

Corso base di



pythonTM

per data scientists

Donato Summa
donato.summa@istat.it

Roma, 24-25/09/2019

Sommario

- Note sul corso
- Introduzione al linguaggio
- Concetti propedeutici
- Numeri e variabili
- Stringhe e sequenze
- Liste, tuple, set, dizionari
- Variabili booleane e istruzioni condizionali
- Iterazione
- Comprehension
- Funzioni
- Errori ed eccezioni
- Moduli
- Lavorare con i file
- Jupyter notebook

Sommario

- **Note sul corso**
- Introduzione al linguaggio
- Concetti propedeutici
- Numeri e variabili
- Stringhe e sequenze
- Liste, tuple, set, dizionari
- Variabili booleane e istruzioni condizionali
- Iterazione
- Comprehension
- Funzioni
- Errori ed eccezioni
- Moduli
- Lavorare con i file
- Jupyter notebook

Note sul corso

- Giorni e orario delle lezioni
- Piattaforma corsi
- Firme presenza
- Obiettivi e premesse

Sommario

- Note sul corso
- **Introduzione al linguaggio**
- Concetti propedeutici
- Numeri e variabili
- Stringhe e sequenze
- Liste, tuple, set, dizionari
- Variabili booleane e istruzioni condizionali
- Iterazione
- Comprehension
- Funzioni
- Errori ed eccezioni
- Moduli
- Lavorare con i file
- Jupyter notebook

Introduzione al linguaggio

Python

Linguaggio di programmazione ideato dall'olandese **Guido van Rossum** che si definisce il “benevolo dittatore a vita”.

Il nome è un omaggio alla serie televisiva comica britannica "Monty Python's Flying Circus".

Prima versione rilasciata nel 1991.

Python

Rappresenta attualmente il linguaggio con il più alto tasso di crescita.

<https://www.tiobe.com/tiobe-index/>

Utilizzatori tipici:

- Informatici
- Matematici
- Analisti dei dati
- Commercialisti
- Bambini (ideale come primo linguaggio)

Caratteristiche salienti

1) Asciutto (bastano poche righe di codice)

2) Multipurpose

Data analysis

AI / ML

Automazione

Web Apps

Mobile Apps

Desktop Apps

SW testing

Hacking

3) Multiparadigma (procedurale, funzionale, OO)

Caratteristiche salienti

- 4) Codice facilmente leggibile
- 5) Semplice da imparare
- 6) Open source
- 7) Scripting e programmi stand-alone
- 8) Gratuito
- 9) Portabile

Anche i Big usano Python



<https://realpython.com/world-class-companies-using-python/>
<https://www.fullstackpython.com/companies-using-python.html>

Python e gli altri linguaggi

C#

C++

JS

php

Java™

C

Scala

R

Ruby

Era veramente necessario un altro linguaggio?

Python VS R

Entrambi :

- sono linguaggi open-source
- hanno una grande comunità alle spalle
- hanno numerose librerie
- sono orientati alla data science

Ma :

- R è maggiormente usato per analisi statistica
- Python ha un approccio più generale alla DS

Python VS R

R rispetto a Python :

- ha un ecosistema più ricco per la data analysis
- ha ottimi tool per visualizzare i risultati
- più vicino al mondo degli statistici (è stato scritto da statistici)

Python VS R

Python rispetto ad R :

- permette di scalare con maggiore facilità
- codice più facile da mantenere
- codice più robusto
- cutting-edge API per ML e AI
- multipurpose

Python VS R

L'ideale sarebbe conoscere entrambi per poter beneficiare dei rispettivi punti di forza

ma . . .

se si parte da zero probabilmente Python è la scelta migliore per i motivi appena discussi e per la **curva di apprendimento** più favorevole.

Python 2 o Python 3 ?

Python 2 :

- prima release nel 2000
- ultima release (2.7) nel 2010
- la % di progetti Py2 è in calo continuo
- rappresenta sostanzialmente il passato
- non sarà più mantenuto dopo il 2020

(<https://pythonclock.org/>)

A meno di casi molto specifici non ha più senso investirci tempo.

Python 2 o Python 3 ?

Python 3 :

- prima release nel 2008
- è e sarà supportato in futuro
- continua migrazione da Py2 a Py3
- community più ampia
- nuovi pacchetti
- migliori performance di esecuzione
- diversi miglioramenti rispetto a Py2

In questo corso ci concentreremo su Python 3

File Python

Ogni file contenente codice Python deve avere estensione **.py**

(Formalmente ciò è necessario per i soli file importati ma è convenzione largamente diffusa farlo sempre per motivi di consistenza)

Es.

mioSorgente.py

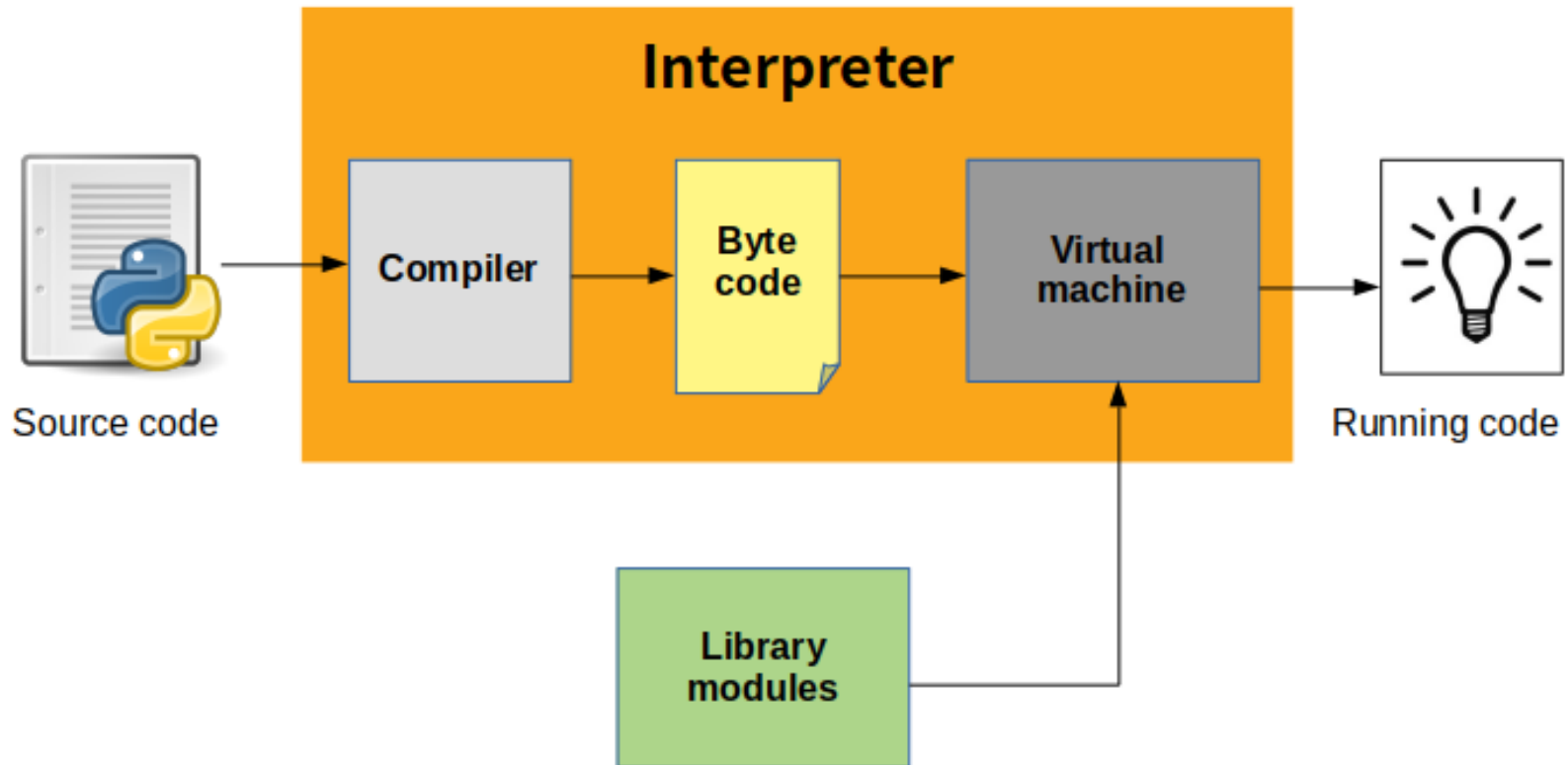
```
# Il mio primo programma  
# in Python  
print("Hello world !")
```

Sommario

- Note sul corso
- Introduzione al linguaggio
- **Concetti propedeutici**
- Numeri e variabili
- Stringhe e sequenze
- Liste, tuple, set, dizionari
- Variabili booleane e istruzioni condizionali
- Iterazione
- Comprehension
- Funzioni
- Errori ed eccezioni
- Moduli
- Lavorare con i file
- Jupyter notebook

Concetti propedeutici

Interprete e PVM



Esecuzione di un programma

Si può eseguire un programma Python:

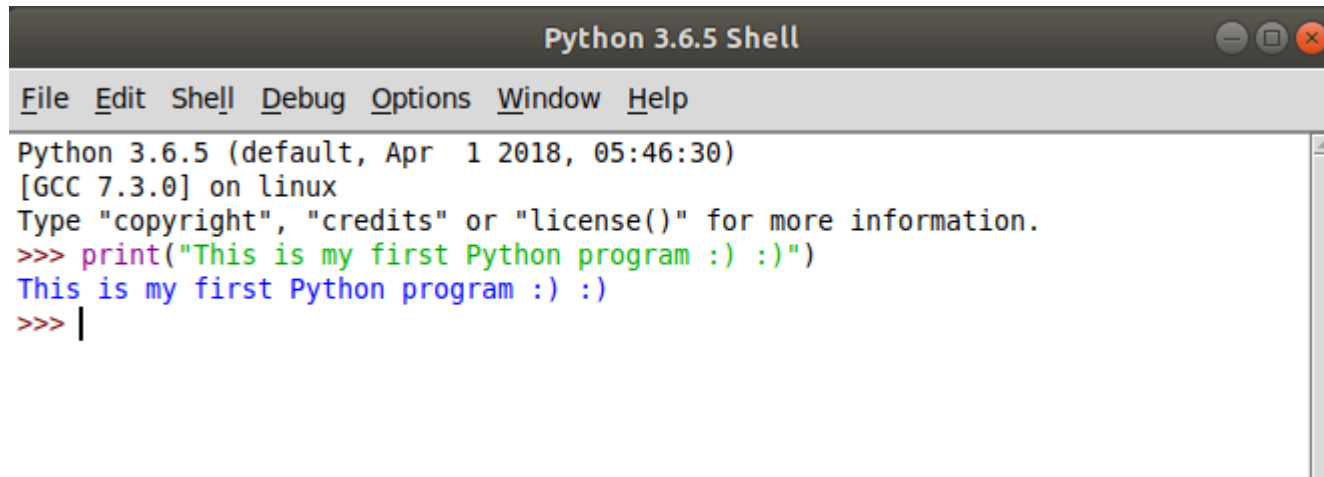
- 1) da linea di comando (OS terminal)
- 2) tramite interactive prompt (es. IDLE)
- 3) tramite doppio click sull'icona del file
- 4) dall'IDE di sviluppo
- 5) attraverso un import
- 6)

Esecuzione di un programma

Si può eseguire un programma Python:

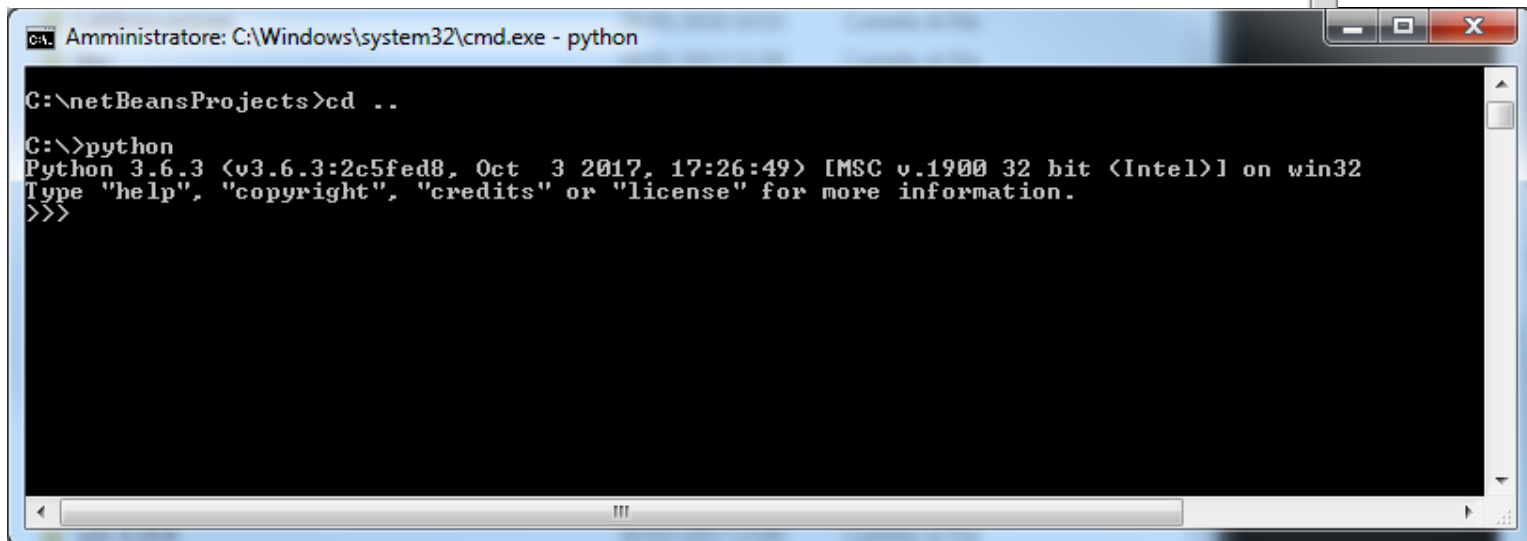
- 1) da linea di comando (OS terminal)**
- 2) tramite interactive prompt (es. IDLE)
- 3) tramite doppio click sull'icona del file
- 4) dall'IDE di sviluppo
- 5) attraverso un import
- 6)

Shell – Terminale – Command line



A screenshot of a Python 3.6.5 Shell window. The title bar reads "Python 3.6.5 Shell". The menu bar includes "File", "Edit", "Shell", "Debug", "Options", "Window", and "Help". The text area shows the following content: "Python 3.6.5 (default, Apr 1 2018, 05:46:30)", "[GCC 7.3.0] on linux", "Type 'copyright', 'credits' or 'license()' for more information.", a prompt ">>>" followed by the command "print('This is my first Python program :) :)", the output "This is my first Python program :) :", and another prompt ">>>" with a cursor.

```
Python 3.6.5 Shell
File Edit Shell Debug Options Window Help
Python 3.6.5 (default, Apr 1 2018, 05:46:30)
[GCC 7.3.0] on linux
Type "copyright", "credits" or "license()" for more information.
>>> print("This is my first Python program :) :)")
This is my first Python program :) :)
>>> |
```



A screenshot of a Windows Command Prompt window titled "Administrator: C:\Windows\system32\cmd.exe - python". The text area shows the following content: "C:\netBeansProjects>cd ..", "C:\>python", "Python 3.6.3 (v3.6.3:2c5fed8, Oct 3 2017, 17:26:49) [MSC v.1900 32 bit (Intel)] on win32", "Type 'help', 'copyright', 'credits' or 'license' for more information.", and a prompt ">>>".

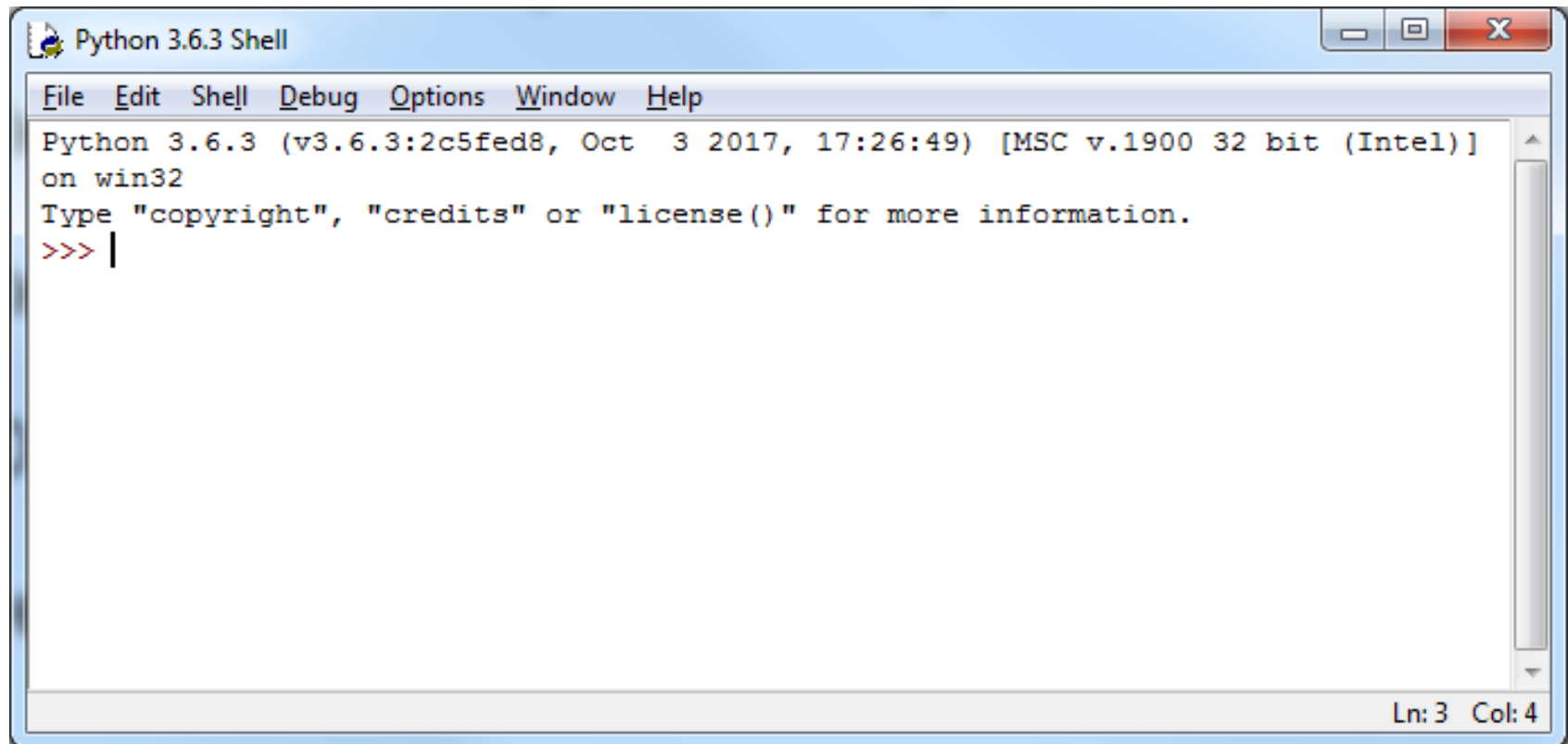
```
Administrator: C:\Windows\system32\cmd.exe - python
C:\netBeansProjects>cd ..
C:\>python
Python 3.6.3 (v3.6.3:2c5fed8, Oct 3 2017, 17:26:49) [MSC v.1900 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Esecuzione di un programma

Si può eseguire un programma Python:

- 1) da linea di comando (OS terminal)
- 2) tramite interactive prompt (es. IDLE)**
- 3) tramite doppio click sull'icona del file
- 4) dall'IDE di sviluppo
- 5) attraverso un import
- 6)

Python IDLE

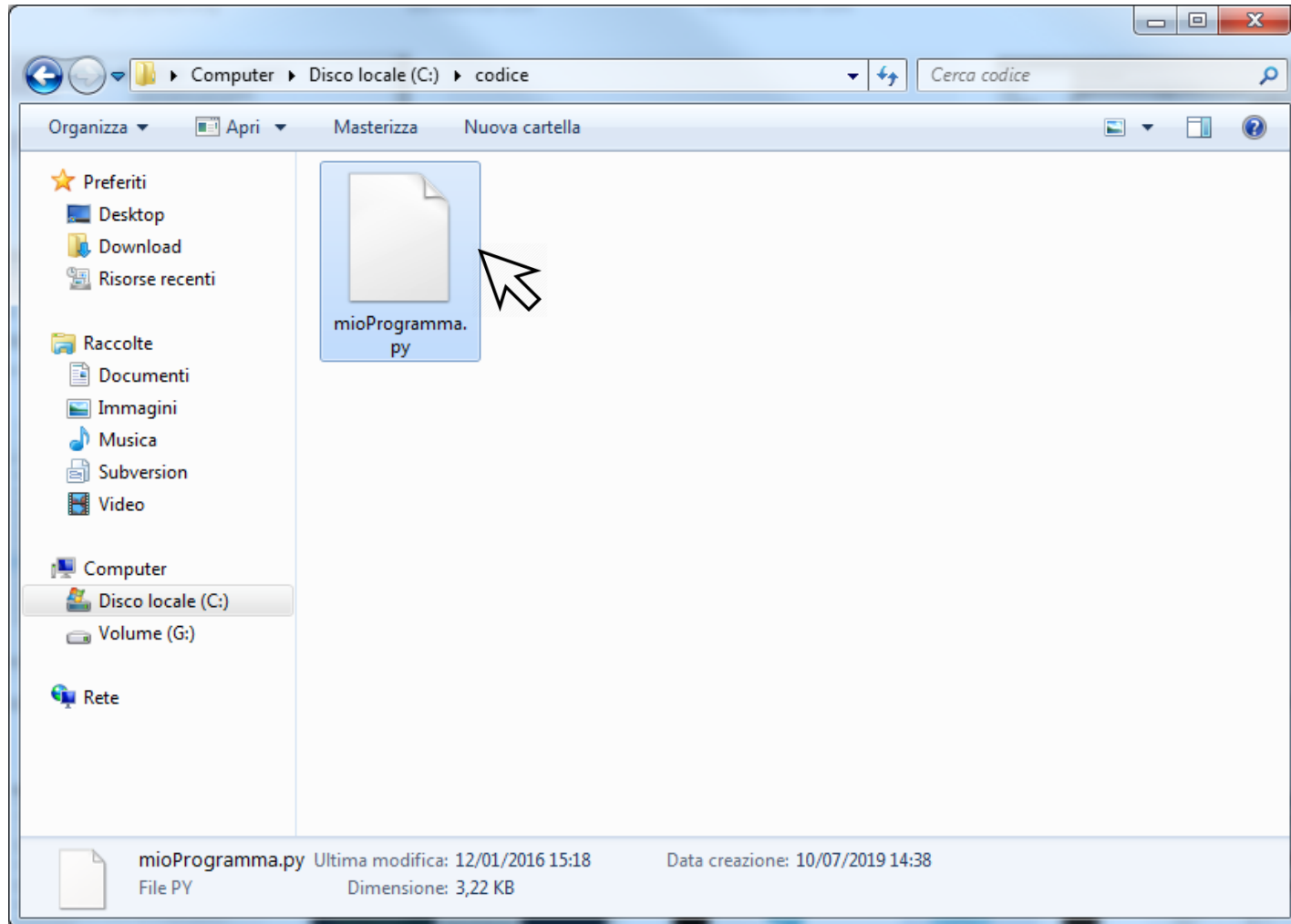


Esecuzione di un programma

Si può eseguire un programma Python:

- 1) da linea di comando (OS terminal)
- 2) tramite interactive prompt (es. IDLE)
- 3) tramite doppio click sull'icona del file**
- 4) dall'IDE di sviluppo
- 5) attraverso un import
- 6)

Esecuzione con doppio click



Esecuzione di un programma

Si può eseguire un programma Python:

- 1) da linea di comando (OS terminal)
- 2) tramite interactive prompt (es. IDLE)
- 3) tramite doppio click sull'icona del file
- 4) dall'IDE di sviluppo**
- 5) attraverso un import
- 6)

Ambiente di sviluppo

Un **IDE** (**I**ntegrated **D**evelopment **E**nvironment) è un programma che integra diversi componenti SW al fine di agevolare l'attività di sviluppo.

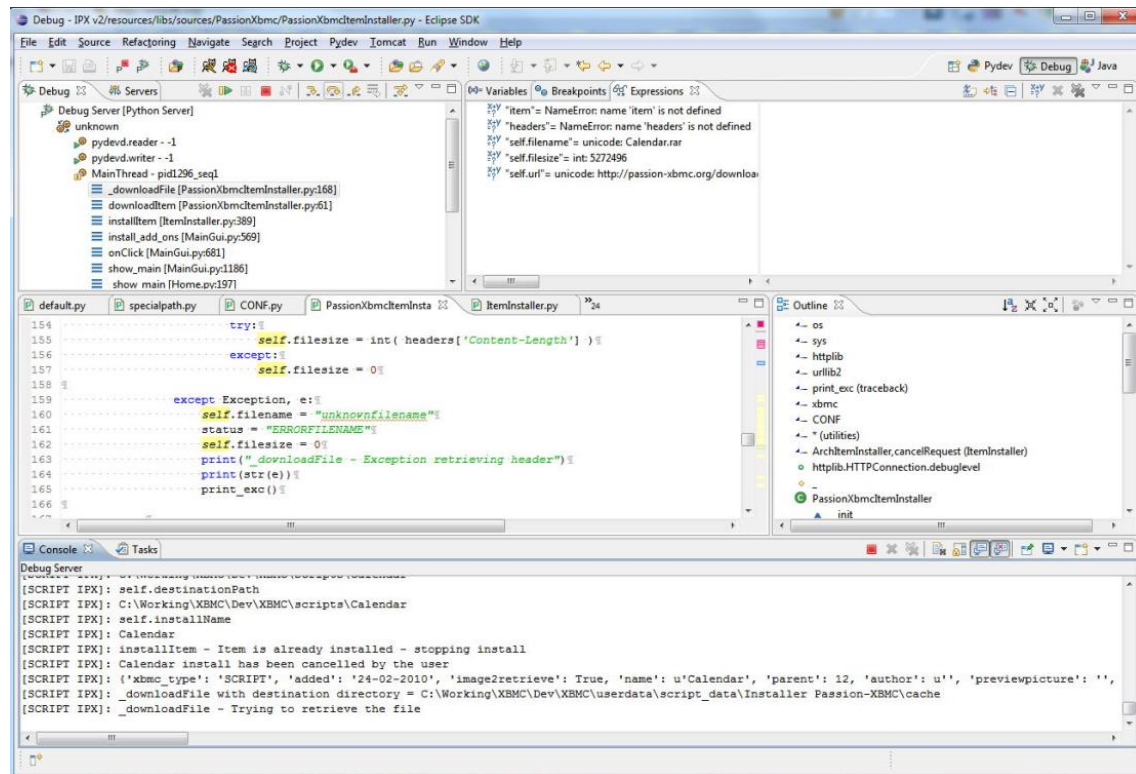
Solitamente racchiude almeno:

- un editor di testo
- un compilatore e/o un interprete
- un tool di building automatico
- un debugger

Ambiente di sviluppo

Normalmente esistono diversi IDE per ogni linguaggio, tra i più comuni per Python:

- Eclipse + Pydev
- PyCharm
- Komodo IDE
- Spyder
- Jupyter



Modalità di esecuzione

È possibile eseguire codice Python in 2 modalità:

1) Interattiva

2) Non interattiva

Modalità Interattiva

Ogni singola istruzione digitata viene eseguita immediatamente ed eventuali risultati vengono stampati in automatico.

Tramite shell o IDLE.

Il codice digitato **NON** viene salvato.

Si usa essenzialmente per :

- scopo didattico
- testare al volo pezzi di codice
- prototipazione rapida di funzionalità

Modalità Interattiva

- digitare solo comandi Python (non comandi di sistema)
- la stampa dei risultati non richiede istruzioni print (al contrario della programm. nei file)
- l'indentazione non è necessaria (viene scritta una sola riga alla volta)
- eventuali istruzioni multiriga richiedono una riga vuota finale

Modalità non Interattiva

Quando i programmi diventano grandi e complessi è impensabile doverli riscrivere ogni volta da zero una riga per volta . . .

Il codice viene dunque scritto all'interno di uno o più file che andranno a comporre il programma.

Il lancio del programma comporterà l'esecuzione automatica delle righe di codice in esso contenute secondo una sequenza stabilita dalle regole del linguaggio.

Indentazione

In tutti i linguaggi indentare il codice è pratica comune (oltre che buona norma) perchè agevola la lettura e la comprensione.

Python non solo non fa eccezione ma rende **OBBLIGATORIA la corretta indentazione** del codice.

Il non rispetto delle regole si traduce in:

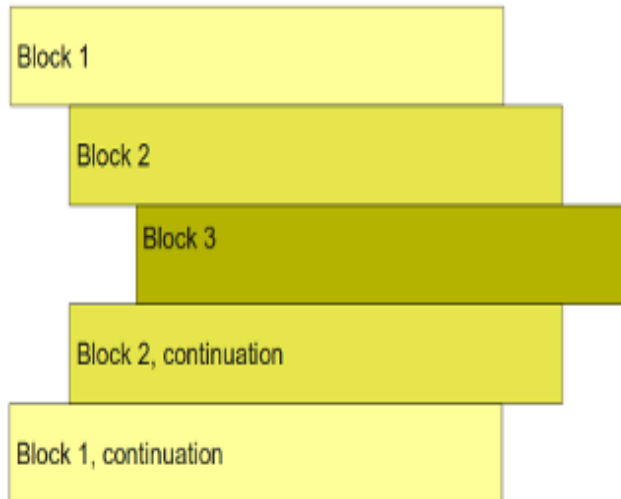
- 1) Errori sintattici (caso fortunato)
- 2) Errori semantici (caso molto GRAVE !!!)

Indentazione

Le **regole di base** da rispettare sono 2:

- 1) inserire **una sola istruzione per riga** (sono a volte possibili eccezioni che noi non tratteremo)
- 2) **indentare consistentemente** le istruzioni di un blocco di codice annidato (es. istruzioni sotto un for, if, while,...)

Indentazione



```
import threading, socket, time
class sock(threading.Thread):
    def __init__(self):
        self.sck=socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        threading.Thread.__init__(self)
        self.flag=1
    def connect(self, addr, port, func):
        try:
            self.sck.connect((addr, port))
            self.handle=self.sck
            self.todo=2
            self.func=func
            self.start()
        except:
            print "Error: Could not connect"
    def listen(self, host, port, func):
        try:
            self.sck.bind((host, port))
            self.sck.listen(5)
            self.todo=1
            self.func=func
            self.start()
        except:
            print "Error: Could not bind"
    def run(self):
        while self.flag:
            if self.todo==1:
                x, ho=self.sck.accept()
                self.todo=2
                self.client=ho
                self.handle=x
            else:
                dat=self.handle.recv(4096)
                self.data=dat
                self.func()
    def send(self, data):
        self.handle.send(data)
    def close(self):
        self.flag=0
        self.sck.close()
```

Indentazione

È possibile trovare la guida allo stile all'URL:

<https://www.python.org/dev/peps/pep-0008>

Sarebbe buona norma:

- usare sempre 4 spazi per livello di indentazione
- evitare l'uso del tabulatore
- non mischiare mai tabulazioni e spazi

In questo corso ci limiteremo ad adottare uno **stile consistente** per le "rientranze" del codice.

Compound statements

Un compound statement (istruzione composta) è uno statement che :

- si articola generalmente su più righe
- può contenere altri statement (es. if annidati)
- controlla l'esecuzione delle istruzioni contenute

Compound statements in Python:

if

for

while

try

def

with

class

(altri che non tratteremo)

Case sensitive

I linguaggi di programmazione generalmente sono sensibili alla differenza tra lettere maiuscole e minuscole.

Es. Nomi di variabili e funzioni

Totale \neq totale \neq TOTALE \neq ToTaLe
somma \neq Somma \neq SOMMA

Python è case sensitive !

Commenti

Singola riga

questo è un commento su una sola riga

Multiriga

"""

Commento su
più righe

"""

3 doppi apici

oppure

"""

Altro modo di
commentare su più
righe

"""

3 singoli apici

Concetto di funzione



Es.

3 4

6.4

"ciao"

+

round()

print()

7

6

stampa ciao

La funzione print()

Stampa a schermo ciò che gli viene passato in input:

Es.

`print(35)` stamperà 35

`print("totale")` stamperà totale

`print(totale)` stamperà il valore di totale
(se esiste) o darà errore

La funzione input()

Ritorna **come stringa** ciò che viene digitato dall'utente:

Es.

```
>>>numero = input("immetti un numero: ")
```

```
immetti un numero: 7
```

```
>>>numero
```

```
'7'
```

L'utente ha digitato 7 nel terminale

```
>>>nome = input("immetti un nome: ")
```

```
immetti un nome: Cinzia
```

```
>>>nome
```

```
'Cinzia'
```

L'utente ha digitato Cinzia nel terminale

Le funzioni **dir()** e **help()**

Normalmente usate in modalità interattiva

dir() stampa a schermo la lista degli attributi disponibili per l'oggetto passato in input come argomento

Es. supponendo che S sia una stringa

```
>>> dir(S)
['_add_', '__class__', '__contains__', '__delattr__', '__dir__', '__doc__',
'__eq__', '__format__', '__ge__', '__getattr__', '__getitem__',
'__getnewargs__', '__gt__', '__hash__', '__init__', '__iter__', '__le__',
'__len__', '__lt__', '__mod__', '__mul__', '__ne__', '__new__', '__reduce__',
'__reduce_ex__', '__repr__', '__rmod__', '__rmul__', '__setattr__', '__sizeof__',
'__str__', '__subclasshook__', 'capitalize', 'casefold', 'center', 'count',
'encode', 'endswith', 'expandtabs', 'find', 'format', 'format_map', 'index',
'isalnum', 'isalpha', 'isdecimal', 'isdigit', 'isidentifier', 'islower',
'isnumeric', 'isprintable', 'isspace', 'istitle', 'isupper', 'join', 'ljust',
'lower', 'lstrip', 'maketrans', 'partition', 'replace', 'rfind', 'rindex',
'rjust', 'rpartition', 'rsplit', 'rstrip', 'split', 'splitlines', 'startswith',
'strip', 'swapcase', 'title', 'translate', 'upper', 'zfill']
```

Le funzioni **dir()** e **help()**

help() stampa a schermo le informazioni disponibili su uno specifico attributo passato in input come argomento

Es. supponendo che S sia una stringa

```
>>> help(S.replace)
Help on built-in function replace:
```

```
replace(...)
    S.replace(old, new[, count]) -> str
```

```
Return a copy of S with all occurrences of substring
old replaced by new.  If the optional argument count is
given, only the first count occurrences are replaced.
```


Documentazione Python

Online

<http://www.python.org>

<http://www.python.org/doc>

Offline

Start => Programmi => Python 3.X => Manuals

Start => Programmi => Python 3.X => Modules doc

Core objects data types

Nome	Esempi
Numeri	12 3.456 769365970 745383.23623521850
Stringhe	"ciao" "Marco" "123"
Lists	frutta = ["mele" , "pere" , "arance" , "banane"]
Dictionaries	auto = {"marca":"Ford", "modello":"Fiesta", "anno":1983}
Tuples	nomi = ("Vito" , "Lucia" , "Antonio" , "Cinzia" , "Carmela")
Sets	giorni = set("lun", "mar", "mer", "gio", "ven", "sab","dom")
Files	files
Boolean	True False
None	None

Core objects data types

Sono parte integrante del linguaggio Python

Sono sempre disponibili (import non necessarie)

Si utilizzano come componenti base per costruire oggetti più complessi (es. classi)

Largamente utilizzati nella scrittura di script

È fondamentale conoscerli e padroneggiarli

Sommario

- Note sul corso
- Introduzione al linguaggio
- Concetti propedeutici
- **Numeri e variabili**
- Stringhe e sequenze
- Liste, tuple, set, dizionari
- Variabili booleane e istruzioni condizionali
- Iterazione
- Comprehension
- Funzioni
- Errori ed eccezioni
- Moduli
- Lavorare con i file
- Jupyter notebook

Numeri e variabili

Numeri

Esistono diversi tipi di numeri, in questo corso :

Interi	3	45	9736	8999886346
Float	1.2	482.234947	0.634	32.98

La separazione tra parte intera e parte decimale avviene tramite “.” e non tramite virgola

È possibile applicare conversioni e altre operazioni tra tipi tramite apposite funzioni:

int()

float()

round()

Numeri

int → **float**

script.py	
<code>float(10)</code>	
Output	10.0

float → **int**

script.py	
<code>int(4.3)</code>	
Output	4

script.py	
<code>round(4.99)</code>	
Output	5

script.py	
<code>round(4.5)</code>	
Output	4

script.py	
<code>round(4.3)</code>	
Output	4

Operatori aritmetici

+	addizione	$2 + 2 = 4$
-	sottrazione	$5 - 3 = 2$
/	divisione	$9 / 4 = 2.25$
//	divisione intera	$9 // 4 = 2$
%	modulo (resto divisione)	$9 \% 4 = 1$
*	moltiplicazione	$2 * 3 = 6$
**	elevamento a potenza	$2 ** 3 = 8$

Common shorthands

Scorciatoia sintattica	Esempio di codice	Risultato	Codice equivalente senza scorciatoie sintattiche
<code>+=</code>	<code>X = 4</code> <code>X += 2</code>	6	<code>X = 4</code> <code>X = X + 2</code>
<code>-=</code>	<code>X = 4</code> <code>X -= 2</code>	2	<code>X = 4</code> <code>X = X - 2</code>
<code>*=</code>	<code>X = 4</code> <code>X *= 2</code>	8	<code>X = 4</code> <code>X = X * 2</code>
<code>/=</code>	<code>X = 4</code> <code>X /= 2</code>	2	<code>X = 4</code> <code>X = X / 2</code>
<code>**=</code>	<code>X = 4</code> <code>X **= 2</code>	16	<code>X = 4</code> <code>X = X ** 2</code>

Precedenza degli operatori

Le operazioni aritmetiche in Python seguono lo stesso ordine usato in matematica:

- 1) parentesi
- 2) esponenziali
- 3) divisioni e moltiplicazioni
- 4) addizioni e sottrazioni

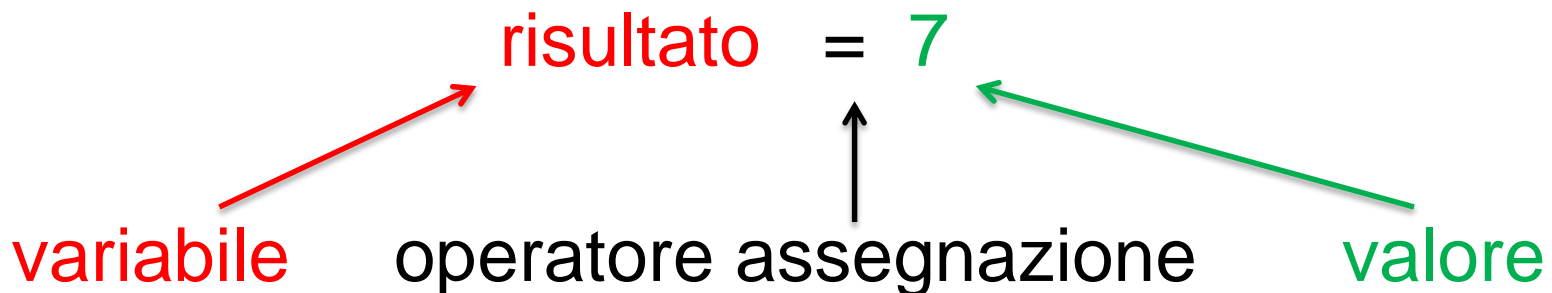
Es. $4 + 2 * 10 = 24$
 $(4 + 2) * 10 = 60$

Variabili

Una variabile è un nome che si riferisce ad una zona della memoria contenente un valore (non necessariamente un numero !!!).

Possiamo paragonare una variabile ad una etichetta posta su un contenitore.

Es.



Variabili

I nomi delle variabili

possono

- essere lunghi a piacere
- contenere lettere, numeri, underscores

NON possono

- iniziare con un numero
- contenere spazi o caratteri speciali
- essere parole riservate del linguaggio

Parole riservate di Python

and	except	lambda	with
as	finally	nonlocal	while
assert	false	None	yield
break	for	not	
class	from	or	
continue	global	pass	
def	if	raise	
del	import	return	
elif	in	True	
else	is	try	

Variabili

In Python **NON sono tipizzate staticamente**,
ciò garantisce maggiore flessibilità.

```
var = 5
```

```
var = "ciao"
```

```
var = 7.89
```



Questa sequenza di istruzioni
è perfettamente legittima

Viaggiare senza cintura aumenta la libertà di
movimento, ma se vai a sbattere

Assegnazione (=) VS uguaglianza (==)

risultato = 7

Informalmente diremo che risultato è uguale a 7.

Propriamente si dovrebbe dire che è stato inserito il valore 7 nella locazione di memoria etichettata “risultato”.

variabile1 == variabile2

variabile1 == 7

Si usa l'operatore di uguaglianza per effettuare confronti tra valori (espliciti o rappresentati da variabili)

Questo tipo di istruzioni restituiscono valori booleani e si usano tipicamente con gli if

Sommario

- Note sul corso
- Introduzione al linguaggio
- Concetti propedeutici
- Numeri e variabili
- **Stringhe e sequenze**
- Liste, tuple, set, dizionari
- Variabili booleane e istruzioni condizionali
- Iterazione
- Comprehension
- Funzioni
- Errori ed eccezioni
- Moduli
- Lavorare con i file
- Jupyter notebook

Stringhe e sequenze

Stringhe

Stringa = sequenza di caratteri alfanumerici e caratteri speciali racchiusi tra apici doppi (“”) o singoli (‘’)

Es: “Python” ‘Python’ “_3.5 : ”

È possibile assegnare una stringa ad una variabile nel solito modo:

Es:

```
mia_variabile_stringa = “oggi sono sereno”  
stringvar = ‘2 + 2 fa 4’
```

Stringhe

A volte è necessario includere nel testo della stringa proprio i caratteri “” e ‘’

Se nel testo c'è il carattere ‘ racchiuderò la stringa tra “”

Es: “Non ho visto l'appartamento”

Se nel testo c'è il carattere “ racchiuderò la stringa tra ‘

Es: ‘Il film “La casa” mi è piaciuto’

E se ci sono entrambi?

Stringhe

Si scelgono i delimitatori da usare (“” o ‘’)

Si effettuerà l'**escape** dei caratteri problematici con \

Es:

“ Ho dormito all'hotel \“President\” ”

‘ Ho dormito all\'hotel “President” ’

Stringhe

È possibile inserire nelle stringhe anche alcuni caratteri speciali che normalmente sono preceduti dal carattere di escape **** :

Es:

"Vai a capo alla fine **\n**"

"Il carattere **\t** funge da tabulatore"

"Il backslash si scrive così: ****"

Per la lista completa si può fare riferimento al link

https://docs.python.org/3/reference/lexical_analysis.html

Concetto di sequenza

Una collezione di oggetti ordinati per posizione da sinistra a destra.

Gli oggetti contenuti nella sequenza sono salvati e recuperati per mezzo della loro posizione.

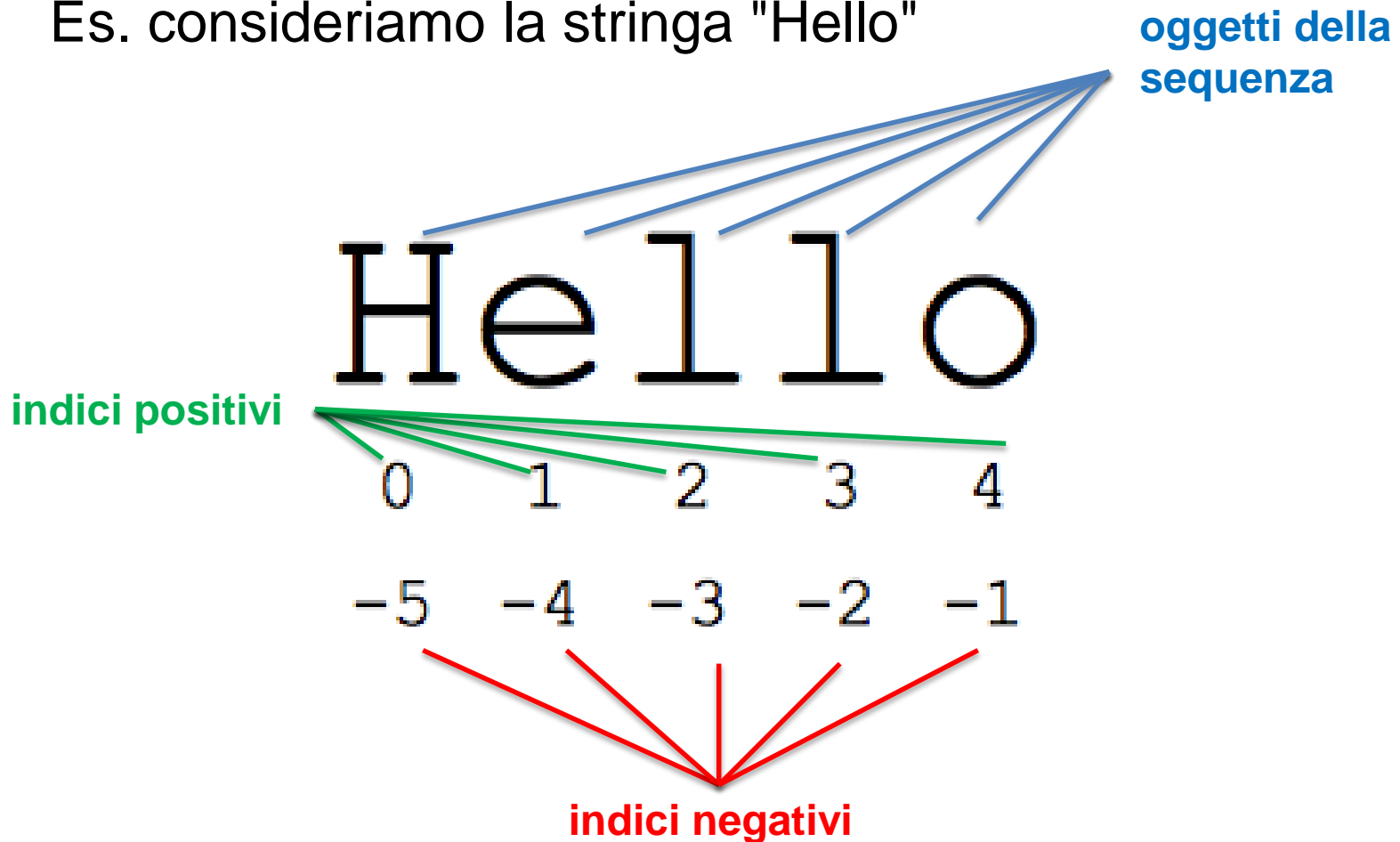
Le operazioni praticabili sulle sequences assumono un ordinamento posizionale degli elementi.

La **numerazione delle posizioni** degli oggetti contenuti in una sequence **comincia da 0**.

Esempi di sequences sono: Stringhe, Liste, Tuple

Concetto di sequenza

Es. consideriamo la stringa "Hello"



Operazioni sulle sequenze

Alcuni delle più comuni operazioni sulle sequenze sono :

Indexing

Slicing

Concatenation

Repetition

Operazioni sulle sequenze - Indexing

S = "Hello"

He l l o

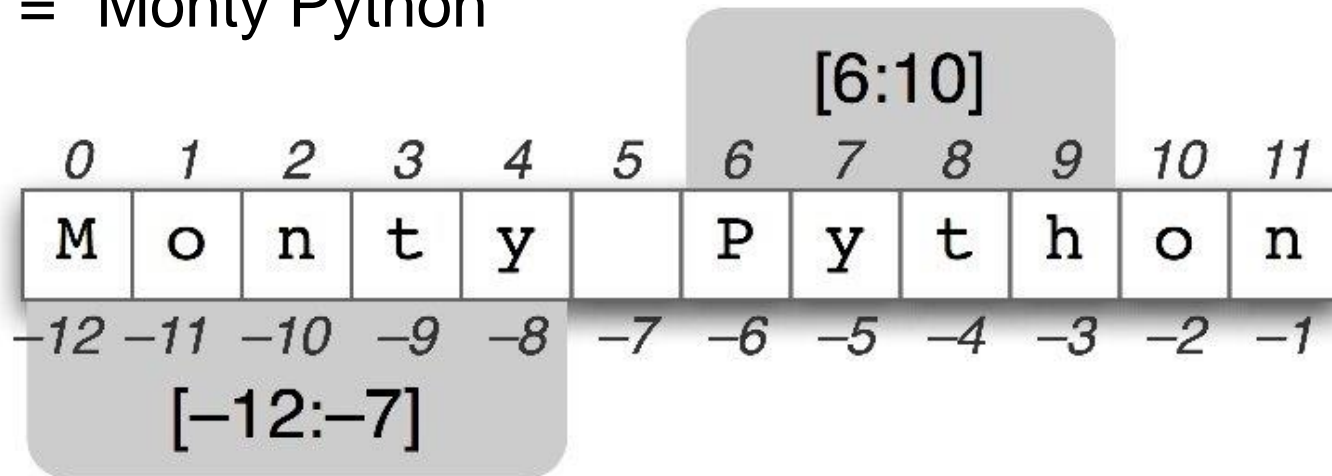
0 1 2 3 4
-5 -4 -3 -2 -1

S[0]	→	"H"	S[-1]	→	"o"
S[1]	→	"e"	S[-2]	→	"l"
S[2]	→	"l"	S[-3]	→	"l"
S[3]	→	"l"	S[-4]	→	"e"
S[4]	→	"o"	S[-5]	→	"H"

Qualsiasi altro
indice produce
un IndexError !

Operazioni sulle sequenze - Slicing

S = "Monty Python"



S[6:10]	→	"Pyth"
S[1:]	→	"onty Python "
S[0:5]	→	"Monty"
S[:5]	→	"Monty"
S[:-1]	→	"Monty Pytho"
S[:]	→	"Monty Python"

Operazioni sulle sequenze - Concatenation

```
S = "Pasta"
```

```
newString = S + "Pizza"
```

```
print(newString)
```

→ PastaPizza

```
print(S + newString)
```

→ PastaPastaPizza

L'operatore **+** richiede operandi dello stesso tipo:

```
print("Hello " + "World")
```

→ Hello World

```
print("Hello " + S)
```

→ Hello Pasta

```
print(3 + 2)
```

→ 5

```
n = 2
```

```
print("numero " + 3)
```

→ **ERRORE**

```
print("numero " + n)
```

→ **ERRORE**

```
print("numero " + str(3))
```

→ numero 3

```
print("numero " + str(n))
```

→ numero 2

Operazioni sulle sequenze - Repetition

S = "Pizza"

print(S * 4)

➔ PizzaPizzaPizzaPizza

Immutabilità delle stringhe

In Python **le stringhe sono oggetti immutabili**, NON è quindi possibile modificare una stringa dopo la sua creazione.

```
>>> s = "test"
```

```
>>> s[0] = "a"
```

```
Traceback (most recent call last):
```

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

```
TypeError: 'str' object does not support item assignment
```

Poiché tutte le operazioni sulle stringhe producono come output una nuova stringa, ciò che si può fare è riutilizzare il nome di una variabile assegnando alla stessa una nuova stringa.

Alcuni metodi della classe String

S.find()

S.replace()

S.split()

S.upper()

S.lower()

S.isalpha()

S.rstrip()

S.lstrip()

S.index()

S.startswith()

S.endswith()

S.count()

len(S)

...

Sommario

- Note sul corso
- Introduzione al linguaggio
- Concetti propedeutici
- Numeri e variabili
- Stringhe e sequenze
- **Liste, tuple, set, dizionari**
- Variabili booleane e istruzioni condizionali
- Iterazione
- Comprehension
- Funzioni
- Errori ed eccezioni
- Moduli
- Lavorare con i file
- Jupyter notebook

Liste

Tuple

Set

Dizionari

Liste

List

- sono collezioni ordinate di oggetti (no fixed type !)
- non hanno una dimensione fissa
- sono mutabili (al contrario delle stringhe)
- sono la tipologia di sequenza più generica in Python
- rappresentano una estensione del concetto di array

Sintatticamente gli elementi di una lista sono separati da **virgole** e racchiusi in **parentesi quadre**:

Es. mia_lista = [123 , "racchiusi" , 3.45]

	↑	↑	↑
	1° elemento	2° elemento	3° elemento
	(indice 0)	(indice 1)	(indice 2)

List

In quanto sequenze è possibile applicare alle liste tutte le operazioni applicabili alle sequenze.

Es. lista= [123, "spam", 1.23, "eggs"]

Indexing

lista[0] → 123 lista[-1] → "eggs"

Slicing

lista[0,2] → [123, "spam"]

Concatenation

lista2 = [8]

lista = lista + lista2 → [123, "spam", 1.23, "eggs", 8]

Repetition

lista = lista * 2 → [123, "spam", 1.23, "eggs", 8,
123, "spam", 1.23, "eggs", 8]

List

È possibile sostituire elementi di una lista ricorrendo all'indicizzazione in questo modo:

```
listaNumeri = ["uno", "due", "tre", "four", "cinque"]
```

```
listaNumeri[3] = "quattro"
```

```
print(listaNumeri) → ["uno", "due", "tre", "quattro", "cinque"]
```

```
listaNumeri[19] = "venti" → ERRORE
```

Se dal valore di un elemento si vuole ricavare l'indice:

```
indice = listaNumeri.index("uno")
```

```
print(indice) → 0
```

```
indice = listaNumeri.index("ciao") → ERRORE
```

List

È possibile aggiungere elementi ad una lista esistente:

- in **coda** con il metodo **append()**:

```
listaNumeri = ["uno", "due", "tre"]  
listaNumeri.append("quattro")  
print(listaNumeri) → ["uno", "due", "tre", "quattro"]
```

- in **posizione arbitraria** con il metodo **insert()**:

```
giorni = ["lun", "mer", "gio", "ven", "sab", "dom"]  
listaNumeri.insert(1, "mar")  
print(giorni) → ["lun", "mar", "mer", "gio", "ven", "sab", "dom"]
```

List

È possibile rimuovere elementi da una lista ricorrendo :

- all'operatore **del** (usa l'indice dell'elemento.)

Es.

```
listaNumeri = ["uno", "due", "tre", "quattro", "cinque"]  
del listaNumeri[3]  
print(listaNumeri) → ["uno", "due", "tre", "cinque"]  
del listaNumeri[0:2]  
print(listaNumeri) → ["tre", "cinque"]
```

- al metodo **remove** (usa il valore dell'elemento)

Es.

```
listaNumeri = ["uno", "due", "tre", "quattro", "cinque"]  
listaNumeri.remove("due")  
print(listaNumeri) → ["uno", "tre", "quattro", "cinque"]
```

List

È possibile ordinare gli elementi di una lista con il metodo `sort()` :

Es.

```
lettere = ["q", "a", "n", "t", "c"]
```

```
lettere.sort() → ["a", "c", "n", "q", "t"]
```

```
lettere.sort(reverse=True) → ["t", "q", "n", "c", "a"]
```

```
numeri = [83, 52, 25, 90, 12, 2]
```

```
numeri.sort() → [2, 12, 25, 52, 83, 90]
```

```
numeri.sort(reverse=True) → [90, 83, 52, 25, 12, 2]
```

Se si mescolano nella stessa lista numeri e lettere l'ordinamento non funzionerà !

List

Anche per le liste (come per tutti gli oggetti in Python) è possibile ottenere:

- l'elenco degli attributi disponibili tramite la funzione **dir()**
- informazioni circa uno specifico attributo tramite la funzione **help()**

```
>>> lista=[1,2,3]
>>> dir(lista)
['_add_', '_class_', '_contains_', '_delattr_', '_delitem_', '_dir_', '_doc_', '_eq_', '_format_', '_ge_', '_getattr_', '_getitem_', '_gt_', '_hash_', '_iadd_', '_imul_', '_init_', '_init_subclass_', '_iter_', '_le_', '_len_', '_lt_', '_mul_', '_ne_', '_new_', '_reduce_', '_reduce_ex_', '_repr_', '_reversed_', '_rmul_', '_setattr_', '_setitem_', '_sizeof_', '_str_', '_subclasshook_', 'append', 'clear', 'copy', 'count', 'extend', 'index', 'insert', 'pop', 'remove', 'reverse', 'sort']
```

```
>>> help(lista.append)
Help on built-in function append:
```

```
append(object, /) method of builtins.list instance
Append object to the end of the list.
```


Liste e stringhe

Sia per le stringhe che per le liste sono disponibili l'operatore **in** e la funzione **len**:

Es.

```
stringa = "ciao"
```

```
len(stringa)           → 4
```

```
"c" in stringa         → True
```

```
"z" in stringa         → False
```

```
"z" not in stringa     → True
```

```
listaNumeri = [1, 2, 3, 4, 5, 6, 7]
```

```
len(listaNumeri)       → 7
```

```
1 in listaNumeri       → True
```

```
8 in listanumeri       → False
```

Liste e stringhe

È possibile convertire una stringa in una lista di caratteri con la funzione **list()**:

Es.

```
stringa = "ciao"
```

```
lista = list(stringa)
```

```
lista → ["c", "i", "a", "o"]
```

L'operazione inversa richiede la funzione **join()**:

Es.

```
lista = ["c", "i", "a", "o"]
```

```
stringa = "".join(lista) → "ciao" (solo se la lista ha tutti char)
```

```
stringa = ",".join(map(str, lista)) → "c,i,a,o" (caso generale)
```

Tuple

Tuple

- sono collezioni ordinate di oggetti "hashabili"
- hanno una dimensione fissa
- sono immutabili
- occupano meno memoria (rispetto alle liste)
- sono più veloci (rispetto alle liste)

Sintatticamente gli elementi di una tupla sono separati da **virgole** e racchiusi in **parentesi tonde**:

Es. `mia_tupla = (123 , "Python" , 3.45)`

Tuple

L'accesso agli elementi avviene nel solito modo:

Es. `mia_tupla = (123 , "Python" , 3.45)`
 `mia_tupla[0]` \rightarrow 123

Poiché immutabili non è possibile (dopo la creazione):

- modificare il valore di un elemento
- aggiungere elementi
- rimuovere elementi

Si usano quando è essenziale:

- impedire la modifica della collezione
- migliorare le performance di spazio o tempo (iterazioni)

Tuple

Tecnicamente è possibile (sebbene non consigliabile) creare una tupla anche evitando l'uso delle parentesi:

Es. `t1 = 123 , "Python" , 3.45`
 `t2 = (123 , "Python" , 3.45)`
 `t1 == t2 ➔ True`

In alcuni casi l'uso delle parentesi è obbligato al fine di impedire il verificarsi di problemi sintattici:

Es. `len((123, "Python", 3.45)) ➔ 3`
 `len(123, "Python", 3.45) ➔ ERRORE`

Tuple

Poiché anche le tuple sono sequenze (come le stringhe e le liste) ammettono le seguenti operazioni:

```
t = ('abc', 123, 45.67)
```

```
t[0] ➔ "abc"           # indexing
```

```
t[:2] ➔ ("abc", 123)  # slicing
```

```
123 in t ➔ True       # op. di contenimento "in" e "not in"
```

```
t + ('xyz', 890) ➔ ('abc', 123, 45.67, 'xyz', 890)
```

```
# concatenazione (ritorna una nuova tupla)
```

```
t * 2 ➔ ('abc', 123, 45.67, 'abc', 123, 45.67)
```

```
# ripetizione (ritorna una nuova tupla)
```

Tuple

Come sempre è possibile ottenere:

- l'elenco degli attributi disponibili tramite la funzione **dir()**
- informazioni circa uno specifico attributo tramite la funzione **help()**

```
>>> dir(tuple)
['__add__', '__class__', '__contains__', '__delattr__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattr__', '__getitem__', '__getnewargs__', '__gt__', '__hash__', '__init__', '__init_subclass__', '__iter__', '__le__', '__len__', '__lt__', '__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__rmul__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__', 'count', 'index']

>>> help(tuple.count)
Help on method_descriptor:

count(...)
    T.count(value) -> integer -- return number of occurrences of value
```


Set

Set

Un set è una **collezione non ordinata di elementi distinti**

Non sono ammessi elementi duplicati

Gli elementi possono essere di tipi differenti

Sono mutabili (tranne i frozenset)

Sintatticamente gli elementi di un set sono separati da **virgole** e racchiusi in **parentesi graffe**:

Es. `mio_set = { 10 , "Python" , True , 3.45 }`

NB: nonostante sia set che dizionari usino le parentesi graffe, Python è in grado di distinguerli perché i dizionari contengono chiavi e valori separati dai due punti

Set

Non esiste una sintassi dedicata per la creazione di un set vuoto, contrariamente a:

liste → mia_lista = []

tuple → mia_tupla = ()

dizionari → mio_dizionario = {}

nel caso di set **NON è possibile scrivere mio_set = {}**
poiché tale sintassi è già usata per i dizionari.

Per creare un set vuoto si usa:

```
mio_set = set()
```

Set

Poiché non sono ammessi duplicati istruzioni come:

```
mioSet = {1, 2, 2, 3, 4, 4, 5}
```

produrranno automaticamente output del tipo:

```
{1, 2, 3, 4, 5}
```

È possibile creare un nuovo set partendo dagli elementi contenuti in un altro oggetto iterabile:

```
s = set("python")           # creo un set da una stringa  
s ➔ {'y', 'n', 'p', 'h', 't', 'o'} # nei set l'ordine non è garantito
```

Set

Operazioni di base con i set:

```
nums = {10, 20, 30, 40}
```

<code>len(nums)</code>	→ 4	# numero di elementi
<code>min(nums)</code>	→ 10	# elemento più piccolo
<code>max(nums)</code>	→ 40	# elemento più grande
<code>10 in nums</code>	→ True	# contenimento
<code>50 not in nums</code>	→ True	# contenimento
<code>60 in nums</code>	→ False	# contenimento

Set

Metodi comunemente usati con i set:

```
nums = {1, 2, 3, 4}
```

```
nums.add(5)           ➔ {1, 2, 3, 4, 5}
```

```
nums.remove(1)        ➔ {2, 3, 4, 5}
```

```
nums.remove(6)        ➔ ERRORE
```

```
nums.discard(2)       ➔ {3, 4, 5}
```

```
nums.discard(6)       ➔ Non fa niente
```

```
nums.copy()           ➔ Restituisce una copia del set
```

```
nums.clear()          ➔ {}
```

Set

Supportano una serie di operazioni tipiche degli insiemi. Alcune di queste operazioni sono implementate sia tramite metodi (per leggibilità) sia tramite operatori (per

Operatore	Metodo	Descrizione
	<code>s1.isdisjoint(s2)</code>	Restituisce True se i due set non hanno elementi in comune
<code>s1 <= s2</code>	<code>s1.issubset(s2)</code>	Restituisce True se s1 è un sottoinsieme di s2
<code>s1 < s2</code>		Restituisce True se s1 è un sottoinsieme <i>proprio</i> di s2
<code>s1 >= s2</code>	<code>s1.issuperset(s2)</code>	Restituisce True se s2 è un sottoinsieme di s1

Set

Operatore	Metodo	Descrizione
$s1 > s2$		Restituisce True se $s2$ è un sottoinsieme <i>proprio</i> di $s1$
$s1 s2 \dots$	<code>s1.union(s2, ...)</code>	Restituisce l'unione degli insiemi (tutti gli elementi)
$s1 \& s2 \& \dots$	<code>s1.intersection(s2, ...)</code>	Restituisce l'intersezione degli insiemi (elementi in comune a tutti i set)
$s1 - s2 - \dots$	<code>s1.difference(s2, ...)</code>	Restituisce la differenza tra gli insiemi (elementi di $s1$ che non sono negli altri set)
$s1 \wedge s2$	<code>s1.symmetric_difference(s2)</code>	Restituisce gli elementi dei due set senza quelli comuni a entrambi

Set

Operatore	Metodo	Descrizione
$s1 \mid= s2 \mid \dots$	<code>s1.update(s2, ...)</code>	Aggiunge a s1 gli elementi degli altri insiemi
$s1 \&= s2 \& \dots$	<code>s1.intersection_update(s2, ...)</code>	Aggiorna s1 in modo che contenga solo gli elementi comuni a tutti gli insiemi
$s1 -= s2 \mid \dots$	<code>s1.difference_update(s2, ...)</code>	Rimuove da s1 tutti gli elementi degli altri insiemi
$s1 \wedge= s2$	<code>s1.symmetric_difference_update(s2)</code>	Aggiorna s1 in modo che contenga solo gli elementi non comuni ai due insiemi

Set

Come sempre è possibile ottenere:

- l'elenco degli attributi disponibili tramite la funzione **dir()**
- informazioni circa uno specifico attributo tramite la funzione **help()**

```
>>> dir(set)
['_and_', '__class__', '__contains__', '__delattr__',
'_dir_', '__doc__', '__eq__', '__format__', '__g
e_', '__getattr__', '__gt__', '__hash__', '__ia
nd_', '__init__', '__init_subclass__', '__ior__', '__
isub_', '__iter__', '__ixor__', '__le__', '__len__',
'_lt_', '__ne__', '__new__', '__or__', '__rand__',
'_reduce_', '__reduce_ex__', '__repr__', '__ror__',
'_rsub_', '__rxor__', '__setattr__', '__sizeof__',
'_str_', '__sub_', '__subclasshook__', '__xor__',
'_add_', 'clear', 'copy', 'difference', 'difference
update', 'discard', 'intersection', 'intersection_upd
ate', 'isdisjoint', 'issubset', 'issuperset', 'pop',
'remove', 'symmetric_difference', 'symmetric_differen
ce update', 'union', 'update']
```

```
>>> help(set.union)
Help on method_descriptor:
```

```
union(...)
    Return the union of sets as a new set.

    (i.e. all elements that are in either set.)
```

Dizionari

Dictionary

I dizionari sono strutture dati non ordinate che :

- contengono coppie chiave-valore
- hanno una dimensione variabile
- sono mutabili

Sono concettualmente simili ai dizionari fisici in cui ad ogni parola (chiave) è associata una definizione (valore).

Sintatticamente gli elementi di un dizionario (coppie chiave:valore) sono separati da **virgole** e racchiusi in **parentesi graffe**:

Es. `dict = {"a": 1, "b": 2, "c": 3}`

Dictionary

Le **chiavi** sono solitamente stringhe univoche (è possibile usare anche altri tipi a condizione che siano "**hashabili**", liste e dizionari non lo sono).

I **valori** possono essere di **qualsiasi tipo**.

Funzionamento tipico: si fornisce in input una chiave per avere in output il valore corrispondente

Es. `dict = {"nome": "Vito", "eta": 30, "numeri": [6, 52, 10]}`
 `dict["nome"]` **→ "Vito"**
 `dict["cognome"]` **→ ERRORE**

Dictionary

Tramite le chiavi è possibile aggiungere, recuperare, modificare, eliminare elementi del dizionario.

Es.

<code>dict = {}</code>	<code># creo dizionario vuoto</code>
<code>dict["rosso"] = "red"</code>	<code># aggiungo un elemento</code>
<code>dict["bianco"] = "white"</code>	<code># aggiungo un elemento</code>
<code>dict["verde"] = "green"</code>	<code># aggiungo un elemento</code>
<code>dict["verde"]</code>	<code># recupero un elemento</code>
<code>dict["verde"] = "GREEN"</code>	<code># modifico un elemento</code>
<code>del dict["verde"]</code>	<code># elimino un elemento</code>

Dictionary

In caso di dizionari con moltissimi elementi non è facile ricordare tutte le chiavi, ci vengono in aiuto gli operatori **in** e **not in**.

Es. `dict = {"nome": "Vito", "eta": 30, "numeri": [6, 52, 10]}`
 `"cognome" in dict` **→ False**
 `"cognome" not in dict` **→ True**

Al fine di evitare errori dovuti a chiave mancante si potrebbe ogni volta verificare la presenza/assenza della stessa con questi operatori.

È tecnicamente fattibile e perfettamente funzionante ma è scomodo e noioso !

Dictionary

Si preferisce quindi usare il metodo **get()** che, data una chiave, restituisce alternativamente:

- il valore dell'elemento (se la chiave esiste)
- None (se la chiave non esiste)

È anche possibile fare in modo che, in caso di chiave non esistente, `get()` restituisca un valore di default.

Es. `dict = {"a": "uno", "b": "due", "c": "tre"}`
`dict.get("a")` \rightarrow "uno" # chiave esistente
`dict.get("H")` \rightarrow None # chiave non esist.
`dict.get("H", "zero")` \rightarrow "zero" # valore default

Dictionary

Alcuni metodi comuni usati con i dizionari sono:

Metodo	Descrizione
d.items()	Restituisce gli elementi di d come un insieme di tuple
d.keys()	Restituisce le chiavi di d
d.values()	Restituisce i valori di d
d.get(chiave, default)	Restituisce il valore corrispondente a chiave se presente, altrimenti il valore di default (None se non specificato)
d.pop(chiave, default)	Rimuove e restituisce il valore corrispondente a chiave se presente, altrimenti il valore di default (dà KeyError se non specificato)
d.update(d2)	Aggiunge gli elementi del dizionario d2 a quelli di d
d.copy()	Crea e restituisce una copia di d
d.clear()	Rimuove tutti gli elementi di d

lista completa con le spiegazioni → funzioni dir(), help()

Sommario

- Note sul corso
- Introduzione al linguaggio
- Concetti propedeutici
- Numeri e variabili
- Stringhe e sequenze
- Liste, tuple, set, dizionari
- **Variabili booleane e istruzioni condizionali**
- Iterazione
- Comprehension
- Funzioni
- Errori ed eccezioni
- Moduli
- Lavorare con i file
- Jupyter notebook

Variabili booleane e Istruzioni condizionali

Variabili booleane

Una variabile booleana è una variabile che può assumere solo 2 possibili valori:

True

False

In entrambi i casi la prima lettera **DEVE** essere maiuscola

Sono tipicamente utilizzate nei test (es. if)

Spesso rappresentano il risultato di una condizione (es. `x<5` , `isMaggiorenne(21)` , ...)

NB: `int(True)` → 1 `int(False)` → 0

Vero o Falso?

Vero

True

tutti i numeri diversi da 0

tutti gli oggetti non vuoti

Falso

False

il numero 0

tutti gli oggetti vuoti (es. "", [], (), {})

oggetto None

Operatori booleani

X == Y

Vero se X e Y sono uguali

Falso altrimenti

X != Y

Vero se X e Y sono diversi

Falso altrimenti

Operatori booleani

X and Y

Vero se sia X che Y sono veri

Falso altrimenti

X or Y

Vero se almeno uno tra X e Y è vero

Falso altrimenti

not X

Vero se X è falso

Falso altrimenti

If statement

Si utilizza per l'esecuzione condizionale di istruzioni e nella sua forma più generale si presenta come segue:

if test1 :	# if test
numero variabile di istruzioni	
elif test2 :	# elif opzionale
numero variabile di istruzioni	
...	
elif testN :	# elif opzionale
numero variabile di istruzioni	
else:	# else opzionale
numero variabile di istruzioni	

If statement – esempio 1

a = 10

b = 20

if a == b :

print("numeri uguali")

else :

print("numeri diversi")

If statement – esempio 2

opzione = 2

if opzione == 1 :

 print("Hai selezionato l'opzione 1")

elif opzione == 2 :

 print("Hai selezionato l'opzione 2")

elif opzione == 3 :

 print("Hai selezionato l'opzione 3")

else :

 print("Nessuna opzione valida selezionata")

If statement – esempio 3

```
num = 12
```

```
if num % 2 == 0 :
```

```
    print("num è pari")
```

```
    if num % 3 == 0 :
```

```
        print("ed è anche multiplo di 3")
```

```
    else :
```

```
        print("ma non è multiplo di 3")
```

```
else :
```

```
    print("num è dispari")
```

Sommario

- Note sul corso
- Introduzione al linguaggio
- Concetti propedeutici
- Numeri e variabili
- Stringhe e sequenze
- Liste, tuple, set, dizionari
- Variabili booleane e istruzioni condizionali
- **Iterazione**
- Comprehension
- Funzioni
- Errori ed eccezioni
- Moduli
- Lavorare con i file
- Jupyter notebook

Iterazione

While statement (loop o ciclo while)

Si usa quando è necessario ripetere l'esecuzione di un blocco di codice (una o più istruzioni) un **numero indefinito di volte** fino al verificarsi di una certa condizione.

Nella forma più generale si presenta così:

while test :

una o più istruzioni

loop test

corpo del loop

else :

una o più istruzioni

else opzionale

codice eseguito se non si è
usciti dal loop con un break

Cicli (pass, continue, break)

Nel corpo dei cicli è spesso possibile trovare (o dover usare) tre istruzioni particolari:

pass

continue

break

Cicli (pass)

Lo scopo della parola riservata **pass** è:
non fare assolutamente niente !

Si usa normalmente in 2 situazioni :

- 1) come segnaposto (**debug**)
- 2) quando la sintassi del linguaggio richiede una o più istruzioni che vogliamo inserire successivamente (**prototipazione**)

Cicli (pass)

def funzione():

pass

prototipazione

while condizione:

pass

prototipazione

istruzioni

pass

debug breakpoint

istruzioni

Cicli (continue)

In un ciclo, **continue** forza l'inizio della successiva iterazione (eventuali istruzioni residue dell'iterazione corrente non verranno eseguite)

Normalmente segue un test (if).

```
x = 10
```

```
while x:
```

```
    x = x - 1
```

```
    if x % 2 == 0:
```

```
        continue      # impedisce la stampa dei dispari
```

```
    print(x)
```

Cicli (break)

In un ciclo, **break** causa l'uscita forzata dal ciclo (eventuali istruzioni residue dell'iterazione corrente non verranno eseguite)

Normalmente segue un test (if).

```
while True:
```

```
    nome = input("Inserisci un nome: ")
```

```
    if nome == "stop":
```

```
        break
```

```
    print("Hai inserito il nome " + nome)
```

Cicli (continue VS break)

Entrambi :

- si usano solitamente nel blocco di un ciclo
- interrompono l'esecuzione dell'iterazione corrente del ciclo

continue innesca l'iterazione successiva

break causa l'uscita forzata dal ciclo

For statement (loop o ciclo for)

Si usa quando è necessario ripetere l'esecuzione di un blocco di codice (una o più istruzioni) un **numero definito di volte** pari alla numerosità degli elementi di un oggetto iterabile (es. sequenze)

È un generico iteratore che scandisce gli elementi di una sequenza (su cui è possibile eseguire operazioni).

Funziona su qualsiasi oggetto iterabile (es. stringhe, liste, tuple, ...)

Può contenere pass, continue e break.

For statement (loop o ciclo for)

Nella forma più generale si presenta così:

for target in object :	# assegna gli elem. di object a target
una o più istruzioni	# corpo del loop
else :	# else opzionale
una o più istruzioni	# codice eseguito se non si è usciti dal loop con un break

Esempio:

```
for s in "ciao" :  
    print("s")  
else :  
    print("Esecuzione terminata senza incontrare break")
```

For statement (loop o ciclo for)

Se è davvero necessario effettuare conteggi come normalmente si fa in altri linguaggi (**cosa che in Python andrebbe evitata**) è possibile usare un ciclo for con la funzione range().

Java

```
for (int i = 0; i < 10; i++) {  
    print(i);  
}
```

Python

```
for i in range(10):  
    print(i)
```

Il for in Python va inteso come un for-each, usandolo come sopra:

- lo si snatura
- si perde di efficienza

Funzione range

`range([start,] stop [, step])`

ritorna un oggetto iterabile di tipo range

<code>range(5)</code>	→	0 , 1 , 2 , 3 , 4
<code>range(0,5)</code>	→	0 , 1 , 2 , 3 , 4
<code>range(0,5,1)</code>	→	0 , 1 , 2 , 3 , 4
<code>range(0,5,2)</code>	→	0 , 2 , 4
<code>range(0,5,3)</code>	→	0 , 3
<code>range(10,5,-1)</code>	→	10 , 9 , 8 , 7 , 6
<code>range(-2,2)</code>	→	-2 , -1 , 0 , 1
<code>range(100,20,-20)</code>	→	100 , 80 , 60 , 40

Sommario

- Note sul corso
- Introduzione al linguaggio
- Concetti propedeutici
- Numeri e variabili
- Stringhe e sequenze
- Liste, tuple, set, dizionari
- Variabili booleane e istruzioni condizionali
- Iterazione
- **Comprehension**
- Funzioni
- Errori ed eccezioni
- Moduli
- Lavorare con i file
- Jupyter notebook

Comprehension

Comprehension

Le **comprehension** servono a creare nuovi oggetti (partendo da oggetti iterabili esistenti) senza dover:

- creare manualmente un oggetto vuoto
- aggiungere gli elementi individualmente

Sono considerate ***syntactic sugar***, in buona sostanza rappresentano un modo compatto di eseguire operazioni che richiederebbero più linee di codice.

Comprehension

Vengono tipicamente usate per :

- creare nuove liste/set/dizionari
- filtrare elementi
- trasformare elementi

List comprehension

Supponiamo di avere la seguente lista

$$L = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]$$

e di voler trasformare ogni suo elemento ad esempio elevandolo al quadrato in modo da ottenere

$$L2 = [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]$$

List comprehension

Tale operazione è fattibile con il seguente codice:

```
L2 = []                # creo una lista vuota che conterrà i quadrati
for num in L:          # scandisco tutti gli elem della lista originale
    L2.append(num**2)  # popolo la nuova lista con i quadrati
```

Questo frammento di codice :

- rappresenta il modo "classico" di procedere
- è simile a quello di altri linguaggi
- è perfettamente lecito
- è intuitivo ed autoesplicativo

List comprehension

In alternativa si può scrivere:

```
L = [num**2 for num in L]
```

usando una comprehension che risulta:

- sicuramente più compatta
- funzionalmente equivalente
- ma a prima vista DECISAMENTE poco intuitiva

Ha davvero senso imparare le comprehension ??

SI

List comprehension

È necessario impararle poiché:

- permettono di risparmiare codice
- sono largamente diffuse
- possono essere anche 2 volte più veloci (C)

Comprehension

La sintassi cambia in accordo al tipo di oggetto iterabile utilizzato:

list → **[expr for elem in seq]**

set → **{expr for elem in seq}**

dict → **{expr: expr for elem in seq}**

Comprehension: esempi

>>> # **list** comprehension che crea una lista di quadrati

>>> **[x**2 for x in range(10)]**

[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]

>>> # **set** comprehension che crea un set di cubi

>>> **{x**3 for x in range(10)}**

{0, 1, 64, 512, 8, 343, 216, 729, 27, 125}

>>> # **dict** comprehension che mappa lettere lowercase
all'equivalente uppercase

>>> **{c: c.upper() for c in 'abcde'}**

{'c': 'C', 'e': 'E', 'a': 'A', 'b': 'B', 'd': 'D'}

Comprehension

Tecnicamente in una comprehension è possibile:

- aggiungere degli if

```
>>> [x/2 for x in range(20) if x%2 == 0]  
[0.0, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0]
```

```
res = []  
for x in range(20):  
    if x % 2 == 0:  
        res.append(x / 2)  
  
>>> res  
[0.0, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0]
```

- inserire un numero arbitrario di for ognuno avente un suo if opzionale

```
>>> [x + y for x in 'abc' for y in 'lmn']  
['al', 'am', 'an', 'bl', 'bm', 'bn', 'cl', 'cm', 'cn']
```

```
res = []  
for x in 'abc':  
    for y in 'lmn':  
        res.append(x + y)  
  
>>> res  
['al', 'am', 'an', 'bl', 'bm', 'bn',  
'cl', 'cm', 'cn']
```

Comprehension

È uno strumento pensato per eseguire **semplici tipologie di iterazioni** in maniera compatta

Se la complessità dell'iterazione si spinge troppo oltre (es. 3 for con if incorporati) è decisamente preferibile rinunciare alla compattezza offerta dallo strumento in favore di:

- leggibilità
- comprensibilità
- facilità di modifica

Se qualcosa è troppo difficile da capire probabilmente non è una buona idea !

Funzioni `map()` e `filter()`

Sono 2 funzioni builtin che svolgono un compito simile alle comprehension:

`map(func, seq)`

applica la funzione `func` a tutti gli elementi di `seq` e ritorna un nuovo oggetto iterabile

`filter(func, seq)`

ritorna un oggetto iterabile contenente tutti gli elementi di `seq` per cui `func(elel)` è true

Funzione **map()**

`map(func, seq)`

è simile a

`[func(elem) for elem in seq]`

```
>>> # definisco una funzione che ritorna il quadrato di un numero
>>> def square(n):
...     return n**2
...
>>> squares = map(square, range(10))
>>> squares # map ritorna un oggetto iterabile
<map object at 0xb6730d8c>
>>> list(squares) # convertendolo in lista si possono vedere gli elementi
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
>>> # la seguente listcomp è equivalente a usare list(map(func, seq))
>>> [square(x) for x in range(10)]
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Funzione **filter()**

`filter(func, seq)`

è simile a

`[elem for elem in seq if func(elem)]`

```
>>> # definisco una funzione che ritorna True se il numero è pari
>>> def is_even(n):
...     if n%2 == 0:
...         return True
...     else:
...         return False
...
>>> even = filter(is_even, range(10))
>>> even # filter ritorna un oggetto iterabile
<filter object at 0xb717b92c>
>>> list(even) # convertendolo in lista si possono vedere gli elementi
[0, 2, 4, 6, 8]
>>> # la seguente listcomp è equivalente a usare list(filter(func, seq))
>>> [x for x in range(10) if is_even(x)]
[0, 2, 4, 6, 8]
```

Sommario

- Note sul corso
- Introduzione al linguaggio
- Concetti propedeutici
- Numeri e variabili
- Stringhe e sequenze
- Liste, tuple, set, dizionari
- Variabili booleane e istruzioni condizionali
- Iterazione
- Comprehension
- **Funzioni**
- Errori ed eccezioni
- Moduli
- Lavorare con i file
- Jupyter notebook

Funzioni

Funzioni

Spesso in un programma bisogna eseguire molte volte un dato calcolo o un dato compito (logica applicativa) con diversi parametri di input.

(Es. calcolo interessi, inserimento cliente, ...)

Un possibile modo per gestire questa esigenza è

- sviluppare il codice necessario
- ricopiarlo ogni volta cambiando l'input

Questa soluzione può avere senso solo se applicata in caso di codice usa e getta !

Funzioni

La **duplicazione di codice** presenta diversi inconvenienti:

- allunga il programma
- rende il codice meno leggibile
- è spesso causa di errori
- genera problemi di manutenibilità

VA EVITATA !

Funzioni

La corretta soluzione al problema consiste nella **modularizzazione** del codice attraverso l'impiego delle funzioni:

- si scrive il codice in un "contenitore" (la funzione)
- si richiama la funzione quando necessario

Vantaggi:

- programma più breve
- codice più ordinato e leggibile
- meno errori
- manutenibilità
- riutilizzo

Funzioni

Esistono molte funzioni già disponibili in Python che coprono esigenze comuni (es. `print`, `sqrt`, ...)

Per esigenze specifiche è possibile definire nuove funzioni. (es. chiusura mese su URBI :-D)

Una funzione :

- può avere 0 o più parametri di input
- restituisce sempre un valore

Funzioni - Definizione

Sintassi:

```
def nomeFunzione(eventuali_parametri):  
    istruzioni
```

Es.

```
def calcolaAreaRettangolo(lato1, lato2):  
    return lato1 * lato2
```

```
def isNumeroPari(num):  
    if num % 2 == 0 :  
        return True  
    return False
```

Funzioni - Chiamata

Sintassi:

```
nomeFunzione(eventuali_parametri)
```

Es.

```
area = calcolaAreaRettangolo(5, 10)
if isNumeroPari(area):
    print("rettangolo con area pari")
print("rettangolo con area dispari")
```

NB: il numero e l'ordine dei parametri di input in fase di chiamata deve rispettare la firma

Funzioni - return

La keyword return nel corpo di una funzione serve a ritornare un valore al chiamante

Il codice della funzione può contenere 0 o più return (se non ci sono return viene ritornato None)

Eseguito un return la funzione termina ritornando al chiamante il valore che segue return (se tale valore non esiste viene ritornato None)

Ne consegue che in presenza di più return verrà eseguito solo il primo ad essere raggiunto

Funzioni - Parametri

Quando una funzione ammette parametri di input questi possono essere di 2 tipi:

- obbligatori
- default

I parametri obbligatori (quelli usati fino ad ora) **devono precedere** eventuali parametri di default

Eventuali parametri di default prevedono una sintassi differente.

Funzioni - Parametri

Es.

```
def cm(feet=0, inches=0):  
    inches_to_cm = inches * 2.54  
    feet_to_cm = feet * 12 * 2.54  
    cm = inches_to_cm + feet_to_cm  
    return cm
```

Parametri di
default
nome = valore

```
def totale(stipendio, extra=0):  
    return stipendio + extra
```

Parametri
obbligatori
precedono quelli
di default

Funzioni - Documentazione

È buona norma aggiungere come prima riga di codice di una funzione un commento speciale (**docstring**) che descrive lo scopo della funzione.

Es.

```
def totale(stipendio, extra=0):  
    """Calcola entrate mensili"""  
    return stipendio + extra
```

Questo commento sarà visibile se richiamiamo la funzione di aiuto help sulla funzione → **help(totale)**

Sommario

- Note sul corso
- Introduzione al linguaggio
- Concetti propedeutici
- Numeri e variabili
- Stringhe e sequenze
- Liste, tuple, set, dizionari
- Variabili booleane e istruzioni condizionali
- Iterazione
- Comprehension
- Funzioni
- **Errori ed eccezioni**
- Moduli
- Lavorare con i file
- Jupyter notebook

Errori ed Eccezioni

Errori ed Eccezioni

Nella scrittura ed esecuzione di un programma non sempre tutto fila liscio, l'errore è sempre dietro l'angolo!

In Python possiamo distinguere 2 tipi fondamentali di errori:

- 1) errori di sintassi
- 2) eccezioni

Errori di sintassi

Gli **errori di sintassi** (noti anche come errori di parsing) sono il tipo più comune di messaggio di errore che si riceve quando si sta ancora imparando Python.

```
>>> while True print("ciao")
      File "<stdin>", line 1
        while True print("ciao")
                        ^
      SyntaxError: invalid syntax
```

In questi casi il **parser** riporta la riga incriminata e mostra una piccola freccia in corrispondenza del punto in cui è stato rilevato l'errore (più o meno).

Nel caso l'input provenga da un file, vengono stampati anche il nome del file e il numero di riga.

Eccezioni

Vengono chiamate **eccezioni** tutte le condizioni di errore che possono verificarsi in un programma.

Quando un programma esegue un'operazione non valida, essa viene rilevata dall'**interprete** che genera un'eccezione.

Le eccezioni si propagano automaticamente finchè non vengono catturate e gestite (se non vengono gestite, il programma mostra un messaggio di errore e termina)

È possibile catturare, gestire, e riportare eccezioni.

Eccezioni

test.py

```
def funzione1(x, y):  
    return x / y  
  
def funzione2(x, y):  
    return funzione1(x, y)  
  
def funzione3(x, y):  
    return funzione2(x, y)  
  
ris = funzione3(3,0)
```

Le eccezioni includono un **traceback** che riporta informazioni sulla sequenza di operazioni che hanno portato all'errore durante l'esecuzione del programma

Il traceback mostra questa sequenza di chiamate in ordine cronologico

La chiamata più recente (quella che ha originato il problema) si trova alla fine.

```
C:\code>python test.py  
Traceback (most recent call last):  
  File "test.py", line 10, in <module>  
    ris = funzione3(3,0)  
  File "test.py", line 8, in funzione3  
    return funzione2(x, y)  
  File "test.py", line 5, in funzione2  
    return funzione1(x, y)  
  File "test.py", line 2, in funzione1  
    return x / y  
ZeroDivisionError: division by zero
```

Eccezioni

I diversi tipi di eccezioni esistenti sono organizzati in una **gerarchia** che include eccezioni più o meno specifiche.

Ad esempio, l'eccezione *ZeroDivisionError* è un caso particolare di *ArithmeticError*, che è un sotto-tipo di *Exception*, che a sua volta è un sotto-tipo di *BaseException*

Gerarchia completa delle eccezioni disponibile qui:
<https://julien.danjou.info/python-exceptions-guide/>

Eccezioni

Non è una buona idea affidarsi alla fortuna e ignorare le molte possibili cause di eccezione di un programma che possono causarne la chiusura forzata . . .

Le eccezioni vanno catturate e gestite !

In Python si usa :

- il costrutto **try / except / finally** per la corretta gestione delle eccezioni
- il comando **raise** per rilanciare eccezioni al chiamante senza gestirle

Eccezioni – try / except / finally

La forma più generale del costrutto è la seguente:

try:

istruzioni a rischio

except [*type* [*as value*]]:

istruzioni da eseguire in caso di errore

[**except** [*type* [*as value*]]:

istruzioni da eseguire in caso di errore]*

[**else:**

istruzioni da eseguire se NON si verificano errori]

[**finally:**

istruzioni da eseguire alla fine in ogni caso]

Eccezioni – try / except / finally

In questo corso ci limitiamo a trattare la forma seguente:

try:

istruzioni a rischio

[except type:

istruzioni da eseguire in caso di errore]+

[finally:

istruzioni da eseguire alla fine in ogni caso]

Eccezioni – try / except / finally

try:

risultato = 7 / 0

except ArithmeticError:

print("Operazione non valida !")

try:

risultato = 7 / 0

except ZeroDivisionError:

print("Operazione non valida !")

Il meccanismo funziona in entrambi i casi poiché ZeroDivisionError è un sottotipo di ArithmeticError

Eccezioni – try / except / finally

Se gli errori nel blocco try possono essere diversi si accodano altri blocchi except:

try:

```
a = float(input("Inserisci un numero: "))  
b = float(input("Inserisci un altro numero : "))  
print(a / b)
```

except ValueError:

```
print("Hai inserito un valore non corretto!")
```

except ZeroDivisionError:

```
print("Operazione non valida !")
```

Eccezioni – try / except / finally

La clausola finally contiene codice che viene eseguito in ogni caso, normalmente esegue flush di risorse.

```
numeri = []  
dati = open("fileTesto.txt", "r")  
try:  
    for num in dati:  
        numeri.append(float(num))  
except ValueError:  
    print("Dati non corretti!")  
finally:  
    dati.close()
```


Eccezioni – raise

A volte è preferibile non gestire localmente una eventuale eccezione e rilanciarla al chiamante

In questi casi si usa il comando raise in questi modi:

raise # rilancia l'eccezione più recente

raise instance # es. raise IndexError()

raise class # es. raise IndexError

```
def div(num, den):
```

```
    if den == 0:
```

```
        raise ZeroDivisionError('Impossibile dividere per 0')
```

```
    return num / den
```

Sommario

- Note sul corso
- Introduzione al linguaggio
- Concetti propedeutici
- Numeri e variabili
- Stringhe e sequenze
- Liste, tuple, set, dizionari
- Variabili booleane e istruzioni condizionali
- Iterazione
- Comprehension
- Funzioni
- Errori ed eccezioni
- **Moduli**
- Lavorare con i file
- Jupyter notebook

Moduli

Moduli

Modulo è il nome generico associato ad un file di testo contenente codice Python.

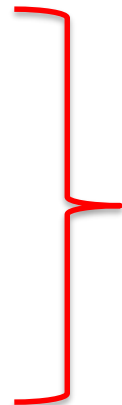
All'interno di un modulo si trovano tipicamente:

variabili

costanti

funzioni

classi



Attributi del modulo

richiamabili anche al di fuori del file.

Moduli - Librerie

Python include "out of the box" una vasta lista di moduli standard (standard library)

Una libreria è un **insieme di moduli** importabili in un programma per svolgere agevolmente operazioni più o meno complesse limitando considerevolmente la scrittura del codice necessario. (Invece di scrivere da zero la soluzione al problema ne uso una già pronta)

<https://docs.python.org/3/library/>

Moduli - Import

In aggiunta alla standard library è possibile scaricare o definire nuovi moduli.

Per accedere al contenuto di un modulo è necessario prima di tutto importarlo nello script o nel programma che si sta scrivendo.

Le istruzioni di import sono le seguenti:

import nomeModulo [**as** nome]

from nomeModulo **import** nomeAttr [**as** nome]

Moduli - Esempio

```
import math # importo il modulo math

def calcolaAreaCerchio(raggio):
    return raggio * raggio * math.pi # uso l'attributo PI del modulo

area = calcolaAreaCerchio(2)

print("l'area del cerchio di raggio 2 è " + str(area))
```

- il modulo math include vari attributi `dir(math)`
- una volta importato, Python crea una variabile math tramite la quale richiamare gli attributi del modulo
- noi abbiamo usato la costante pi richiamandola con la sintassi:

nomeModulo.nomeAttr ossia math.pi

Moduli – From ... Import

L'istruzione "import nomeModulo" importa tutti gli attributi di nomeModulo (richiamabili tramite ".")

Se ci interessa un solo attributo specifico possiamo usare la sintassi :

from nomeModulo **import** nomeAttr

es. **from** math **import** pi

e usare l'attributo scrivendone direttamente il nome:

es. raggio * raggio * **pi**

Moduli – As

Può capitare che i nomi di alcuni moduli o attributi siano:

- particolarmente lunghi (scomodi da usare)
- simili a nomi già in uso (confusione)
- uguali a nomi già in uso (problemi)
- difficili da ricordare

In questi casi è possibile effettuare un rename tramite:

```
import nomeModulo as mioNome
```

```
es.    import math as matem
```

```
from nomeModulo import nomeAttr as mioNomeAttr
```

```
es.    from math import sqrt as radice
```

Sommario

- Note sul corso
- Introduzione al linguaggio
- Concetti propedeutici
- Numeri e variabili
- Stringhe e sequenze
- Liste, tuple, set, dizionari
- Variabili booleane e istruzioni condizionali
- Iterazione
- Comprehension
- Funzioni
- Errori ed eccezioni
- Moduli
- **Lavorare con i file**
- Jupyter notebook

Lavorare con i file

File

Il tipo "file" in Python è un core-type, ciò significa che è possibile usare funzioni built-in per eseguire le più comuni operazioni :

- creazione
- apertura
- lettura
- scrittura
- chiusura

Ulteriori operazioni richiedono l'importazione di moduli:

- cancellazione
- spostamento
- ...

File - Apertura

Per aprire un file si usa la funzione **open** solitamente con 2 parametri di tipo stringa:

- 1) path del file da aprire
- 2) modalità

es. `mioFile1 = open("c:/dati/punteggi.txt", "r")`
 `mioFile2 = open("corso.txt", "r")`
 `mioFile3 = open("guida.txt")`

Se per il primo parametro non si esplicita il path completo, il file verrà cercato nella directory locale.

Se viene omesso il secondo parametro si sottointende "r"

File - Apertura

Le modalità di apertura (processing mode) di un file sono :

- r** apertura in modalità lettura
- w** crea (se già non esiste) un file e lo apre in scrittura
- a** apertura in scrittura in append

e possono avere opzioni aggiuntive come:

- b** modalità binaria
- +** abilita sia lettura che scrittura

La funzione open può avere anche un terzo parametro di input opzionale tramite il quale controllare:

- la modalità di scrittura (buffering)
- l'encoding

File - Metodi

Una volta aperto un file è possibile chiamare sul riferimento ottenuto i vari metodi a disposizione tra cui i più comuni:

Operazione	Significato
<code>S = input.read()</code>	Legge l'intero contenuto del file
<code>S = input.read(N)</code>	Legge i successivi N caratteri (o bytes)
<code>S = input.readline()</code>	Legge la linea successiva (compreso <code>\n</code>)
<code>mylist = input.readlines()</code>	Legge l'intero file convertendolo in una lista di stringhe
<code>S = input.write(stringa)</code>	Scrive una stringa di caratteri (o bytes)
<code>S = input.writelines(lista)</code>	Scrive tutte le stringhe contenute nella lista
<code>S = input.close()</code>	Chiude il file
<code>S = input.flush()</code>	Forza la scrittura
<code>S = input.seek(N)</code>	Cambia la posizione nel file dell'offset N

File - Metodi

Si può come sempre usare `dir` per la lista completa dei metodi:

es:

```
>>> miofile = open("c:/code/test.py","r")  
>>> dir(miofile)    # lista completa dei metodi
```

È importante ricordare che:

- ciò che viene letto da file viene acquisito come stringa
- ciò che viene scritto su file va prima convertito in stringa con `str()`

Lavorare con i file

È **importante ricordarsi di chiudere un file aperto** in precedenza nel programma al fine di evitare:

- problemi sul file (es. mancata scrittura)
- spreco di risorse (memoria allocata)

Normalmente ciò viene fatto tramite una apposita istruzione che potrebbe essere omessa per:

- imperizia
- distrazione / dimenticanza

Ciò può essere causa di problemi !

with

Questa istruzione rappresenta una delle migliorie apportate al linguaggio nel corso degli anni.

È un compound statement (come if, try, for, ...) che permette di risparmiare codice.

Viene usata in fase di apertura di un file con lo scopo di chiuderlo automaticamente al termine delle istruzioni contenute nel blocco.

Data la comodità dello strumento è frequente vederlo in uso nel codice.

with

Esempio di lettura di un file senza with :

```
mioFile = open("murphy.txt","r")  
for riga in mioFile:  
    print(riga)  
mioFile.close()
```



Ogni soluzione genera
nuovi problemi

A scopo didattico verifichiamo che il file sia stato
effettivamente chiuso in questo modo:

```
print(mioFile.closed)
```



True

with

Con with si risparmia la riga del close():

```
with open("murphy.txt","r") as mioFile:  
    for riga in mioFile:  
        print(riga)
```



Ogni soluzione genera
nuovi problemi

e non si corre il rischio di dimenticare il file aperto, infatti
se si testa l'effettiva chiusura il risultato è:

```
print(mioFile.closed)
```



True

with

with ci fa risparmiare anche il blocco finally (se questo viene usato solo per chiudere il file) :

senza with

```
numeri = []
dati = open("dati_importanti.txt","r")
try:
    for num in dati:
        numeri.append(float(num))
except ValueError:
    print("Errore!!!")
finally:
    dati.close()
```

con with

```
numeri = []
with open("dati_importanti.txt","r") as dati:
    try:
        for num in dati:
            numeri.append(float(num))
    except ValueError:
        print("Errore!!!")
```

Sommario

- Note sul corso
- Introduzione al linguaggio
- Concetti propedeutici
- Numeri e variabili
- Stringhe e sequenze
- Liste, tuple, set, dizionari
- Variabili booleane e istruzioni condizionali
- Iterazione
- Comprehension
- Funzioni
- Errori ed eccezioni
- Moduli
- Lavorare con i file
- **Jupyter notebook**

Jupyter notebook

Jupyter notebook

È una webapp che gira su un server locale del PC

Serve a creare e condividere documenti multimediali chiamati **notebook** che possono contenere:

- codice (che può essere eseguito al momento)
- testo formattato (in Markdown o HTML)
- immagini
- grafici
- contenuti multimediali (es. video Youtube)

Strumento didattico molto valido (aula, blog, github)

Jupyter notebook

Permette a ricercatori e istituzioni scientifiche di pubblicare i risultati ottenuti corredandoli con le procedure seguite e le opportune spiegazioni del caso.

Permette ai fruitori di replicare interattivamente il processo (anche con dati differenti) ed effettuare contestualmente modifiche al codice.

Comunemente utilizzato in ambito Data Science e Machine Learning data la facilità di utilizzo e i vantaggi offerti.

Jupyter notebook

È un componente **non incluso nell'installazione standard** di Python, va quindi installato separatamente tramite il gestore di pacchetti pip digitando da terminale:

pip install jupyter

L'alternativa solitamente più utilizzata dai fruitori dello strumento è quella di installare **Anaconda**, una particolare distribuzione di Python che include :

- diverse librerie per la DataScience e il ML
- diversi tool tra cui Jupyter

Jupyter notebook

Un file notebook è formato da un insieme numerato di celle di diverso tipo.

La numerazione indica l'ordine di esecuzione

La tipologia di una cella può essere:

- **Code** (conterrà codice)
- **Markdown** (conterrà testo esplicativo)
- **RawNBConvert** (testo che non subirà formattazione)
- **Heading** (usato per l'inserimento di titoli)

Un file notebook ha estensione **.ipynb** ma può facilmente essere convertito/esportato in diversi formati (pdf, html,...)

Jupyter notebook

Le modalità di utilizzo di un file notebook sono 2:

- edit mode
- command mode

In edit mode è possibile modificare il contenuto delle celle che saranno contornate di verde (e icona della matita in alto a destra).

In command mode le celle sono invece contornate di azzurro/grigio (no icona della matita).

Jupyter notebook

È inoltre possibile all'interno delle celle di tipo Code inserire comandi particolari (non Python) che forniscono:

- accesso alla console del sistema operativo
cominciano con il carattere "!"
es. `!ls` mostra il contenuto della directory
- ulteriori funzionalità
cominciano con "%" e con "%%"
sono chiamati magics
il comando `%lsmagic` mostra l'elenco completo

Jupyter notebook

Per avviare Jupyter :

Start → Programmi → Anaconda3 → Jupyter Notebook

oppure

creare una nuova cartella, entrarci da terminale ed eseguire il comando "jupyter"

L'applicativo sarà visibile tramite browser all'indirizzo:

<http://localhost:8888/tree>

Grazie per l'attenzione !