Un contenitore vuoto([], (), set(), {}, ...) equivale a *False*, altrimenti *True*
  - *None* equivale a *False*

funzioni/1

- Le funzioni si dichiarano con *def*, seguito dal nome della funzione e dalla lista dei parametri.
- Come al solito, non si dichiara nulla, né il tipo del valore di ritorno, né il tipo dei parametri: *duck typing!*

funzioni/2

```
>>> def sum(a, b):
...   return a+b
...
>>>
>>> print sum(3, 4)
7
>>> print sum("Ciao, ", "mondo!")
Ciao, mondo!
>>> print sum(3.2, 9.1)
12.3
>>> print sum(6, "ccc")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in sum
TypeError: unsupported operand type(s) for +: 'int' and 'str'
>>>
```

funzioni/3

```
>>> def factorial(n):
...   if n < 2:
...     return 1
...   else:
...     return n*factorial(n-1)
...
>>> for i in (0, 1, 5, 10, 200):
...   print factorial(i)
...
1
1
120
3628800
788657867364790503552363213932185062295135977687173263294742533244359449963403342920304284011984623904177212138919638830257642790242637105061926624952829931113462857270763317237396988943922445621451664240254033291864131227428294853277524242407573903240321257405579568660226031904170324062351700858796178922227896237038973747200000000000000000000000000000000000000000000000000
>>>
```

funzioni/4

Qualsiasi funzione ha un valore di ritorno. Per default, questo valore è None. Se si vuole dare un valore di ritorno specifico, ad esempio 4, ad una funzione, basta aggiungere lo statement *return 4*.

Uno statement *return* senza espressioni alla destra equivale a *return None*.

Se una funzione termina senza aver incontrato alcun *return*, esegue implicitamente un *return None.*

passaggio di parametri/1

- Gli argomenti di una funzione possono essere passati per posizione o per nome:

```
>>> def count(lst, val):
...     c = 0
...     for el in lst:
...         if el == val: c += 1
...     return c
...
>>> print count([1,2,1,3,2,4], 2)
2
>>> print count(val=2, lst=[1,2,1,3,2,4])
2
```

passaggio di parametri/2

- Dopo aver passato almeno un argomento per nome, non è più possibile passarli per posizione:

```
>>> print count(val=2, [1,2,1,3,2,4,1])
  File "<stdin>", line 1
SyntaxError: non-keyword arg after keyword
arg
>>>
```

argomenti di default/1

- Gli argomenti di una funzione possono avere valori di default:

```
>>> def count(lst, val=1):
...     c = 0
...     for el in lst:
...         if el == val: c += 1
...     return c
...
>>> print count([1,2,1,3,2,4,1], 2)
2
>>> print count([1,2,1,3,2,4,1])
3
>>>
```

argomenti di default/2

- Se un argomento accetta un valore di default, anche tutti i successivi argomenti devono avere un valore di default:

```
>>> def f(a, b=0, c):   # ambiguità
...     print a+b
...
  File "<stdin>", line 1
SyntaxError: non-default argument follows
default argument
>>>
```

arbitrary argument list/1

Una funzione può avere argomenti posizionali opzionali; questi argomenti vengono inseriti in una tupla:

```
>>> def f(a, *l):
...   print a
...   print l
...
>>> f("a")
a
()
>>> f("a", 2)
a
(2,)
>>> f("a", 2, 5, 'y')
a
(2, 5, 'y')
>>>
```

arbitrary argument list/2

```
>>> def sum(a, *l):
...   for i in l:
...     a += i
...   return a
...
>>> print sum(10)
10
>>> print sum(10, 1, 100)
111
>>> print sum(“a”, “bc”, “d”)
abcd
>>>
>>> print sum([1], [], [2, 3], [4, 5, 6], range(7, 10))
[1, 2, 3, 4, 5, 6, 7, 8, 9]
>>>
```

arbitrary keyword arguments

Gli argomenti opzionali possono essere previsti anche per nome; in tal caso, vengono mantenuti in un dizionario:

```
>>> def g(a, **kw):
...   print a
...   print kw
...
>>> g(5, y=9, z=5, x=1)
5
{'y': 9, 'x': 1, 'z': 5}
>>> g(5, y=9, z=5, x=1, a=1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: g() got multiple values for keyword argument 'a'
>>>
```

forma generale di una funzione

- Riassumendo, la forma generale di una funzione è:
- >>> def f(a, b, c=0, *l, **kw):
- ... print a, b, c, l, kw
- ...
- >>> f(1, 2)
- 1 2 0 () {}
- >>> f(1, 2, 3)
- 1 2 3 () {}
- >>> f(1, 2, 3, 4, 5, 6, 7)
- 1 2 3 (4, 5, 6, 7) {}
- >>> f(1, 2, 3, 4, 5, 6, 7, x=0, y=1, alfa=2)
- 1 2 3 (4, 5, 6, 7) {'y': 1, 'x': 0, 'alfa': 2}
- >>>

unpacking argument list

A volte può capitare di avere, in una lista, il valore degli argomenti posizionali di una funzione:

```
>>> def f(a, b, c):
...     return a*b-c
...
>>> args = [3, 5, -2]
>>> print f(*args) ### => f(3, 5, -2)
17
>>>
```

unpacking keyword arguments

A volte può capitare di avere, in un dizionario, una lista di argomenti da passare per nome ad una funzione:

```
>>> def f(x, y, z):
...     return x**2 + y**2 + z**2
...
>>> args = {'x': 3, 'y': 4, 'z': 5}
>>> print f(**args) ### => f(x=3, y=4, z=5)
50
>>>
```

doc string/1

È possibile associare ad una funzione una *doc string*, una stringa arbitraria di documentazione per la funzione. La stringa diventa parte integrante della funzione; se il nome della funzione è *fun*, *fun.__doc__* è la sua *doc string*.

Per default, la doc string vale *None*.

doc string/2

```
>>> def sum(a, b, *l):
...   """Somma due o piu' numeri; ad esempio,
...      >>> print sum(2, 4, 10, 20, 30)
...      66
...      >>>"""
...   r = a + b
...   for el in l:
...     r += el
...   return r
...
>>> print sum.__doc__
Somma due o piu' numeri; ad esempio,
     >>> print sum(2, 4, 10, 20, 30)
     66
     >>>
>>>
```

doc string/3

```
>>> print range.__doc__ # le funzioni built-in hanno una doc!
range([start,] stop[, step]) -> list of integers

Return a list containing an arithmetic progression of integers.
range(i, j) returns [i, i+1, i+2, ..., j-1]; start (!) defaults
to 0.
When step is given, it specifies the increment (or decrement).
For example, range(4) returns [0, 1, 2, 3].  The end point is
omitted!
These are exactly the valid indices for a list of 4 elements.
>>>
```

pass

- Pass è una generica istruzione che non fa niente, e può stare in qualsiasi posizione del codice. È utile perché in alcune situazioni è necessario uno statement per ragioni sintattiche. Ad esempio, il corpo di una funzione non può essere omesso. Se si vuole una funzione vuota, si fa così:

```
>>> def f(a, b, c, d=0):
...     pass
...
>>>
```