Corso Python 2018

Niccolò Izzo

izzo.niccolo@gmail.com

D. Raffaele Jr. Di Campli

dcdrj.pub@gmail.com

POLITECNICO OPEN
unix LABS
Come hack with us.

Potete ottenere questa presentazione recandovi all’indirizzo:

slides.poul.org/2018/python

Perché Python?

- Ottimo per i principianti
- Facile da usare
- Estremamente diffuso
- Moduli per qualunque cosa

C

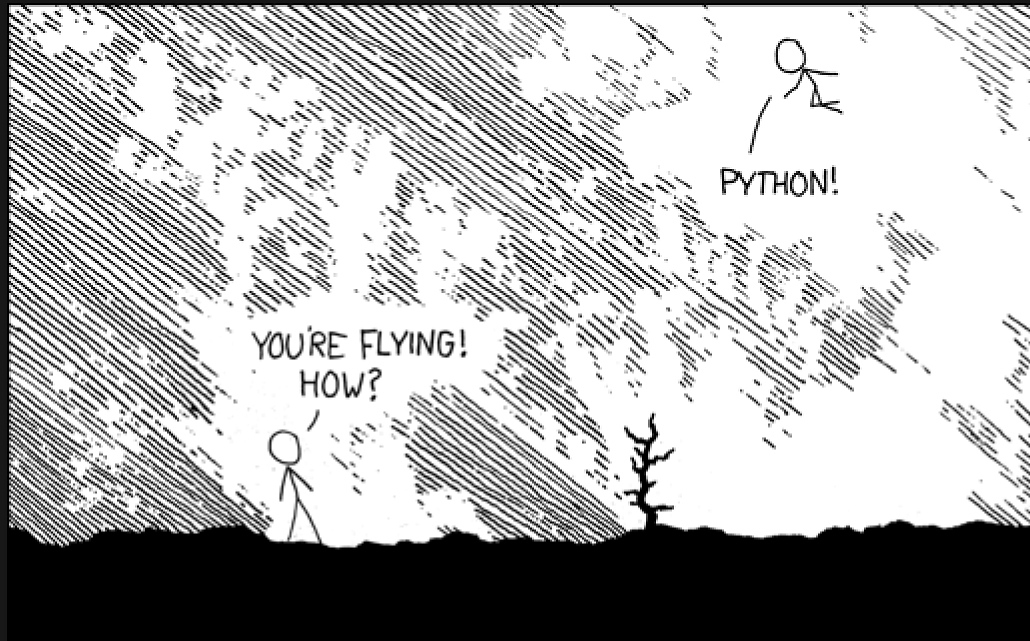
```c
#include <stdio.h>

int main() {
    printf("Hello, world!\n");
    return 0;
}
```

Java

```java
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Goodbye, world!");
    }
}
```

Python

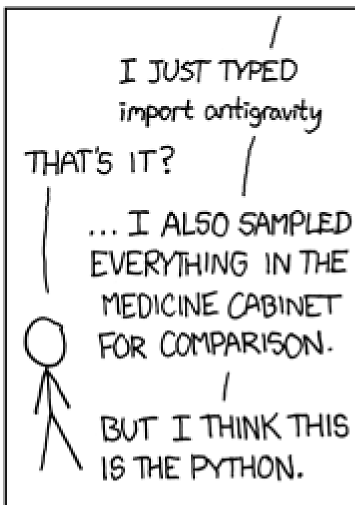
```
print('Hello, world!')
```



YOU'RE FLYING! HOW?
PYTHON!

I LEARNED IT LAST NIGHT! EVERYTHING IS SO SIMPLE!
HELLO WORLD IS JUST print "Hello, world!"

I DUNNO... DYNAMIC TYPING? WHITESPACE?
COME JOIN US! PROGRAMMING IS FUN AGAIN! IT'S A WHOLE NEW WORLD UP HERE!
BUT HOW ARE YOU FLYING?



I JUST TYPED import antigravity
THAT'S IT?
... I ALSO SAMPLED EVERYTHING IN THE MEDICINE CABINET FOR COMPARISON.
BUT I THINK THIS IS THE PYTHON.

Breve storia

- Nasce nel 1991 per mano di Guido van Rossum
- Prende il nome dai Monty Python
- È in continua evoluzione
- Convivono due versioni (2 e 3)
- Noi useremo Python 3
- Fatelo anche voi

Interpretato vs. compilato

- Compilato: tradotto in linguaggio macchina
- Interpretato: eseguito da un altro programma
- Python è interpretato (circa)

Cosa serve

- Per scrivere il codice, un editor di testo:
  - Multipiattaforma: Atom
  - Windows: Notepad++
- In quanto linguaggio interpretato, l’interprete Python
  - Linux: generalmente preinstallato, altrimenti utilizzare il package manager
  - macOS: Installer
  - Windows: 32bit o 64bit

Il linguaggio

REPL

Read, Eval, Print Loop

```
$ python3
Python 3.6.1 (default, Mar 27 2017, 00:27:06)
[GCC 6.3.1 20170306] on linux
Type "help", "copyright", "credits" or "license" for more info...
>>> print('Hello, world!')
Hello world!
```

Per uscire: `quit()` o `CTRL-D`

Espressioni

```
>>> 2 + 2
4
```

```
>>> 3 * 3
9
```

```
>>> 4 < 5
True
```

Commenti

Tutto ciò posto dopo il cancelletto (#) non è interpretato.

```
>>> 1 + 1  # qui posso scrivere quello che voglio
2
```

Errori

```
>>> 3 / 0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
```

Il segreto degli hacker? Leggere le scritte!

Variabili

```
>>> x = 6
>>> y = 2
>>> x
6
>>> y
2
```

```
>>> 2 * y
4
>>> x + y
8
```

```
>>> z = x + y
>>> z
8
```

Tipi

Cosa ho di fronte?

```
>>> type(3)
<class 'int'>
```

```
>>> type(3.14)
<class 'float'>
```

Numeri interi (`int`)

```
12          # notazione decimale
0xAB        # notazione esadecimale
0b101       # notazione binaria
```

Numeri decimali (`float`)

```
>>> 3.14 + 1.23
4.3740000000000006
```

```
>>> 100 / 3
33.333333333333336
```

```
>>> 100 // 3
33
```

Casting

- È l'azione di conversione del tipo
- Si usa il nome del tipo destinazione

```
>>> int(4.20)
4
```

```
>>> x = float(13)
>>> x
13.0
```

```
>>> type(x)
float
>>> type(13)
int
```

Booleani (`bool`)

- Possono assumere due valori: **True** o **False**
- Supportano operazioni logiche (**and**, **or**, **not**)
- Sono il risultato delle operazioni di confronto

```
>>> 4 < 5
True
```

```
>>> 3.14 < 2
False
```

```
>>> (4 < 5) or (3.14 < 2)
True
```

```
>>> (4 < 5) and (3.14 < 2)
False
```

Short-circuit evaluation

```
>>> (3.14 < 2) or print('Hello world!')
Hello world!
```

```
>>> (3.14 < 2) and print('Hello world!')
False
```

- È necessario che il valore di solo un membro dell’espressione sia falso per rendere tutta l’espressione falsa. Tutti gli altri termini non vengono valutati.
- Python valuta i membri di una espressione finché ritiene sia necessario per la corretta esecuzione del codice

Collezioni

Stringhe (`str`)

- Testo
- Racchiuse tra singoli apici (`'`) o virgolette (`"`)
- Per stringhe su più righe racchiuse tra tripli apici (`'''`) o virgolette (`"""`)
- Sono *immutabili*

```
>>> a = "Hello"
>>> b = "World"
>>> a + ' ' + b
'Hello World'
```

Liste (`list`)

- Collezione **ordinata** di elementi, anche di tipi diversi
- Si definiscono ponendo all'interno di parentesi quadre gli elementi intervallati da una virgola

```
lista = [False, 1, "due", 3.0, 4, 5]
```

- Si accede a un elemento della lista aggiungendo l'indice dell'elemento desiderato tra parentesi quadre
- Il primo elemento ha indice 0 (zero-based)

```
>>> lista[2]
'due'
```

```
>>> lista = [False, 1, "due", 3.0, 4, 5]
>>> lista
[False, 1, 'due', 3.0, 4, 5]
```

```
>>> lista[0] = 0.0
>>> lista
[0.0, 1, 'due', 3.0, 4, 5]
```

```
>>> lista.append('sei')
>>> lista
[0.0, 1, "due", 3.0, 4, 5, 'sei']
```

```
>>> lista.remove('sei')
>>> lista
[0.0, 1, "due", 3.0, 4, 5]
```

```
>>> lista.reverse()
>>> lista
[5, 4, 3.0, 'due', 1, 0.0]
```

```
>>> len(lista)
6
```

Slicing

- Permette di formare sottoinsiemi di elementi contigui
- Bisogna indicare un intervallo di indici, separando inizio e fine con due punti (`:`)
- L’elemento finale non è incluso

```
>>> lista = [0.0, 1, "due", 3.0, 4, 5, 'sei']
```

```
>>> lista[2:5]
['due', 3.0, 4]
```

Slicing

- È possibile indicare tutti gli elementi dall'inizio o fino alla fine di una lista omettendone l'indice

```
>>> lista[:4]
[0.0, 1, 'due', 3.0]
>>> lista[4:]
[4, 5, 'sei']
```

- Utilizzando indici negativi si parte a contare dal fondo

```
>>> lista[-4:]
[3.0, 4, 5, 'sei']  # ultimi 4 elementi
```

Unpacking

Estrarre valori da un contenitore

```
>>> t = (1, 2)
```

```
>>> a, b = t
>>> b
2
```

```
>>> a, b = b, a
>>> a
2
```

```
>>> c, _ = t
>>> c
1
```

L’operatore `in`

Valuta l’appartenenza di un elemento ad una collezione

```
>>> lista = ['a', 'b', 'c']
>>> 'c' in lista
True
```

- È un’espressione booleana
- Si può usare con l’operatore `not`

```
>>> lista = ['a', 'b', 'c']
>>> 'c' not in lista
False
```

Tuple (`tuple`)

- Esattamente come le liste, ma **immutabili**
- Sono quindi anche **ordinate**
- Si definiscono con le parentesi tonde

```
tupla = ('a', 1, 2, '3.0')
```

```
>>> tupla[1] = 1.0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

```
>>> tupla.append('sei')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'tuple' object has no attribute 'append'
```

Riferimenti

- Quando assegniamo un oggetto ad una variabile, la variabile contiene solo un *riferimento* all'oggetto
- **Non** l'oggetto stesso.

```
>>> spesa_a = ['mela', 'mango']
```

```
>>> spesa_b = spesa_a
>>> spesa_b.append('carota')
>>> spesa_b
['mela', 'mango', 'carota']
```

```
>>> spesa_a
['mela', 'mango', 'carota']
```

Insiemi (set)

- Come gli insiemi matematici
- **Non sono ordinati**
- Non contengono elementi duplicati

```
>>> frutta = {"mele", "pere", "zucchine", "mele"}
>>> frutta
{'mele', 'zucchine', 'pere'}
```

Supportano le operazioni insiemistiche

```
>>> frutta = {"mele", "pere", "zucchine", "mele"}
>>> verdure = {"zucchine", "verze", "coste", "porri"}
```

```
>>> frutta.union(verdure)
{'coste', 'mele', 'pere', 'porri', 'verze', 'zucchine'}
```

```
>>> frutta.intersection(verdure)
{'zucchine'}
```

```
>>> frutta.difference(verdure)
{'pere', 'mele'}
```

```
>>> verdure.difference(frutta)
{'verze', 'coste', 'porri'}
```

La sintesi è anche espressività

```
>>> frutta = {"mele", "pere", "zucchine", "mele"}
>>> verdure = {"zucchine", "verze", "coste", "porri"}
```

```
>>> frutta | verdure
{'porri', 'verze', 'pere', 'zucchine', 'coste', 'mele'}
```

```
>>> frutta & verdure
{'zucchine'}
```

```
>>> frutta - verdure
{'pere', 'mele'}
```

```
>>> verdure - frutta
{'verze', 'coste', 'porri'}
```

Dizionari (`dict`)

- Associano ogni chiave ad un valore
- Le chiavi devono essere immutabili (stringhe, tuple)

```
d = {
    "chiave": "valore",
    "nome":"Tancredi",
    "cognome":"Orlando",
    ('immutable', 'types'): ['are', 'cool']
    }
```

Si accede ai campi come si accede agli elementi di una lista, usando la chiave al posto dell'indice.

```
>>> d['nome']
'Tancredi'
```

```
>>> print('these things ' + ' '.join(d[('immutable', 'types')]))
these things are cool
```

Controllo del flusso

`if`

Se una condizione è vera esegue un blocco di codice.

```
if 3 > 2:
    print('Condizione vera')
```

Blocchi

- Sono corpi di codice allineati con una spaziatura multipla coerente
- Per bontà agli dèi usate quattro spazi ed evitate le tabulazioni come la peste

`else`

- Se la condizione di una istruzione `if` è falsa viene eseguito il blocco della istruzione `else` (se presente)

```
if 3 > 2:
    print('Condizione vera')
else:
    print('Condizione falsa')
```

`while`

- Valuta una espressione
- Se è vera esegue un blocco di codice
- Vai al primo punto

```
>>> x = 0
>>> while x < 3:
...     x = x + 1
...     print("ora x vale", x)
...
ora x vale 1
ora x vale 2
ora x vale 3
```

`for`

- Esegue un blocco per ogni elemento di una sequenza

```
>>> for i in [0, 1, 2, 3, 4]:
...     print(i)
0 1 2 3 4
```

- La funzione `range()` genera sequenze da usare all’interno di istruzioni `for`

```
>>> for i in range(0, 5):
...    print(i)
0 1 2 3 4
```

`break` e `continue`

- Sono entrambe istruzioni da usare all'interno di un ciclo, `while` o `for`
- Dopo una istruzione `break` tutto il ciclo è interrotto e la condizione non sarà più valutata
- Dopo una istruzione `continue` solo l'iterazione corrente è interrotta, il ciclo ripende a partire dalla valutazione della condizione

```
>>> for i in [0, 1, 2, 3, 4, 5, 6, 7]:
...     if i == 2:
...         continue
...     if i == 6:
...         break
...     print(i)
0 1 3 4 5
```

Funzioni

- Sono porzioni di codice a cui è assegnato un nome
- È possibile eseguire il codice di una funzione similmente a come si ottiene il valore contenuto da una variabile
- Permettono di riutilizzare il codice che scriviamo
- Si definiscono con l'istruzione `def`

```
>>> def say_hello():
...     print('Hello world!')
```

```
>>> say_hello()  # chiama la funzione
Hello world!
>>> say_hello()  # chiama la funzione nuovamente
Hello world!
```

- Le funzioni possono ottenere dati in ingresso, e fornire dati in uscita
- Per tale capacità permettono la riusabilità del codice

Parametri

- La funzione può essere “predisposta” per ricevere parametri
- I parametri sono i dati in ingresso di una funzione
- I parametri diventano variabili visibili all’interno della funzione per essere usati

Valore di ritorno

- Il valore di ritorno è ciò che fornisce in uscita la funzione, il proprio risultato
- Lo si specifica inserendo l'istruzione `return` seguita dal valore di ritorno
- La funzione termina subito dopo l'istruzione return

```
>>> def somma(a, b):
...     return a + b
```

```
>>> somma(1, 2)
3
```

```
>>> a, b = 3, 14
>>> c = somma(a, b)
>>> c
17
```

Parametri di default

- Alcuni parametri possono essere opzionali
- Se non specificati assumono un valore di default

```python
def f(a, b=3, c=6):
    print(a, b, c)
```

```python
>>> f(1)
1 3 6

>>> f(1, 2)
1 2 6

>>> f(1, c=4)
1 3 4
```

Scoping

- Lo scoping determina la **visibilità** delle variabili
- Le variabili definite all'interno di uno scope sono accessbili solo al suo interno
- Gli scope sono definiti da funzioni, classi e moduli
- Le classi sono contenitori di funzioni e di variabili
- I moduli sono contenitori di classi, di funzioni e di variabili

```
>>> x = 'globale'
>>> def f():
...     x = 'locale'
```

```
>>> f()
>>> print(x)
globale
```

- All’interno della funzione `f()` viene creata una seconda varibile di nome `x` accessibile solo all’interno della funzione
- La variabile `x` **all’interno della funzione** è chiamata **locale**
- La variabile `x` **al di fuori della funzione** è chiamata **globale**
- Sono due variabili **diverse**

Se all’interno della funzione non è stato usato il nome di una variabile globale è possibile accedere ad essa in lettura.

```
>>> x = 'globale'
>>> def f():
...     print(x)
>>> f()
globale
```

Tuttavia non è possibile modificarla.

```
>>> x = 'globale'
>>> def f():
...    print(x)
...    x = 'variabile ' + x
>>> f()
UnboundLocalError: local variable 'x' referenced before assignment
```

L’interprete ci informa che stiamo chiamando una variabile x che non è stata assegnata. Infatti dentro la funzione la variabile x non esiste.

None

- Equivale al `NULL` di C
- Una funzione ritorna `None` se priva di `return`
- Una variabile si può confrontare a None con l’istruzione di comparazione `is`

```
>>> def f():
...     print('funzione')
...     return None
```

```
>>> f()
funzione
>>> x = f()
>>> x
None
```

```
>>> if x is None:
...    print('esatto!')
esatto!
```

Moduli

- I file contenenti codice Python hanno estensione `.py`
- Ogni file è un **modulo**
- È possibile importare variabili, funzioni e classi da altri moduli con l'istruzione `import`
- modulo.py

```
default_lim = 3
def conta(lim=default_lim):
    for x in range(lim):
        print(x)
```

- programma.py

```
import modulo
x = modulo.default_lim + 2
modulo.conta(x)
```

- È possibile importare solo alcuni elementi da un modulo con la sintassi `from <modulo> import <oggetto>`
- Aiuta la leggibilità del codice
- Si omette il modulo di provenienza
- modulo.py

```
default_lim = 3
def conta(lim=default_lim):
    for x in range(lim):
        print(x)
```

- programma.py

```
from modulo import default_lim, conta
x = default_lim + 2
conta(x)
```

Il codice posto nello scope del modulo è comunque eseguito, anche se sono importati alcuni elementi con la sintassi `from ... import`.

- modulo.py

```
a = 'variabile'
def f():
    print('funzione')
print('modulo')
```

- dalla REPL

```
>>> from modulo import f
modulo
>>> f()
funzione
```

È possibile eseguire del codice solo se il file è eseguito direttamente e non è stato importato come modulo

if **name** == ‘**main**‘: print(‘Programma eseguito direttamente’) else: print(‘Questo modulo è stato importato’) ` ` `

- La variabile `__name__` contiene il nome del modulo corrente
- Se un modulo è eseguito direttamente la variabile `__name__` assume valore ‘`__main__`’

- Python dispone di numerosi moduli già inclusi
- Le capacità del linguaggio si possono estendere con moduli esterni
- È possibile installare ulteiori moduli tramite `pip`

```
$ pip install --user shouty
Collecting shouty
Installing collected packages: shouty
Successfully installed shouty-0.1.dev6
```

Input e Output

- Purtroppo a volte è necessario interagire con l’utente
- È possibile ottenere un valore dall’utente
- È possibile accedere a un file presente sul disco

Formattazione dell’output

```
>>> s = 'Corsi {} {}'
>>> s.format('Python', '2017')
'Corsi Python 2017'
>>> s.format('Linux', 'base')
'Corsi Linux base'
```

```
>>> s = 'Corsi {argomento} {tipo}'
>>> s.format(tipo='avanzati', argomento='antani')
'Corsi antani avanzati'
```

```
from math import pi
>>> '{:.2f}'.format(pi)
3.14
```

Input dall’utente

```
>>> a = input()
asd
>>> a
'asd'
```

```
>>> nome = input("Inserisci il tuo nome: ")
Gattuso
>>> 'Il tuo nome è {}.'.format(nome)
'Il tuo nonme è Gattuso.'
```

Aprire i file

```
f = open("documento.txt", "w")
f.write("contenuto")
f.close()
```

- `open(filename, mode)` ritorna un riferimento al file
- Il parametro `mode` specifica la modalità

mode	descrizione
r	Apertura in sola lettura
w	Apertura in sola scrittura
r+	Apertura in lettura e scrittura
a	Scrittura in coda

Usare i file

- Per scrivere su un file

```
>>> f = open("documento.txt", "w")
>>> f.write("Contenuti serissimi.")
20
>>> f.close()
```

- Per leggere da un file

```
>>> f = open("documento.txt", "r")
>>> f.read()
'Contenuti serissimi.'
>>> f.close()
```

- I metodi `readline` e `writeline` leggono e scrivono una riga alla volta
- Ricordiamoci di chiudere sempre il file con la funzione `close()`

Una scorciatoia

```
with open("documento.txt", "w") as f:
    f.write("contenuto")
```

- All’interno del blocco, `f` è un riferimento al file
- Non serve chiudere esplicitamente
- Se vengono sollevati errori o eccezioni il file viene chiuso
- Usate `with` quando potete

Guardiamoci le spalle

- Gli errori non sono sempre causati dal programmatore
- Si possono presentare situazioni in cui è plausibile che si verifichi un errore

```
>>> f = open("documento.rtf", "r")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
FileNotFoundError: [Errno 2] No such file or directory ...
```

- Queste situazioni si chiamano **eccezioni**.

Eccezioni

- Il codice che potrebbe dare errori si include in un blocco `try`
- Si indica l'errore con l'istruzione `except`
- Si indica il codice da eseguire in caso si presenti l'errore dentro un blocco `except`

```
>>> try:
...     with open("documento.txt", "r") as f:
...         print('Do stuff')
... except FileNotFoundError:
...     print("Ops")
```

I più ardimentosi possono anche “sollevare” eccezioni all’interno del proprio codice con l’istruzione `raise`.

```
>>> raise Exception("No internet connection!")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
Exception: No internet connection!
```

Nella prossima puntata

Giovedì 19 alle 17.15 in aula 5.0.2.

- Classi
- Metodi
- Ereditarietà
- List comprehensions
- Decoratori
- yield

```
>>> 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.
[...]
```

Thank you!

POLITECNICO OPEN
unix LABS
Come hack with us.