Corso Python base

dr. Alessandro Dentella

slides: https://corsi.e-den.it

Alessandro Dentella
sandro.dentella@gmail.com

Presentazione personale

- Fisico prestato all'informatica da sempre (con intrusioni nell'insegnamento)
- Prima principalmente sistemista Linux ora principalmente programmatore
- Uso Linux dal '92
- Uso Python dal 2005
- In passato tcl/tk, awk, php, perl
- Uso emacs, recentemente vscode, non capisco vi...

Alessandro Dentella
sandro.dentella@gmail.com

Organizzazione giornata

- Divisione tempo fra lezione frontale e (molti) esercizi personali per sedimentare le cose che vengono spiegate e per prendere confidenza con gli errori ed il debug

- Velocita' dipendente dal gruppo. 5 giornate per la parte Python e strumenti di sviluppo (hg, PyCharm, documentazione...)

- Slides: http://docs.thux.it/

Alessandro Dentella
sandro.dentella@gmail.com

IDE

- Useremo Colab (google), VsCode o PyCharm versione community, preinstallato sulle macchine.
- Personalmente ho sempre e solo usato emacs... ma VsCode/PyCharm permettono un apprendimento più rapido grazie alla facile navigazione del codice ed una maggiore facilità nella gestione dei *virtualenv*

python

Alessandro Dentella
sandro.dentella@gmail.com

Sviluppo del corso - Python

- Sintassi
- Tipi di base
- Eccezioni
- Idiomi del linguaggio:
  - Generatori
  - Decoratori
- Moduli/import
- Libreria std:
  - builtin-functions
  - os/sys
  - Date / datetime

- Classi / ereditarietà
- Accesso database
- Debug
- Linguaggio come “glue code” di altre script (subprocess)
- Moduli/packages – virtualenv
- Scrivere documentazione
- concorrenza (asincio & multiprocessing)
- Tests

Alessandro Dentella
sandro.dentella@gmail.com

Scheda

- General-purpose language , nato nel 1990 circa. Deve il nome al *“Mounthy Python's Flying Cyrcus”*
- Guido Von Rossum è l'autore
- Usato da Google, YouTube, NASA e innumerevoli altri enti, google app engine
- Scritto in C
- Il codice viene compilato in un bytecode e successivamente eseguito. La compilazione trova errori sintattici non semantici ed avviene al più tardi nel momento in cui un modulo viene importato

Alessandro Dentella
sandro.dentella@gmail.com

Scheda presentazione

Linguaggio interpretato

Linguaggo dinamico:

- Molte operazioni che altri linguaggi fanno in fase di compilazione, vengono fatti a runtime (termine non sempre rigoroso) :
  - Estensioni codice
  - Estensione oggetti
  - Modifiche ai tipi
- Tipizzato dinamicamente: il tipo è collegato al valore, non alla variabile (Python è fortemente tipato)

Alessandro Dentella
sandro.dentella@gmail.com

Py2 vs. Py3

- Python2 non è più supportato dal 2020
- Il corso usa Python3
- Esiste ancora una grande quantità di codice scritto in Python2

Alessandro Dentella
sandro.dentella@gmail.com

Implementazioni

- Esistono implementazioni in java -jython, C# iron-python, python (!) - pypy
- Psyco genera codice assembly da python
- Pyrex and Cython genera codice c da python
- Ogni novità viene introdotta dopo approfondite discussioni che passano tramite le PEP (Python Enhence Proposal)
- Sito: www.python.org

python

Alessandro Dentella
sandro.dentella@gmail.com

Community

- stackoverflow molto attiva
- Mailing list in italiano molto disponibile
- Conferenza italiana annuale
- Sito ricchissimo di documentazione (ita e mondo)
- Cookbook molto ricco ospitato da activestate: http://code.activestate.com/recipes/langs/python/
- Newgroup e mailing list internazionali di altissimo livello.

python

Alessandro Dentella
sandro.dentella@gmail.com

Documentazione

- http://python.org/doc ha ricca documentazione:
  - Tutorial
  - Libreria di sistema
  - PEP (Python Enhance Proposal)
  - https://realpython.com/
  - https://docs.python.org/3/glossary.html

python

Alessandro Dentella
sandro.dentella@gmail.com

La galassia Python

Applicazioni:
Zope
Plone
OpenObject
OpenERP

Embedded:
cellulari

Database:
Python DB-API
ODBC
PostgreSQL
MySQL
MSSQL
...
SQLAlchemy (ORM)
SqlObject (ORM)
...

LDAP:
ldap
pumpkin (ORM style)

GUI:
GTK
Qt
Wx
...

Docs:
docutils
sphinx

python powered

ipython

Math:
numpy
Scipy
Modellazione
Simulazione
matplotlib

Network Programming
Twisted (asincrona)
Monitoring tools (pymon)

Web:
cgi
django
web2py
pylons
turbogears

Report:
Reportlab (pdf)
OpenOffice
Relatorio
PyChart
.xls (lettura/scrittura)

Img:
PIL

Devel:
Nose, py.test
ide (eclipse, komodo, wind, ...)
Setuptools (PyPi)
Mercurial dVCS
Pyrex (Python → c)

Libreria **standard**: → utilizzata!!!

Built-in:
functions (~80)
types
exceptions

Moduli (necessitano 'import ...'):

String services	Threading	GUI (tk)
Data types(datetime, ...)	Xml	Introspection
Numeric modules	Internet protocols	Packaging
File & directories	Debug	Internationalization
Data persistence (sqlite..)	Profiling	Net services (cli/server)
Os Interaction - I/O	Test	...

Linee guida che prediligono leggibilità (import this)
PEP (Python Enhancement Proposal)

Object Oriented
Dinamico fortemente tipato
Sintassi essenziale

Linux/Unix
Windows
Mac OS/2
Amiga
Java vm
.NET vm

OpenSource

python™

Facile da imparare

Grazie ad alcune scelte sintattiche e di stile risulta un linguaggio particolarmente facile da apprendere. In generale la leggibilità è una regola di stile a cui un *pythonista* tiene molto. La sintassi aiuta in questo. L' indentazione è il modo di raggruppare gli statement in luogo delle parentesi:

```
if a == b:
    print('a e b sono uguali')
```

python

Alessandro Dentella
sandro.dentella@gmail.com

Python – The zen of Python

- Beautiful is better than ugly.
- Explicit is better than implicit.
- Simple is better than complex.
- Complex is better than complicated.
- Flat is better than nested.
- Sparse is better than dense.
- Readability counts.
- Special cases aren't special enough to break the rules.
- Although practicality beats purity.

Alessandro Dentella
sandro.dentella@gmail.com

Import this

- Errors should never pass silently.
- Unless explicitly silenced.
- In the face of ambiguity, refuse the temptation to guess.
- There should be one-- and preferably only one --obvious way to do it.
- Although that way may not be obvious at first unless you're Dutch.
- Now is better than never.
- Although never is often better than *right* now.
- If the implementation is hard to explain, it's a bad idea.
- If the implementation is easy to explain, it may be a good idea.
- Namespaces are one honking great idea -- let's do more of those!

Alessandro Dentella
sandro.dentella@gmail.com

ipython

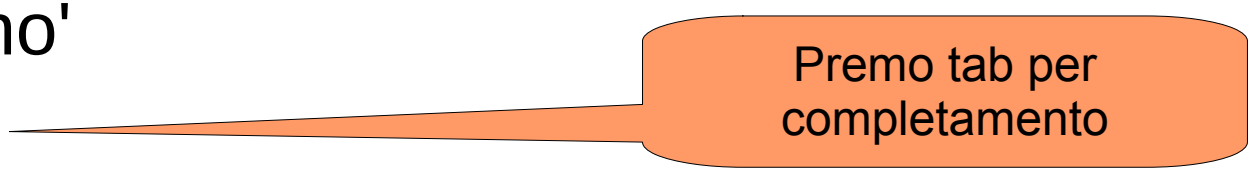
- `Ipython` è un interprete interattivo che mi permette di eseguire codice python, lanciare programmi, analizzare la visibilità delle variabili, lanciare il debugger... Il prompt cambia da “>>>” a “In[1]”
- Ottimo strumento di esplorazione dei moduli, delle classi, dell'help
- 'Tab' completa le parole chiave, 'Control-l' anche metodi ed attributi (In Unix tab completa tutto)
- Da emacs è possibile eseguire il codice direttamente nella finestra di editing

Alessandro Dentella
sandro.dentella@gmail.com

ipython

Ottimo strumento di esplorazione dei moduli, delle classi, dell'help:

```
In [3]: a = 'uno'
a.s (tab)
In [4]: a.s
a.split       a.startswith  a.swapcase
a.splitlines  a.strip
```



python

Alessandro Dentella
sandro.dentella@gmail.com

Ipython: introspezione

'?' dopo il metodo

Docstring: stringa definita subito dopo “def”, documenta la func.

```
In [2]: a.split?
Type:        builtin_function_or_method
String form: <built-in method split of str object at 0x7fa5e246eab0>
Docstring:
S.split(sep=None, maxsplit=-1) -> list of strings

Return a list of the words in S, using sep as the
delimiter string.  If maxsplit is given, at most maxsplit
splits are done. If sep is not specified or is None, any
whitespace string is a separator and empty strings are
removed from the result.
```

Alessandro Dentella
sandro.dentella@gmail.com

jupyter notebook

ipython nel tempo è stato separato in più componenti

- jupyter (il motore, agnostico rispetto al linguaggio, il nome fonde Julia, Python and R)
- ipython (il motore/backend/kernel Python di jupyter)

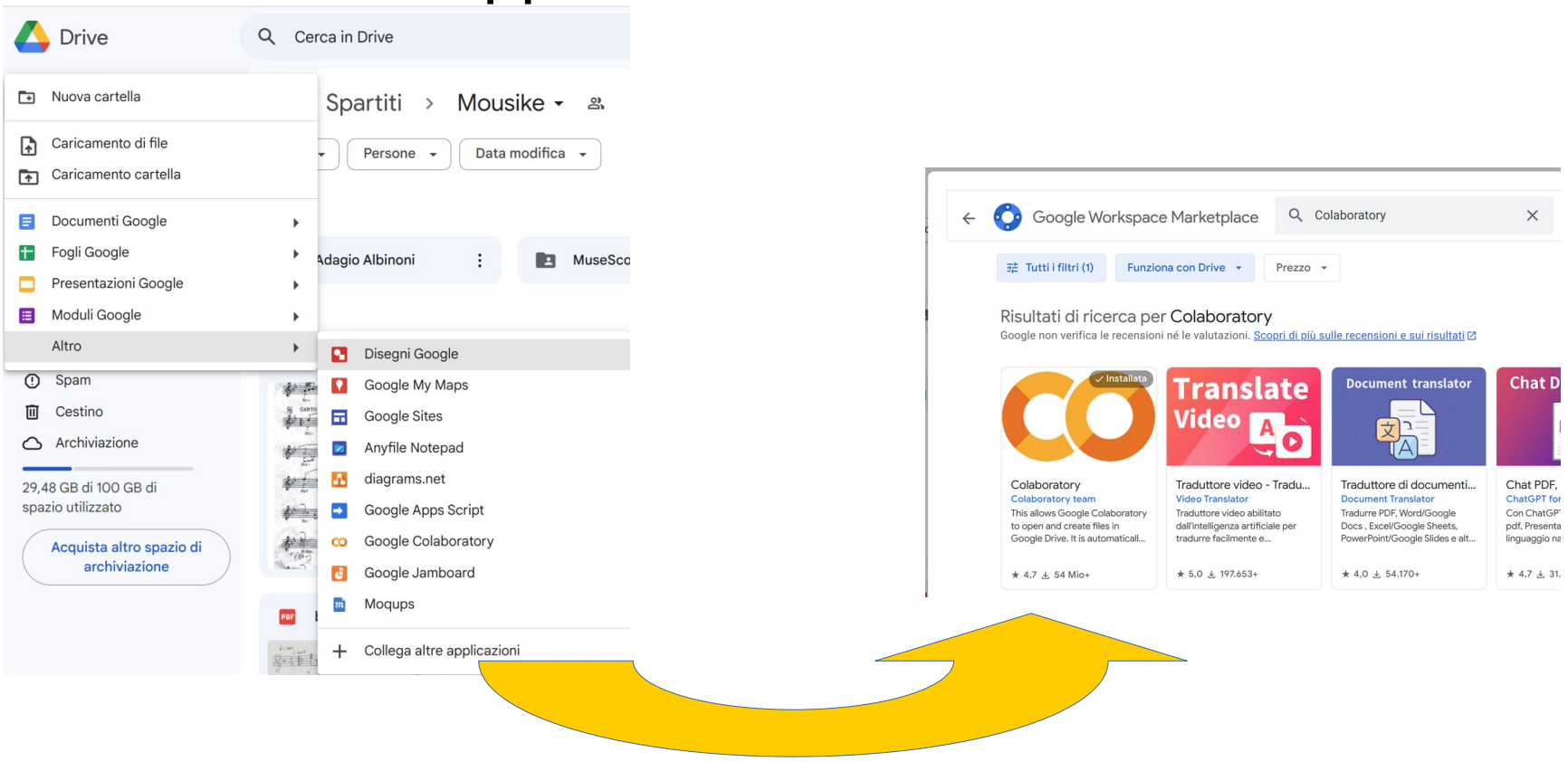
Google ha colab, una piattaforma per eseguire codice Python all'interno di notebook jupiter.

Si può utilizzarla direttamente o eseguendo un file con estensione .ipynb in google drive (occorre abilitare l'app colab)

Alessandro Dentella
sandro.dentella@gmail.com

Colab

- Per abilitare l'app colab



python

Alessandro Dentella
sandro.dentella@gmail.com

colaboratory

- Andando su https://colab.research.google.com/
  Possiamo leggere il benvenuto che è già un notebook jupiter
- Andando su:
  https://corsi.e-den.it/colab

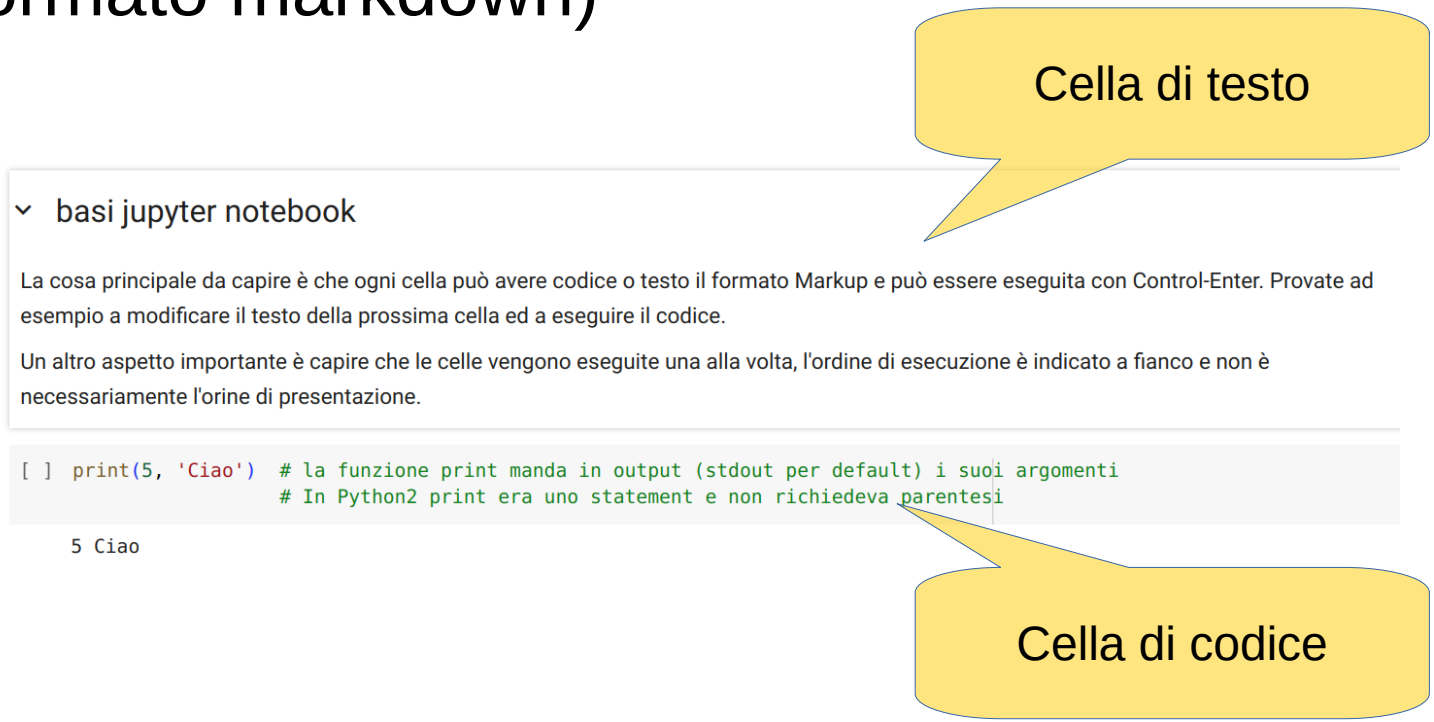
  possiamo vedere il notebook con gli esempi della lezione

python

Alessandro Dentella
sandro.dentella@gmail.com

Struttura di un notebook jupiter

- Composto di celle che possono contenere codice o testo (formato markdown)



Alessandro Dentella
sandro.dentella@gmail.com

Per cominciare

- Commenti: `#`
- Assegnamenti: `a = 1`
- Estensione file: `.py`
- Modulo: un file contenente definizioni e statement
- `import module_name / python module_name.py`
-
```
def my_func(txt):
    print(txt)
```

Alessandro Dentella
sandro.dentella@gmail.com

Numeri

```
>>> 1 + 2
3
>>> a, b = 3, 4      # assegnazione multipla
>>> a + b
7
>>> 10/3
3.3333333335
>>> 10//3
3
>>> 1.1 + 2.2
3.3000000000000003
>>> print('0.2 = {0:.17f}'.format(0.2))
0.20000000000000001
>>> from decimal import Decimal
>>> Decimal('1.1') + Decimal('2.2')
Decimal('3.3')
```

Alessandro Dentella
sandro.dentella@gmail.com

Calcolatrice & more...

```
In [1]: 2 * 3
Out[1]: 6

In [2]: _   * 6
Out[2]: 36

In [3]: a = 'Ciao Mondo!

In [4]: a * 3
Out[4]: 'Ciao Mondo!... Ciao Mondo!... Ciao Mondo!...
'

In [5]: a + a
Out[5]: 'Ciao Mondo!... Ciao Mondo!...'
```

_ contiene il valore ritornato dalla riga precedente

La somma indica la concatenazione

Alessandro Dentella
sandro.dentella@gmail.com

Assegnazione multipla

```
# Posso assegnare più variabili contemporaneamente
In [6]: m, n = 2, 5

In [7]: m
Out[7]: 2

In [8]: n
Out[8]: 5

In [9]: m, n
Out[9]: (2, 5)

In [10]: m, (x, y, z) = 2, (6, 7, 8)
In [11]: y
Out[11]: 7

In [12]: a = b = c = 0   # tutti avranno valore 0
```

Questo oggetto non è un numero...
È una tupla, ovvero una lista immutabile

In questo caso l'assegnazionemultipla arriva in profondità a spacchettare la tupla

python

Alessandro Dentella
sandro.dentella@gmail.com

Python3, ass multipla

Python3 aggiunge un' ulteriore possibilità all'assegnazione multipla tramite l'operatore '*', questo risulta anche più comodo dell'operatore slicing che vedremo fra poco

```
In [1]: first, *rest = range(6)

In [2]: first
Out[2]: 0

In [3]: rest
Out[3]: [1, 2, 3, 4, 5]

In [4]: first, *rest, last = range(6)

In [5]: last
Out[5]: 5

In [6]: rest
Out[6]: [1, 2, 3, 4]
```

python

Alessandro Dentella
sandro.dentella@gmail.com

Stringhe

- `"`, `'`, `"""`, `'''`
- La forma con 3 ripetizioni di ' o " permette di andare a capo e può al suo interno avere qualunque apice, doppio o singolo:

```
A = '''
Testo lungo
'''
```

- Normalmente usato nelle docstring
- Esistono anche `u'unicode'` (in py3 tutto è unicode...), `r'raw'`, `b'bytestring'` ed altri meno usati

Alessandro Dentella
sandro.dentella@gmail.com

slicing

Lo ***slicing*** è una operazione fatta sulle stringhe (ma poi vedremo in generale sulle liste) referenziando le lettere in base alla posizione. Potremo quindi prendere dei sottoinsiemi definendoli in modo molto efficace:

```
In [1]: a = "Ciao Mondo"

In [2]: a[0]
Out[2]: 'C'

In [3]: a[1]
Out[3]: 'i'

In [4]: a[1:6]
Out[4]: 'iao M'

In [5]: a[-2:]
Out[5]: 'do'
```

Alessandro Dentella
sandro.dentella@gmail.com

Hello world

```
print('ciao mondo')   # obbligatorio da 3.0,
print('ciao ' + “mondo”) # concatenazione: +
dest = 'mondo'           # definizione variabile
print('ciao', dest)        # concatenazione implicita
print('ciao', end='')     # Evita \n
print('mondo')
print('ciao {}'.format(dest))  # versione ufficiale
print('{y} {dest}'.format(dest=dest, y='ciao')) #con var
print('ciao %s' % (dest))  # output formattato + tupla
print('ciao mondo'.title())    # ogni stringa e' un
                                oggetto con metodi
print(f'ciao {dest}')    # da py3.6
```

Alessandro Dentella
sandro.dentella@gmail.com

String formatting

- Questa forma è stata introdotta con python 2.6. La PEP 3101 la spiega in gran dettaglio

```
txt = '''Mi chiamo {0} {cognome} e ho un bimbo di nome {1}'''

print(txt.format('Alessandro', 'Nicolas', cognome='Dentella'))

'Mi chiamo Alessandro Dentella e ho un bimbo di nome Nicolas'
```

- `.format` è metodo delle stringhe
- Sono ammessi argomenti posizionali e nominali. Gli argomenti posizionali {0}, {1} fanno riferimento agli argomenti posizionali passati alla funzione. Possono anche essere impliciti gli indici:
  `'Mi chiamo {} ed ho un bimbo di nome {}'`
- Le formattazioni ammesse sono molte di più e comprendono la dotted notatione per accedere agli attributi, oltre alla formattazione specifica di numeri ed altri tipi, fare riferimento alla documetazione ad alla PEP

Alessandro Dentella
sandro.dentella@gmail.com

f-string

```
nome = 'Sandro'
cognome = 'Dentella'
msg = f'mi chiamo {nome} {cognome}'
```

A differenza dal metodo `.format`, non richiede che gli argomenti per la sostituzione siano passati, ma li preleva dal namespace corrente

python

Alessandro Dentella
sandro.dentella@gmail.com

Unicode

```
In [1]: why_u = u'perch\xe9'

In [2]: why = b'perch\xc3\xa9'

In [3]: print(why)
perché

In [4]: print(why.decode(
           'utf-8'))
perché

In [5]: len(why), len(why_u)
Out[5]: (7, 6)
```

Dalla doc di Python:

A character is the smallest possible component of a text. ‘A’, ‘B’, ‘C’, etc., are all different characters. So are ‘È’ and ‘Í’. (...)

The Unicode standard describes how characters are represented by **code points.** A code point is an **integer value**, usually denoted in base 16. In the standard, a code point is written using the notation U+12ca to mean the character with value 0x12ca (4810 decimal). A ***Unicode string is a sequence of code points***, which are numbers from 0 to 0x10ffff. This sequence needs to be represented as a set of bytes (meaning, values from 0-255) in memory. The rules for translating a Unicode string into a sequence of bytes are called an **encoding**.

python

Alessandro Dentella
sandro.dentella@gmail.com

Unicode in Python2 vs Python3

- **Python2:** `unicode` type creates an unicode string. The unicode() constructor has the signature `unicode(string[, encoding, errors])`. All of its arguments should be 8-bit strings. The first argument is converted to Unicode using the specified encoding; if you leave off the encoding argument, the ASCII encoding is used for the conversion, so characters greater than 127 will be treated as errors
  - Another important method is `.encode([encoding], [errors='strict'])`, which returns an 8-bit string version of the Unicode string, encoded in the requested encoding.
  - Python's 8-bit strings have a `.decode([encoding], [errors])` method that interprets the string using the given encoding:
- **Python3:** `str` type contain Unicode characters
- Python3 now accepts u'' strings (clearly 'u' doesn't do anything here)
- In py2 you can force the behaviour of py3 just importing:

  ```
  from __future__ import unicode_literals
  ```

- http://www.joelonsoftware.com/articles/Unicode.html Is and interesting article:

  The Absolute Minimum Every Software Developer Absolutely, Positively Must Know About Unicode and Character Sets

Alessandro Dentella
sandro.dentella@gmail.com

Encoding – da python doc

- Suggerisco la lettura integrale di https://docs.python.org/2/howto/unicode.html
- Encodings don’t have to handle every possible Unicode character, and most encodings don’t. For example, Python’s default encoding is the ‘ascii’ encoding. The rules for converting a Unicode string into the ASCII encoding are simple; for each code point:
  - If the code point is < 128, each byte is the same as the value of the code point.
  - If the code point is 128 or greater, the Unicode string can’t be represented in this encoding. (Python raises a UnicodeEncodeError exception in this case.)
- UTF-8 is one of the most commonly used encoding. UTF stands for “Unicode Transformation Format”, and the ‘8’ means that 8-bit numbers are used in the encoding. (There’s also a UTF-16 encoding, but it’s less frequently used than UTF-8.) UTF-8 uses the following rules:
  - If the code point is <128, it’s represented by the corresponding byte value.
  - If the code point is between 128 and 0x7ff, it’s turned into two byte values between 128 and 255.
  - Code points >0x7ff are turned into three- or four-byte sequences, where each byte of the sequence is between 128 and 255.

Alessandro Dentella
sandro.dentella@gmail.com

Unicode vs. UTF-8

Unicode is just a standard that defines a character set (UCS) and encodings (UTF) to encode this character set. *But in general, Unicode is refered to the character set and not the standard.*

UTF-8 usa un pattern dinamico, quindi usa alcuni bit per informare quanti ulteriori bytes formano il carattere.

Binary format of bytes in sequence					
1st Byte	2nd Byte	3rd Byte	4th Byte	Number of Free Bits	Maximum Expressible Unicode Value
0xxxxxxx				7	007F hex (127)
110xxxxx	10xxxxxx			(5+6)=11	07FF hex (2047)
1110xxxx	10xxxxxx	10xxxxxx		(4+6+6)=16	FFFF hex (65535)
11110xxx	10xxxxxx	10xxxxxx	10xxxxxx	(3+6+6+6)=21	10FFFF hex (1,114,111)

http://stackoverflow.com/questions/643694/utf-8-vs-unicode

Alessandro Dentella
sandro.dentella@gmail.com

Inserto sintassi

- Il codice Python assomiglia molto allo pseudocodice usato in molti esempi astratti.
- È intuitivo per:
  - Uso furbo dell'indentazione che rende chiarissimo il rapporto fra uno statement ed il prec/successivo
  - Assenza di parentesi per definire i blocchi di codice
  - Stile non compatto ma esplicito

```
for x in 1,2,3,4,5:
    print(x)
```

L'indentazione indica che
'print x' è lo statement da iterare

Alessandro Dentella
sandro.dentella@gmail.com

if

Nel test si usa ==, non è possibile fare una assegnazione
Durante un test: è un errore troppo diffuso, Python lo evita alla radice in quanto non lo permette

- Analogamente:

```
if x == '3':
    print("'x' vale", 3)
```

python

Alessandro Dentella
sandro.dentella@gmail.com

funzioni

- Le funzioni vengono definite con lo statement def, come già visto.
- Il nome della funzione soggiace agli stessi limiti del nome delle variabili (`[a-zA-Z_]+[_a-zA-Z0-9]*`)
- Una funzione è un oggetto, come tutto in Python...

```
def ciao():
    return 'Ciao mondo'
```

python

Alessandro Dentella
sandro.dentella@gmail.com

Ciao mondo in funzione

```
def ciao():
    '''stampa un saluto
       questa e' una docstring di una funzione
    '''
    print('ciao mondo')   # notare l' indentazione

>>> ciao()       # >>> indica il mio input
ciao mondo
>>> print(ciao)
<function ciao at 0x844802c>
>>> print(ciao.__doc__)
stampa un saluto
    questa e' una docstring di una funzione
>>> type(ciao)
<type 'function'>
>>> type(ciao())
<type 'NoneType'>
```

La docstring viene salvata nell'attributo __doc__

Alessandro Dentella
sandro.dentella@gmail.com

Prendi esempio

```
# coding: utf-8
"""
Nel mezzo del cammin di nostra vita
mi ritrovai per una selva oscura,
ché la diritta via era smarrita.

Ahi quanto a dir qual era è cosa dura
esta selva selvaggia e aspra e forte
che nel pensier rinova la paura!
"""
def stampa(txt=__doc__, n=1):
    """
    crea una lista delle prime lettere di ogni verso
    """
    ret = []

    for line in txt.split('\n'):
        ret += [line[:n]]
    return ret

if __name__ == '__main__':

    for string in stampa():
        print string
```

Dichiarazione encoding usato nel codice

Docstring (modulo):
Stringa iniziale (a parte coding e sh-bang)

Docstring di funzione

Forma idiomatica per capire se il modulo sta girando come script (il suo attributo __name__ è '__main__ o importato da altri

Alessandro Dentella
sandro.dentella@gmail.com

Function annotation

Le funzioni possono essere *annotate*, questo serve agli editor per dare suggerimenti, ma non influisce sull'esecuzione del codice Python. Esiste un modulo della libreria apposta per generare eventuali nuovi types: `typing`

```
def sum_two_numbers(a: int, b: int) -> int:
    return a + b
```

Alessandro Dentella
sandro.dentella@gmail.com

Inserto sintassi

- Il codice Python assomiglia molto allo pseudocodice usato in molti esempi astratti.
- È intuitivo per:
  - Uso furbo dell'indentazione che rende chiarissimo il rapporto fra uno statement ed il prec/successivo
  - Assenza di parentesi per definire i blocchi di codice
  - Stile non compatto ma esplicito

```
for x in 1,2,3,4,5:
    print(x)
```

L'indentazione indica che 'print(x)' è lo statement da iterare

Alessandro Dentella
sandro.dentella@gmail.com

Assegnazione

```
In [5]: a = 'Ciao Mondo'
# len è la funzione che restituisce la lunghezza di un oggetto
In [6]: len(a)
Out[6]: 10
# ipython permette di scriverlo senza () per brevità
In [7]: len a
------> len(a)
Out[7]: 10
# L'assegnazione di un item non è permessa per una stringa
In [8]: a[0] = 'x'
---------------------------------------------------------
TypeError                                 Traceback (most re

/home/misc/src/hg/isi/isi-2.0/<ipython console> in <module>(

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

Un errore solleva una eccezione.
Ogni eccezione ha un proprio tipo
Questa è di tipo TypeError

python

Alessandro Dentella
sandro.dentella@gmail.com

Altre eccezioni

```
In [9]: a = uno
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)

/home/misc/src/hg/isi/isi-2.0/<ipython console> in <module>()

NameError: name 'uno' is not defined

In [10]: 1 + ''
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)

/home/misc/src/hg/isi/isi-2.0/<ipython console> in <module>()

TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Alessandro Dentella
sandro.dentella@gmail.com

None

- NoneType è un tipo base che ha un solo valore: None

```
In [1]: a = None


In [2]: type(a)
Out[2]: <type 'NoneType'>
In [3]: a
In [4]: print(a)
Out[4]: None
```

Alessandro Dentella
sandro.dentella@gmail.com

Boolean

- True
- False
- La funzione bool(arg) converte un valore in boolean. Vengono valutate a False:
  - '' # Stringhe vuote
  - (), [], {}, set() # tuple, liste, dizionari vuoti, set vuoti
  - 0

python

Alessandro Dentella
sandro.dentella@gmail.com

Liste

```
# in una lista gli oggetti possono non essere omogenei
In [1]: l = [1, 'm', 'n']

In [2]: len l
------> len(l)
Out[2]: 3
# è ammessa l'assegnazione di un item
In [3]: l[0] = 'x'

In [4]: l
Out[4]: ['x', 'm', 'n']
# L' operatore += aggiunge (append) oltre la dimensione corrente
In [5]: l += ['o', 'p', 2.5]

In [6]: l
Out[6]: ['x', 'm', 'n', 'o', 'p', 2.5]

# Lo slicing funziona come già visto
In [12]: l[-3:]
Out[12]: ['o', 'p', 2.5]
```

Alessandro Dentella
sandro.dentella@gmail.com

Mutable/Immutable

```
In [1]: a = 'uno'

In [2]: id(a)
Out[2]: 160911296

In [3]: a += 'due'

In [4]: id a
------> id(a)
Out[4]: 160919744
```

Id ritorna un identificativo unico dell'oggetto

Dopo la concatenazione la stringa Non ha più lo stesso identificativo. Non è Lo stesso oggetto modificato ma è un **altro** oggetto

```
In [5]: l = [1, 2]

In [6]: id l
------> id(l)
Out[6]: 148581228

In [7]: l += [3]

In [8]: l
Out[8]: [1, 2, 3]

In [9]: id l
------> id(l)
Out[9]: 148581228
```

La lista anche dopo la modifica mantiene lo stesso identificativo

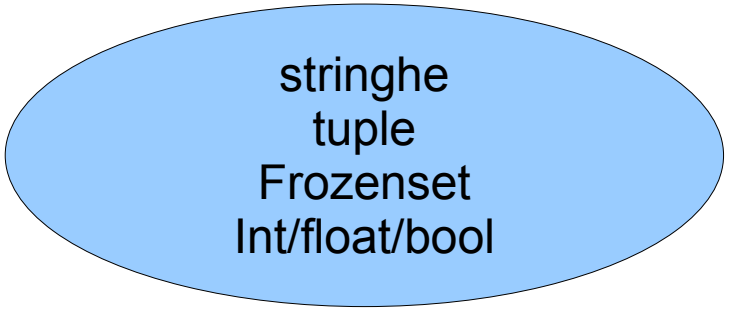
Alessandro Dentella
sandro.dentella@gmail.com

Mutable/Immutable 2

Questa differenza risulta molto importante:

- **per lo scopo (visibilità) delle variabili nelle funzioni**
- per efficienza: risulta più efficiente aumentare una lista e poi "joinarla" in un colpo che aumentare una stringa molte volte, nel quale caso il testo viene riscritto in continuazione.

Un'intera gamma di oggetti saranno divisi in base a questo criterio



Alessandro Dentella
sandro.dentella@gmail.com

Creare liste

- `l1 = [1, 2 , 'mangio', 'bevo']`
- Molte funzioni restituiscono liste (o tuple)

```
l2 = list(range(5))
```

- Questa è una forma **molto usata**, molto compatta ed efficiente chiamata *list comprehension*:

```
l3 = [x for x in range(5) if x % 2]
```

```
l3 = [x*x for x in range(5) if x % 2]
```

```
[mio_format(u) for u in users if
u.email.endswith('gmail.com')]
```

- `l4 = list((1,2))`

Esercizio:
Creare lista di mesi anno:
- Estrarre mesi primaverili
- Estrarre quelli che cominciano per 'a'
- Stessa lista senza list comprehension
- Stessa lista con 'filter'

Alessandro Dentella
sandro.dentella@gmail.com

Esercizi

- Varianti di funzione che estrae i primi 25 numeri pari:
  - Con ciclo for
  - Con list comprehension
  - Con filter map
  - Con range
- Misurare l'efficacia relativa delle diverse soluzioni (usa timeit di ipython)

python

Sequence: tuple e liste

Le sequence sono container ordinati di oggetti: stringhe, tuple, liste
*Stringhe e tuple sono **immutabili**, le liste sono **mutabili.** Per creare una tupla basta la “,”*

a = m, n

```
a = (1, 2, 'b')                 # tupla: possono contenere tipi
                                # differenti, al pari delle liste
aa = ((1,2), (2,3, 'b'))        # quindi anche altre tuple
>>> len(aa)          # len non è un metodo ma un builtin
2                    # funzione su ogni oggetto che definisce __len__
b = [ 1, 2 ]           # la lista
b += ['a']             # augmented assignement
>>> b
[1, 2, 'a']
b += ('x', 'y')
```

Esercizi:
- Creare una lista di un solo elemento
- Creare una tupla di un solo elemento

Alessandro Dentella
sandro.dentella@gmail.com

Iterazioni

- Scrivere una funzione che produca la media di una lista in ingresso. Gestire anche il caso di 0 elementi.
- Scrivere una funzione `space` che data una stringa aggiunga spazi fra una lettera e l'altra.
- Scrivere una funzione che date in argomento 2 liste produca una lista con gli elementi presi a turno dalle 2 liste.

python

Inserto sintassi

Spesso nella assegnazione di valori è utile un costrutto simile:

```
if value:
   my_var = value

else:

   my_var = 10
```

Che può essere sintetizzato in:

```
my_var = value or 10

name = user and user.last_name or “Missing”
```

- Esiste anche la forma con and
- Esiste anche la forma (ternary operation):

```
my_var = 10 if a == b else value

name = user.last_name if user else “Missing”
```

Alessandro Dentella
sandro.dentella@gmail.com

Set e frozenset

- Sono collezioni non ordinate di elementi unici
- Permettono di fare operazioni insiemistiche:
  - Unione: `a.union('b')`, `a = b`

    `a.update('b')`, `a |= b`
  - Differenza
  - Intersezione
  - Differenza simmetrica
- Frozenset è la versione immutabile di un set

python

Alessandro Dentella
sandro.dentella@gmail.com

Set in azione

```
In [3]: a = set(x for x in range(5))

In [4]: a
Out[4]: set([0, 1, 2, 3, 4])

In [5]: a.add(4)

In [6]: a
Out[6]: set([0, 1, 2, 3, 4])

In [7]: a.add(6)

In [8]: a
Out[8]: set([0, 1, 2, 3, 4, 6])

In [9]: 6 in a
Out[9]: True

In [10]: 10 in a
Out[10]: False
```

Esercizio:
Creare 4 set per le 4 stagioni
partendo dai mesi
Creare set mesi caldi partendo dai set delle stagioni

Alessandro Dentella
sandro.dentella@gmail.com

Dizionari

- Un *mapping* è una collezione di oggetti indicizzata da valori arbitrari chiamati *key*
- Le chiavi devono essere immutabili. La chiave è un valore ricavato tramite la funzione hash, questa funzione solleva una eccezione TypeError se eseguita su oggetti mutabili.
- A = { 1 : 'uno',

  'due' : 2,

  date.today() : 'ciao', # una data è immutabile

  }

Assegnazione: A['foo'] = 'bar'

- In python3 esiste dict-comprehension:

```
{x: x**2 for x in range(10)}
{k: v*2 for k, v in ((1,2), (3,4))}
```

Alessandro Dentella
sandro.dentella@gmail.com

update & co

- `dict1 = {1:1, 2:2, 3:3}`

  `dict2 = {3:'tre', 4:4}`

- `dict1 | dict 2`
  `>>> {1: 1, 2: 2, 3: 'tre', 4: 4}`
  `dict1` NON è cambiato

- `dict1 |= dict2`
  `dict1`
  `>>> {1: 1, 2: 2, 3: 'tre', 4: 4`

  dict1 è cambiato

L'operatore | per dizionari è stato aggiunto nella versione Python 3.9

Before:

```
self.attrs =
self.known_attrs.copy()
self.attrs.update(attrs)
```

After:

```
self.attrs = self.known_attrs |
attrs
```

python

Alessandro Dentella
sandro.dentella@gmail.com

Metodi dizionari

- `.keys(), .values(), .items()` # in py3 return "views" not lists!
- `.get(key, default=None)  l[key]`
- `.update(other_dict)`
- `.pop(), .popitem()`
- `.setdefault(key, None)`
- `.clear()`

python

Alessandro Dentella
sandro.dentella@gmail.com

dict

- È possibile iterare sulle chiavi, sui valori, su entrambe:

```
for k in mydict.keys():  … (analoga a:)
for k in mydict:  ...
for val in mydict.values():
for key, val in mydict.items(): # iteritems() in Py2
```

- `if  key in mydict:`

  testa la presenza della chiave nel dizionario

- Esiste una versione di dizionario che ricorda l'ordine di inserimento `collections.OrderedDict`

python

Alessandro Dentella
sandro.dentella@gmail.com

Esercizi sui dizionari

- Creare un dizionario `mesi_dict` che abbia per chiave i mesi e per valore il numero di giorni usando dict_comprehension + modifiche a mano (ovvero assegnazioni esplicite)
- Trovare quanti gg ha un anno
- A partire da `mesi_dict`, creare un secondo dizionario `gg_dict` che abbia per chiave i giorni e per valore i nomi del mese
- A partire da `mesi_dict` create `gg_dict_set,` che abbia per chiave i gg e per valore un set con i mesi, in tutto 3 elementi: 28, 30, 31
- Verificare che un *mutable* non può essere usato come *key* di un dizionario
- Estrarre tutti i nomi dei mesi in ordine alfabetico

python

Dizionari e namespace

- Il dizionario in Python è **molto** usato
- I namespace (mapping fra nome ed oggetto) è implementato come dizionario che può essere analizzato guardando `__dict__` di un modulo/oggetto
- `locals()` e `globals()` restituiscono dizionari
- `exec/execfile` accettano come argomenti anche dizionari come namespace globale/locale:

```
exec(compile(open('test.py').read(),
             'test.py', 'exec'), {os: os})
```

esegue `test.py` che avrà a disposizione il modulo os come se fosse stato preventivamente importato

Alessandro Dentella
sandro.dentella@gmail.com

Oggetti e metodi

- Ogni typo di Python è una classe, ed ogni oggetto (una lista, una tupla, una stringa un numero) ne è una istanza.
- Possiamo vedere quali metodi hanno i nostri oggetti in modo empirico, da ipython usando Control-l:

  ```
  a = 'ciao mondo'
  ```

  `a.`(Control-l) → restiusce un elenco di attributi e metodi
- Possiamo usare '?' per cercare la descrizione

python

Alessandro Dentella
sandro.dentella@gmail.com

Esercizi sui tipi base

Usando i metodi base e la documentazione proviamo a:

- Stringhe:
  - creare una lista iterando su una stringa
  - maiuscolizzare (capital, upper, title)
- Liste:
  - Aggiugere elementi
  - Trovare posizione di un elemento (che indice ha 'aprile'?)
- Dizionari:
  - Estendere il dizionario (update)
  - Trovare il valore o un default
  - Cancellare un elemento

Esercizi

- Creare una funzione che stampi i primi100 numeri attuando questa sostituzione
  - Per ogni multiplo di 3 scriva ‘fizz’
  - Per ogni multiplo di 5 scriva ‘buzz’

    (e quindi scriva fizz buzz nei casi comuni)
- Stampare la tavola pitagorica
- Creare una lista di numeri contenuti della tavola pitagorica

python

Namespace

- È il mapping fra nomi e oggetti
- Normalmente implementato come dict
- Esempi:
  - Built-in names (len, dir, open, …)
  - Global namespace di un modulo
  - Variabili locali ad una funzione (comprese eventuali altre funzioni/classi)
- Un modulo vede solo ciò che è definto nel modulo stesso, importato o built-in.
- Una funzione vede anche le variabili definite nel modulo dove è definita la funzione se non viene mascherata da una variabile locale con lo stesso nome anche se non passata come argomento

Esercizio:
- Verificare che una funzione **Vede** variabili definite nel modulo
  - Verificare se può modificarle

Alessandro Dentella
sandro.dentella@gmail.com

Passaggio parametri

Parametri posizionali (obbligatori)

```
def area_rettangolo(b, h):
    return b * h
```

```
area_rettangolo(10, 3)
area_rettangolo(b=10, h=3)
area_rettangolo(h=3,  b=10)
```

Parametri nominali (con default)

```
def ivato(prezzo, iva=22):
    return prezzo * (1 + iva/100)
```

```
ivato(100)
ivato(100, iva=23)
ivato(100, 23)
ivato(iva=22, prezzo=100)
```

Alessandro Dentella
sandro.dentella@gmail.com

Funzioni: Passaggio parametri

I parametri possono essere passati in modo **posizionale o nominale**.

- Posizionale: nell'ordine in cui sono definiti nella funzione. Sono obbligatori. Se non li metto python solleva un `TypeError: missing positional argument`
- Nominale: nella forma `arg1=valore1`, `arg3=None`… possono essere facoltativi (hanno un default)
- Questo permette la completa flessibilità rispetto all'uso dei valori di default ed alla chiarezza e leggibilità della chiamata alla funzione
- I valori di default possono essere di qualunque tipo, ma **vengono valutati in fase di parsing** (import del modulo), quindi se usiamo come valore di default `date.today()`, sarà sempre la stessa data ad essere passata, la data valutata nel momento dell'import del modulo. Questo è un punto delicato, se non lo capite **non** usate liste come default.

Alessandro Dentella
sandro.dentella@gmail.com

Parametri: visibilità

La visibilità di una variabile all'interno di una funziona segue una regola molto semplice:

- Prima viene cercata la variabile nel namespace locale (ovvero viene controllato se esiste una chiave con il nome della nostra variabile in `locals()`)
- Se la variabile non viene trovata allora viene cercata anche nel namespace globale. Questo indipendetemente dal fatto che la variabile sia passata o meno come argomento
- La **scrittura** è sempre nel namespace locale, a meno che sia stata resa globale con il comando `global`

python

Alessandro Dentella
sandro.dentella@gmail.com

Parametri: visibilità (2)

Il passaggio è *per valore* se però l'oggetto passato è *mutable* la funzione può richiedere cambiamenti all'oggetto (es.: lista). Per argomenti mutable è più proficuo pensare che viene passato il puntatore. Se il valore è un puntatore, posso modificare il valore ma non riassegnarlo. Ovvero:

- Un **immutable** (es.: stringa) non v'è dubbio è passata per valore. Ogni eventuale variazione non viene vista all'esterno. Essendo non mutabile anche se scrivo `a += 'b'`, di fatto sto riassegnandole un valore
- Il valore di una lista è solo il puntatore alla lista. Quindi se all'interno della funzione modifico la lista, questa modifica è vista anche all' esterno, se la riassegno (`a = [1,2]`), sto collegando (*binding*) questo identificativo ad un **nuovo** indirizzo di memoria

Alessandro Dentella
sandro.dentella@gmail.com

Namespace e dizionari

Il comportamento appena descritto risulta facilmente comprensibile se si pensa alla visibilità implementata attraverso i namespace.

Esistono quindi 2 dizionari, uno delle variabili locali ed uno di quelle globali. Quando si chiede una variabile a quello locale, se Python non la trova nel dizionario locale, la cerca in quello globale:
(`loc.get('myvar', glob.get('myvar')`)

Quando assegnamo, assegnamo sempre in quello locale, oscurando quindi quello globale.

Alessandro Dentella
sandro.dentella@gmail.com

Esercizi

- Creare una funzione che ritorni l'area del cilindro con signature: `area_cilindro(r, h)`
- Creare una funzione che prenda in input un numero di secondi e restituisca ore, muniti, secondi (usare `divmod`)
- Creare funzione `scontato(tot, percentuale)` che restituisca il valore scontato

python

Passaggio parametri: es

```
def params(i0, lst0, lst1):
    i0 = 23
    lst0 = [1,2,3]
    lst1.append(60)
    lst1[0] = 10
```

Rebind oggetto immutable: non cambia il valore all'esterno (77 resta 77)

Rebind mutable: non influenza il modo in cui viene visto fuori in quanto riassegna il valore. *(Pensare all'implementazione del namespace Attraverso i dizionali aiuta a capire questo comportamento)*

```
i_0 = 77
lst_0 = [99]
lst_1 = [50, 51]
```

La manipolazione dell'oggetto list viene vista anche dall'esterno sia che sia passato come argomento (lst_1) sia che non lo sia (lst_2, prox dia)

```
>>> params(i_0, lst_0, lst_1)
>>> print(i_0, lst_0, lst_1)
(77, [99], [10, 51, 60])
```

Esercizio:
- Come si comporta l'operatore +=?
- Si comporta uguale per stringa/tupla/lista?

Alessandro Dentella
sandro.dentella@gmail.com

Visibilità variabili globali

La variabile viene vista regolarmente

```
def params():

    print(lst_2)
    # print(lst_3)  # delicato:
    # lst_3 += [100]
    lst_2.append(100)
     # lst_2 += 100

    print(b)


lst_2 = ['non', 'passata']
b = 'non_passata in arg'
```

La manipolazione dell'oggetto list viene vista anche dall'esterno sia che sia passato come argomento (lst_1) sia che non lo sia (lst_2)

Rebind: .append si comporta diversamente da +=. Python se vede += la pensa variabile locale qui usata Prima della definizione

Vista anche se non passata in argomento

```
>>> params()
['non', 'passata']
non_passata in arg
>>> print(b, lst_2)
'non_passata in arg', ['non', 'passata', 100]
```

Esercizio:
- Come si comporta l'operatore +=?
- Si comporta uguale per stringa/tupla/lista?
- Cosa succede se decommentiamo lst_3?

Alessandro Dentella
sandro.dentella@gmail.com

*args, **kwargs

Nella signature di una funzione possono anche esserci una o entrambe le forme `*args` e `**kwargs`. Se presenti devono seguire i parametri (es.e07)

```
def bevi(quando, cosa='caffe', come='in tazza', **kw):
      '''grande flessibilità nei keyword args'''

     dove = kw.get('dove', '')
     print(f"{quando} bevi {cosa} {come} {dove}")

bevi('oggi', come='a canna', dove='in gita')
```

Alessandro Dentella
sandro.dentella@gmail.com

*args, **kwargs (unpacking)

È possibile passare parametri posizionali a partire da una lista e/o nominali a partire da un dizionario, usando gli operatori * e **:

```
myfunc(*args, **kwargs)
req = ('ora',)
info = {
   'cosa': 'caffe',
   'dove': 'qui'
}
bevi(*req, **info)
```

o -nella funzione precedente- avremmo potuto usare

```
print("{quando} bevi {cosa} {come} {dove}".format(**locals()))
```

Alessandro Dentella
sandro.dentella@gmail.com

Esempi *args e **kw

- Molto usato nelle classi quando si vuole chiamare un metodo di un genitore:

```
def __init__(self, *args, **kwargs):
    return super(MyClass, self).__init__(*args, **kwargs)
```

- Ad esempio quando si usa una funzione per “wrapparne” un' altra:

```
def my_func(*args, **kw):
        logger.debug("args: %r kw: %r", args, kw)

        # ed “inoltra”
        original_func(*args, **kw)
```

Alessandro Dentella
sandro.dentella@gmail.com

- Scrivere una script che stampi a video la docstring che sia stata usata come template per scrivere una lettera con sostituzione delle variabili scritte in un dizionario. Es.:

```
persone = [
    {'name': 'Nicolas', 'age': 8},
    {'name': 'Sandro', 'age': 54}]
```

  Vogliamo stampare per ogni persona :
  xx ha yy anni usando la formattazione vista in precedenza. Come passo i parametri?

- Creare una tabella html con lo stesso criterio

python

Prendere decisioni: Operatori

- Booleani: not, and, or
- Relazionali: <, >, <=, >=, ==, !=
- Appartenenza: `item in list/set, key in dict`

python

Alessandro Dentella
sandro.dentella@gmail.com

Eccezioni

- **Ogni** errore solleva eccezioni.
- Le Eccezioni… non sono eccezioni, sono la norma!!
- Try / except
- Try/ finally
- Try / except / finally
- Le eccezioni si propagano fino ad un handler. Per default se non gestite, viene stampato un traceback dell'errore con indicazione precise del punto. La raffigurazione dell'errore può essere personalizzata (django)
- È prassi normale gestirle piuttosto che prevenirle. Questa prassi ha un nome:

  EAFP: easier to ask for forgiveness than permission

Alessandro Dentella
sandro.dentella@gmail.com

Eccezioni: esempio

```
def just_numbers(l):
    clean = []
    for x in l:
        try:
            x/2
            clean += [x]
        except TypeError as e:
            pass
    return clean
```

Avremmo alternativamente potuto testare
Prima se l'elemento è un numero ma try/except risulta spesso più efficiente

Esercizio:
Creare un dizionario e fare un loop su possibili chiavi e stampare quelle inesistenti

```
In [21]: just_numbers([10, 12, 14, 'spam'])
Out[21]: [10, 12, 14]
```

Alessandro Dentella
sandro.dentella@gmail.com

Eccezioni

- Le eccezioni sono organizzate gerarchicamente per cui è possibile essere più o meno granulari nel gestirle:

  http://docs.python.org/library/exceptions.html#exception-hierarchy

- E' prassi normale crearne ad hoc per l'applicazione.
- Le eccezioni possono essere sollevate con il comando raise:

```
if not key in mydict:
    raise MissingObjException("L'oggetto non esiste")
```

Alessandro Dentella
sandro.dentella@gmail.com

Esercizi

- Rivedere funzione media con uso Eccezioni
- Creare una funzione `distinct` che elimini ripetizioni di una lista data in input
- Creare una funzione che scriva in lettere le cifre 0-9
- Modificare la funzione precedente in modo da accettare un parametro per la lingua

python

Debug

- Esistono alcuni strumenti molto potenti in Python per potere debuggare il codice, il principale è il modulo pdb, di cui il comando principale è set_trace()
- Pdb permette di procedere passo-passo, di impostare dei break-point e di indagare ogni step di esecuzione.

python

Alessandro Dentella
sandro.dentella@gmail.com

Debug con ipython

- Ipython ci permette di fare debug di una script passo-passo.
  ```
  $ ipython
  [1] run my_script  # esegue la script ed                    #
  al termine NON esce
  ```
  Se incontra un errore è possibile entrare nel debugger con il comando `debug`
  ```
  [2] run -d my_script  # esegue debug passo-passo
  ```
  - `l` (*list*) mostra le 10 righe di codice successive
  - `n` (*next*) esegue la riga successiva
  - `s` (*step into*) entra nella funzione
  - `u` (*up*) sale di un livello (chiamante)
  - `d` (*down*) scende di un livello
  - `c` (continue fino al termine programma o al prossimo brake point)

python

Ipython + pdb

- Pdb a partire da ipython è particolarmente comodo in quanto permette di eseguire del codice (ipython script.py) e saltare direttamente nel punto in cui viene sollevata una eccezione (senza neanche avere dovuto preparare la script con ipython). Con il debugger è possibile passare da un frame all'altro (u/d == up/down) ed ispezionare variabili o rilanciare comandi
- Alternativamente dalla shell di ipython si può usare il comando %run per eseguire una script direttamente dal debugger
- In ogni caso è possibile usare il completamento dall'interno del debugger di ipython

Alessandro Dentella
sandro.dentella@gmail.com

ipdb

- Anche in una script python (non ipython) è possibile entrare in una shell interattiva stile ipython usando il modulo ipdb:

  …

  import ipdb; ipdb.set_trace()

  …

- Con ipdb è possibile usare il completamento!
- Potete installare ipdb con: “pip install ipdb”

Alessandro Dentella
sandro.dentella@gmail.com

import

- Ogni file contenente definizioni di variabili, funzioni o classi costituisce un modulo, il modulo chiamato direttamente nell'invocare Python ha nome '__main__'
- Lo statement import rende disponibile il modulo in argomento nello scopo corrente:

```
import os

os.environ
```

Esercizio:
Analizzare la gerarchia di cartelle della installazione Python

- Quando si esegue `'import os'`, python cerca:
  - Un file `os.py` in sys.path (la cartella corrente è normalmente dentro)
  - Una cartella '`os`' (per Python2: con all'interno un file `__init__.py`)
  - Altri meccanismi meno diretti (es. file .egg)

Alessandro Dentella
sandro.dentella@gmail.com

__init__.py

- Lo statement ‘import name’ non richiede che l’oggetto importato sia un file o una cartella. Questo rende possibile spezzettare una file cresciuto troppo in una cartella.
- In questo caso il file __init__.py ha funzione di inizializzatore, ovvero può contenere codice dove ad esempio si importano oggetti da altri moduli
- Quando si importa un modulo di fatto *si esegue* il modulo stesso.

python

Alessandro Dentella
sandro.dentella@gmail.com

Moduli: __file__, .pth

- I moduli importati tramite un file, hanno definito un attributo `__file__` che dice quale è la posizione del modulo nel filesystem.

  Spesso è utile analizzarlo per capire dove è

- Si può estendere il `sys.path` mettendo nel `sys.path` file con estensione `.pth` che elencano le cartelle aggiuntive

Esercizio:
- Creare due moduli a piacere
- Importare uno nell'altro ed eseguire una funzione
- Controllare cosa viene aggiunto nella cartella

.pyc

- In Python2, dopo avere eseguito l' esercizio precedente notiamo che è comparso un file `.pyc` nella cartella degli esercizi: è il file compilato in un formato binario. Questa compilazione avviene al momento dell' import o quando si usa il comando builtin `compile()`. I file `*pyc` in Python3 sono nella cartella `__pycache__`
- E` possibile distribuire solo i file `.pyc` se si vuole oscurare il codice, ma è possibile -se veramente determinati- ricostruire il sorgente.

Alessandro Dentella
sandro.dentella@gmail.com

Moduli vs. package

- Un modulo è un file contente definizioni e statement
- Un package è un modo di strutturare il namespace dei moduli con la notazione puntata: `os.path`, `email.mime.multipart`, `sound.formats.waveread`

```
sound/                          Top-level package
      __init__.py               Initialize the sound package
      formats/                  Subpackage for file format conversions
              __init__.py
              wavread.py
              auread.py
              auwrite.py
              ...
      effects/                  Subpackage for sound effects
              __init__.py
              surround.py
              reverse.py
              ...
```

Alessandro Dentella
sandro.dentella@gmail.com

PYTHONPATH

- Per rendere disponibile il pacchetto ad un interprete (i)python dobbiamo modificar eil sys.path in modo che venga aggiunto il percorso. Possiamo farlo impostando la variabile d'ambiente `PYTHONPATH`, che deve essere *esportata* perché sia vista dai processi figli della shell:

```
export PYTHONPATH=~/corso_python
```

- Questa operazione di esportazione va fatta in ogni terminale
- Quando affronteremo i virtualenv vedremo un altro modo che diventerà il nostro standard per ottenere lo stesso risultato
- Ora sarà possibile eseguire:

```
In [1]: from corso import lezioni

In [2]: lezioni

Out[2]: <module 'corso.lezioni' from ...corso_python/corso/lezioni/__init__.py'>
```

Questa è la rappresentazione
Che python usa per il modulo (__repr__)

Verificate che dopo l'esportazione
di `PYTHONPATH`
l'interprete avrà
`corso_python` nel `sys.path`

entella
sandro.dentella@gmail.com

Installare packages

- Il modo normale di installare moduli Python è tramite una script chiamata `setup.py` ed ultimamente `pyproject.toml`:
  ```
  python setup.py install
  ```
  è cura di chi prepara il modulo scrivere la script che usa a sua volta un package (distutils)
- Esiste opzione *develop* che lascia in loco ma cambia `sys.path` di sistema
- Esistono poi altri 3 packages che aggiungono varie funzioni indispensabili fra cui le dipendenze:
  - Setuptools (easy_install) può installare anche le eggs (vedi oltre)
  - Pip ottimo basato su setuptools

Alessandro Dentella
sandro.dentella@gmail.com

Pypi

- Esiste un repository Pypi (Python Package Index) su cui chi produce un modulo python può caricare il package
- Per installare i moduli qui contenuti è sufficiente:
  `pip install nome_modulo`
  Il comando viene scaricato e installato
- Se un modulo va compilato, in genere questo meccanismo non funziona, bisogna accertarsi di avere gli strumenti (compilatore, librerie, ...), in alternativa vengono offerti pacchetti già compilati

Alessandro Dentella
sandro.dentella@gmail.com

Moduli: versioni

- Python importa sempre e solo l'ultima versione disponibile localmente
- Per utilizzare versioni differenti occorre creare degli ***environment*** differenti in cui il `sys.path` è impostato in modo da mostrare solo i moduli che ci interessano.
- ***Virtualenv*** è lo strumento ottimale per questo tipo di setup, internamente a Thux usiamo una alternativa – ***buildout***- preferita in quanto ha una sintassi dichiarativa che ho reputato più chiara

python

Alessandro Dentella
sandro.dentella@gmail.com

Eggs & Wheels

- Il mondo del packaging in Python è tutt'ora in grande fermento e non ci sono ancora dei punti fissi sugli strumenti da utilizzare
- Un formato comodo in alcuni casi sono le eggs: cartelle (eventualmente zippate) che contengono un package
- Un formato che sostituirà le eggs e risolverà il problema della compilazione sotto Windows/Mac OS sono le Wheels

python

Alessandro Dentella
sandro.dentella@gmail.com

Built-in function

- Metto qui solo le più frequentemente usate:

basestring
bool
dict
list
str / repr / unicode
float / int
set / frozenset
isinstance / issubclass
type
tuple

callable (py2)
setattr / getattr / delattr / hasattr
dir
locals / globals
enumerate
execfile / exec / eval
file / open
filter/map
help
id
iter
input
min / max
print
property
range
reload
sorted
sum
vars
__import__

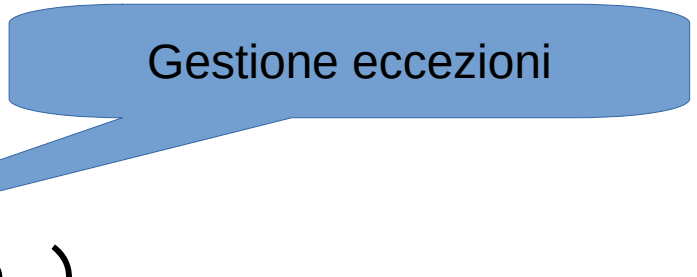
Alessandro Dentella
sandro.dentella@gmail.com

Controllo di flusso

- `if / elif /else`
- `while cond: pass`
- `for var in iterable: pass`
  `#ogni oggetto che implementi l'interfaccia 'iterable'`

  `(esistono però anche loop impliciti in c usando map, filter reduce)`

  `break/ continue`
- `try / except / finally ( raise )`
- `with open('/tmp/f') as f:  pass`



Alessandro Dentella
sandro.dentella@gmail.com

With & Context Manager

La sintassi appena vista con with per i file è un caso particolare di implementazione di un context manager, con lo statement with che ci permette di isolareusi standard di `try/finally`.

Un context manager fornisce i metodi `__enter__` e `__exit__` che vengono invocati entrando ed uscendo dal body dello statement `with`:

```
with open(filename) as f:
    for line in f:
        print(line)
```

Target: l'oggetto file ha metodi __enter__ e __exit__

L'esecuzione dello statement restituisce un context manager

Lo statement with garantisce che se __enter__ viene eseguito senza errori __exit__ verrà eseguito (anche eventualmente in presenza di errori)

Alessandro Dentella
sandro.dentella@gmail.com

Esercizi break/continue

- Trovare i primi 10 numeri primi.
- `write_multiples(step=2, max=10, output='/tmp/multiples.txt')` che scriva l'output in un file

python

Libreria std - sys

- `sys`: interazioni con il sistema
- `sys.path`: elenco delle cartelle dove verranno cercati i moduli, pacchetti
- `sys.argv`: una lista di argomenti passati alla script. Il primo elemento è l'interprete stesso
- `sys.stdout`: l'handler aperto dell'output
  `sys.stderr`: l'handler aperto dell'err
  `sys.stdin`: l'handler aperto dell'input
- `sys.version`: la versione corrente di Python

Alessandro Dentella
sandro.dentella@gmail.com

Libreria std - os

- `os`: fornisce un modo portabile di usufruire di funzionalità di sistema. Cambio cartella, interrogazione os, cambio attributi file...
- `os.path`: raggruppamento di funzione di manipolazione path

Creare una script che prenda in argomento un nome di cartella e crei la suddetta cartella con all'interno Un file **readme** leggibile a tutti ed un file **script.sh** Eseguibile solo dal proprietario con dentro un testo preso dalla docstring del comando stesso (es.:

```
#!/bin/bash
 echo "ciao mondo"
```

)

Alessandro Dentella
sandro.dentella@gmail.com

Libreria std - datetime

- Manipolazione date:
  - date
  - time
  - datetime
  - timedelta
  - Tzinfo: time information
- Fra i tanti metodi hanno `strftime` e `strptime` (*format* and *parse*)
- È usato da ogni driver di database che quindi ritornano oggetti date/datetime/timedelta

```
In [1]: from datetime import datetime
In [2]: datetime.now().strftime('%x %X')
Out[2]: '01/25/15 22:29:58'
In [3]:
In [4]: import os
In [5]: locale.setlocale(
            locale.LC_ALL, os.environ['LANG'])
Out[5]: 'it_IT.UTF-8'
In [6]: datetime.now().strftime('%x %X')
Out[6]: '25/01/2015 22:30:59'
In [7]: datetime.strptime("21/01/2015",
                                "%d/%m/%Y").date()
Out[7]: datetime.date(2015, 1, 21)
```

python

Alessandro Dentella
sandro.dentella@gmail.com

date

- Creare funzione che dato un anno di inizio ed uno di termine restituisca la lista di date 29/2 usando tentativi e gestione delle eccezioni.
- Verificare che algebra delle date è supportata:
  date + datetime
  date + timedelta
  datetime + timedelta
  …
- Stampare la data di oggi in diversi formati
- Fare una script che legga il file di log /var/log/syslog (1^ argomento) ed estrarre dal secondo e terzo campo la data.

python

Libreria std: re

- Implementa regexp stile perl sia per unicode che non
- Le operazioni con le regexp sono disponibili sia come funzioni di modulo (re.sub, re.match...) che come metodi dell' oggetto RegexpObject
- Alla sintassi perl viene aggiunta una sintassi specifica Python che aumenta la leggibilità di alcune regexp.
- Esiste un bel tutorial delle espressioni regolari per Python:

  https://docs.python.org/2/howto/regex.html

python

Regexp: il problema

Il problema che le regexp risolvono è il seguente: esprimere una moltitudine di possibilità in modo compatto e sintetico. Esempio:

Copiare tutte le immagini da una cartella data.

Supponiamo di basarci sulla estensione dei file, dovremo copiare tutto ciò che termina per .jpg, .png, .gif... ma anche .jpeg e le loro versioni maiuscole.

Esiste una versione di espressione regolare usata dalla shell (sh, bash, …) implementata dal modulo `glob` ed esiste la versione implementata dal modulo `re`

```
mv *.png *.jpg *jpeg *PNG *JPG *JPEG /dest_dir
```

```
re.search('(\.png|\.jpg|\.jpeg|\.gif)$', filename, re.IGNORECASE)
re.search('\.(png|jpe?g|gif)$', filename, re.i)
```

Alessandro Dentella
sandro.dentella@gmail.com

Regexp sintesi

- Molto usate anche nel router di django
- Permettono
  - Ricerche di stringhe (pattern) in un testo
  - Split rispetto ad un pattern
  - Modifiche di una stringa
- Esempio:
  - re.search('[A-Z][a-z]+', txt)
  - m = re.search('(?P<title>[A-Z][a-z]+)', txt)
    if m:
      print m.groupdict()

python

Regexp: basi

- . “matcha” qualunque carattere
- [a-z]: una lettera qualunque minuscola
- [aeiou]: le vocali
- ?: il match precedente può esserci o no
- + il match precedente può esserci 1 o + volte
- * il match precedente può esserci 0 o più volte
- gruppi di lettere: \d (digits) \s (space) \w (writable)
- () crea un gruppo che può essere referenziato dopo. Python aggiunge la possibilità di dare un nome

```
(?P<mail>[a-z._\d]+@example\.com)
```

python

Funzioni semplici

- `re.sub('[aeiou]', '', 'Monte Bianco', n_sost=None)`
- `re.match(pattern, string, opts)` `# ancorato all' inizio`
- `re.search(pattern, string, opts)`
- `re.findall(pattern, string)`
- `re.finditer(pattern, string)` `# return iterator`

Suggerimento: regexr.com, regex101.com

Alessandro Dentella
sandro.dentella@gmail.com

Problema della \

- Sia le espressioni regolari che python usano \. nelle espressioni regolari si usano per esprimere ad esempio spazi: \s.
- Volendo cercare le ricorrenze di '\section' dovremo passare al modulo re '\\section'. Essendo un literal però python interpreta la prima come protezione della seconda, costringendoci a scrivere: '\\\\section'
- Python permette di usare il modo raw per evitare troppe controbarre: r'\\section'. In questo modo python non aggiunge il livello di interpretazione delle literal.
- print(r'a\nb') stampa un solo a capo (il fine riga).

Alessandro Dentella
sandro.dentella@gmail.com

Oggetto match

- Il modo di avere in mano il match di una funzione è però di scrivere:

```
m = re.match(pattern, string)
```

- Questo genera un oggetto match che ha a sua volta i metodi:
  - `.groups()`: tutti i match
  - `.group(i)`: match numero i
  - `.group('name')`: match per nome
  - `.groupdict()`: dizionario con i gruppi

Alessandro Dentella
sandro.dentella@gmail.com

(?P<name>pattern)

- La sintassi python per dare il nome ad un gruppo è scritta sopra. Risulta molto efficace perchè il nome è indipendente dalla posizione e sicuramente più espressivo.
- Ci piacerebbe scrivere:

  print(“in {data} hai inviato mail a {dest}”.format(m))

  ma l'oggetto m non ha un *mapping*... come lo aggiungiamo? Cosa è un mapping?

Alessandro Dentella
sandro.dentella@gmail.com

compile

- Quando una opzione viene ripetuta più volte, si può valutare di compilare separatamente l' oggetto:

```
PAT = re.compile('[abcde]*', re.UNICODE)
```

- Ed usarlo in seguito direttamente con la stringa:

```
m = PAT.search(testo)
```

python

Alessandro Dentella
sandro.dentella@gmail.com

Opzioni compilazione

- Le opzioni di compilazioni sono
  - re.I, re.IGNORECASE
  - re.L, re.LOCALE # Make \w, \W, \b, \B, \s and \S # dependent on the current locale.
  - re.M, re.MULTILINE # ^ e $ matchano per ogni linea
  - re.S, re.DOTALL # . match newline as well
  - re.X, re.VERBOSE # sintassi che permette commenti
  - re.U, re.UNICODE
- Le opzioni si combinano con |:

```
re.VERBOSE | re.DOTALL
```

Alessandro Dentella
sandro.dentella@gmail.com

GREP

- Creare una script che agisca come **grep:** accetta un pattern in ingresso a uno o più nomi di file e stampa a video tutte le righe che contengono quel pattern
- Creare una funzione che stampi a video tutti i file nel path di sistema che terminano per .pth ed il loro contenuto

python

```
#!/usr/bin/python
# coding: utf-8
"""
Script che si comporta come grep

Ricorda di eseguire chmod +x pygrep
"""
import sys

def grep(pattern, filename):
    """Return all lines in filename that match pattern

    :arg pattern: the pattern we want to find
    :arg filename: the filename in which we search
    """
    #import ipdb; ipdb.set_trace()
    with open(filename) as f:
        for line in f:
            if re.search(pattern, line):
                print(line)

if __name__ == '__main__':
    pattern = sys.argv[1]
    filenames = sys.argv[2:]
    for filename in filenames:
        grep(pattern, filename)
```

Questa riga informa la shell di chiamare l'interprete Python per eseguire questa script

Suppone che questa script sia chiamata 'pygrep'

Per debug di una script che non sia invocata da ipython possiamo usare (i)pdb

Alessandro Dentella
sandro.dentella@gmail.com

optionparse

```
"""
In questo esempio, la docstring di help è essa stessa la fonte
per la definizione delle opzioni. Non possono proprio essere
disallineate!!!

usage: %prog [options]
   -p, --port=PORT: the port to make opeoffice to listen to
   -s, --server=SERVER: the server name or ip of the machine
   -v, --verbose: be verbose
"""
import optionparse

opts, args = optionparse.parse(__doc__)

if opts.verbose:
    print opts
```

Alessandro Dentella
sandro.dentella@gmail.com

Implementazione optionparse

- Più tardi, nella parte sul trattamento testi analizziamo optionparse
- Faremo una digressione per vedere
  - Come lavora optionparse, che è basato sulle regexp
  - Come usare ipdb, il debugger

python

Alessandro Dentella
sandro.dentella@gmail.com

Iteratori

- Le sequence (stringhe, tuple, liste...) sono oggetti che implementano il protocollo di iterazione che può essere implementato da ogni oggetto ed è basato su due semplici elementi:
  - Un iteratore (`__iter__`)
  - Un metodo `.__next__()` *sull'iteratore*
- Il comando `iter` restituisce un iteratore da una sequence
- Il ciclo `for` agisce prima chiedendo un iteratore e poi chiamando `next()` fino ad esaurirlo

```
In [33]: iter([0, 1])
Out[33]: <listiterator object at
0xb76751cc>

In [34]: i=iter([0, 1])

In [35]: next(i)
Out[35]: 0

In [36]: next(i)
Out[36]: 1

In [37]: next(i)

StopIteration Traceback (most recent call
last)
```

Alessandro Dentella
sandro.dentella@gmail.com

Iteratori: file

Gli iteratori permettono di mantenere una sintassi essenziale in molte circostanze:

```
with open('/etc/fstab') as f:
    for line in f:
        print(line)
```

Questa forma risulta la più veloce per accedere ad un file una riga alla volta

python

Alessandro Dentella
sandro.dentella@gmail.com

Esempio iteratore

start_loop.py

```
import time
with open('/tmp/tubo') as f:
    for line in f:
        time.sleep(1)
        print(line)
```

(dopo avere lanciato start_loop.py)

```
$ for i in {6..7}; do
echo $i >> /tmp/tubo; done
```

```
$ for i in {1..5} ; do
echo $i >> /tmp/tubo; done
python start_loop.py
1
2
...
6
7
...
```

Se scriviamo prima che start_loop abbia terminato ma dopo che sia stato lanciato, possiamo vedere l'effetto del che il generatore agisce solo nel momento della richiesta next()

python

Alessandro Dentella
sandro.dentella@gmail.com

Generatori

- Funzioni che contengono yield
- **Ritornano iteratori**. Chiamando `iteratore.next()` causa una esecuzione della funzione fino allo statement yield.
- Un return o il raggiungimento di `StopIteration` fanno si che la procedura finisca
- La funzione, fra una chiamata e l'altra **ricorda lo stato in cui era per cui non serve tenere traccia del punto dove era o del valore.** (es. 50/51)

```
def gen():
    for i in range(20):
        if not i % 3:
            yield (i, i**2)
```

Il generatore ricorda lo stato interno (il valore di i in questo caso) nella successiva chiamata

```
>>> for i in gen():
...     print(i)
(0, 0)
(3, 9)
(6, 36)
(9, 81)
(12, 144)
(15, 225)
(18, 324)
```

Alessandro Dentella
sandro.dentella@gmail.com

generator expression

- permettono di creare un generatore in modo molto compatto, esattamente come la list comprehension:

```
gen_dispari = (i for i in range(10) if x % 2 )

for x in gen_dispari:
    print(x)
```

- Nota che qui non compare yield!!!

python

Alessandro Dentella
sandro.dentella@gmail.com

- Creare un generatore con la funzione dell'esempio precedente e chiamare esplicitamente il metodo `.__next__()` fino ad esaurirlo. Ora provate a eseguire il codice del ciclo for: che succede?
- Usare la stessa funzione usando `return`, cosa cambia?
- Creare un generatore che -partendo da un dizionario di mesi/gg (gg_mesi_dict)- restituisca solo i mesi di 31 gg e creare una lista usando la list comprehension che cicli su questo generatore

Decoratori

- Una funzione che ritorna un'altra funzione
- La sintassi del decoratore '@' è chiamata 'Syntactic sugar' in quanto semanticamente equivalente a:

```
def func(...):
    ...

f1 = function(func)

@function
def func()
```


```
from decorator import decorator

@decorator
def login_required(func, request, *args, **kwargs):
    """Un decoratore che controlla che l'utente
    collegato sia autenticato e redirige a /login
    in caso contrario
    """

    if not request.user.is_authenticated():
        return Redirect('/login')
    return func(request, *args, **kwargs)


@login_required
def my_page(request):

    return ...
```

@decorator

- Nell'esempio precedente abiamo fatto uso del pacchetto esterno `decorator` per semplicità di implementazione (e27b)
- Il modulo decorator è usato da ipython ed altre librerie scientifiche quindi lo troviamo già nell'environment.
- Alternativamente avremmo dovuto creare una funziona wrapper interna (closure)

Alessandro Dentella
sandro.dentella@gmail.com

- creare un modulo chiamato 'my_decorators' in cui mettere un decoratore per fare il log delle chiamate
- Aggiungere questo decoratore alla funzione che trova i numeri primi
- Da ipython chiamare la funzione che genera numeri primi per vedere che effettivamente logga le chiamate

Lambda functions

- Python supporta la creazione di funzioni anonime che ammettono una espressione e non richiedono alcun return (il valore è sempre ritornato)
- Spesso usate in contesti come:
  - `map`
  - `filter:`

```
dispari = filter(lambda x: x % 2, range(11))
```

  `sorted`

```
sorted(student_tuples, key=lambda student: student[2])
```

- Idonea per funzioni di una sola righa, permette una sintassi molto compatta.

Alessandro Dentella
sandro.dentella@gmail.com

Garbage collector

- Quando una variabile non è più vista da alcun punto viene cancellata dal garbage collector
- Una variabile locale ad una funzione viene eliminata quando la funzione ritorna
- Esiste un modulo che permette di ispezionare il garbage collector: gc
- Esiste un modo di referenziare un oggetto in modo debole, ovvero tale che non sia suff. a tenere in memoria l' oggetto, si crea con il modulo weakref

python

Alessandro Dentella
sandro.dentella@gmail.com

Glue code: ovvero alternative alla shell

- Python offre funzioni che permettono di eseguire programmi esterni, possono quindi essere usati come collante per altri programmi
- In modo analogo è possibile pensare che all' interno di un programma Python sia necessario chiamare un comando esterno o avviare un processo ad es.:
  - Da una interfaccia web, riavviare un servizio
  - Da una applicazione avviare un browser web
  - Da una script interagire con comandi di sistema

python

Alessandro Dentella
sandro.dentella@gmail.com

scenari

- Il programma da fare partire è breve, attendiamo il risultato ed il codice di ritorno
- Il programma è lungo, vogliamo solo avviarlo (es.: un servizio)
- Il programma è lungo e vogliamo:
  - Controllare input, output, error
  - Codice di ritorno, pid
  - Interagire (ad esempio spegnerlo)
- Nel tempo si sono usati vari programmi che sono attualmente nella libreria std

Alessandro Dentella
sandro.dentella@gmail.com

scelte

- I molti programmi che troviamo si distinguono tutti per la quantità di cose che ci permettono di controllare.
- Il modulo **`subprocess`**, presente dalla versione 2.4 racchiude tutte le possibilità degli altri moduli/funzioni che ancora sono presenti solo per compatibilità con il passato, ovvero
  - os.system
  - os.popen*
  - os.spawn*
  - popen2.* (py2)
  - commands.* (py2)

Alessandro Dentella
sandro.dentella@gmail.com

subprocess

```
class subprocess.Popen(
    args,                # What to run and how. Can be a string
    bufsize=0,           # Subprocess files bufsize opt (as open)
    executable=None,     # if None first arg is used
    stdin=None,          # an open file or PIPE
    stdout=None,
    stderr=None,
    preexec_fn=None,     # a callable
    close_fds=False,
    shell=False,  # If True look executable and defult sh or cmd
    cwd=None,             # a directory to change to before
    env=None,             # a mapping {'PATH' : '/sbin'}
    universal_newlines=False,
    startupinfo=None,     # windows specific
    creationflags=0       # windows specific
)
```

Alessandro Dentella
sandro.dentella@gmail.com

Attributi di subprocess.Popen

- Una istanza di subproces ha questi attibuti:
  - pid: il pid del processo
  - returncode:
    - None se il processo non è terminato
    - 0 se è terminato correttamente
    - > 0 se è terminato con un errore
    - < 0 se è terminato per un segnale
  - stdin,stderr, stdout:
    - None se il rispettivo argomento era None
    - File like object se era PIPE

Alessandro Dentella
sandro.dentella@gmail.com

Metodi di Popen

- communicate(input=None) => output, error
- poll
- wait
- send_signal (dal 2.6)
- terminate (dal 2.6)
- kill (dal 2.6)

python

Alessandro Dentella
sandro.dentella@gmail.com

Esempi

- Cominciamo con la serie 30 per gli esempi su subprocess.
- Il comando può essere una stringa se:
  - Lo interpreta la shell o non ha opzioni
  - Nel caso abbia opzioni, non deve avere opzioni con spazi
- Per catturare stdin e stdout occorre chiamare la classe Popen con subprocess.PIPE, altrimenti vanno direttamente nella sys.stdin e sys.stdout di sistema

python

Alessandro Dentella
sandro.dentella@gmail.com

Pipeline nel comando

- È possibile avere una pipe nel comando
- È più semplice farlo con la shell es. 31
- Per farlo in modo indipendente dalla piattaforma occorrono 2 processi (es. 32) . in questo caso l' output del primo processo è messo come input del secondo:

```
p1 = Popen(['cat', '/etc/fstab'], stdout=PIPE)

p2 = Popen(['grep', 'sda'], stdin=p1.stdout, stdout=PIPE)

output, error = p2.communicate()
```

python

Alessandro Dentella
sandro.dentella@gmail.com

Alimentando con input

L' esempio 33 mostra come alimentare l' input:

```
p = Popen("tr -d aeiou", stdout=PIPE, stdin=PIPE,  shell=True)
output, error = p.communicate('Monte Bianco')
print(outout)
```

```
Mnt Bnc
```

Alessandro Dentella
sandro.dentella@gmail.com

Fare partire un servizio

- L' esempio 34 mostra come fare partire un servizio, -nella fattispece il server di openoffice-
- Questo esempio introduce il parsing delle opzioni fatto in modo estremamente “agile” attraverso un modulo non di libreria chiamato 'optionparse' che si basa sul modulo di libreria 'optparse'.

python

Alessandro Dentella
sandro.dentella@gmail.com

Classi

Alessandro Dentella
sandro.dentella@gmail.com

OOP

- La programmazione OOP (Object Orented) è uno dei principali paradigmi di programmazione (altri sono imperativa, funzionale, logica...)
- Si basa su oggetti, ovvero strutture di dati -attributi- e codice -metodi- (... che poi sono sempre attributi dell'oggetto!)
- Noi vogliamo concentrarci su come Python implementa le classi, non su come ragionare per classi... ma vediamo di acclimatarci con il concetto delle classi.
- Di fatto anche se la programmazione imperativa permette gli stessi risultati, la programmazione ad oggetti risulta spesso più chiara e intuitiva

Alessandro Dentella
sandro.dentella@gmail.com

Classi vs. func

- Vediamo la stessa operazione eseguita utilizzando una funzione esterna o un metodo dell'oggetto..

```
from my_utils import lower

lower('Mario Rossi')
```

```
'Mario Rossi'.lower()
```

- Costruttori

```
user = create_user(username='Mario')
user = User(username='Mario')
```

- Python permette che i due paradigmi siano usati contemporaneamente
- La struttura dati stringa (che è essa stessa un oggetto) o User hanno in sé -perché definite all'origine, nella creazione della classe- le informazioni sulla manipolazione dei dati che viaggeranno sempre con l'oggetto

Alessandro Dentella
sandro.dentella@gmail.com

- È possibile ottenere lo stesso effetto con una funzione ma bisognerebbe portarsi dietro l'import di tutte le funzioni sempre
- Grazie alla possibilità di introspezione, in sessioni interattive ho una visibilità delle operazioni che posso fare sull'oggetto (Tab-completion)

Alessandro Dentella
sandro.dentella@gmail.com

Classi in Python

- Le classi in python sono ottenuto con un minimo di sintassi (`class`)

```
class MyClass(base_class,[base_class],...):
      Statement1
      Statement2
      …
      StatementN
```

- L'ereditarietà permette anche di avere molte *base classes*
- I tipi base possono essere usati come classi da estendere

Alessandro Dentella
sandro.dentella@gmail.com

Definizione della classe

- Non pongono barriere forti fra il programmatore e la struttura interna, lasciando alla buona educazione il compito di non guastare il meccanismo (vedi oltre)
- Nel linguaggio usato da c++, tutti i dati sono *pubblici* ed i metodi *virtuali*
- *Lo guide-style di Python raccomanda di usare* la lettera maiuscola per le classi.
- Esiste anche la possibilità di definire una classe *in modo dinamico*, ovvero passando il nome tramite variabile ed usando il comando `type`

python

Alessandro Dentella
sandro.dentella@gmail.com

esempio

```
class Prova:

    name = 'abc'
    children = []

    def get_name(self):
        return self.name
```

Base class(es). In Python3 non è necessario ereditare da *object*

Ogni metodo viene definito con l' oggetto stesso come primo argomento, aggiunto in modo implicito.
È ***consuetudine*** chiamarlo self

Chiamando una classe si crea una ***istanza*** della classe

`self.name`: la notazione puntata indica che si vuole l'attributo `name` del'oggetto `self`. L'attributo può essere sia privato dell'oggetto che della classe

```
>>> p = Prova()
>>> p.name
'abc'
>>> p.get_name()
'abc'
```

Attenzione!

- Come posso da un metodo referenziare un attributo della classe?
- Che differenza c'è fra *name* e *children*?

Alessandro Dentella
sandro.dentella@gmail.com

Introspezione della classe

- `hasattr` testa se un oggetto ha definito un attributo passato in argomento come stringa
- `p` ha definito l'attributo `name`
- Il metodo `get_name` è un attributo esso stesso
- Interattivamente p.<Tab> completa con gli attributi definiti esplicitamente
- `p` ha anche l'attributo `__init__` che non abbiamo definito. Da dove viene?

```
>>> p = Prova()
>>> p.name
'abc'
>>> hasattr(p, 'name')
True
>>> hasattr(p, 'get_name')
True
>>> p.
p.children  p.get_name  p.name
>>> hasattr(p, '__init__')
True
```

python

Alessandro Dentella
sandro.dentella@gmail.com

- Creare una classe Tour, con un attributo cities
- Creare 2 istanze differenti di questa classe ed impostare due percorsi differenti

```
t_ve = Tour()
t_ve.cities += ['Venezia', 'Vicenza']
t_roma = Tour()
t_roma.cities += ['Roma', 'Orvieto']
t_roma.cities … cosa leggete?
```

- Verificare quanto scritto: vi torna?

python

Spiegazione

- In modo analogo a quanto succede per la visibilità e e la modifica delle liste all'interno di una funzione, una variabile di classe viene vista (e modificata!!!) da tutte le istanze. Solo se riassegnata (rebind) “diventa privata”:

```
tour_ve.cities.append('Venezia') # Tutte le istanze la
                                 # vedono
tour_ve.cities = ['Venezia'] # la vede solo il tour 'tour'
```

- Se si vuole una lista di proprietà dell'istanza va creata nell'__init__

Alessandro Dentella
sandro.dentella@gmail.com

Classes: __init__ & __str__

```
class Tour:
    """
    Giro turistico per le città. Un Tour ha

      * una lunghezza in giorni
      * una guida
      * una lista di città passata come
        dizionario ordinato {citta: gg_visita}
      """
    guida = None
    days = 0


    def __init__(self, name, cities=None, days=None):
        self.name = name
        self.days = days or 0
        self.cities = cities or {}

    def __str__(self):
        return "<Tour: {}>".format(self.cities.keys())
```

Alessandro Dentella
sandro.dentella@gmail.com

```
>>> from collections import OrderedDict
>>> cities = OrderedDict([
        ('Roma', 3),
        ('Orvieto', 1),
    ])
>>> cities['Perugia'] =  2
>>> tour_roma = Tour('Roma e Umbria', cities)
>>> tour_roma.guida = 'Mario Rossi'
>>> print(tour_roma)
    "<Tour: ['Roma', 'Orvieto', 'Perugia']>"
```

possiamo accedere agli attributi ed anche impostarli direttamente

Esercizio:
Data questa classe, creare un metodo per la validazione del dato "cities", che verifichi che sia un dizionario e che i valori siano dei numeri. Chiamarla all'interno dell'__init__

Alessandro Dentella
sandro.dentella@gmail.com

Metodi speciali

- Alcuni metodi hanno nomi “speciali”, che cominciano e terminano con “__”. Hanno funzioni particolari ed in generale implementano funzioni/protocolli determinati. Nessun metodo è necessario
- __init__: serve quando si voglia cominciare in uno stato noto
- __str__(/__unicode__): serve a dare una rappresentazione a stringa/unicode dell'oggetto (Unicode solo pe py2)
- __len__: implementa il concetto di lunghezza
- __iter__: restituisce un iteratore per l'oggetto se si vuole si comporti come una sequence
- __contains__: implementa in costrutto “key in obj”
- __repr__, __add__, __mul__....

Alessandro Dentella
sandro.dentella@gmail.com

- Aggiungere un metodo `__len__` in modo che

```
len(my_tour)
```

  restituisca il numero di giorni di durata del tour
- Modificare la `__str__` in modo che compaia anche la guida o la scritta `"Non Definita"`
- Aggiungere all'interno di `__init__` un attributo `self.days` con il valore calcolato dei giorni
- Scrivere un metodo `__contains__` in modo da potere usare questo Test:

```
'Roma' in my_tour
```

str/repr

- Da ipython istanziamo un tour stampiamo a video
- Ci piace come rappresentazione del tour?

  ['pisa', 'lucca']
- Possiamo migliorarla? Cosa dobbiamo cambiare?
  Implementiamo __str__ in modo che `print(tour)` sia soddisfacente.
- Ora eseguiamo:

```
>>> print([umbria])

  [<tour.Tour object at 0xb753dccc>]
```

Python usa una una rappresentazione differente dell'oggetto quella generata da __repr__, proviamo a migliorarla

Alessandro Dentella
sandro.dentella@gmail.com

repr

```
def __repr__(self):
    return "<Tour {}: {} ({})>".format(self.name, "/".join(self.cities)), self.days)
```

```
In [1]: umbria
Out[1]: <Tour Umbria: perugia/assisi (0)>
```

Che chiaramente chiede di essere migliorato per quanto riguarda il numero di gg.

Esercizio:

- Creiamo un dizionario di gg di visita per città
- **Aggiungiamo una funzione che conti il numero di gg**
- **Usiamo questa in __repr__**

python

Alessandro Dentella
sandro.dentella@gmail.com

getattr / setattr / delattr

In molte circostanze dobbiamo cambiare un attributo il cui nome è contenuto in una variabile, non è quindi possibile usare la *dot.notation*:

```
instance.attr_name
```

Useremo:

```
getattr(instance, attr_name, [default value])
setattr(instance, attr_name, value)
delattr(instance, attr_name)
```

python

Alessandro Dentella
sandro.dentella@gmail.com

- Creare una classe `Anno` che richieda l'anno come unico argomento, e popolare una istanza di `Anno` di attributi con i nomi dei mesi

```
anno = Anno(2009)
anno.gennaio = 31
anno.febbraio = 28
```

- Nell'`__init__` mettere un controllo se l'anno è bisestile usando `datetime.date` e valorizzando un attributo 'bisestile'
- Nel metodo `__str__` aggiungete un asterisco se è bisestile
- Aggiungere un metodo in modo da potere scrivere:

```
for mese in anno:
    print(getattr(anno, mese))
```

- creare anche un ciclo che che estrae i mesi con 31 giorni da questa istanza (NON da mesi_gg *;-)

__iter__ e __next__

Il protocollo di iterazione già visto precedentemente in azione, a livello di oggetti è implementato da due metodi

- `__iter__`: restitusce un iteratore
- `__next__`: restituisce il prossimo elemento

Di fatto il metodo `__next__` **deve essere implementato sull'iteratore** (restituito da `__iter__`). In alcune circostanze, quando l'oggetto stesso ha un metodo `__next__, __iter__` ritorna sé stesso. In rete ci sono molte implementazioni così con un attributo `i` che fa da counter ma è molto meglio usare un generatore per implementare `__iter__`

Esercizio:
- rendiamo la classe tour iterabile sulle città che tocca
- perché è meglio un generatore per iter?

Alessandro Dentella
sandro.dentella@gmail.com

Metodi speciali 2

- `__getattr__`: se presente permette di implementare un modo differente di accedere agli attributi di classe. È ad esempio possibile implementare un dizionario a cui si possa accedere agli item come se fossero attributi. `__getitem__` viene invocato ogni volta che si cerca di accedere ad un attributo che non esiste. Non va confuso con `__getattribute__` che viene chiamato sempre
- `__getitem__`: analogo al precedente per gli item
- `__delattr__:` analogo per cancellazione attributi
- `https://docs.python.org/3/reference/datamodel.html#special-method-names`

Non confondere questi con gli identificativi che iniziano con almeno __ e terminano con non più di un _ (es.: __spam). Questi vengono sostituiti con _classname__spam ed è un modo di fornire attributi/metodi privati della classe

entella
ail.com

__getattr__

- Cosa succede quando cerchiamo di accedere ad un attributo della classe (usando quindi `obj.attr`)?
- Se l'attributo è definito nell'oggetto (`obj.__dict__`) viene restituito quel valore (eventualmente trovato in una classe fra quelle dichiarate in `__bases__`) altrimenti viene invocato un metodo speciale `__getattr__`. Che a sua volta deve restituire AttributeError in caso non esista. Questo permette di esporre con questa sintassi qualunque oggetto.
- In modo analogo il metodo `__setattr__` implementa l'assegnazione

Alessandro Dentella
sandro.dentella@gmail.com

__getitem__

- Analogamente a quanto visto per gli attributi, nel caso dei dizionari, gli item vengono cercati tramite una chiave dal metodo `__getitem__`
- Analogamente esiste `__setitem__` per impostare il valore

python

Alessandro Dentella
sandro.dentella@gmail.com

days & get_days

- Aggiungiamo una città assegnandola direttamente a `self.cities`
- Abbiamo creato una situazione in cui `self.days` e `self.get_days()` ritornano concettualmente la stessa cosa ma di fatto una ritorna una quantità immagazzinata e l'altra il frutto di un calcolo
- È facile immaginare casi in cui il numero di gg. dipenda anche da altri fattori per cui possa variare e non abbia molto senso archiviarlo corretto ogni volta

python

Alessandro Dentella
sandro.dentella@gmail.com

Property

- In questi casi si usa una property ovvero un attributo speciale per cui viene definito
  - Un getter (una funzione che viene valutata ed il cui valore viene restituito)
  - Un setter (eventuale)
  - Un deleter (eventuale)
- `days = property(get_days)`
- Questa tecnica è molto usata soprattuto quando si vuole permettere un accesso più intuitivo ai dati e quando si vgliono aggiungere controlli ad un dato ma non si vuole cambiare la sintassi.

Alessandro Dentella
sandro.dentella@gmail.com

Property con setter

In questo caso non esiste una variabile reale che immagazzini il numero di gg. Nel caso invece esista è prassi abbastanza diffusa che abbia nome che inizia con '_': `self._days`. Una possibile implementazione sarebbe:

days **non** _days

```
def _get_days(self):
    return self._days or sum(self.cities.values())

def _set_days(self, i):
    if i >= self.days:
        self._days = i
    else:
        raise Exception("Viaggio troppo corto")
days = property(_get_days, _set_days)
```

_set_days, _get_days **senza** self. Perché?

Alessandro Dentella
sandro.dentella@gmail.com

Property con decoratori

- Esiste un modo alternativo di creare una property, usando i decoratori
- L'attributo `leader` dell'istanza p è una stringa
- L'attributo `leader` della classe Tour è una *property*
- Si può anche impostare *@leader.deleter*

```
@property
def leader(self):
    return getattr(self, '_leader', None)

@leader.setter
def leader(self, name):
    self._leader = name
```

```
>>> t = Tour('Milano')
>>> t.leader = 'Sandro'
>>> t.leader
'Sandro'
>>> getattr(t, 'leader')
'Sandro'
>>> getattr(Tour, 'leader')
<property object at 0x7f4c3cc158e8>
```

python

Alessandro Dentella
sandro.dentella@gmail.com

Composizione

- L'oggetto Tour ha un attributo che è un oggetto dict collegato all'etichetta *cities*:

```
>>> tour_roma.__dict__
{ 'start_city': 'Milano',
  'info': OrderedDict(
                [('Roma', 3), ('Orvieto', 1), ('Perugia', 2)]),
  'name': 'Roma e Umbria'}
```

- Questa è un pattern molto comune che si chiama ***composizione***. In molti casi è preferibile all'ereditarietà (vedi oltre).
- Esempi:
  user.organization.addresses[0]
  book.chapters.chapter1.title
  book.chapters[1].title

Alessandro Dentella
sandro.dentella@gmail.com

- Creare
  - un dizionario con key: città e valore: una lista di monumenti della città
  - una classe City, con un attributo `main_monuments`, che nell'init viene impostata prendendo dal dizionario del punto precedente. Creare una istanza per ogni città visitata dal tour
  - creare un attributo `visited_cities` di Tour che contenga tutte le istanze `City` del tour
- Creare un metodo `display` che elenchi le città visitate, una per riga con indentazione variabile (parametro passato quando invoco il metodo)
- Creare un metodo `get_program` che stampi a video una mini presentazione del Tour partendo da un template che create voi usando opportunamente la formattazione delle stringhe

Ereditarietà

- Se analizziamo gli attributi di un oggetto con il comando dir(t) ci accorgiamo che sono definiti molti attributi che non abbiamo definito direttamente
- Sono gli attributi definiti per la classe da cui abbiamo fatto derivare Tour (per noi quindi *object*)

```
>>> dir(t)
['__class__',
 '__contains__',
 '__delattr__',
 '__dict__',
 '__doc__',
 '__format__',
 '__getattribute__',
 '__hash__',
 '__init__',
 '__iter__',
 '__len__',
 '__module__',
 '__new__',
...
 '_leader',
 'add_info',
 'cities',
 'days',
 'display',
 'get_days',
 'get_program',
 'leader',
 'name',
 'start_city']
```

python

Alessandro Dentella
sandro.dentella@gmail.com

Ereditarietà

Python nella dichiarazione della classe permette di indicare una o più classi da cui la classe che stiamo definendo ***eredita,*** ovvero classi i cui attributi verranno usati qualora non siano ridefiniti. Questa tecnica permette di evitare ripetizioni nel caso ci servano classi simili ad altre ma con varianti in alcune parti.

Alessandro Dentella
sandro.dentella@gmail.com

Esempio

Finora non abbiamo tenuto conto del tempo del viaggio. Qualora ne volessimo tenere conto dovremmo tenere anche in conto del mezzo di trasporto. Potremmo quindi alternativamente:

Arricchire il metodo `get_days`

- Usare l' ereditarietà e fare metodi differenti per tipi di viaggio differenti

```
class BusTour(Tour):
    def get_days(self):
        return Tour.get_days(self) + (len(self.cicities)-1)/2.
    days = property(get_days)
```

Base class

Chiamo il metodo della base class, alternativamente super().get_days()

È necessario ripetere la property

½ gg di spostamento per cambio citta

sandro.dentella@gmail.com

- Modificare il metodo `add_info` in modo che aggiunga per ogni città contenuta in `self.cities` un attributo `self.<nome_citta>_days`.

  Alla fine dobbiamo vedere

```
>>> t.roma_days
3
>>> t.orvieto_days
1
```

- Mettete l'attributo tutto minuscolo

Creiamo un dizionario che implementi anche una interfaccia con notazione puntata

```
>>> m = MyDict()
>>> m['uno'] = 1
>>> m.uno
1
>>> hasattr(m, 'uno')
True
```

```
class MyDict(dict):
    def __getattr__(self, key):
        return self[key]
        # return self.__getitem__(key)
```

Ereditarietà e sostituzione __getattr__

- Tour.cities è una lista di città (per ora nomi di città). In quanto lista non possiamo scrivere:

  ```
  umbria_tour.cities.assisi
  ```

  ma:
  ```
  umbria_tour.cities[1]
  ```

- Implementiamo una lista che implementa un `__getattr__` che cerca la città, ne trova l'indice e poi estrae l' elemento corrispondente

Alessandro Dentella
sandro.dentella@gmail.com

```
class Cities(list):
    """
    Contenitore di città cui si può accedere anche in dotted
    notation (container.city)

      >>> cities = Cities([City('roma'), City('pisa')])
      >>> cities.roma
      'Roma'
    """

    def __getattr__(self, key):
        try:
            idx = 0
            for c in self:
                if c.name.lower() == key.lower():
                    return self[idx]
                idx += 1
        except ValueError as e:
            raise AttributeError(
                "Attribute '%s' does not exists",format(key))
```

Alessandro Dentella
sandro.dentella@gmail.com

Ereditarietà multipla

- Una classe può ereditare da più classi:

```
class NewClass(Base1, Base2): pass
```

  i metodi saranno cercati secondo un algoritmo ben definito prima nella classe Base1 e nei suoi antenati e poi in Base2 e nei suoi antenati.

Questo algoritmo cerca in una sequenza che è scritta nell'attributo `__mro__` (Method Resolution Order)

Alessandro Dentella
sandro.dentella@gmail.com

mixin

- L'ereditarità multipla permette di scrivere classi che uniscono “nature” differenti. Risulta quindi possibile segregare alcune funzionalità in una classe che viene poi importata da più classi differenti evitando la duplicazione del codice.
- L'utilizzo eccessivo dei mixin (ed in genere quindi della ereditarietà multipla) viene considerato negativo per il fatto che rischia di nascondere in modo silenzioso alcuni metodi se non ci si ricorda con precisione cosa ci sia
- Django fa uso esteso dei mixin, ad esempio nelle viste a classi.

python

Alessandro Dentella
sandro.dentella@gmail.com

- Creare un mixin -PrintMixin- che si occupi della stampa (il metodo attuali `display` e `get_program`). Questo Mixin lo farete **senza** `__init__`
- Ristrutturare la classe usando quei mixin
- Creare `BusTour` che eredita da `Tour` e da `Print`
- Modificate `__init__` in modo da usare gli operatori di (un)packing (* e **)

python

Isinstance / issubclass

L'ereditarietà rende poco utile un codice che si usi `type()` per decidere se eseguire una certa operazione:

```
if type(obj) == dict:
    pass
```

Mi basta che erediti da dict (in generale) quindi scriveremo:

```
if isinstance(obj, dict):
    pass
if isinstance(obj, (list, set, tuple)):
    pass
```

Alessandro Dentella
sandro.dentella@gmail.com

- Creare class `Person`
- Creare una metodo `add_person` che aggiunga al tour (attribute `persons`) una persona passata in argomento come stringa o come oggetto `Person`, e nel primo caso istanzi un oggetto `Person`

Alessandro Dentella
sandro.dentella@gmail.com

classmethod

- Esistono dei metodi che non operano su una istanza, ma da un punto di vista logico appartengono alla classe: li chiameremo classmethod e vengono dichiarati tramite un decoratore:

```
class Test:
        @classmethod
        def objects(cls):
                …
```

- Come si nota al posto dell'istanza, Python chiama questi metodi passando in argomento la classe

Alessandro Dentella
sandro.dentella@gmail.com

staticmethod

- Esistono metodi che non agiscono neanche sulla classe ma sono logicamente legati alla classe. Vengono invocati **senza** alcun parametro aggiuntivo
- Sono poco usati e qualche teorico dice che non vanno usati… ma permettono di raccogliere funzioni strumentali ad altri metodi, nello stesso contesto. Sarà poi possibile usarli senza doverli importare separatamente

python

Alessandro Dentella
sandro.dentella@gmail.com

TDD

Test driven development

Alessandro Dentella
sandro.dentella@gmail.com

Il problema

- Scrivere codice non è una operazione facile, normalmente inseriamo nel codice errori di vario tipo:
  - Sintassi. Facili da riconoscere e risolvere.
  - Logici. A volte anche molto difficili da comprendere
- Un errore può presentarsi a runtime a seconda del tipo di dato che in quel momento ha una data variabile.
- Molte volte le nostre funzioni possono avere molte casistiche che ci son ben presenti quando scriviamo per la prima volta il codice e che ci siamo dimenticati in una successiva revisione, o si basano su assunzioni non condivise
- Molti errori arrivano quando facciamo refactoring di una funzione e dimentichiamo di contemplare tutte le casistiche o cambiamo l'interfaccia

Alessandro Dentella
sandro.dentella@gmail.com

Refactoring

Il refactoring è la ristrutturazione del codice che si rende spesso necessaria quando ci accorgiamo che la prima impostazione data non era la più opportuna

- Il codice è cresciuto
- Le esigenze sono cambiate
- Il codice non può affrontare le casistiche di interesse
- Abbiamo scoperto una nuova libreria che funziona meglio
- Ogni IDE, anche pycharm/visual studio hanno strumenti appositi per il refactoring e per i test!

Alessandro Dentella
sandro.dentella@gmail.com

workflow

Il normale workflow è così:

- Si ricevono le specifiche
- Si concorda una interfaccia di libreria (api)
- Si implementano
- Si verificano, magari su più interpreti/OS/browser/db...

L'anello che vogliamo aggiungere ora è la produzione di codice che fa solo la sentinella sul buon funzionamento * **anche nel futuro ed in modo automatico**

python

Alessandro Dentella
sandro.dentella@gmail.com

test

- Questo codice prende il nome di test e a seconda della tipologia di funzionalità che verifica potrà essere
  - **Unit test**: testa il comportamento di una porzione limitata del codice
  - **Functional test**: testa un comportamento
  - **Integration test**: testa svariate componenti contemporaneamente e nella relazione reciproca

python

Alessandro Dentella
sandro.dentella@gmail.com

unit vs. integration

- Se un integration test fallisce rimaniamo con l'onere di capire quale componente si è rotto. Per analogia, se la luce dell'automobile non funziona, sarà la lampadina, la batteria, l'alternatore, un cavo staccato…?
- Usando unit test abbiamo immediatamente la risposta di quale sia il componente che non funziona come previsto

python

Alessandro Dentella
sandro.dentella@gmail.com

Metodologia per unit test

La logica è molto semplice: si scrive il minimo codice che utilizza una certa funzione, metodo, componente con dati di ingresso noti e si confronta con il risultato atteso:

```
assert sum([1, 3, 5]) == 9, "Doveva fare 9!!!"
```

python

Alessandro Dentella
sandro.dentella@gmail.com

unittest

La libreria base ha un modulo chiamato `unittest` che ci permette non solo di scrivere i test ma anche di lanciarli ed avere un output di sintesi dove ogni test che passa è raffigurato da un "." e ogni test fallito da "F":

```
python test_sum_unittest.py
.F
======================================================================
FAIL: test_sum_tuple (__main__.TestSum)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "test_sum_unittest.py", line 9, in test_sum_tuple
    self.assertEqual(sum((1, 2, 2)), 6, "Should be 6")
AssertionError: Should be 6
----------------------------------------------------------------------
Ran 2 tests in 0.001s
FAILED (failures=1)
```

python

Alessandro Dentella
sandro.dentella@gmail.com

testrunner

Esistono alcune librerie che permettono di lanciare i test in modo diverso:

- **`unittest`**: nella lib std. Prevede che i test siano raggruppati in classi, quindi sono nella forma di metodi, e assert è un metodo (ne esistono molte varianti)
- **`nose2:`** accetta tutti i test di unittest ma è più flessibile ed accetta filtri per eseguire solo alcuni test
- **`pytest`**: accetta che i test siano semplici funzioni, uso assert standard, può eseguire solo i test falliti in un run precedente.

Alessandro Dentella
sandro.dentella@gmail.com

Come strutturare i test

I test devono provare le "units"

- con parametri differenti per tipo, per valore, ...
- nei casi in cui è previsto funzionare
- nei casi in cui è previsto fallire

python

Alessandro Dentella
sandro.dentella@gmail.com

Workflow con TDD

- Si comprendono le specifiche
- Si concorda una API
- Si scrivono i test, che quindi funzionano come una implementazione del controllo delle specifiche
- Si verifica che falliscono
- Si implementa il codice
- Si verifica che i test passano
- Ad ogni modifica del codice viene rieseguita tutta la suite di test.

Alessandro Dentella
sandro.dentella@gmail.com

CI / CD

- è possibile creare un setup che invochi automaticamente i test ad ogni commit. Si chiama CI (continuous integration)
- è possibile anche spingere oltre l'automazione ed arrivare al deploy automatico e continuo: CD

python

Alessandro Dentella
sandro.dentella@gmail.com

Test in pratica

- Quasi ogni serio programma opensource ha una suite di test anche molto ricca
- I test sono anche una documentazione concreta e inoppugnabile di come andrebbe usato il codice
- Django ha circa 122k righe di test a fronte di 213k righe di codice
- Questo garantisce che siano ridotti in modo quasi totale gli errori
- I test hanno un costo elevato nel prepararli ma contribuiscono in modo essenziale alla qualità del prodotto

Alessandro Dentella
sandro.dentella@gmail.com

Doctest vs. unittest

Esiste un modo di scrivere i test in una docstring

```
>>> from sqlkit.misc import datetools
as dt
>>> from datetime import *
>>> dt.TEST = True

>>> dt.TODAY = date(2007, 5, 15)
>>> dt.string2date('d')
datetime.date(2007, 5, 15)

>>> dt.string2date('m')
datetime.date(2007, 5, 1)
>>> dt.string2date('M')
datetime.date(2007, 5, 31)
```

python

Alessandro Dentella
sandro.dentella@gmail.com

unittest

Il modo più utilizzato è di scrivere i test singoli in metodi di una classe che eredita da

```
from django.test import TestCase, Client

class FirstTest(TestCase):

    def setUp(self):
        self.client.login(username='admin', password='')

    def test_00(self):
        "Prova di test"
        self.assert_(True)
```

python

Alessandro Dentella
sandro.dentella@gmail.com

- Creiamo un test per verificare che la funzione per trovare i numeri primi funzioni correttamente
- Creiamo una cartella tests su gitlab in modo da cominciare una suite di test per tutti gli esercizi proposti: dobbiamo modificare il codice degli esercizi?

python

Database

Alessandro Dentella
sandro.dentella@gmail.com

Accesso ai database

- L' accesso ai database può essere fatto sia in modo diretto che attraverso un layer intermedio che può avere varie componenti. Quella che analizziamo è l' ORM.
  - Acceso diretto
  - Accesso via ORM (Object Relational Mapper), ovvero un layer che ci ritorna un oggetto per ogni record del database i cui attributi sono in relazione con gli attributi della tabella sottostante. Sia php che python hanno un considerevole numero di ORM disponibili.

python

Alessandro Dentella
sandro.dentella@gmail.com

Accesso diretto

- Python ha unificato l' interfaccia che i singoli driver devono avere nella PEP 249 “database API specification”, quindi tutti i moduli hanno la medesima interfaccia: http://www.python.org/dev/peps/pep-0249/
- Python ha moduli per interfacciarsi a tutti i più comuni database: sqlite, mysql, postgres, oracle, db2, mssql, firebird, ms-sql, odbc...

python

Alessandro Dentella
sandro.dentella@gmail.com

Db di prova

- Gli esempi vengono fatti con il db sqlite di prova ove questo è sufficiente
- Sqlite non ha un vero tipo data (e comunque non richiede il rispetto del tipo di dato nella colonna), quindi faremo anche alcune prove su db esterno.

python

Alessandro Dentella
sandro.dentella@gmail.com

Python

- Tutti i driver per database che aderiscono alle database API v2 hanno la medesima interfaccia, quindi i comandi sono i medesimi. (es. 20, 21, 22)

```
cnx = psycopg2.connect("user=%s dbname=%s" % (user, dbname))
```

Alessandro Dentella
sandro.dentella@gmail.com

API v2

- Le API v2 unificano le interfacce che deve avere un driver
  - connect()
  - paramstyle (qmark, numeric, named, format, pyformat)
  - apilevel
  - threadsafety
- La connection deve avere:
  - .close()
  - .commit()
  - .rollback()
  - .cursor()


```python
import sqlite3

# Connessione al database SQLite
conn = sqlite3.connect('database.db')

# Creazione di un cursore per eseguire le query SQL
cursor = conn.cursor()

# Esecuzione di una query SQL per selezionare tutti i record dalla tabella
cursor.execute("SELECT * FROM nome_tabella")

# Recupero di tutti i record restituiti dalla query
rows = cursor.fetchall()

# Stampa dei risultati
for row in rows:
  print(row)

# Chiusura del cursore e della connessione
cursor.close()

conn.close()
```

Alessandro Dentella
sandro.dentella@gmail.com

Paramstile

per evitare la sql injection la dbapi prevede che i valori per uno statement siano passati separatamente, tramite parametri che possono avere vari stili:

```
# Stile 'qmark' (con il punto interrogativo '?')
cursor.execute("INSERT INTO utenti (nome, cognome) VALUES (?, ?)", (
    nome_utente, cognome_utente))

# Stile 'named' (con i nomi dei parametri)
cursor.execute("INSERT INTO utenti (nome, cognome) VALUES (:nome, :cognome)", {
    'nome': nome_utente, 'cognome': cognome_utente})

# Stile 'format' (con segnaposto in stile Python)
cursor.execute("INSERT INTO utenti (nome, cognome) VALUES (%s, %s)" % ('?', '?'), (
    nome_utente, cognome_utente))

# Stile 'pyformat' (con segnaposto in stile Python)
cursor.execute("INSERT INTO utenti (nome, cognome) VALUES (%(nome)s, %(cognome)s)", {
    'nome': nome_utente, 'cognome': cognome_utente})
```

Alessandro Dentella
sandro.dentella@gmail.com

python

Errori

Le API definiscono che ogni errore generato debba ricadere in uno di questi

```
StandardError
        |__Warning
        |__Error
            |__InterfaceError
            |__DatabaseError
                |__DataError
                |__OperationalError
                |__IntegrityError
                |__InternalError
                |__ProgrammingError
                |__NotSupportedError
```

python

Alessandro Dentella
sandro.dentella@gmail.com

Il cursore

- .description
- .rowcount
- .callproc(procname, parameters)
- .close()
- .execute(operation, parameters)
- .executemany()

- .fetchone()
- .fetchmany()
- .fetchall()
- .nextset()
- .arraysize
- .setinputsizes(sizes)
- .setoutputsize(size)

python

Alessandro Dentella
sandro.dentella@gmail.com

paramstyle

- Per questioni di sicurezza, è opportuno passare i valori come 'bind parameters'
- Esistono vari stili:
  - qmark: ???
    i dati sono passati come tupla a .execute()
  - named: :first_name, :last_name
    i dati sono passati come mapping a .execute()
  - pyformat: %(first_name)s %(last_name)s
  - numeric: :1
  - format: ansi c format code

Alessandro Dentella
sandro.dentella@gmail.com

Data Science

Alessandro Dentella
sandro.dentella@gmail.com

Python ha un numero vasto di librerie nel campo della analisi dei dati che possiamo dettagliare così:

- **Esplorazione e analisi dei dati:** Pandas; NumPy; SciPy; un aiuto dalla libreria standard di Python.

- **Visualizzazione dei dati**. Un nome piuttosto autoesplicativo. Prendere i dati e trasformarli in qualcosa di colorato: Matplotlib; Seaborn; Datashader; altri.

- **Apprendimento automatico classico.** Concettualmente, potremmo definirlo come qualsiasi compito di apprendimento supervisionato o non supervisionato che non sia *deep learning* (vedi sotto). Scikit-learn è di gran lunga lo strumento principale per implementare classificazione, regressione, clustering e riduzione della dimensionalità, mentre StatsModels è meno attivamente sviluppato ma ha ancora diverse funzionalità utili.

- **Deep learning** Questo è un sottoinsieme dell'apprendimento automatico che sta vivendo una rinascita ed è comunemente implementato con Keras, tra altre librerie. Keras, TensorFlow e molti altri.

- **Archiviazione dei dati e framework per big data.** Il big data è meglio definito come dati che sono o letteralmente troppo grandi per risiedere su una singola macchina, o non possono essere elaborati in assenza di un ambiente distribuito. Le connessioni Python alle tecnologie Apache hanno un ruolo importante qui. Apache Spark; Apache Hadoop; HDFS; Dask; h5py/pytables.

- **Questioni varie.** Include sottotematiche come l'elaborazione del linguaggio naturale e la manipolazione delle immagini con librerie come OpenCV. nltk; Spacy; OpenCV/cv2; scikit-image; Cython.

Alessandro Dentella
sandro.dentella@gmail.com

NumPy e Pandas

- affrontiamo solo a livello promozionale NumPy e Pandas
- NumPy offre la struttura base, l'array multidimensionale
- Pandas offre una facilità d'uso ed integrazione con la grafica offerta da matlib

Esempi in Jupiter notebook (colab)

python

Alessandro Dentella
sandro.dentella@gmail.com

NumPy

- NumPy è una libreria fondamentale per il calcolo scientifico in Python.
- Fornisce **supporto per array multidimensionali** ad alte prestazioni e funzioni matematiche per operare su di essi.
- NumPy è ampiamente utilizzato in settori come l'analisi dei dati, l'apprendimento automatico, l'elaborazione delle immagini e altro ancora.
- La sua **velocità e la sua efficacia** nel gestire grandi quantità di dati lo rendono una scelta popolare tra gli scienziati e gli ingegneri.

Alessandro Dentella
sandro.dentella@gmail.com

NumPy deve la sua velocità principalmente a due fattori principali:

- **Implementazione in C:** Gran parte del codice di NumPy è scritto in linguaggio C, che è notevolmente più veloce di Python. Le operazioni di calcolo principali, come le operazioni sugli array e le funzioni matematiche, sono implementate in C per massimizzare l'efficienza e la velocità.

- **Operazioni vettorializzate:** NumPy utilizza operazioni vettorializzate per eseguire operazioni su interi array anziché su singoli elementi. Questo significa che le operazioni sono ottimizzate per essere eseguite su grandi blocchi di dati in modo efficiente utilizzando istruzioni SIMD (Single Instruction, Multiple Data) o altre ottimizzazioni di basso livello. Questo approccio riduce notevolmente il tempo di esecuzione rispetto alla scrittura di loop Python per elaborare singoli elementi.

- Inoltre, NumPy è altamente **ottimizzato e continuamente sviluppato dalla comunità degli sviluppatori Python** per massimizzare la sua efficienza e la sua velocità. La combinazione di queste caratteristiche rende NumPy uno strumento fondamentale per il calcolo scientifico ad alte prestazioni in Python.

Alessandro Dentella
sandro.dentella@gmail.com

Elementi

- **Array multidimensionali:** NumPy offre un potente oggetto **array** multidimensionale che consente di eseguire operazioni matematiche su intere matrici o vettori in modo efficiente.
- **Funzioni matematiche:** NumPy include una vasta gamma di funzioni matematiche integrate per operazioni come la somma, la media, la deviazione standard, il prodotto scalare e altro ancora.
- **Broadcasting:** NumPy supporta il broadcasting, che consente di eseguire operazioni tra array di dimensioni diverse in modo automatico e efficiente.
- **Integrazione con altre librerie:** NumPy è ampiamente integrato con altre librerie Python, come SciPy per il calcolo scientifico avanzato e Matplotlib per la visualizzazione dei dati.

Alessandro Dentella
sandro.dentella@gmail.com

Pandas

- Pandas è una potente libreria open-source per la **manipolazione e l'analisi** dei dati in Python.
- **Fornisce strutture dati flessibili e performanti**, ideali per la manipolazione e l'analisi di dati tabellari e serie temporali.
- Pandas è ampiamente **utilizzato in settori come l'analisi dei dati, la scienza dei dati, il machine learning e altro ancora**.
- Offre un'**interfaccia intuitiva e ricca di funzionalità** per caricare, manipolare, filtrare, aggregare e visualizzare dati in modo efficiente.

Alessandro Dentella
sandro.dentella@gmail.com

Elementi

- **Strutture dati principali:** Pandas offre due principali strutture dati: **DataFrame**, per rappresentare dati tabellari, e **Series**, per rappresentare una singola colonna o riga di dati.
- **Potenti operazioni di indicizzazione e selezione**: Pandas offre numerose opzioni per selezionare, filtrare e manipolare dati all'interno di DataFrame e Series utilizzando metodi come **loc[] e iloc[].**
- **Gestione di dati mancanti:** Pandas fornisce strumenti per gestire dati mancanti, come dropna() per eliminare righe o colonne con valori mancanti e fillna() per sostituire i valori mancanti con valori specificati.
- **Facilità di integrazione con altre librerie:** Pandas si integra facilmente con altre librerie Python, come NumPy per calcoli numerici e Matplotlib per la visualizzazione dei dati.

Alessandro Dentella
sandro.dentella@gmail.com