

Strutturare il codice

Informatica@SEFA 2018/2019 - Lezione 7

Massimo Lauria <massimo.lauria@uniroma1.it>
<http://massimolauria.net/courses/infosefa2018/>

Mercoledì, 10 Ottobre 2018

Valori indefiniti

Valore None

None indica un valore **non definito**.

```
>>> x = None
>>> type(x)
<class 'NoneType'>
>>> x
>>> print(x)
None
```

Nella sessione interattiva l'espressione `x` con valore `None` non dà nessun valore. Confrontate col caso seguente...

```
>>> x = 10
>>> type(x)
<class 'int'>
>>> x
10
>>> print(x)
10
```

Variabili e valori indefiniti

Variabile indefinita

```
>>> (2 + sconosciuto)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'sconosciuto' is not defined
```

Variabile **definita** associata a valore indefinito

```
>>> sconosciuto = None
>>> (2 + sconosciuto)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'NoneType'
```

Motivazione

A che serve rappresentare valori non definiti?

- dati incompleti
- espressioni ancora da calcolare
- espressioni non calcolabili (dati mancanti)

E.g. Il minimo di $\{4, 7, 10, -3, 1\}$ è il numero -3 . Ma il minimo dell'insieme vuoto non è una nozione ben definita.

Testare se una variabile è None

```
if x is None:                                     1  
    print("il valore di x è INDEFINITO")          2
```

```
if x is not None:                                 1  
    print("il valore di x è DEFINITO")            2
```

Le espressioni

- `x is None`
- `x is not None`

hanno valori booleani, e testano rispettivamente se la variabile `x` abbia un valore indefinito o meno.

Ancora sul test il valore indefinito None

```
if x is not None:  
    print("il valore di x è DEFINITO")
```

1
2

è equivalente ma più leggibile, e quindi preferibile, a

```
if not (x is None):  
    print("il valore di x è DEFINITO")
```

1
2

Funzioni che non restituiscono un valore

Una funzione **non restituisce un valore** quando

- esegue `return` senza espressione
- termina senza eseguire un `return`

```
def esempio(x):  
    if x > 10:  
        return  
    elif x < 0:  
        return 10  
    y = 6          # istruzione inutile
```

1
2
3
4
5
6

Esempio

```
print("Nessun valore restituito.")      1
print(esempio(15))                     2
print("Restituito un valore intero.")  3
print(esempio(-1))                     4
print("Raggiunta la fine della funzione.") 5
print(esempio(4))                      6

# La variabile X ha valore indefinito  7
X = esempio(4)                         8
print( "Il valore di X è indefinito? --> " , X is None) 9
                                         10
```

```
Nessun valore restituito.
None
Restituisce un valore intero.
10
Raggiunta la fine della funzione.
None
True
```

Indentazione

Indentazione

L'inserimento di spazio vuoto all'inizio della riga, per

- identificare blocchi logici di codice
- rendere il codice più leggibile

Esempio di indentazione annidata

```
def scontato(prezzo,sconto):           1
    if sconto < 0 or sconto > 100:      2
        print("Errore nell'input")      3
        print("Lo sconto deve essere tra 0 e 100")  4
        return                          5
                                        6

    percentuale = 100 - sconto           7
    prezzo_scontato = prezzo * percentuale / 100  8
    return prezzo_scontato              9
                                        10

print(scontato(1000,20))                11
print(scontato(500,15))                 12
```

In Python l'indentazione è importante

Le istruzioni nello stesso blocco devono essere **allineate**

```
print("Prima riga")  
    print("seconda riga")
```

1
2

```
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
    File "/var/folders/kf/p7km5ptj1p52hvv5nl6j9gr80000gn/T/babel-oj12Dc/python-Q  
        print("seconda riga")  
        ^  
IndentationError: unexpected indent
```

```
print("Prima riga")  
print("seconda riga")
```

1
2

```
Prima riga  
seconda riga
```

Tab vs Spazi

Il carattere “tabulazione” (TAB) indica “aggiungi un livello di indentazione”. Sfortunatamente

- è visivamente uguale a una sequenza di spazi
- la sequenza ha lunghezza differente (2,3,4,8... spazi) a seconda della visualizzazione.

```
<spazio><spazio><spazio><spazio>istruzione1  
<tabulazione>istruzione2
```

In editor o terminali diversi si ottiene:

```
istruzione1  
istruzione2
```

```
istruzione1  
istruzione2
```

```
istruzione1  
    istruzione2
```

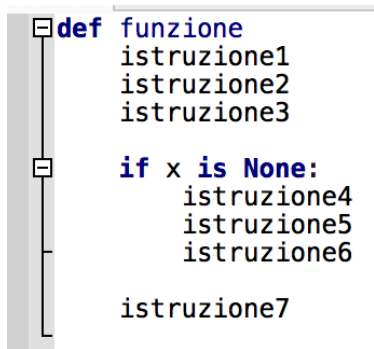
Tab vs Spazi in python3

In python3

- è vietato mischiare Tab e Spazi nell'identazione
- si consiglia di usare solo spazi (tipicamente 4 per livello)
- è possibile impostare l'editor così che inserisca 4 spazi ogni volta che si preme TAB.

Tab vs Space nell'editor Geany (I)

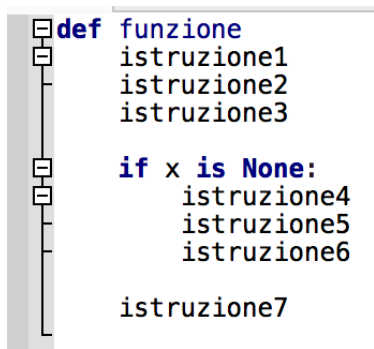
Geany evidenzia i livelli di indentazione. Linee spurie possono indicare che Tab e Spazi si sono mischiati.



```
def funzione
    istruzione1
    istruzione2
    istruzione3

    if x is None:
        istruzione4
        istruzione5
        istruzione6

    istruzione7
```

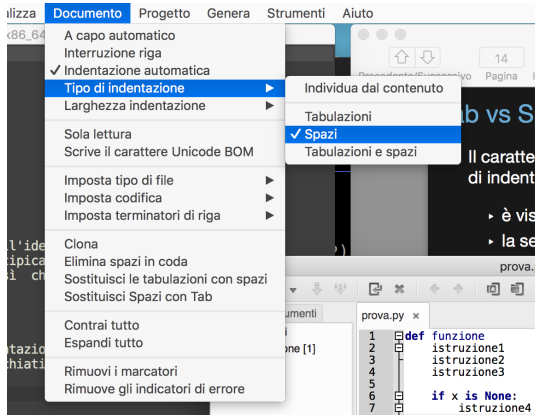


```
def funzione
    istruzione1
    istruzione2
    istruzione3

    if x is None:
        istruzione4
        istruzione5
        istruzione6

    istruzione7
```


Tab vs Space nell'editor Geany (II)



Quanto indentare

Io suggerisco 4 spazi.

- la lunghezza dell'indentazione è facoltativa
- non compromettete la leggibilità

```
x = 12                                     1
print("Primo livello di indentazione, 0 spazi") 2
if x > 0:                                     3
    print("Secondo livello di indentazione, 2 spazi") 4
    if x<100:                                 5
        print("Terzo livello di indentazione, 1 spazio") 6
else:                                         7
    print("Secondo livello di indentazione, 5 spazi") 8
```

De-indentare

Ridurre l'indentazione comunica al Python che la nuova istruzione fa parte di un blocco di codice più esterno, al quale questa **deve** essere allineata.

```
x = 10 1
2
def gruppo_istruzioni(): 3
    print("Tizio") 4
    print('Caio') 5
    print("Sempronio") 6
7
gruppo_istruzioni() 8
```

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

File "/var/folders/kf/p7km5ptj1p52hvez5nl6j9gr80000gn/T/babel-oj12Dc/python-u

gruppo_istruzioni()
~

IndentationError: unindent does not match any outer indentation level

Commenti e righe vuote

Ignorati da python. L'indentazione è sempre considerata rispetto alla precedente riga contenente vero codice.

```
x = 10 1
2
def gruppo_istruzioni(): 3
    print("Tizio") 4
    # commento mal indentato. Brutto ma corretto 5
    print('Caio') 6
    print("Sempronio") 7
8
    # altro commento mal indentato 9
gruppo_istruzioni() 10
```

```
Tizio
Caio
Sempronio
```

Intermezzo: gruppi di valori

Funzione che restituisce un **gruppo** di valori.

```
def quoziente_e_resto(N,D):           1
    if D==0:                           2
        return None    # pessima gestione dell'errore  3
                                        4

    quoziente = N // D                 5
    resto = N % D                     6

    return quoziente, resto           7
                                        8
print( quoziente_e_resto(10,4) )      9
print( quoziente_e_resto(7, 2) )     10
                                     11
```

(2, 2)

(3, 1)

La dimensione del gruppo è arbitraria

```
def valorimultipli(N):
    if N < 2 or 4 < N:
        return None # pessima gestione dell'errore

    if N == 2:
        return "uno","due"
    elif N == 3:
        return "uno","due","tre"
    else:
        # qui N vale 4
        return "uno","due","tre","quattro"

print( valorimultipli(1) )
print( valorimultipli(2) )
print( valorimultipli(3) )
print( valorimultipli(4) )
```

```
None
('uno', 'due')
('uno', 'due', 'tre')
('uno', 'due', 'tre', 'quattro')
```

Valutare le espressioni

Precedenze di operatori

1. Aritmetici

- `**` (unico valutato da destra a sinistra)
- segni `+` e `-` (per esempio `-2` e `+2.4`)
- `/`, `//`, `%`
- `+`, `-`

2. Confronti (stessa precedenza)

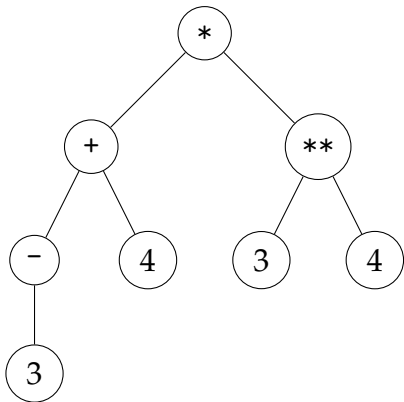
- `in`, `not in`, `is`, `is not`, `<`, `>`, `<=`, `>=`, `==`, `!=`

3. Logici

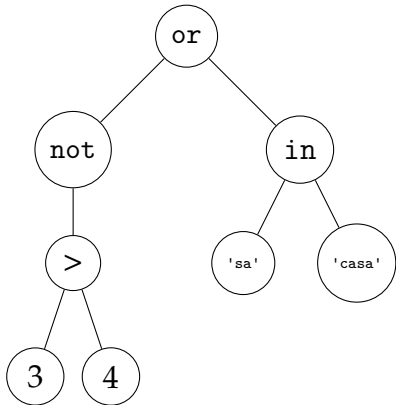
- `not` prima di `and` prima di `or`

Gli altri operatori sono nella **documentazione**

`(- 3 + 4) * 3**4`



`not 3 > 4 or 'sa' in 'casa'`



Esercizio: calcolare la seguente espressione

```
not -5//2**4 < -1 and 3 ** 2 ** (5 + - 3) >= 2*4
```

1

Questionario del laboratorio

bit.ly/INFO2018-07a

Usare e scrivere moduli Python

Modulo

Un file python è un **modulo**, ovvero un'unità che contiene funzioni e variabili pronte per essere riutilizzate.

```
import math  
print(math.pi * math.sin(0.4))
```

1

2

```
1.2233938033699718
```

I moduli python sono documentati

```
import math
```

1

```
help(math)
```

2

3

Help on module math:

NAME

math

MODULE REFERENCE

<https://docs.python.org/3.6/library/math>

The following documentation is automatically generated from the Python source files. It may be incomplete, incorrect or include features that are considered implementation detail and may vary between Python implementations. When in doubt, consult the module reference at the location listed above.

<.. TANTE ALTRE INFORMAZIONI....>

Anche le funzioni sono documentate

```
import math  
help(math.log)
```

1

2

Help on built-in function log in module math:

```
log(...)  
    log(x[, base])
```

Return the logarithm of x to the given base.
If the base not specified, returns the natural
logarithm (base e) of x.

Spazio dei nomi

In ogni punto e momento del programma esiste uno

spazio dei nomi

- nomi delle variabili e funzioni definite, moduli importati
- ad ogni nome corrisponde una sola entità python

<code>temp = 4.2</code>	1
<code>def temp():</code>	2
<code>return 5</code>	3
<code>print(type(temp))</code>	4
	5
	6

```
<class 'function'>
```

Usare i moduli inclusi in python

```
import nome_modulo
```

1

- nome_modulo va nello spazio dei nomi
- si accede via nome_modulo alle sue funzioni

```
import math  
print(math.pi)  
print(math.cos(0.3))
```

1

2

3

```
3.141592653589793  
0.955336489125606
```

Importare delle funzioni da un modulo

```
from nome_modulo import fun1  
from nome_modulo import fun2
```

1

2

```
from nome_modulo import fun1, fun2
```

1

- fun1, fun2 va nello spazio dei nomi
- nome_modulo **non è accessibile** nel programma

```
from math import pi,cos  
print(pi)  
print(cos(0.3))
```

1

2

3

```
3.141592653589793  
0.955336489125606
```

```
>>> print(math.cos(0.4))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'math' is not defined
```

```
>>> print(cos(0.4))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'cos' is not defined
```

```
>>> from math import sin,cos
```

```
>>> print(math.cos(0.4))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'math' is not defined
```

```
>>> print(cos(0.4))
0.9210609940028851
```

```
>>>
```

Riassumendo

```
import nome_modulo
```

1

- nome_modulo va nello spazio dei nomi
- si accede via nome_modulo alle sue funzioni

```
from nome_modulo import fun1, fun2
```

1

- fun1, fun2 va nello spazio dei nomi
- nome_modulo **non è accessibile** nel programma

Scriviamo il nostro file `primomodulo.py`

```
def quadrato(x):  
    return x**2  
  
def cubo(x):  
    return x**3  
  
# Un po' di codice di prova per testare  
print('codice che prova il modulo')  
print(cubo(2)+quadrato(3))
```

1
2
3
4
5
6
7
8
9

```
$ python3 primomodulo.py  
codice che prova il modulo  
17
```

Usiamo primomodulo.py come un modulo

```
import primomodulo 1
2
print("Codice principale:") 3
print(primomodulo.cubo(3)) 4
```

```
codice che prova il modulo
17
Codice principale:
27
```

Possiamo riutilizzare le funzioni di `primomodulo.py` !

Importare un modulo esegue **tutto** il suo codice. Ma in questo caso il codice di prova ci disturba!

Scriviamo secondomodulo.py

```
def quadrato(x):                                1
    return x**2                                  2
def cubo(x):                                    3
    return x**3                                  4

if __name__ == '__main__':                      5
    print('codice che prova il modulo')         6
    print(cubo(2)+quadrato(3))                  7
                                                8
```

```
$ python3 secondomodulo.py
codice che prova il modulo
17
```

```
import secondomodulo                            1
print("Codice principale:")                     2
print(secondomodulo.cubo(3))                    3
```

```
Codice principale:
27
```


Lecture

- Capitolo 4
- Paragrafo 5.5