Codice Corso – Python

Maggio 2021

Materiale → **http://bit.ly/MPSPY**

https://gitlab.wicome.com/fabrizio/pythonbase.git

Piattaforma **https://train.wicome.com/guacamole**

(per chi non ha installazioni in locale)

Rev 5.50

Partecipanti credenziali

- utente student1 ove x è il vostro numero progressivo
- la password è per tutti *studentwicome*
- *Sulle macchina individuali: (ove previsto)*
  - *utente student*
  - *pwd studentwicome*

Anaconda

- Enterprise-ready Python distribution for
  - data analytics,
  - processing,
  - scientific computing.
  - Python 2.7 or Python 3.4
  - 100+ cross-platform tested and optimized Python packages.

```
.condarc file → conda config
   e.g.  conda config --add channels
conda-forge
   e.g.  conda config --set
auto_update_conda False
            yaml format
 for a spec env
~/miniconda3/envs/flowers/.condarc
.conda directory

.continuum directory
```

- Get ready
  - conda - - version; conda update conda
  - allows you to to create separate environments containing files, packages and their dependencies <u>that will not interact with other environments</u>.
  - conda create --name snowflakes biopython ; conda info –envs ; python --version
  - [*source*] [*de*]activate snowflakes also conda activate xxx on linux

    linux
  - conda create --name snakes python=3.5 (force with a specific interpreter)
  - conda search ; conda list;conda install

Installazione di miniconda

- miniconda prompt
- conda install jupyter spyder pandas numpy

Anaconda Packages & Environments

```
Python 3.3+ has Implicit Namespace Packages that
allow to create a packages without an __init_.py
```

```
cards/
    __init__.py
    standard.py
    blackjack.py
    poker.py
```

- Module
  - a file that contains Python programming.
  - can be brought into another program via the import statement, or it can be executed directly as the main script of an application program
  - A **library module** is expected to contain definitions of classes, functions and module variables.
  - A script (or application or “main” module) does the useful work of an application.
- Packages → a number of modules logically related; a directory that contains modules, generally a directory structure
  - allows us to use qualified module names
  - the package structure clarifies the relationships among the modules.
    - If we have several modules related to the game of craps, we might have the urge to create a craps_game.py module and a craps_player.py module. As soon as we start structuring the module names to show a relationship,

      we can use a package instead.

```
casino/
    craps/
        dice.py
        game.py
        player.py
    roulette/
        wheel.py
        game.py
        player.py
    blackjack/
        cards.py
        game.py
        player.py
    srategy/
        basic.py

martingale.py
        bet1326.py

cancellation.py
```

Anaconda

- Install as a user (not administrative or root permission)
  - 32- or 64-bit computer; Miniconda 400 MB; Anaconda 3 GB minimum
  - Windows, macOS or Linux ; Python 2.7, 3.4, 3.5...7.
  - pycosat; PyYaml; Requests.
- Verify PATH if there are previous or different installation of Python
  - echo %PATH%; echo $PATH; which | where python
- Channels
  - where to pick packages

  - Other refs : go https://conda.io/docs/user-guide

Alcune note per l’ambiente Windows

- Attenzione se installate Python preso da Python.org su windows
  - per default vi viene presentata la versione 32 bit
  - installate solo per il vostro utente altrimenti vi serviranno i diritti amministrativi
  - aggiungete il PATH di Python
  - ricordate il CALL invece che lo START nei bat

Phyton VENV

```
su -# subscription-manager repos --enable
rhel-7-server-optional-rpms \  --enable
rhel-server-rhscl-7-rpms# yum -y install
@development# yum -y install rh-
python36 # yum -y install rh-python36-
numpy \ rh-python36-scipy \  rh-python36-
python-tools \ rh-python36-python-six #
exit
```

- Creare un venv
  - python3 -m venv /path/to/new/virtual/environment
  - source /path/to/new/virtual/environment/venv/bin/activate
- Specificare un python
  - pip install virtualenv
  - virtualenv -p /home/example_username/opt/python-3.6.2/bin/python3 my_project
- Python impacchettamento
  - pip install venv-pack
    - $ venv-pack -o my_env.tar.gz   # Pack the current environment into my_env.tar.gz
    - $ venv-pack -p /explicit/path/to/my_env # Pack an environment located at an explicit path into my_env.tar.gz
  - On the target machine
    - $ mkdir -p my_env   # Unpack environment into directory `my_env`
    - $ tar -xzf my_env.tar.gz -C my_env
    - $ source my_env/bin/activate   # Activate the environment. This adds `my_env/bin` to your path
- /usr/local/bin → programmi
- /usr/local/lib → librerie e moduli

The Spyder enviroment

- a powerful interactive development environment for the Python language with advanced editing, interactive testing, debugging and introspection features
- numerical computing environment thanks to the support of
  - IPython (enhanced interactive Python interpreter)
  - NumPy (linear algebra),
  - SciPy (signal and image processing)
  - matplotlib (interactive 2D/3D plotting)
- code completion, calltips, goto definition ^g, info ^i, occurrence highlighting
- real time static analysis via pylint (code quality), errors warnings via pyflakes
- TODO FIXME XXX   # TODO: blabla
- debugger
- Consoles
  - each in a separate process
  - variable explorer (with import data form several sources and file types: text, !NumPy, MatLab)
  - data visualization

Spyder concepts CODE CELL

- Similar to MATLAB 'cell' → but no cell mode in Spyder
- A Block: a block of lines to be executed at once in the current interpreter (Python or Ipython).
  - Every script may be divided in as many cells as needed.
  - Starting with
    - #%% (standard cell separator)
    - # %% (standard cell separator, when file has been edited with Eclipse)
    - # <codecell> (IPython notebook cell separator)

Ipython Console

- Interactive usage oriented
- Is a full two-process IPython session where a lightweight front-end interface connects to a full IPython kernel on the back end
- Spyder can launch IPython Console instances that attach to kernels that are managed by Spyder itself or it can connect to external kernels that are managed by IPython Qt Console sessions or the Ipython Notebook
    - Support for Variable explorer only for kernel created by local Spyder not for external kernel
- Please note that variable explorer support a lot of different types and moreover can show them a as a plot or image
- object inspector (inside console or editor highlight an object then inspect)
- See tutorial

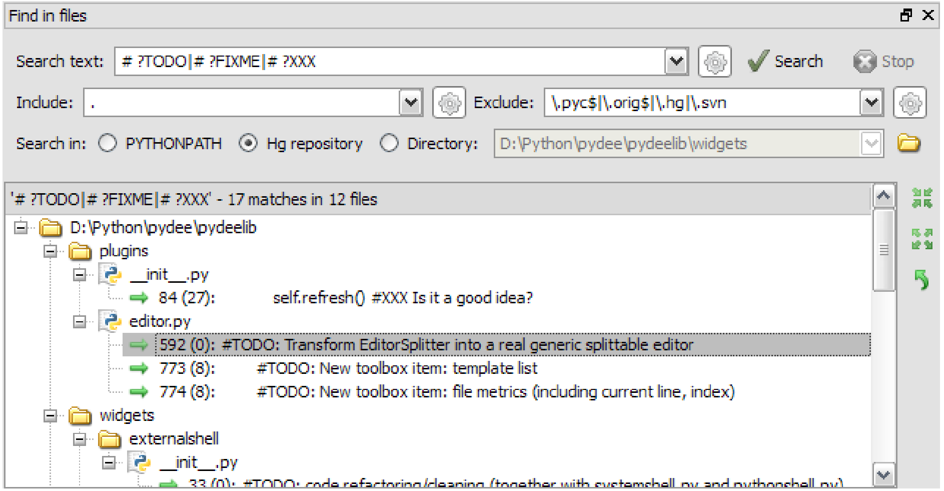
```
1.Pandas DataFrame, TimeSeries and
DatetimeIndex
2.NumPy arrays and matrices
3. PIL/Pillow images
4. datetime dates
5. Integers
6. Floats
7. Complex numbers
8. Lists
9. Dictionaries
10. Tuples
11. Strings
```

Debugging

- Partially integrated in Spyder
- Breakpoints may be defined in the Editor.
  - Simple breakpoints can be set from the Run menu, by keyboard shortcut (F12 by default), or by doubleclick
  - Conditional breakpoints can also be set from the Run menu, Shift+F12 by default or by Shift+double-click.
  - the cyrrent debugging step is highlighted in the Editor.
  - At each breakpoint, globals may be accessed through the Variable Explorer.

Spyder Projects

- Associate a given directory with a project
  - a list of file
  - project path added
  - find in files with regular expr support



Search text: # ?TODO|# ?FIXME|# ?XXX

Include: .

Exclude: \.pyc$|\.orig$|\.hg|\.svn

Search in: ○ PYTHONPATH ● Hg repository ○ Directory: D:\Python\pydee\pydeelib\widgets

```
'# ?TODO|# ?FIXME|# ?XXX' - 17 matches in 12 files
⊟ D:\Python\pydee\pydeelib
   ⊟ plugins
      ⊟ __init__.py
            84 (27):        self.refresh() #XXX Is it a good idea?
      ⊟ editor.py
            592 (0): #TODO: Transform EditorSplitter into a real generic splittable editor
            773 (8):       #TODO: New toolbox item: template list
            774 (8):       #TODO: New toolbox item: file metrics (including current line, index)
   ⊟ widgets
      ⊟ externalshell
         ⊟ __init__.py
               33 (0): #TODO: code refactoring/cleaning (together with systemshell.py and pythonshell.py)
```

Jupiter Notebooks

- Jupyter Documents also "notebook" or "notebook documents" contain both code and rich text elements, such as figures, links, equations
  - produced by the Jupyter Notebook App.
  - the ideal place to bring together
    - analysis description
    - its results
    - more they can be executed to perform the data analysis in real time.
    - jupyter notebook --notebook-dir='C:\Your\Desired\Start\Directory\'
- Jupyter Notebook App
  - is a server-client application, that allows to edit and run notebooks via a web browser
  - two components kernel and dashboard
- Three tabs when opening in browser
  - The "Files" tab is where all your files are kept, the "Running" tab keeps track of all your processes and the third tab, "Clusters", is provided by IPython parallel, IPython's parallel computing framework
  - New Noteboook click new on files tab
    - also terminal, folders, file ...

Jupyter reference

- http://jupyter.readthedocs.io/en/latest/

Jupyter

- Step 1 import libraries
- Step 2 add, remove, edit cell code
  - and explanatory text or Titles!! That's what makes a notebook a notebook in the end.
  - if LaTex $$ prefix → $$c = \sqrt{a^2 + b^2}$$
  - from IPython.display import display, Math, Latex
  - display(Math(r'\sqrt{a^2 + b^2}'))
  - import pandas as pd
    import numpy as np
    df = pd.DataFrame(data=np.array([[1,2,3],[4,5,6]], dtype=int), columns=['A','B','C'])

Jupiter Markdowns

- **Headings**:
  - Use #s followed by a blank space for notebook titles and section headings:
  - # title (also === below the title)
  - ## major headings (also --- below the title)
  - ### subheadings
  - #### 4th level subheadings

```
\begin{equation*}
P(E)   = {n \choose k} p^k (1-p)^{ n-k}
\end{equation*}
```

- **Emphasis**: Use this code: Bold: __string__ or **string** Italic: _string_ or *string*
- **Mathematical symbols**: Use this code: $ mathematical symbols $ per MathJax subset of LaTex ($$ $$
  - $\sqrt{3x-1}+(1+x)^2$ → inline
- **Monospace font**: 'monospace'
- **Line breaks**: <br>.
- **Colors**: <font color=blue|red|green|pink|yellow>Text</font>
- **Indenting**: > text
- **Bullets**: - (two spaces) or - → circular bullet. To create a sub bullet, use a tab followed a dash and two spaces. also use an asterisk instead of a dash.
- **Numbered lists**: Start with 1. followed by a space, then it starts numbering for you. Start each line with some number and a period, then a space. Tab to indent to get subnumbering.
- **Internal links** [section title](#section-title)
- **External links** __[link text](http://url)__
  - vedi anche https://sourceforge.net/p/jupiter/wiki/markdown_syntax/#md_ex_links

Jupyter Markdowns

- Colored note boxes:
  - <div class="alert alert-block alert-info">Tip: Use blue boxes for Tips and notes. If it’s a note, you don’t have to include the word “Note”.</div>
  - <div class="alert alert-block alert-warning">Example: Use yellow boxes for examples that are not inside code cells, or use for mathematical formulas if needed.</div>
  - <div class="alert alert-block alert-success">Up to you: Use green boxes sparingly, and only for some specific purpose that the other boxes can't cover. For example, if you have a lot of related content to link to, maybe you decide to use green boxes for related links from each section of a notebook. </div>
  - <div class="alert alert-block alert-danger">Just don't: In general, just avoid the red boxes.</div>
- Graphics <img src="url.gif" alt="Alt text that describes the graphic" title="Title text" />
- Horizontal lines: Use three asterisks: ***
- Internal links: To link to a section, use this code: [section title](#section-title)

Python – il linguaggio

Python features

Lutz, *Programming Python*

no compiling or linking	rapid development cycle
no type declarations	simpler, shorter, more flexible
automatic memory management	garbage collection
high-level data types and operations	fast development
object-oriented programming	code structuring and reuse, C++
embedding and extending in C	mixed language systems
classes, modules, exceptions	"programming-in-the-large" support
dynamic loading of C modules	simplified extensions, smaller binaries
dynamic reloading of C modules	programs can be modified without stopping

Python features

Lutz, *Programming Python*

universal "first-class" object model	fewer restrictions and rules
run-time program construction	handles unforeseen needs, end-user coding
interactive, dynamic nature	incremental development and testing
access to interpreter information	metaprogramming, introspective objects
wide portability	cross-platform programming without ports
compilation to portable byte-code	execution speed, protecting source code
built-in interfaces to external services	system tools, GUIs, persistence, databases, etc.

Python

- elementi di C++, Modula-3 (modules), ABC, Icon (slicing)
- Apparentemente della stessa famiglia Perl, Tcl, Scheme, REXX, BASIC dialects
- Ma
  - multiparadigma per cui un C++, Perlato, Lisposo, Javizzato ;)
  - approccio molto scripting (il deploy può essere dal complesso al molto complesso)
- Python 3 e Python 2

Using python

- /usr/local/bin/python
  - #! /usr/bin/env python
- interactive use

```
fabrizio@frolix-large:~$ python
Python 2.7.10 (default, Oct 14 2015, 16:09:02)
[GCC 5.2.1 20151010] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

- python –c *command* [*arg*] …
- python script.py [arg]
- python –i *script.py*
  - read script first, then interactive

Python structure

- modules: Python source files or C extensions
  - import, top-level via from, reload
- statements
  - control flow
  - create objects
  - indentation matters – instead of {}
  - ; (deprecated) to multiple statement on a line
- objects
  - **everything is an object**
  - automatically reclaimed when no longer needed

Primissimi esempi

- Fondamentali
  - print("Ciao mondo")
  - help(“print”)
  - import this
- Definizioni
  - a=1
  - b=2*a
  - c=3.0
  - d='questa è una stringa'
  - print(a)
  - print(b)
  - print(c)
  - print(d)

Righe
No terminatore
No def per le var ne tipo solo =
Se uso una variabile non assegnata mi da NameError

```
c=3.0
print("tipo di c = ",type(c))
tipo di c =  <class 'float'>
```

Alcune convenzioni utili

- Notare l'indentazione

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

# import modules used here -- sys is a very standard one

import sys
a=3

if a == 0:
  b= 5
else:
  k = 44

d=33

# Gather our code in a main() function
def main():
    print 'Hello there', sys.argv[1]
    # Command line args are in sys.argv[1], sys.argv[2] ...
    # sys.argv[0] is the script name itself and can be ignored

# Standard boilerplate to call the main() function to begin
# the program.
if __name__ == '__main__':
    main()
```

Operazioni fondamentali: variabili

- Assegnazione:

```
size = 40
a = b  = c = 3
```

- Numeri
  - integer, float
  - complex numbers: 1j+3, abs(z)
- Stringhe

```
r'c:\temp\'   → c:\temp\    c:    emp
'hello world', 'it\'s hot'
"bye world"
continuation via \ or use """ long text """
a = "trentatre trentini trottellavano \
tortellamente"
```

Primi esempi

- help(dir)   → lo spazio dei nomi di quell'oggetto
- dir()
    - se invocata senza argomenti, indica i nomi definiti nel contesto (scope) corrente:
        - 'builtins'
        - 'doc '
        - 'name '
        - 'package '
        - 'a' (nomi delle variabili che abbiamo definito)
        - 'b'
    - #creo un intero
    - print(type(a))
    - print(dir(a))

```
>>> a=3
>>> print(type(a))
<class 'int'>
>>> dir(a)                # proprietà ed attributi di a
['__abs__', '__add__', '__and__', '__bool__', '__ceil__', '__class__', '__delattr__',
'__dir__', '__divmod__', '__doc__', '__eq__', '__float__', '__floor__', '__floordiv__',
'__format__', '__ge__', '__getattribute__', '__getnewargs__', '__gt__', '__hash__',
'__index__', '__init__', '__int__', '__invert__', '__le__', '__lshift__', '__lt__', '__mod__',
'__mul__', '__ne__', '__neg__', '__new__', '__or__', '__pos__', '__pow__', '__radd__',
'__rand__', '__rdivmod__', '__reduce__', '__reduce_ex__', '__repr__', '__rfloordiv__',
'__rlshift__', '__rmod__', '__rmul__', '__ror__', '__round__', '__rpow__', '__rrshift__',
'__rshift__', '__rsub__', '__rtruediv__', '__rxor__', '__setattr__', '__sizeof__', '__str__',
'__sub__', '__subclasshook__', '__truediv__', '__trunc__', '__xor__', 'bit_length',
'conjugate', 'denominator', 'from_bytes', 'imag', 'numerator', 'real', 'to_bytes']
```

E' tutto uno spazio dei nomi

- from importlib import metadata
- metadata.version("pip")   → '19.3.1'
- pip_metadata = metadata.metadata("pip")
- list(pip_metadata)
- In realtà pip_metadata è un dict
  - pip_metadata["Home-page"]   → https://pip.pypa.io/

Naming style

- The following special forms using leading or trailing underscores are recognized (these can generally be combined with any case convention):
  - **_pippo** _single_leading_underscore: weak internal use" indicator. E.g. from M import * does not import objects whose name starts with an underscore.
  - **pippo_** single_trailing_underscore_: used by convention to avoid conflicts with Python keyword, e.g.
    - **Tkinter.Toplevel(master, class_='ClassName')**
  - **__pippo** __double_leading_underscore: when naming a class attribute, invokes name mangling (inside class FooBar, __boo becomes _FooBar__boo; see below).
  - **__pippo__** __double_leading_and_trailing_underscore__: "magic" objects or attributes that live in user-controlled namespaces. E.g. __init__, __import__ or __file__. **Never invent such names; only use them as documented**.

__builtin__

modulo di python che integra nuove funzioni o anche nuovi tipi di oggetti.

- In [ ]: #builtin
- print(type(__builtins__))
  - __builtin__ unico modulo che viene sempre caricato dall'interprete quando questo e' usato e definisce i tipi e le funzioni predefinite
- dir(__builtins__)
  - __builtins__ ottengo una lista in cui ci sono tutti i nomi (di funzioni e tipi di dato ad esempio) definiti nel modulo

Commenti

```
# Questo è un commento
“”” Questo è un commento lungo
molto, molto, molto
Ma molto molto….
“””

A=3   # questo è un altro commento
```

Tipi di base

- Numeri interi (non mutevoli)
  - **NON SEGUITI DA PUNTO!!!!!**
  - Dinamica infinita gestita dal sistema

```
In [4]: a = 2
b = 1+1 #come risultato di un'operazione
c=2.0 #e' un numero reale
print(type(1))
print(type(b))
print(type(c))#!!
```

- Operazioni

```
In [7]: 1+2    #addizione
In [8]: 6-10   #sottrazione
In [9]: 5*4    #moltiplicazione
In [10]: 2**3  #potenza
In [11]: print(13%9) #modulo: restituisce il resto della divisione
print(type(13%9))
print(10/4 , type(10/4))   #divisione con resto (troncata)
In [13]: print(10//4 , type(10//4))     #divisione senza resto
```

- Conversione di base
  - 0b binario
  - 0o ottale
  - 0x esa

```
In [14]: a=0b10 #notazione binaria
print(a,type(a))
In [15]: a=0o10 #notazione ottale
print(a)
In [16]: a=0xF #notazione esadecimale
b=0x10
print(a)
print(b)
```

```
n [8]: print('rappresentazione di 35:')
print('base 2:', bin(35))
print('base 8:',oct(0x35))
print('base 16:',hex(35))
da binario:
In  [9]:  print('rappresentazione  di 0b100011:')
```

Operatori bitwise

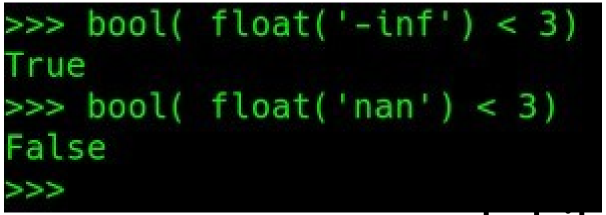
- & **and** iandj = bin(i&j)
  - a = 3 → 000.....011
  - b = 2 → 000....010
  - a & b 00 ... 0 10
- | **or** iorj = bin(i|j)
  - a | b → 00....0011
- ^ **xor** ixorj = bin(i^j)
  - a ^ b 0000 ..00011
- ~ Tilde complemento ad 1 → si invertono semplicemente tutti i bit della parola
- >> shift a dx idx = i>>3
- << shift a sx isx = i<<3

```
In [2]: i = 0b11010011
j = 0b11101100
print('i=',i,bin(i))
print('j=',j,bin(j))
i= 211 0b11010011
j= 236 0b11101100
```

Float (reali non mutevoli)

- A = 2.1
- B = 2.1e6
- Limiti sistema
-
- Numeri speciali
  - Inf (infinito) → math.inf
  - NaN → float('nan')
  - None
- Ciò che e' memorizzato e' il più vicino numero rappresentabile come frazione binaria.
- Il ché rende a volte i confronti avventurosi

```
In [29]: #informazioni accessibili grazie al modulo sys
import sys
print(sys.float_info)

sys.float_info(max=1.7976931348623157e+308,max_exp=1024,max_10_exp=308,
min=2.2250738585072014e-308,min_exp=-1021,min_10_exp=-307,dig=15,mant_dig=53,
epsilon=2.220446049250313e-16,  radix=2,  rounds=1)
```



```
>>> bool( float('-inf') < 3)
True
>>> bool( float('nan') < 3)
False
>>>
```

a=0.1
b=0.10000000000000001
c=0.1000000000000000055511151231257827021181583404541015625
Stessa rappresentazione binaria – a==b == c

Decimal (decimali non mutevoli)

- il modulo decimal "provides support for decimal floating point arithmetic." e definisce un tipo Decimal tale che:
  - il numero è memorizzato non in codifica binaria ma come una sequenza di cifre decimali
  - ogni numero e' rappresentato da una numero fissato di cifre
  - si tratta comunque di un numero floating point cioe', dato il numero di cifre, la virgola può stare ovunque

```
 import decimal
a=decimal.Decimal(0.1)
print(a)
0.1000000000000000055511151231257827021181583404541015625
 a+a+a
 0.30000000000000004
a=decimal.Decimal(“0.1”)
0.1
a+a+a
0.3
```

Problematiche con i decimali

- ## Somma di float vs somma di decimal

```
   somma di float
In [41]: 0.1 + 0.1 + 0.1 – 0.3   →       5.551115123125783e-17

   somma di decimal
In [42]: a=decimal.Decimal('0.1')
         b=decimal.Decimal('0.3')
         a+a+a-b
Out[42]:   Decimal('0.0')
```

- ## Precisione

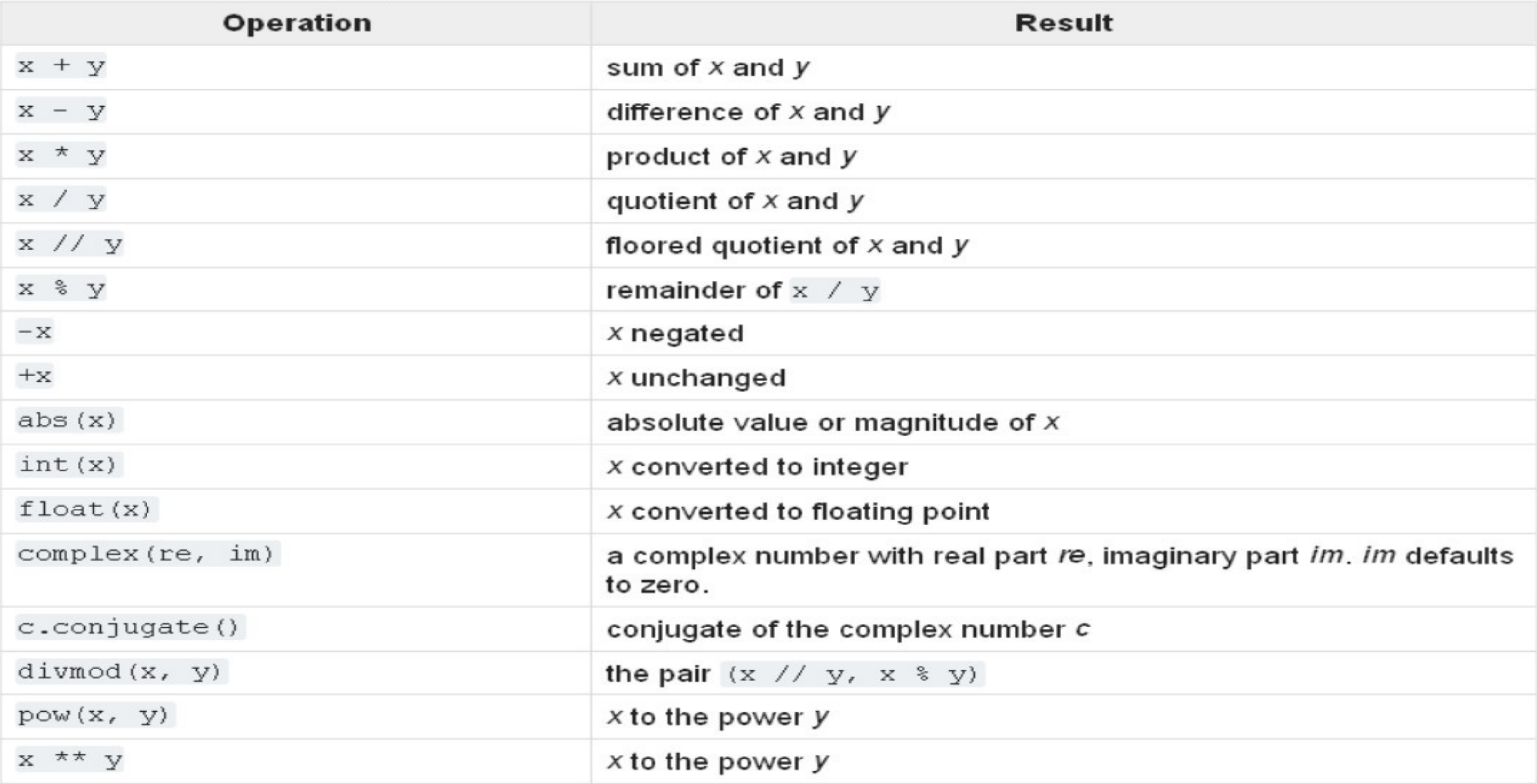
numeri decimali con precisione finita

```
In [51]: #modifico la precisione
decimal.getcontext().prec
4
A=decimal.Decimal(1)/decimal.Decimal(3)
print(A)  → 0.3333
In [52]: #modifico la precisione
         decimal.getcontext().prec=10
         A=decimal.Decimal(1)/decimal.Decimal(3)
         print(A)  → 0.3333333333
```

in ogni caso:

```
In [53]: A+A+A
Out[53]:  Decimal('0.9999999999')
In [54]: A+A+A == decimal.Decimal('1.0')
Out[54]:  False
```

Riepilogo operatori - others in math



Operation	Result
`x + y`	sum of *x* and *y*
`x - y`	difference of *x* and *y*
`x * y`	product of *x* and *y*
`x / y`	quotient of *x* and *y*
`x // y`	floored quotient of *x* and *y*
`x % y`	remainder of `x / y`
`-x`	*x* negated
`+x`	*x* unchanged
`abs(x)`	absolute value or magnitude of *x*
`int(x)`	*x* converted to integer
`float(x)`	*x* converted to floating point
`complex(re, im)`	a complex number with real part *re*, imaginary part *im*. *im* defaults to zero.
`c.conjugate()`	conjugate of the complex number *c*
`divmod(x, y)`	the pair `(x // y, x % y)`
`pow(x, y)`	*x* to the power *y*
`x ** y`	*x* to the power *y*

Operatori

```
In [58]: a = 2.0
b = 7.3
#definisco le operazioni come stringhe
c = 'a+b'                   #addizione
d = 'a-b'                  #sottrazione
e = 'a*b'                  #moltiplicazione
f = 'a/b'                  #divisione
g = 'a//b'                 #divisione intera
h = 'a**b'                 #elevazione a potenza
i = 'pow(a,b)'             #elevazione a potenza con funzione pow
l = '-a'                   #negazione
m = 'abs(-a)'              #valore assoluto con funzione abs
n = 'b%a'                  #resto
#uso la funzione eval per 'stampare' i risultati delle operazioni definite come stringhe
```

```
for esp in(c,d,e,f,g,h,i,l,m,n):
    print(esp , '=' ,eval(esp))
```

Vero e falso

- Ogni cosa può essere testata come:
  - vero → builtin **T**rue
  - falso → builtin **F**alse
- È falso
  - None
  - False
  - zero di qualsiasi tipo numerico, ad esempio, 0, 0,0, 0j.
  - qualsiasi sequenza vuota, per esempio,'', (), []
  - dizionari vuoti {}
  - istanze di classi definite dall'utente, se la classe definisce un metodo bool () o len () e il metodo restituisce il valore
  - intero zero o bool False.
- Tutti gli altri valori sono considerati veri. anche float('nan')

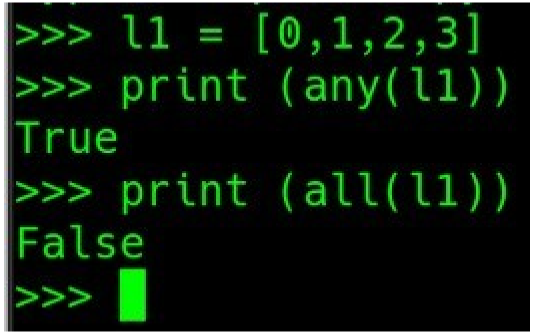
Bool, all, any

```
A = True
B = False
In [58]: a = True
b = False
c = False
```

```
In [59]: a or b and c
# and viene calcolato
# prima di or
# a or (b and c)
Out[59]:  True
```

```
In [60]: (a or b) and c  # parentesi
Out[60]:  False
```

```
# Since all are false, false is returned
print (any([False, False, False, False]))

# Here the method will short-circuit at the
# second item (True) and will return True.
print (any([False, True, False, False]))

# Here the method will short-circuit at the
# first (True) and will return True.
print (any([True, False, False, False]))
```



```
>>> l1 = [0,1,2,3]
>>> print (any(l1))
True
>>> print (all(l1))
False
>>> 
```

```
# ALL
# All the iterables are True so all
# will return True and the same will be printed
print (all([True, True, True, True]))

# Here the method will short-circuit at the
# first item (False) and will return False.
print (all([False, True, True, False]))

# This statement will return False, as no
# True is found in the iterables
print (all([False, False, False]))
```

Complex

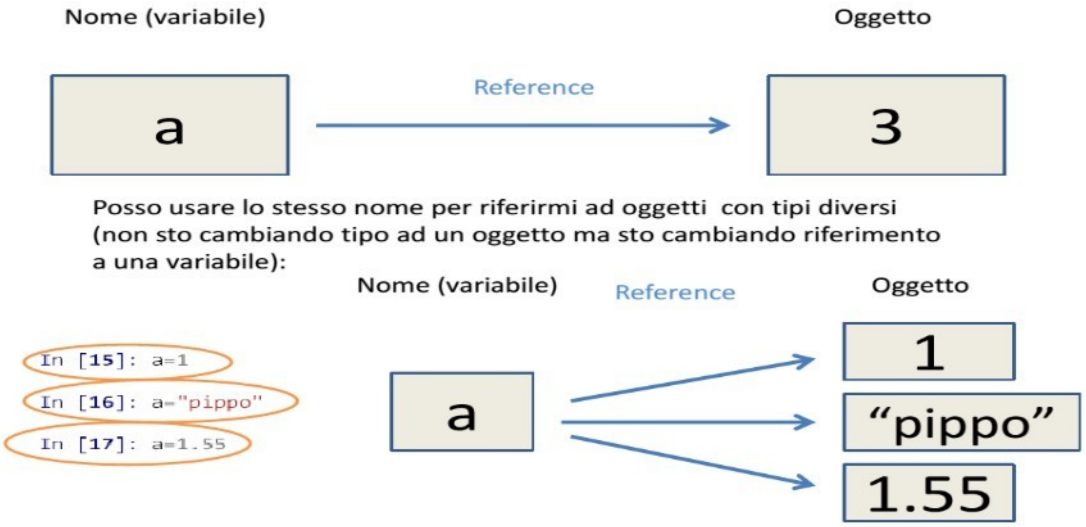
- #inizializzazione di un numero complesso
  - a = 1.0 + 1.5j # suffisso minuscolo
  - b = 2.0 + 2.5J # suffisso maiuscolo
  - c = 3.5J + 3.0 # prima la parte immaginaria
  - d = complex(4.0,4.5) # funzione 'costruttore' con argomenti passati per posizione
  - e = complex(imag=5.5,real=5.0) # funzione 'costruttore' con argomenti passati per nome
- print(type(c))
- print(a,b,c,d,e)

Operatori base

- Assegnazione
  - Va interpretata come:
  - l'espressione sul lato destro dell' = viene valutata;
  - l'oggetto corrispondente all'espressione viene creato/modificato da qualche parte, nella memoria del calcolatore;
  - il nome sul lato sinistro e' assegnato all'oggetto del punto precedente.
  - # creo una lista (a destra dell =) e gli associo il nome a a = [1,2,3]

Dynamic typing

- a = 3 implica una serie di azioni intraprese dal sistema
  - crea un oggetto intero che rappresenta il numero 3.
  - crea una variabile (il nome) a, se ancora non esiste.
  - connette il nome a all’oggetto che rappresenta il numero 3



Posso usare lo stesso nome per riferirmi ad oggetti con tipi diversi (non sto cambiando tipo ad un oggetto ma sto cambiando riferimento a una variabile):

```
In [15]: a=1
In [16]: a="pippo"
In [17]: a=1.55
```

Operatore Walrus

Syntax and semantics

In most contexts where arbitrary Python expressions can be used, a named expression can appear. This is of the form NAME := expr where expr is any valid Python expression other than an unparenthesized tuple, and NAME is an identifier.

The value of such a named expression is the same as the incorporated expression, **with the additional side-effect that the target is assigned that value:**

```
# From: https://www.python.org/dev/peps/pep-0572/#syntax-and-semantics

# Handle a matched regex
if (match := pattern.search(data)) is not None:
    # Do something with match

# A loop that can't be trivially rewritten using 2-arg iter()
while chunk := file.read(8192):
   process(chunk)

# Reuse a value that's expensive to compute
[y := f(x), y**2, y**3]

# Share a subexpression between a comprehension filter clause and its output
filtered_data = [y for x in data if (y := f(x)) is not None]
```

```
while(current := input("Scrivi qualcosa:  ")) != "quit":
...     inputs.append(current)
```

Operatore IS

- possiamo attribuire nomi diversi allo stesso oggetto e quindi e' necessario disporre di strumenti che ci permettano di evidenziare tale occorrenza
- IN[2]
  - a = [1,2,3]   # nomi diversi ma lo stesso oggetto
  - b = a
  - a is b # verifica
- **Out[2]: True**

- In [3]: # oggetti uguali ma distinti
  - a = [1,2,3]
  - b = [1,2,3]

- print(a==b)   → True   # test di uguaglianza
- print(a is b) → False # test di coincidenza dell'oggetto

ID Degli oggetti

- Ogni oggetto che istanziamo ha un suo id univoco. La funzione builtin id restituisce l'id di un oggetto:
  - In [4]: a=[1,2,3]
  - b=a
- #stesso oggetto, stesso id
  - print(id(a))
  - print(id(b))
    - 46320936
    - 46320936
- In [5]:
  - a=[1,2,3]
  - b=[1,2,3]
- #oggetti diversi (anche se con lo stesso contenuto), id diversi
  - print(id(a))
  - print(id(b))
    - 46320776
    - 46320456

Base types

- Numbers 0,1,2,3 → immutabili
  - Integer
  - Float
  - Decimal
  - Complex
- Strings 'abc', 'f' → immutabili, ordinate,iterabili → **sequenze**
- Lists [0,1,2,3] → mutabili, ordinate, iterabili → **sequenze**
- Tuple (0,1,2,3) → immutabili, ordinate, iterabili → **sequenze**
- Dictionary { 'a': 1, 'b':2 ,'paperoga':'ducks', 'quo': 'ducks', 'nip': ('qui','quo')}
  - iterabili, → **sequenze**
  - mutabili, "*ordinati*"
- Set { '3', 'b', 'a'} → mutabili, non ordinati, iterabili → **sequenze**

Altro esempio

```
$ python        ## Run the Python interpreter
Python 2.7.9 (default, Dec 30 2014, 03:41:42)
[GCC 4.1.2 20080704 (Red Hat 4.1.2-55)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> a = 6       ## set a variable in this interpreter session
>>> a           ## entering an expression prints its value
6
>>> a + 2
8
>>> a = 'hi'    ## 'a' can hold a string just as well
>>> a
'hi'
>>> len(a)      ## call the len() function on a string
2
>>> a + len(a)  ## try something that doesn't work
Traceback (most recent call last):
  File "", line 1, in
TypeError: cannot concatenate 'str' and 'int' objects
>>> a + str(len(a))  ## probably what you really wanted
'hi2'
>>> foo         ## try something else that doesn't work
Traceback (most recent call last):
  File "", line 1, in
NameError: name 'foo' is not defined
>>> ^D          ## type CTRL-d to exit (CTRL-z in Windows/DOS terminal)
```

Strutture di controllo

- Non a lot only three :)
- if
- for
- while

Vero e falso (ricordiamo che)

- Ogni cosa può essere testata come vero o falso
- È falso
  - None
  - False
  - zero di qualsiasi tipo numerico, ad esempio, 0, 0,0, 0j.
  - qualsiasi sequenza vuota, per esempio,'', (), []
  - dizionari vuoti {}
  - istanze di classi definite dall'utente, se la classe definisce un metodo bool () o len () e il metodo restituisce il valore
  - intero zero o bool False.
- Tutti gli altri valori sono considerati veri.

IF

```
>>> smiles = "BrC1=CC=C(C=C1)NN.Cl"
>>> bool(smiles)
True
>>> not bool(smiles)
False
>>> if not smiles:
...             print ("The SMILES string is empty")
…   a = 33
```

If finisce dove finisce l'indentazione

The “else” case is always optional

Control flow: if

```
x = int(input("Please enter #:"))
if x < 0:
      x = 0
      print ('Negative changed to zero')
elif x == 0:   print ('Zero')  # Not recommended
elif x == 1:
  print ('Single')
else:
  print ('More')
```

- no case statement

Elif - una forma di case ….

```
>>> mode = "absolute"
>>> if mode == "canonical":
...        smiles = "canonical"
... elif mode == "isomeric":
...        smiles = "isomeric”
... elif mode == "absolute":
...        smiles = "absolute"
... else:
...        raise TypeError("unknown mode")
...
>>> smiles
' absolute '
>>>
```

“raise” is the Python way to raise exceptions

Comparison operators (extended)

Operator	Description	Example
==	If the values of two operands are equal, then the condition becomes true.	(a == b) is not true.
!=	If values of two operands are not equal, then condition becomes true.	(a != b) is true.
>	If the value of left operand is greater than the value of right operand, then condition becomes true.	(a > b) is not true.
<	If the value of left operand is less than the value of right operand, then condition becomes true.	(a < b) is true.
>=	If the value of left operand is greater than or equal to the value of right operand, then condition becomes true.	(a >= b) is not true.
<=	If the value of left operand is less than or equal to the value of right operand, then condition becomes true.	(a <= b) is true.
is	absolute identity	
in	an object contains	

Logica booleana

Python expressions can have “and”s and “or”s:

```
if ben <= 5 and chen >= 10 or chen == 500 and ben != 5:
   print (“Ben and Chen“)

if ben < 5 or bon == 3
    print()
```

RANGE:

```
if 3 <= Time <= 5 :
            print (“Office Hour")
```

Condizioni §

- and and or are short-circuit operators:
  - evaluated from left to right
  - stop evaluation as soon as outcome clear
  - precedenza NOT, AND, OR, dalla più alta alla più bassa
- can check for identity with `is` and `is not`:
  - if a is b
- can check for sequence membership with `in` and `not in`
  - if 4 in vec   if ‘a’ in ‘abcde’
- chained comparisons: a less than b AND b equals c:

  a < b == c

Condizioni

- Can assign result of conditions to variable:

```
>>> s1,s2,s3='', 'foo', 'bar'
>>> non_null = s1 or s2 or s3
>>> non_null
foo
```

- Dalla versione 3.8 è possibile fare un assignment dentro una espressione (a:= 0)
  - operatore := walrus
  - ha come side effect quello di ritornare il valore assegnato

Control flow: for

```
a = ['cat', 'window', 'defenestrate']
for x in a:
  print ( x, len(x))
```

- non usiamo una progressione aritmetica ma una più generica sequenza
  - range(10) → [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

  -
    ```
    for i in range(len(a)):
    print ( i, a[i])
    ```

  o una sequenza rovesciata --> reversed(a)
- **non modificate mai la sequenza sulla quale stiamo iterando**
- tener conto _ meglio niente l=0 l=l+1 e orrori simili
  - for idx, value in **enumerate** (a)

RANGE

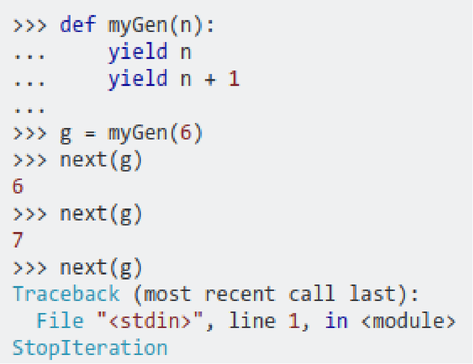
- “range” creates a list of numbers in a specified range
  - **range([start,] stop[, step]) -> list of integers**
  - **first limit enclosed, last excluded**
  - When step is given, it specifies the increment (or decrement).

    ```
    >>> range(5)
    [0, 1, 2, 3, 4]
    >>> range(5, 10)
    [5, 6, 7, 8, 9]
    >>> range(0, 10, 2)
    [0, 2, 4, 6, 8]
    ```
- How to get every second element in a list?

  ```
  for i in range(0, len(data), 2):
      print (data[i])
  ```

Range - yield

- range([start], stop[, step])
- range è un → Generator
- a function which returns an object on which you can call next, such that for every call it returns some value, until it raises a StopIteration exception, signaling that all values have been generated. Such an object is called an iterator.
- Normal functions return a single object using return.
- In Python, however, there is an alternative, called yield.
    - Using yield anywhere in a function makes it a generator.



```
>>> def myGen(n):
...     yield n
...     yield n + 1
...
>>> g = myGen(6)
>>> next(g)
6
>>> next(g)
7
>>> next(g)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

```
import sys
import os
import hashlib

def chunk_reader(fobj, chunk_size=1024):
    """Generator that reads a file in chunks of bytes"""
    while True:
        chunk = fobj.read(chunk_size)
        if not chunk:
            return
        yield chunk
def check_for_duplicates(paths, hash=hashlib.sha1):
    hashes = {}
    for path in paths:
        for dirpath, dirnames,
```

yield – un esempio

- Usare yield rende la funzione un generatore
  - il generatore continuerà a restituire la variabile a ad ogni ciclo, aspettando fino
    - alla prossima invocazione del metodo __next__()
    - alla prossima invocazione
    - quando viene sollevata la Stop iteration

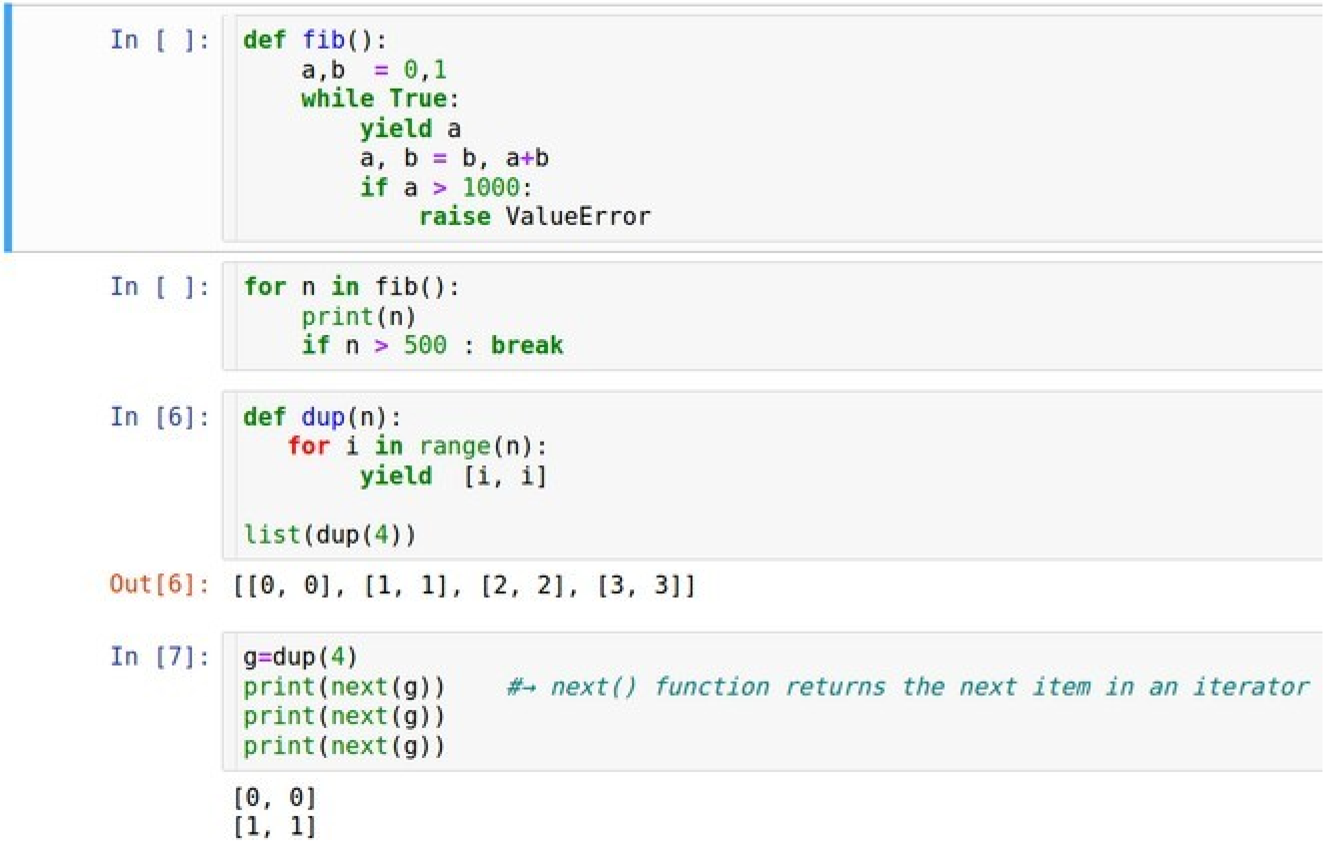
```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Wed Jul  3 09:35:12 2019


@author: fabrizio
"""
def fib():
    a,b  = 0,1
    while True:
        yield a
        a, b = b, a+b
        if a > 1000:
            raise ValueError
# %%


for n in fib():
    print(n)
    if n > 500 : break
```

```
where <expr> is an expression evaluating
to an iterable, from which an iterator
is extracted.
 The iterator is run to exhaustion,
during which time it yields and receives
values directly to or from the caller of
the generator containing the yield from
expression (the "delegating generator").
Furthermore, when the iterator is
another generator, the subgenerator is
allowed to execute a return statement
with a value, and that value becomes the
value of the yield from expression.
```

```
def dup(n):
...     for i in range(n):
...         yield from [i, i]
...
>>> dup(4)
<generator object dup at
0x7f42b52982b0>
>>> list(dup(4))
[0, 0, 1, 1, 2, 2, 3, 3]
>>>
```

```
# %%
# associo il nome g al generatore fib()
g = fib()


print(next(g))    → next() function returns the next item in an iterator
print(next(g))      next(iter, stopdef)  stopdef is returned if generator ends
print(next(g))      attn  next(fib()) non va bene richiama sempre lo stesso oggetto
```

yeld from



In [ ]:

```
def fib():
    a,b  = 0,1
    while True:
        yield a
        a, b = b, a+b
        if a > 1000:
            raise ValueError
```

In [ ]:

```
for n in fib():
    print(n)
    if n > 500 : break
```

In [6]:

```
def dup(n):
   for i in range(n):
        yield  [i, i]

list(dup(4))
```

Out[6]:

```
[[0, 0], [1, 1], [2, 2], [3, 3]]
```

In [7]:

```
g=dup(4)
print(next(g))    #→ next() function returns the next item in an iterator
print(next(g))
print(next(g))
```

```
[0, 0]
[1, 1]
```

yeld 2   also a look on itertools

- Observe that a generator object is generated once,
- **but its code is not run all at once**.
- **Only calls to next actually execute (part of) the code**.
- Execution of the code in a generator stops once a yield statement has been reached, upon which it returns a value.
- he next call to next then causes execution to continue in the state in which the generator was left after the last yield.
- This is a fundamental difference with regular functions: those always start execution at the "top" and discard their state upon returning a value.

- generator è una funzione che può operare come iterator
- Se l'iterator è finito può esaurirsi,
  - se è derivato da una funzione che non si esaurisce no

```
for x in iterator:
    do_something(x)
    break
else:
    it_was_exhausted()
```

Iterators e iterable

- yeld implementa iteratori
- Python iterator object deve implementare due metodi:
  - __iter__() e __next__(), chiamati complessivamente il protocollo iteratore.
- Notate che un oggetto è detto iterable se ne possiamo ricavare un iteratore.
  - Esempio : sono iterabili liste, tuple, stringhe NON ITERATORI.
  - Li possiamo trasformare in iteratori tramite la funzione iter() che ritorna un iteratore da questi oggetti

Iterators ed iterabile

```
a=[1,2,3,4]
next(a)
### TypeError: 'list' object is not an iterator
a=range(10)
next(a)
### TypeError: 'range' object is not an iterator
next(iter(a))
### 0
next(iter(a))
### 0   # notare che non cambia è come se lo reinizializzassi ogni volta
f=iter(a)   # lo trasformo in un iterabile
next(f)
### 0
next(f)
### 1  # Notare che cambia richiamo lo stesso iterabile non lo ricreo
next(f)
### 2
```

while

```
import random
win = 5
p1 = p2 = 0
while p1 < win and p2 < win:
    s1=random.randint(1,6) #i giocatori tirano i dadi
    s2=random.randint(1,6) #i giocatori tirano i dadi
    print(s1,s2)
    if s1>s2:    #valuto gli score ed eventualmente assegno un punto
        p1 += 1
        print('un punto a 1')
    elif s1<s2:
        p2 += 1
        print('un punto a 2')
    else:
        print('patta')
```

Loops: break, continue, else

- break and continue like C
- else after loop exhaustion

```
for n in range(2,10):
    for x in range(2,n):
        if n % x == 0:
            print (n, 'equals', x, '*', n/x)
            break
    else:
        # con il break esce dal for e di conseguenza non esegue else
        # se c'è break e non viene mai preso allora viene eseguito else
        # loop fell through without finding a factor
        print (n, 'is prime')

while True:
 if k == 3:
  break
```

Do nothing

- pass does nothing
- syntactic filler

```
while 1:
        pass
```

Input

- # Get some input from the user.
- variable = input('Please enter a value: ')
- # Do something with the value that was entered.

Liste e tuple

Liste – mutevoli - enumerabili

- Insieme ORDINATO di elementi eterogenei
- A compound data type:
  [0]
  [2.3, 4.5]
  [5, "Hello", "there", 9.8]
  []
- Eterogeneo
  vuol dire …. qualsiasi …
-

```
 import math
l=[math , math.sin ,type(25.)]
#una lista che contiene
# un modulo (l[0]),
# una funzione ed un tipo di dati
```

- Funzioni utili
  - sum(l)
    - Somma gli elementi della lista
    - l = list(range(1,10))
    - sum(l) 45
    - __builtin__.sum(l,5) valore iniziale diverso da zero
  - len(l) → ritorna la lunghezza di una lista
    - > names = [“Ben", “Chen", “Yon"]
    - len(names)   → 3

[ ] to index items in the list

```
names = ['Ben','Chen','Yaquin']
>>> names[0]
‘Ben'
>>> names[1]
‘Chen'
>>> names[2]
‘Yaqin'
>>> names[3]
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
IndexError: list index out of range
>>> names[-1]
‘Yaqin'
>>> names[-2]
‘Chen'
>>> names[-3]
‘Ben'
names[100:1000:1]
```

[0] is the first item.
[1] is the second item
...

Out of range values
raise an exception

Negative values
go backwards from
the last element.

Liste – mutevoli - iterabili

- Lists can be heterogeneous

  a = ['spam', 'eggs', 100, 1234, 4, 5]   a[4:2:-1]
  b = [] # lista vuota
  a = [d,b,c]   != a = ['d','b','c']

- Lists can be indexed and sliced: [start: stop: step]
  - a[0] → 'spam'
  - a[1:3] → ['eggs', 100]
  - a[:2] → ['spam', 'eggs']
  - a[2:] → [100, 1234, 4,5]
  - a[:] → ['spam', 'eggs', 100, 1234, 4, 5]

- Lists can be manipulated

  a[2] = a[2] + 23   → a[2] += 23
  a[0:0] → []
  a[:5:2] → ['spam', 100, 4]
  a[0:2] = [1,12]   → [1,12, 100, 1234, 2*2,
  b = a[:] # b è una copia di a

Alcune operazioni

copia di una stringa con list

```
In [31]: #per creare una lista e riempirla
          l1 = [1 , 'ciao' , 10.6]
          L2 = l1[:]
          l2 = list(l1) #creo una copia della lista passata come argomento
      print(l1,l2)
   [1, 'ciao', 10.6] [1, 'ciao', 10.6]
```

la funzione list restituisce una lista a partire da altri tipi di sequenze:

```
In [32]: l=list('stringa')
print(l)
['s', 't', 'r', 'i', 'n', 'g', 'a']
I range e le liste di interi
```

Metodi sulle liste

- Funzioni member (**in-place**)
  - list.**append**(elem) -- adds a single element to the end of the list. Common error: does not return the new list, just modifies the original
  - list.**insert**(index, elem) -- inserts the element at the given index, shifting elements to the right.
  - list.**extend**(list2) adds the elements in list2 to the end of the list. Using + or += on a list is similar to using extend()
  - list.**index**(elem) -- searches for the given element from the start of the list and returns its index. Throws a ValueError if the element does not appear (use "in" to check without a ValueError)
  - list.**remove**(elem) -- searches for the first instance of the given element and removes it (throws ValueError if not present)
  - list.**sort**() -- sorts the list in place (does not return it). (The sorted() function shown below is preferred.)
  - list.**reverse**() -- reverses the list in place (does not return it)
  - list.**count**(x) how many time element x apperar il the list
- list.pop(**index**) -- removes and returns the element at the given index. Returns the rightmost element if index is omitted (roughly the opposite of append()). items in list
  - create stack (FIFO), or queue (LIFO) → pop(0)

Sort particolari   il parametro key e reversed

Specifica un criterio di ordinamento diverso da quello standard

data una lista di stringhe

```
In [138]:l=['aa','Assfsfsffssw','asdasf','SDDFhoiumn,mi','DGGuuyDSD','dSSDSD']
In [141]:sorted(l,key=len)   # ordino la lista per lunghezza
Out[141]: ['aa', 'asdasf', 'dSSDSD', 'DGGuuyDSD', 'Assfsfsffssw', 'SDDFhoiumn,mi']
```

```
In [142]:def mykey(s):   #Ordino in base al contenuto del secondo carattere
               return s[1]  #restituisco il secondo carattere
         sorted(l,key=mykey)  # ho usato come key la funzione mykey
Out[142]: ['SDDFhoiumn,mi', 'DGGuuyDSD', 'dSSDSD', 'aa', 'Assfsfsffssw', 'asdasf']
```

```
In [144]: def mykey(s):  # voglio trascurare le maiuscole e le minuscole:
               return s[1].lower() #restituisco il secondo carattere
          sorted(l,key=mykey)       sorted(l,key=lambda s: s[1].lower())
Out[144]: ['aa', 'SDDFhoiumn,mi', 'DGGuuyDSD', 'Assfsfsffssw', 'asdasf', 'dSSDSD']
```

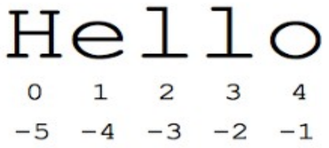
Liste e stringhe condividono qualcosa

- Plus operator overloaded
  - concatenate with + or neighbors
    - `word = 'Help' + ‘x’ → ‘Helpx’`
    - `word = 'Help' 'a' → ‘Helpa’`
- subscripting
  - `'Hello'[2]` → 'l'
  - slice: `'Hello'[1:3]` → 'el'
  - `word[-1]` → last character
  - `len(word)` → 5
  - immutable: cannot assign to subscript ← e qualcosa no !
    - ~~'hello'[1]='a'~~ 'ha' + 'hello'[2:] 'ha' 'llo'

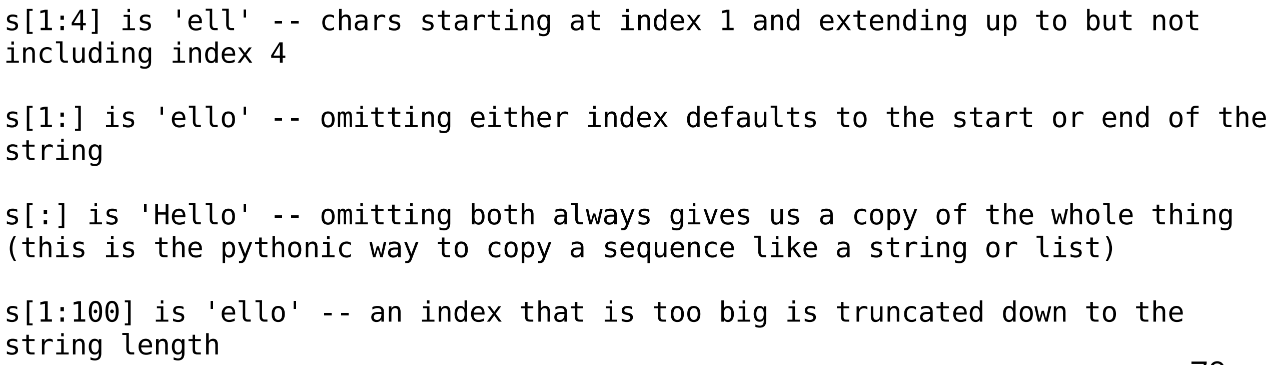
```
Strings are read only
```

Le stringhe condividono con le liste alcuni comportamenti

```
>>> smiles = "C(=N)(N)N.C(=O)(O)O"
>>> smiles[0]
'C'
>>> smiles[1]
'('
>>> smiles[-1]
'O'
>>> smiles[1:5]
'(=N)'
>>> smiles[10:-4]
'C(=O)'
```



Use “slice” notation to get a substring



```
s[1:4] is 'ell' -- chars starting at index 1 and extending up to but not
including index 4

s[1:] is 'ello' -- omitting either index defaults to the start or end of the
string

s[:] is 'Hello' -- omitting both always gives us a copy of the whole thing
(this is the pythonic way to copy a sequence like a string or list)

s[1:100] is 'ello' -- an index that is too big is truncated down to the
string length
```

Inserimenti nelle liste

- l.append(a)   inserisce un elemento a in fondo alla lista
- l.extend(k)   inserisce un elemento k di tipo lista in l
- l.insert(3,'nuovo')
  - [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 'aa', 'bb', 'cc']
  - l = [0, 1, 2, 'nuovo', 3, 4, 5, 6, 7, 8, 9, 'aa', 'bb', 'cc']

- **Via slices**
  - l[4:4]=['x','x','x','x']   # no sovrapposizione
    - [0, 1, 2, 'nuovo', 'x', 'x', 'x', 'x', 3, 4, 5, 6, 7, 8, 9, 'aa', ...
  - l[4:] = ['x','x','x','x'] # sovrapposizione
    - [0, 1, 2, 'nuovo', 'x', 'x', 'x', 'x']

del – rimuovere elementi di lista

- remove by index, not value
- remove slices from list (rather than by assigning an empty list)

```
>>> a = [-1,1,66.6,333,333,1234.5]
>>> del a[0]
>>> a
[1,66.6,333,333,1234.5]
>>> del a[2:4]
>>> a
[1,66.6,1234.5]
```

List comprehension

Sintassi: parentesi quadre che contengono un'espressione seguita da una proposizione for seguita da zero o più proposizioni for o if.
Risultato → una nuova lista risultante dalla valutazione della espressione nel contesto delle proposizioni for ed if che la seguono

```
vsq=[]
 #inizializzo
for val in range(10,100,10): #itero
   vsq.append(val**2)
 #aggiungo
```

Effetti speciali

```
#aggiungiamo una proposizione if ed estraiamo solo i valori tra 1000 e 5000
vsq = [v**2 for v in range(10,100,10) if (v**2 > 1000.0) and (v**2 < 5000.0) ]
print(vsq)   → [1600, 2500, 3600, 4900]
```

```
print(vsq)
[100, 400, 900, 1600, 2500, 3600, 4900, 6400, 8100]
In [89]: vsq = [v**2 for v in range(10,100,10)] # in un colpo solo
print(vsq)
[100, 400, 900, 1600, 2500, 3600, 4900, 6400, 8100]

 l = [-1,-20,40,-90 , 80, -50]
print(l)
#valore assoluto
l1 = [abs(x) for x in l]
print(l,l1)   → [-1, -20, 40, -90, 80, -50] [1, 20, 40, 90, 80, 50]
```

List comprehensions (2.0)

- Create lists without `map(), filter(), lambda`
- = expression seguita
  - da una clausola for
  - zero o +
    - clausole for
    - oppure clausole if

```
>>> vec = [2,4,6]
>>> [3*x for x in vec]
[6, 12, 18]
>>> [{x: x**2} for x in vec]
[{2: 4}, {4: 16}, {6: 36}]
```

List comprehensions : more than a for clause

- cross products:

```
# Similar to have
for x in vec1
   for y in vec2
```

```
>>> vec1 = [2,4,6]
>>> vec2 = [4,3,-9]
>>> [x*y for x in vec1 for y in vec2]
[8,6,-18, 16,12,-36, 24,18,-54]
>>> [x+y for x in vec1 for y in vec2]
[6,5,-7,8,7,-5,10,9,-3]
>>> [vec1[i]*vec2[i] for i in range(len(vec1))]
[8,12,-54]
```

List comprehensions

- possiamo anche usare `if`:

```
>>> [3*x for x in vec if x > 3]
[12, 18]
>>> [3*x for x in vec if x < 2]
[]

[x for b in a for x in b]
```

List comprehension ed operatore ternario

- la = **3 if b == c else 4** ← **Operatore ternario**
  - assign a to **this if cond else** assign a to **that**
- uso di operatore ternario in una list comprehension
- In [94]: [(v**2 if v<5 else v**3) for v in range(1,10)]
- Out[94]: → [1, 4, 9, 16, 125, 216, 343, 512, 729]
- Notare che è come dire
  - (falseValue, trueValue)[bool(testexp)]
  - a=(4,3)[bool(b==c)]

Tuple iterabili – non mutevoli

- A differenza delle liste NON SONO MODIFICABILI IN-PLACE (sono non-mutable).
- vogliamo essere sicuri che il contenuto di una sequenza non vari durante la sua esistenza!
- possono contenere oggetti di tipo eterogeneo
- supportano gli stessi meccanismi di indicizzazione delle liste (in particolare supportano gli slice)
- sono iterabili

Creare una tupla

- parentesi **tonde**, elemento per elemento, separati da virgola
  - #definisco una tupla usando le parentesi tonde
  - #NB: elementi eterogenei
  - t1 = (1,'ciao',65.23,[1,2,3])
- parentesi tonde sono **opzionali**, una serie di elementi separati da virgola e' interpretata come tupla
  - t2 = 1,'ciao',65.23,[1,2,3]
- Operatore + e *
  - t2 = t1 * 3
  - print (t2) –→
    (1, 'ciao', 65.23, [1, 2, 3], 1, 'ciao', 65.23, [1, 2, 3], 1, 'ciao', 65.23, [1, 2, 3])
- Slicing → che ci restituisce una tupla
  - print(t1[1]) → ciao   # E' il secondo elemento
  - print(t1[1:3]) → ('ciao', 65.23)

Tuple

- tupla vuota: ()

```
>>> empty = ()
>>> len(empty)
0
```

- un elemento solo → virgola in fondo
- `>>> singleton = 'foo',`
- `list((3,))`

Tuple iterazione

- È iterabile perciò

```
for elem in t1:
   print(elem)
```

- È non mutevole
  - t1 = (1,'ciao',65.23,[1,2,3])
  - t1[0] = 0 # ERRORE ♠
  - #anche quando e' mutable (t1[3], list)
  - t1[3] = [] # ERRORE ♠
  - t1[3][1]=10*t1[3][1]
    # il quarto elemento è una lista è modificabile in place → chi dice perchè una bambolina !!

Tuple assignment in for loops

```
data = [ ("C20H20O3", 308.371),
    ("C22H20O2", 316.393),
    ("C24H40N4O2", 416.6),
    ("C14H25N5O3", 311.38),
    ("C15H20O2", 232.3181)]
```

```
for (formula, mw) in data:
    print( "The molecular weight of %s is %s" % (formula, mw))

print(f"The molecular weight of {formula} is {mw}")
```

```
The molecular weight of C20H20O3 is 308.371
The molecular weight of C22H20O2 is 316.393
The molecular weight of C24H40N4O2 is 416.6
The molecular weight of C14H25N5O3 is 311.38
The molecular weight of C15H20O2 is 232.3181
```

Allora le stringhe sono …. più che liste strane.. tuple di caratteri

- tuple di soli caratteri e pertanto non mutevoli
- indicizzabili
- slicizzabili
- + e * come per le liste
- iterabili
- con '   '   → consente dentro “
- con “”” “”” → multilinea consente dentro ' e “
- conversione a stringa con str()

String Methods: find, split

```
smiles = "C(=N)(N)N.C(=O)(O)O"
>>> smiles.find("(O)")
15
>>> smiles.find(".")
9
>>> smiles.find(".", 11)
-1    (non lo trova)
>>> k = smiles.split(".")
['C(=N)(N)N', 'C(=O)(O)O']
>>>  ‘.’.join(k)   → "C(=N)(N)N.C(=O)(O)O"
```

String Method: “strip”, “rstrip”, “lstrip” are ways to remove whitespace or selected characters

```
>>> line = " # This is a comment line \n"
>>> line.strip()
'# This is a comment line \n'
>>> line.lstrip()
'# This is a comment line \n'
>>> line.rstrip("\n")
' # This is a comment line '
>>>
```

Funzioni membro

- upper, lower
  - s = 'questa è una stringa minuscola'
  - S = s.upper()
  - S = S.replace('MINUSCOLA','MAIUSCOLA')
  - print(s) → questa è una stringa minuscola
  - print(S) → QUESTA È UNA STRINGA MAIUSCOLA
- s.count('s'),s.count('S') → (3, 0)
- 'stringa' in s , 'Stringa' in s → (True, False)
- s.find('stringa') ,s.find('Stringa') → (13, -1)

Altri metodi vedi anche Python string methods

- s.lower(), s.upper() -- returns the lowercase or uppercase version of the string
- s.strip() -- returns a string with whitespace removed from the start and end
- s.isalpha()/s.isdigit()/s.isspace()... -- tests if all the string chars are in the various character classes
- s.startswith('other'), s.endswith('other') -- tests if the string starts or ends with the given other string
- s.find('other') -- searches for the given other string (not a regular expression) within s, and returns the first index where it begins or -1 if not found
- s.replace('old', 'new') -- returns a string where all occurrences of 'old' have been replaced by 'new'
- s.split('delim') -- returns a list of substrings separated by the given delimiter. The delimiter is not a regular expression, it's just text. 'aaa,bbb,ccc'.split(',') -> ['aaa', 'bbb', 'ccc']. As a convenient special case s.split() (with no arguments) splits on all whitespace chars.
- s.join(list) -- opposite of split(), joins the elements in the given list together using the string as the delimiter. e.g. '---'.join(['aaa', 'bbb', 'ccc']) -> aaa---bbb---ccc
- **Python does not have a separate character type.**

String operators: in, not in

```
if "Br" in "Brother":
  print ("contains brother")

email_address = “clin”
if "@" not in email_address:
  email_address += "@brandeis.edu"
```

Liste in stringhe e stringhe in liste

```
l = list('abcde')  → [ 'a',  'b',' c', 'd', 'e']
list1 = ['1', '2', '3']
str1 = ''.join(list1) → '123'
# aggrego gli elementi di una lista con un elemento  separatore in questo caso
# una stringa vuota
s = {'a','b','c'}
ss=""
for k in s:
   ss += k
print(ss)
```

zipping lists together

```
>>> names
['ben', 'chen', 'yaqin']

>>> gender = [0, 0, 1]
>>> a= zip(names, gender)
      – è come un enumerator
list (a)

  [('ben', 0), ('chen', 0), ('yaqin', 1)]
```

la lunghezza delle liste non deve necessariamente essere uguale viene presa la più corta

Di uso molto comune come prodromo per trasformare un insieme di liste in un dizionario. e.g. Il database

Tuples e sequenze

- lists, strings, **tuples**: sono esempi di sequenza
- tuple = valori anche solo da ,

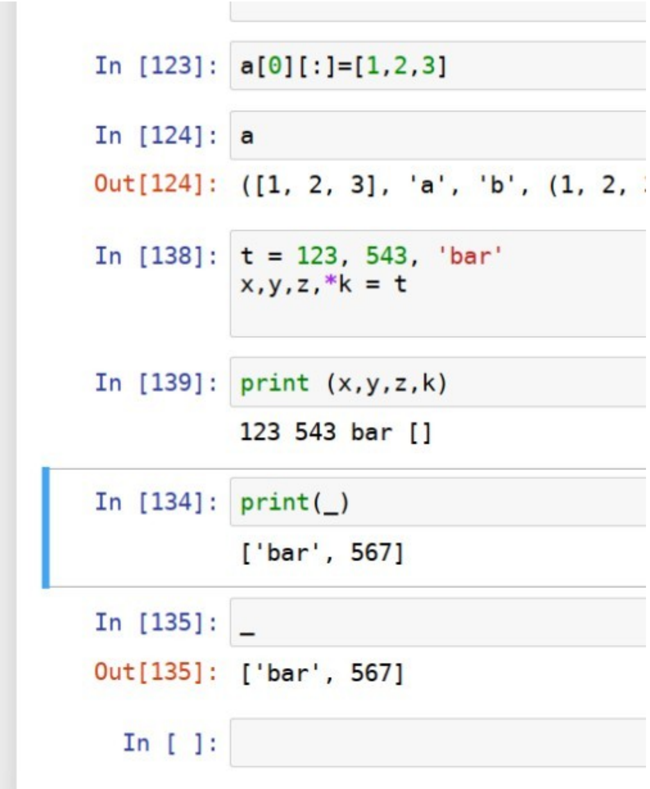
```
>>> t = 123, 543, 'bar'
>>> t[0]
123
>>> t
(123, 543, 'bar')
```

Unpack Tuples

- sequence unpacking → distribute elements across variables

```
>>> t = 123, 543, 'bar'
>>> x, y, z = t
>>> x → 123
for indx,val in enumerate(t):
x,*y = t  → x→ 123 y = [543,'bar']
```

- packing always creates tuple
- unpacking works for any sequence



```
In [123]: a[0][:]=[1,2,3]

In [124]: a
Out[124]: ([1, 2, 3], 'a', 'b', (1, 2,

In [138]: t = 123, 543, 'bar'
          x,y,z,*k = t

In [139]: print (x,y,z,k)
          123 543 bar []

In [134]: print(_)
          ['bar', 567]

In [135]: _
Out[135]: ['bar', 567]

In [ ]:
```

starred assignment

In Python 3, la possibilità di lavorare con un numero indefinito di parametri è stata estesa anche all'operatore di assegnamento

```
l = list(range(5))
a,*b = l
print(f'{a=}')
print('b=',b)
---------------------------------
a= 0  # notare che è un singolo elemento
b= [1, 2, 3, 4]  # notare che è una lista!
```

Formattare stringhe

- vedi esempi di formattazione
  - operatore %
    - "hi there %s" % (name,)
  - operatore format
    - “hi there {}”.format(name)   > 2.5
    - “hi there {0} {1}”.format(name,surname)
    - “hi there {0} {1} from {pr}”.format(name,surname,pr = "Roma"
    - {field_name:conversion}
  - operatore f
    - f”{nome:} è il mio nome”

Formattazione con f → Python >=3.6

- import datetime
- >>> name = 'Fred'
- >>> age = 50
- >>> anniversary = datetime.date(1991, 10, 12)
- >>> f'My name is {name}, my age next year is {age+1}, my anniversary is {anniversary:%A, %B %d, %Y}.'
  - 'My name is Fred, my age next year is 51, my anniversary is Saturday, October 12, 1991.'
  - {espressione o valore:**formato**}
- >>> f'He said his name is {name!r}.'
  - "He said his name is 'Fred'."
  - Novità con 3.8
    - user = 'fab'; member_since = 1959
    - f'{user=} {member_since**=**}' → "user='fab' member_since=1959"

Formattare stringhe con .format

- per **posizione**
  - '{} – zz – {}, {}'.format('a',1.0,100) → 'a – zz – 1.0, 100'
    # parentesi graffe come segnaposto
    # se vuote inserisco gli argomenti di format
    # nell'ordine in cui sono elencati
- per **indice**
  - '{2},{0},{1}'.format('a',1.0,100) → 100,a,1.0
  - # dentro i segnaposto ci metto indici
    # argomenti 0 il primo, 1 il secondo etc.
    # così posso cambiare l'ordinamento dei parametri
    # o ripetere più volte un elemento
    - '{2},{2},{1}'.format('a',1.0,100)
- per **nome**
  - '{c},{a},{b}'.format(a= 'a',b= 1.0,c= 100)

```
data={'da': da, 'a': a[0], 'm': arg}
message = '''From: From {} <{da}>
   To: To {a} <{a}>
   Subject: Message from Interchange
   {m}
    '''.format(**data)
```

Formattazione con format meno comuni

- #ASSOCIAZIONE PER INDICE+ATTRIBUTO
- c = 1+2j
  - 'parte reale: {0.real}, parte immaginaria: {0.imag}'.format(c)
  - #NB: non funziona con le funzioni membro
  - 'parte reale: 1.0, parte immaginaria: 2.0'
- #ASSOCIAZIONE PER INDICE + INDICE
  - l=[1,'due',3.0]
  - 'primo elemento: {0[0]}, secondo: {0[1]}, terzo: {0[2]}'.format(l)
  - 'primo elemento: 1, secondo: due, terzo: 3.0'

Formattazione con %   - metodo antico

- '%s questo è %s' % ('one', 'two')
- '%d %d' % (1, 2)
- '%d' % (1,)
- vedi esempi a https://pyformat.info/

\ → per i caratteri speciali

```
\n -> newline
\t -> tab
\\ -> backslash
...
But Windows uses backslash for directories!
filename = "M:\nickel_project\reactive.smi" # DANGER!
filename = "M:\\nickel_project\\reactive.smi" # Better!
filename = "M:/nickel_project/reactive.smi" # Usually works

filename = r'M:\nickel_project\reactive.smi'
           mette le backslash as is non le considera escape
```

Formattare date

- %a: Returns the first three characters of the weekday, e.g. Wed.
- %A: Returns the full name of the weekday, e.g. Wednesday.
- %B: Returns the full name of the month, e.g. September.
- %w: Returns the weekday as a number, from 0 to 6, with Sunday being 0.
- %m: Returns the month as a number, from 01 to 12.
- %p: Returns AM/PM for time.
- %y: Returns the year in two-digit format, that is, without the century. For example, "18" instead of "2018".
- %f: Returns microsecond from 000000 to 999999.
- %Z: Returns the timezone.
- %z: Returns UTC offset.
- %j: Returns the number of the day in the year, from 001 to 366.
- %W: Returns the week number of the year, from 00 to 53, with Monday being counted as the first day of the week.
- %U: Returns the week number of the year, from 00 to 53, with Sunday counted as the first day of each week.
- %c: Returns the local date and time version.
- %x: Returns the local version of date.
- %X: Returns the local version of time.

Formattare numeri

Conversion	Meaning	Notes
d	Signed integer decimal.	
i	Signed integer decimal.	
o	Unsigned octal.	(1)
u	Unsigned decimal.	
x	Unsigned hexadecimal (lowercase).	(2)
X	Unsigned hexadecimal (uppercase).	(2)
e	Floating point exponential format (lowercase).	
E	Floating point exponential format (uppercase).	
f	Floating point decimal format.	
F	Floating point decimal format.	
g	Same as "e" if exponent is greater than -4 or less than precision, "`f`" otherwise.	
G	Same as "E" if exponent is greater than -4 or less than precision, "F" otherwise.	
c	Single character (accepts integer or single character string).	
r	String (converts any python object using `repr()`).	(3)
s	String (converts any python object using `str()`).	(4)
%	No argument is converted, results in a "%" character in the result.	

Formattazioni particolari

Flag	Meaning
#	The value conversion will use the ``alternate form'' (where defined below).
0	The conversion will be zero padded for numeric values.
-	The converted value is left adjusted (overrides the "0" conversion if both are given).
	(a space) A blank should be left before a positive number (or empty string) produced by a signed conversion.
+	A sign character ("+" or "-") will precede the conversion (overrides a "space" flag).

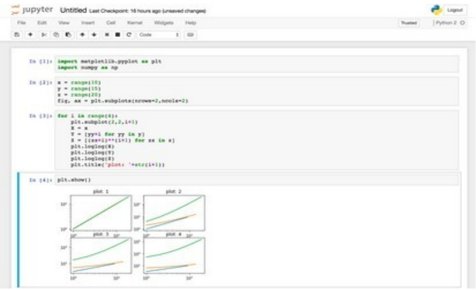
- (1) The alternate form causes a leading zero ("0") to be inserted between left-hand padding and the formatting of the number if the leading character of the result is not already a zero.
- (2) The alternate form causes a leading '0x' or '0X' (depending on whether the "x" or "X" format was used) to be inserted between left-hand padding and the formatting of the number if the leading character of the result is not already a zero.
- (3) The %r conversion was added in Python 2.0.
- (4) If the object or format provided is a unicode string, the resulting string will also be unicode.

Esempi dagli esercizi

```
def front_x(words):
 # +++your code here+++
 '''
 lista_di_quelli_cominciano_perx = [i for i in words if i[0] == 'x']
 lista_di_quelli_cominciano_perx.sort()
 lista_div = [i for i in words if i[0] != 'x']
 lista_div.sort()
 return lista_di_quelli_cominciano_perx + lista_div
 -----------------------------
 x = []
 nonx = []
 for i in words:
     if i[0] == 'x':
         x.append(i)
        else:
         nonx.append(i)
 x.sort()
 nonx.sort()
 return x + non_x
 '''

 return sorted(words,key=lambda x: '0' + x[1:] if x[0]== 'x' else x)
```

Le Date

vedi jupyter
notebook
datetime



```
In [1]: import matplotlib.pyplot as plt
        import numpy as np

In [2]: x = range(10)
        y = range(15)
        z = range(20)
        fig, ax = plt.subplots(nrows=2,ncols=2)

In [3]: for i in range(4):
            plt.subplot(2,2,i+1)
            X = x
            Y = [yy+i for yy in y]
            Z = [(zz+i)**(i+1) for zz in z]
            plt.loglog(X)
            plt.loglog(Y)
            plt.loglog(Z)
            plt.title('plot: '+str(i+1))

In [4]: plt.show()
```

Dizionari

Dictionaries

- like Tcl or awk associative arrays
- indexed by keys
- keys are any immutable type: e.g., tuples
- but not lists (mutable!)
- uses 'key: value' notation

```
>>> tel = {'hgs' : 7042, 'lennox': 7018}
    tel = {}    tel = dict()
    tel['hgs']
    tel['hgs'] = 7042
>>> tel['cs'] = 7000 # assegn. nuova
>>> tel → {'hgs' : 7042, 'lennox': 7018, 'cs' : 7000}
```

Dictionaries

- no particular order
- delete elements with del

```
>>> del tel['foo']
```

- keys() method → unsorted list of keys

```
>>> tel.keys()
['cs', 'lennox', 'hgs']
```

- use has_key() to check for existence

```
>>> tel.has_key('foo') → 0
no in python 3    'foo' in tel
for i in tel : print(tel[i])
```

Dictionaries

- Dictionaries are lookup tables.
- They map from a “key” to a “value”.

  -
    ```
    symbol_to_name = {
    "H": "hydrogen",
    "He": "helium",
    "Li": "lithium",
    "C": "carbon",
    "O": "oxygen",
    "N": "nitrogen"
    }
    ```
- Duplicate keys are not allowed
- Duplicate values are just fine

Dictionaries

Le chiavi possono essere qualsiasi immutable : numbers, strings, tuples, frozenset.

Not mutables list, dictionary, set, ...

```
atomic_number_to_name = {
1: "hydrogen"
6: "carbon",
7: "nitrogen"
8: "oxygen",
}
nobel_prize_winners = {
(1979, "physics"): ["Glashow", "Salam", "Weinberg"],
(1962, "chemistry"): ["Hodgkin"],
(1984, "biology"): ["McClintock"],
}
```

A set is an ~~unordered~~ collection with no duplicate elements.

A frozenset is the application of frozenset method

`frozenset([iterable])`

Dizionari

```
>>> symbol_to_name["C"] → 'carbon'  #Get the value for a given key
>>> "O" in symbol_to_name, "U" in symbol_to_name   # Test if the key
   exists                                (“in” only checks the keys,not
   the values.)
(True, False)
>>> "oxygen" in symbol_to_name
False
>>> symbol_to_name["P"]
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
KeyError: 'P'
>>> symbol_to_name.get("P", "unknown")
'unknown'
>>> symbol_to_name.get("C", "unknown")
'carbon'
```

[] lookup failures raise an exception.
Use “.get()” if you want
to return a default value.

Metodi utili per i dizionari

```
>>> symbol_to_name.keys()
['C', 'H', 'O', 'N', 'Li', 'He']

>>> symbol_to_name.values()
['carbon', 'hydrogen', 'oxygen', 'nitrogen', 'lithium', 'helium']

>>> symbol_to_name.update( {"P": "phosphorous", "S": "sulfur"} )
>>> symbol_to_name.items()
[('C', 'carbon'), ('H', 'hydrogen'), ('O', 'oxygen'), ('N', 'nitrogen'), ('P', 'phosphorous'), ('S', 'sulfur'), ('Li', 'lithium'), ('He',
   'helium')]

>>> del symbol_to_name['C']
>>> symbol_to_name
{'H': 'hydrogen', 'O': 'oxygen', 'N': 'nitrogen', 'Li': 'lithium', 'He': 'helium'}
>>> romanNums = {'I':1, 'II':2, 'III':3, 'IV':4, 'V':5 }
>>> value = romanNums.setdefault('I',””) # Se la chiave I esiste ritorna il valore che c’è
                                         # se non esiste inserisce la chiave  con il valore (secondo parametro fornito se non fornito none)

print("The return value is: ", value)
```

Nuove operazioni con i dizionari >=3.9

- Merge
  - dict1 = {'a': 'Cat', 'b': 10}
  - dict2 = {'c': 'Dog', 'd': 11}
  - d3 = d1 | d2
  - d1 |= d2
  -
  - dict4 = {**dict1, **dict2}

Set

- A set is a collection (**of not mutable elements recursive -hash-**) which is unordered and unindexed. In Python sets are written with curly brackets.
  - Example
  - Create a Set:
    - thisset = {"apple", "banana", "cherry"}
    - print(thisset)
    - Note: Sets are unordered, so you cannot be sure in which order the items will appear.
- Access Items
  - You cannot access items in a set by referring to an index, since sets are unordered the items has no index But you can loop through the set items using a for loop, or ask if a specified value is present in a set, by using the in keyword.
  - Example
    - Loop through the set, and print the values:
    - thisset = {"apple", "banana", "cherry"}
    - for x in thisset:
    -   print(x)
  - Example: check if "banana" is present in the set:
    - thisset = {"apple", "banana", "cherry"}
    - print("banana" in thisset)
- Change Items
  - Once a set is created, you cannot change its items, but you can add new items.

Set 2: Add Items

- To add one item to a set use the add() method.
  - To add more than one item to a set use the update() method.
  - Example: add an item to a set, using the add() method:
    - thisset = {"apple", "banana", "cherry"}
    - thisset.add("orange")
    - print(thisset)
  - Example: add multiple items to a set, using the update() method:
    - thisset = {"apple", "banana", "cherry"}
    - thisset.update(["orange", "mango", "grapes"])
    - print(thisset)
- Get the Length of a Set
  - To determine how many items a set has, use the len() method.
  - Example : get the number of items in a set:
    - thisset = {"apple", "banana", "cherry"}
    - print(len(thisset))

Set 3 : Remove Item

- To remove an item in a set, use the remove(), or the discard() method.
  - Example: remove "banana" by using the remove() method:
    - thisset = {"apple", "banana", "cherry"}
    - thisset.remove("banana")
    - print(thisset)
  - Note: If the item to remove does not exist, remove() will raise an error.
  - Example: remove "banana" by using the discard() method:
    - thisset = {"apple", "banana", "cherry"}
    - thisset.discard("banana")
    - print(thisset)
  - Note: If the item to remove does not exist, discard() will NOT raise an error.
  - You can also use the pop(), method to remove an item, but this method will remove the last item. Remember that sets are unordered, so you will not know what item that gets removed. The return value of the pop() method is the removed item.
  - Example :remove the last item by using the pop() method:
    - thisset = {"apple", "banana", "cherry"}
    - x = thisset.pop()
    - print(x)
    - print(thisset)
  - Note: Sets are unordered, so when using the pop() method, you will not know which item that gets removed.
  - Example : the clear() method empties the set:
    - thisset = {"apple", "banana", "cherry"}thisset.clear()
    - print(thisset)
    - Example : the del keyword will delete the set completely:
    - thisset = {"apple", "banana", "cherry"}
    - del thisset
    - print(thisset)

Set 4 Join Two Sets

- There are several ways to join two or more sets in Python.
  - You can use the union() method that returns a new set containing all items from both sets, or the update() method that inserts all the items from one set into another:
  - Example : the union() method returns a new set with all items from both sets:
    - set1 = {"a", "b" , "c"}
    - set2 = {1, 2, 3}
    - set3 = set1.union(set2)
    - print(set3)
  - Example : the update() method inserts the items in set2 into set1:
    - set1 = {"a", "b" , "c"}
    - set2 = {1, 2, 3}
    - set1.update(set2)
    - print(set1)
  - Note: Both union() and update() will exclude any duplicate items.
    - There are other methods that joins two sets and keeps ONLY the duplicates, or NEVER the duplicates, check the full list of set methods in the bottom of this page.
- **The set() Constructor**
  - It is also possible to use the set() constructor to make a set.
  - Example : using the set() constructor to make a set:
    - thisset = set(("apple", "banana", "cherry")) # note the double round-brackets
    - print(thisset)

Set methods

Method	Description
add()	Adds an element to the set
clear()	Removes all the elements from the set
copy()	Returns a copy of the set
difference()	Returns a set containing the difference between two or more sets
difference_update()	Removes the items in this set that are also included in another, specified set
discard()	Remove the specified item
intersection()	Returns a set, that is the intersection of two other sets
intersection_update()	Removes the items in this set that are not present in other, specified set(s)
isdisjoint()	Returns whether two sets have a intersection or not
issubset()	Returns whether another set contains this set or not
issuperset()	Returns whether this set contains another set or not
pop()	Removes an element from the set
remove()	Removes the specified element
symmetric_difference()	Returns a set with the symmetric differences of two sets
symmetric_difference_update()	inserts the symmetric differences from this set and another
union()	Return a set containing the union of sets
update()	Update the set with the union of this set and others

Comparing sequences

- unlike C, can compare sequences (lists, tuples, ...)
- lexicographical comparison:
  - compare first; if different → outcome
  - continue recursively
  - subsequences are smaller
  - strings use ASCII comparison
  - can compare objects of different type, but by type name (list < string < tuple)

Comparing sequences

```
(1,2,3) < (1,2,4)
[1,2,3] < [1,2,4]
'ABC' < 'C' < 'Pascal' < 'Python'
(1,2,3) == (1.0,2.0,3.0)
(1,2) < (1,2,-1)
```

Dispose object

- del obj (libera il nome)
  - ma un sacco di eccezioni
- gc.collect()
  - ma evitare di usarla

Named tuple

```
>>> import collections
>>> Point = collections.namedtuple('Point', ['x', 'y'])
>>> p = Point(1, y=2)
>>> p.x, p.y
1 2
>>> p[0], p[1]
1 2
```

- per chi viene dal C
  - sembrano structs senza nomi dell'elemento:
    - (x,y) coordinates
    - database records
- ricordiamo che come le stringhe **non sono mutevoli →** non è possible effettuare assegnazioni al singolo item
- Ottime da usare in programmazione funzionale

Esempio

- Partiamo da una lista (mutevole) di dizionari (mutevoli)

```
scientist = [ {'nome' : 'Enrico Fermi', 'campo': ' fisica', 'anno': 1901},
         {'nome' : 'Ennio Maiorana', 'campo': ' fisica', 'anno': 1906},
         {'nome' : 'Federico Faggin', 'campo': ' elettronica', 'anno': 1941}
         {'nome' : 'Piergiorgio Odifreddi', 'campo': ' matematica', 'anno': 1950}

scientist[0]['nome'] → Enrico Fermi
# Se cambio scientist[0]['nome'] = 'Pasquale Fermi' # ho cambiato  gli scientist
# Se inserisco e sbaglio creo un caos scientist[0]['name'] = 'Leonardo da Vinci'
 {'nome' : 'Enrico Fermi', 'name': 'Leonardo da Vinci', 'campo': ' fisica', 'anno': 1901}

import collections
Scientist = collections.namedtuple('Scientist',['nome','campo','anno'])
Scientist(nome='Enrico Fermi', campo='fisica', anno=1901) # non abbiamo più dict key ma keywords
enrico = Scientist(nome='Enrico Fermi', campo='fisica', anno=1901)

enrico.nome → Enrico Fermi   assomiglia molto di più ad un oggetto

ancora meglio trasformare la list di scienziati in una tupla di scienziati (di named tuple)
s = (Scientist(nome= , campo= ….),)
filter(lambda x:x.anno<=1950, s)
```

Funzioni

Defining functions

```
d =512
k = 45
def fib(n):
  """  La funzione f effa
  Print a Fibonacci series up to n.
  param n:
  """
  global d
  print(d)       # → 512
  print(k)       # errore !! → 45
  print(m)       # errore !!
  a, b = 0, 1
  while b < n:
    print (b)
    a, b = b, a+b

>>> fib(2000)
```

- First line is *docstring*
- first look for variables in local, then global
- need global to assign global variables

```
def fib(n):
  """  La funzione f effa
  Print a Fibonacci series up to n.
  param n:
  """
  global d
  print(d)       # → 512

  def ff():
      return 4

  print(k)       # errore !! → 45
  print(m)       # errore !!
  a, b = 0, 1
  while b < n:
    print (b)
    a, b = b, a+b
```

Funzioni: default argument

```
def ask_ok(prompt, retries=4,complaint='Yes or no'):
  while 1:
    ok = input(prompt)
    if ok in ('y', 'ye', 'yes'): return 1
    if ok in ('n', 'no'): return 0
    retries = retries - 1
    if retries < 0:
           raise IOError, 'refusenik error'
    else :
           print (complaint)

>>> ask_ok('Really?')
ask_ok(“Dammi un valore”, 5,’Si o no’)
ask_ok(“Dammi un valore”)
```

Keyword arguments

```
   def parrot(voltage, < / | * >,state='a stiff', action='voom', type='Norwegian blue'):
def parrot(voltage, state='a stiff', action='voom', type='Norwegian blue'):
  print ("-- This parrot wouldn't", action,
         "if you put", voltage, "Volts through it.",
         "Lovely plumage, the ", type,
         "-- It's", state, "!")
parrot(1000)
parrot(action='VOOOM', voltage=100000)
~~parrot(action='VOOOM',100000)~~ ma parrot(100000, action='VOOOM')
def myFunction(a,/,b,*,c):     / tutti quelli a sx solo posizionali
   print(a,b,c)                * tutti quelli a dx solo keyword (solo>3.8)
```

Funzioni

```
>>> def f(a):
...  a = 2*a # i numeri sono immutabili
...
>>> x = 5
>>> f(x)
>>> x
5
>>> def g(a):
...  a[0] = 2*a[0] # le liste sono mutevoli
...
>>> y = [5]
>>> g(y)
>>> y
[10]
```

Qualsiasi argomento che sia una variabile
che punti ad un oggetto mutevole può
cambiare il valore di quell'oggetto
dall'interno della funzione

Nomi delle funzioni

```
>>> def f():
...  print (1)
...
>>> def g():
...  print (2)
...
>>> f()
1
>>> g()
2
>>> [f,g] = [g,f]
>>> f()
2
>>> g()
1
```

Parametri delle funzioni

- I. per posizione
- II. per nome
- III. per posizione in numero variabile
- IV. per nome in numero variabile

```
 def funz(a,b,c,d):
"""Funzione per prova passaggio argomenti
"""
print('-'*15)
print('a = ',a)
print('b = ',b)
print('c = ',c)
print('d = ',d)
return
```

Per posizione

```
: #si associa un valore ad ogni parametro in base all'ordine con cui passo gli argomenti:
funz(1,'ciao',3.55,[1,2,3,4])   def funz(a,b,c,d)
---------------
a = 1
b =  ciao
c =  3.55
d =  [1, 2, 3, 4]
```

Per nome

```
funz(a=33,b=3,c=2,d=1)    def funz(a,b,c,d)
funz(d=1,c=2,b=3,a=33)

  # prima per posizione, poi per nome si il viceversa no

funz(1, 2, d=4, c=3)
funz(1,2,a=4,c=3)  NO

funz(d=3,a=4,5,6)
```

Parametri di default

- I parametri di una funzione, possono anche avere un valore di default, specificato nella definizione, che viene assunto quando non diversamente specificato:
- **def** funz6(a=-1):
  - print('a = ',a)
  - return
  - #se non passo alcun argomento il parametro assume il valore di default
- ---------------------------------

funz6()
funz6(10)
a = -1
a = 10

Occhio ai default mutevoli
l'assegnazione avviene alla definizione
NON ogni volta che chiamo

Posizione e numero variabile di argomenti

```
    def funz2(*args):
        print('-'*15)
        for arg in args:
        print(arg)

funz2([1,2,3])
l = [1,2,3]

In [8]: funz2(1)
          funz2(1,2,3) → funz2(*l)
          funz2(1,[2,3],'ciao')
---------------
1
---------------
1
2
3
---------------
1
[2, 3]
ciao
```

```
def funz3(*argomenti):

print(type(argomenti))
      print('-'*15)
      for arg in
argomenti:
        print(arg)

funz3(1,2,3,4,5,'ciao')
<class 'tuple'>
---------------
1
2
3
4
5
ciao
```

Creare un f che riceve un numero variabile di argomenti e che produce in output un string multiline

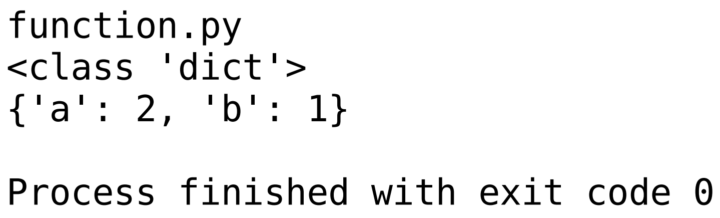
1: valore del primo argomento
2: valore del secondo argomento
3: ......

```
def funz2(*args):
    k = []
    for i in args:
      k += i
    return k

funz2('b','c','d') → bcd

l = ['b','c','d']
funz2(l) → ['b','c','d']

funz2(*l) → 'bcd'
```

*args e' uno standard in Python ma qualsiasi nome dopo l'asterisco.

Per nome in numero variabile

- Doppio asterisco nella definizione della funzione:
  - def funz4(**kwargs):
  - print(type(kwargs))
  - print(kwargs)
  - return



```
function.py
<class 'dict'>
{'a': 2, 'b': 1}

Process finished with exit code 0
```

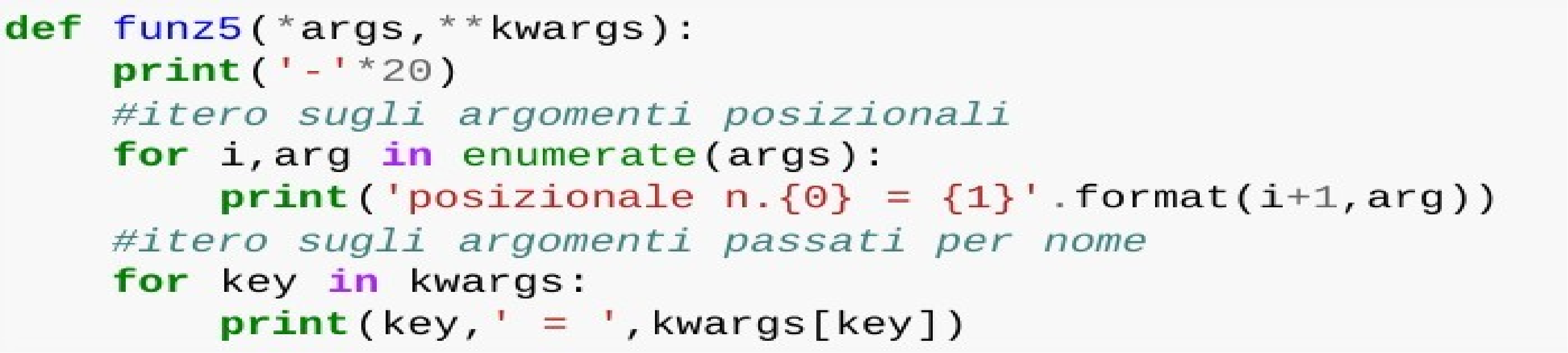
funz4(a=1,b=2)   dd= {‘a’:1,’b’:2}   funz4(**dd)

----------------

a = 1
b = 2

  - Il nome che segue il doppio asterisco) e' un dizionario che contiene come chiavi i nomi dei parametri e come valori il valori del parametro associato.

numero variabile di argomenti per posizione e per nome (*args,**kwargs)



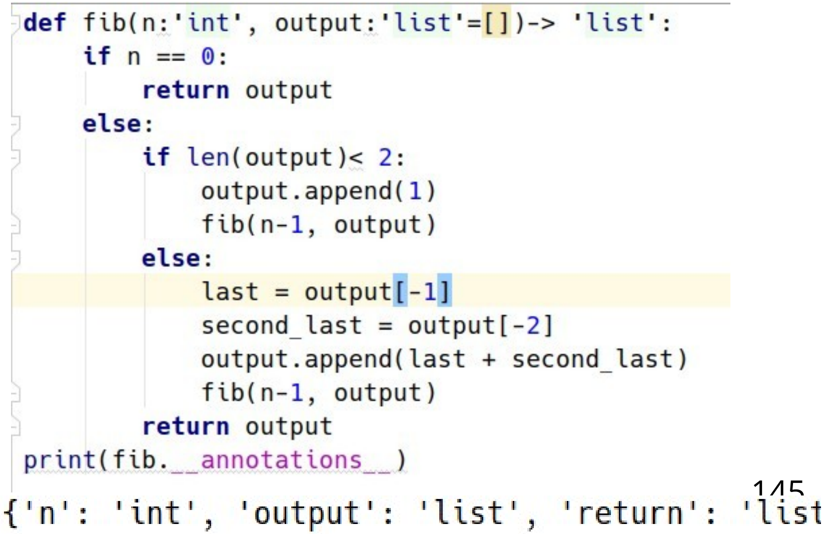
```
def funz5(*args,**kwargs):
    print('-'*20)
    #itero sugli argomenti posizionali
    for i,arg in enumerate(args):
        print('posizionale n.{0} = {1}'.format(i+1,arg))
    #itero sugli argomenti passati per nome
    for key in kwargs:
        print(key,' = ',kwargs[key])
```

Function annotations

- Annotations can be used to collect information about the type of the parameters and the return type of the function to keep track of the type change occurring in the function.
  - **[def foo(a:”int”, b:”float”=5.0) -> ”int”]**
- Python doesn't do anything with the annotations other than put them in an __annotations__ dictionary.

```
>>> def f(x: int) -> float:
...     pass
...
>>> f.__annotations__
{'return': <class 'float'>, 'x':
<class 'int'>}
```

```
def f(a: stuff, b: stuff = 2) -> result:
    ...
```



```
def fib(n:'int', output:'list'=[])-> 'list':
    if n == 0:
        return output
    else:
        if len(output)< 2:
            output.append(1)
            fib(n-1, output)
        else:
            last = output[-1]
            second_last = output[-2]
            output.append(last + second_last)
            fib(n-1, output)
        return output
print(fib.__annotations__)
```

```
{'n': 'int', 'output': 'list', 'return': 'list
```

Parameter passing

- I parametri passati a funzione sono riferimenti a oggetti e non gli oggetti stessi.
- **se il parametro e' di tipo NON-MUTABLE** la funzione (ma non solo lei) non potra' modificare in-place il valore della variabile (*).
- **se il parametro e' di tipo MUTABLE** la funzione puo' EVENTUALMENTE modificare in-place il valore della variabile in modo che tale modifica permanga una volta che la funzione e' terminata.

Return parameters

- con una tupla

```
def f(x):
  y0 = x + 1
  y1 = x * 3
  y2 = y0 ** y3
  return
(y0,y1,y2)
```

- con un dizionario

```
def g(x):
  y0 = x + 1
  y1 = x * 3
  y2 = y0 ** 3
  return {'y0':y0,
'y1':y1 ,'y2':y2 }
```

- con una classe

```
class ReturnValue(object):
  def __init__(self, y0, y1,
y2):
     self.y0 = y0
     self.y1 = y1
     self.y2 = y2
def g(x):
  y0 = x + 1
  y1 = x * 3
  y2 = y0 ** y3
  return ReturnValue(y0, y1, y2)
```

- con una lista

```
def h(x):
…...
  result = [x + 1]
  result.append(x * 3)
  result.append(x ** 3)
  return result
```

- con una named tuple

```
>>> import collections
>>> Point = collections.namedtuple('Point', ['x', 'y'])
>>> p = Point(1, y=2)
>>> p.x, p.y
1 2
>>> p[0], p[1]
1 2
```

Scoping

- Lo scope di una variabile e' determinato da dove questa viene definita.
  - se una variabile e' definita nella funzione stessa si dice **locale** alla funzione
  - se una variabile e' definita in una funzione che racchiude la funzione alla quale ci si riferisce (enclosing def) la funzione si dice **nonlocale** alla funzione
  - se una variabile e' definita all'esterno di tutte le funzioni, e' detta variabile **globale**
- La questione degli scope ha rilevanza a proposito:
  - dove posso usare una variabile
  - cosa succede se assegno lo stesso nome a variabili di contesti diversi
  - dove Python cerca le variabili in base al loro nome ed al punto di definizione

Namespaces

- mapping from name to object:
  - built-in names (`abs()`)
  - global names in module
  - local names in function invocation
- attributes = any following a dot
  - `z.real`, `z.imag`
- attributes read-only or writable
  - module attributes are writeable

Namespaces

- scope = textual region of Python program where a namespace is directly accessible (without dot)
  - innermost scope (first) = local names
  - middle scope = current module's global names
  - outermost scope (last) = built-in names
- assignments always affect innermost scope
  - don't copy, just create name bindings to objects
- global indicates name is in global scope

Lambda forms

- funzioni anonime di una singola riga
- potrebbero non funzionare in versioni datate

```
def make_incrementor(n):
  return lambda x: x + n

f = make_incrementor(42)
f(0) → 42
f(1) → 43
```

```
>>> def f(x):
...     return x*2
...
>>> f(3)
6
>>> g = lambda x: x*2
>>> g(3)
6
>>> (lambda x: x*2)(3)
6
```

Programmazione funzionale

- Descrivere il programma in termini valutazione di funzioni
- Si affida in maniere particolare a struttura dati di tipo immutabile
    - i dati immutabili non hanno stato più semplici da gestire in calcolo parallelo, multithread etc etc ....
- E' uno stile di programmazione
    - riduce il rischio di errori
    - semplifica il debug
    - evita i side effects

MAP REDUCE FILTER

- **Filter creates a list of elements for which a function returns true. Here is a short and concise example:**

```
number_list = range(-5, 5)
less_than_zero = list(filter(lambda x: x < 0,
number_list))
print(less_than_zero)
# Output: [-5, -4, -3, -2, -1]
```

- **map(function_to_apply, list_of_inputs)**
  - Most of the times we want to pass all the list elements to a function one-by-one and then collect the output. For instance:

```
items = [1, 2, 3, 4, 5]
squared = []
for i in items:
   squared.append(i**2)
```

  - Map allows us to implement this in a much simpler and nicer way. Here you go:

```
def f(x):
   return x%2
```

-

- **Reduce:**
  - from functools import reduce
  - product = reduce((lambda x, y: x * y), [1, 2, 3, 4])
- # Output: 24
- E' una funzione che ben si presta a fungere da accumulator (e.g. groupby)
-

Giochini ed un esempio di reduce

`vedi jupyter notebook reducegame`

```
In [1]: import matplotlib.pyplot as plt
        import numpy as np

In [2]: x = range(10)
        y = range(15)
        z = range(20)
        fig, ax = plt.subplots(nrows=2,ncols=2)

In [3]: for i in range(4):
            plt.subplot(2,2,i+1)
            X = x
            Y = [yy+i for yy in y]
            Z = [(zz+i)**(i+1) for zz in z]
            plt.loglog(X)
            plt.loglog(Y)
            plt.loglog(Z)
            plt.title('plot: '+str(i+1))

In [4]: plt.show()
```

Classi

Classes

- mixture of C++ and Modula-3
- multiple base classes
- derived class can override any methods of its base class(es)
- method can call the method of a base class with the same name
- objects have private data
- C++ terms:
  - all class members are public
  - all member functions are virtual
  - no constructors or destructors (not needed)

Programming paradigm

- Python is a OO ?
  - unlike Java, Python does not impose object-oriented programming as the main programming paradigm
  - modules and namespaces in Python gives to developer a way to ensure the encapsulation and separation of abstraction layers, both being the most common reasons to use object-orientation.
  - Trend do not use object-orientation, if it is not required by the business model.
    - Reason to avoid
      - Class glue state and functionalities → prone to concurrency problems, and race conditions
      - using stateless functions is a better programming paradigm.

Programming paradigm

- Carefully isolating functions with context and side-effects from functions with logic (called **pure** functions) allow the following benefits:
  - Pure functions are deterministic: given a fixed input, the output will always be the same.
  - Pure functions are much easier to change or replace if they need to be refactored or optimized.
  - Pure functions are easier to test with unit tests: There is less need for complex context setup and data cleaning afterwards.
  - Pure functions are easier to manipulate, decorate, and pass around.

Classes

- classes (and data types) are objects
- ~~built-in types cannot be used as base classes by user~~
- arithmetic operators, subscripting can be redefined for class instances (like C++, unlike Java)

Class definitions

```
Class ClassName (BaseClass):
    <statement-1>
    ...
    <statement-N>
```

- must be executed
- can be executed conditionally (see Tcl)
- creates new namespace

Esempio

- from time import sleep
- class contatore():

```
    c = 0
    def inc(self, I):
            self.c += I
     def res(self):
            self.c=0
      def set(self,k):
             self.c=k
       def timer(self):
              sleep(self.c)
              self.c -= 1

c1 = contatore()
c1.inc(4)
c1.timer()
 c1.c
c2 = contatore()
```

- c2.t=0
- c1.t → error
- class neo(contatore):

```
    t = 0
    def timer(self,k,m):
         sleep(k)
         self.c += m
c3=neo()
c3.timer(4,5)
```

- class pippo()
-     pass

Namespaces

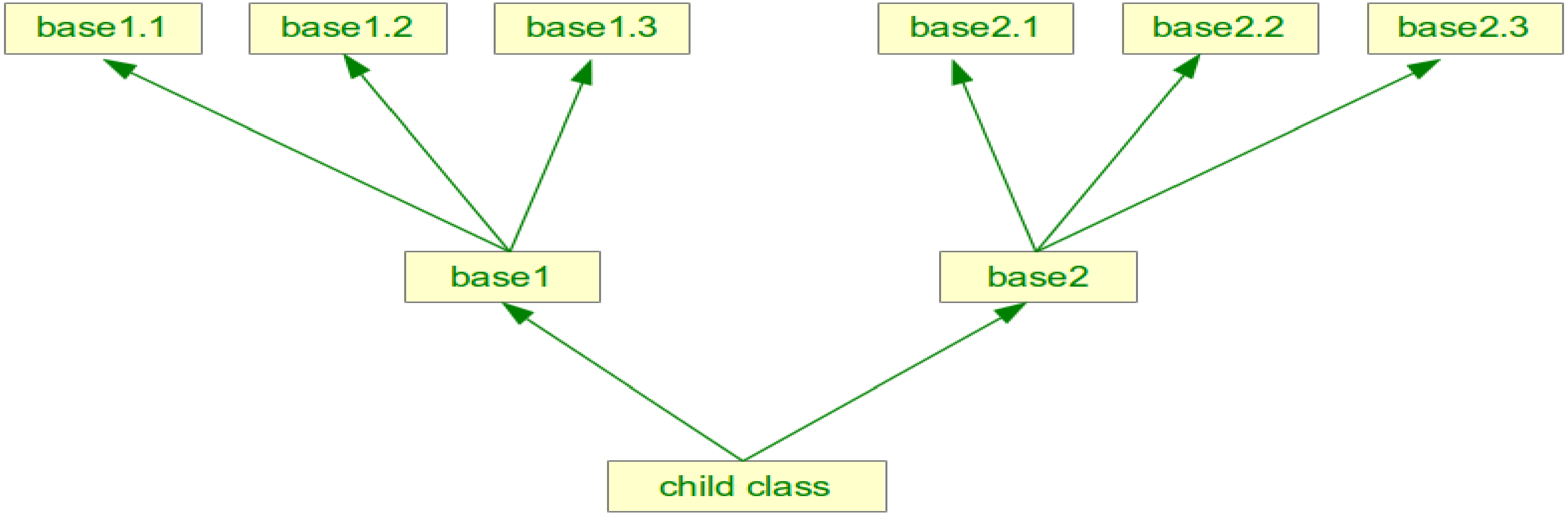
- mapping from name to object:
  - built-in names (`abs()`)
  - global names in module
  - local names in function invocation
- attributes = any following a dot
  - `z.real`, `z.imag`
- attributes read-only or writable
  - module attributes are writeable

Namespaces

- scope = textual region of Python program where a namespace is directly accessible (without dot)
  - innermost scope (first) = local names
  - middle scope = current module's global names
  - outermost scope (last) = built-in names
- assignments always affect innermost scope
  - don't copy, just create name bindings to objects
- global indicates name is in global scope

Subclassing

```
class SubclassName(BaseClass1, BaseClass2, BaseClass3, ...):
    pass
```



Class objects

- obj.name references (plus module!):

```
class MyClass:
  """A simple example class"""
  i = 123
  def f(self):
    return 'hello world'
a = MyClass()
a.i → 123
```

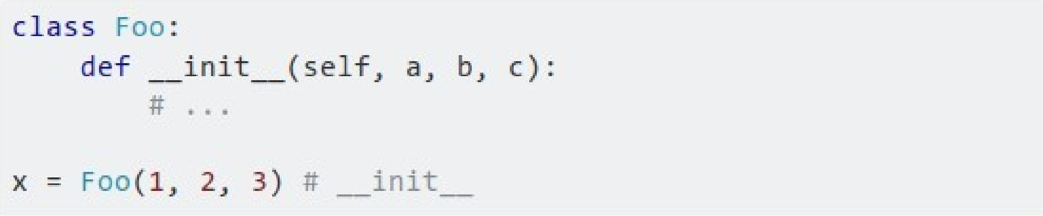
- MyClass.f is method object

Instance objects

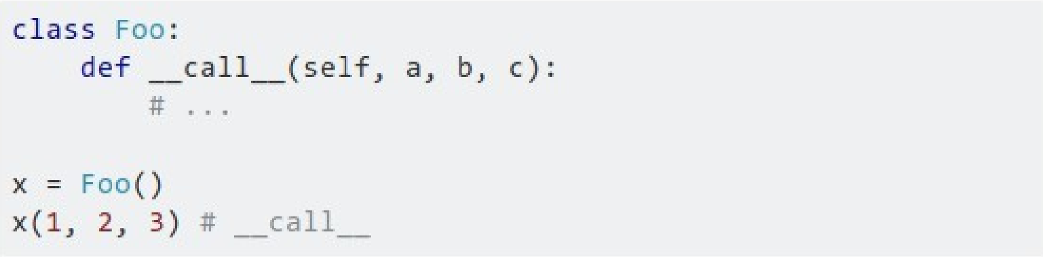
- attribute references
- data attributes (C++/Java data members)
  - created dynamically

```
x.counter = 1
while x.counter < 10:
  x.counter = x.counter * 2
print ( x.counter)
del x.counter
```

Class init e call



```python
class Foo:
    def __init__(self, a, b, c):
        # ...

x = Foo(1, 2, 3) # __init__
```



```python
class Foo:
    def __call__(self, a, b, c):
        # ...

x = Foo()
x(1, 2, 3) # __call__
```

```
In [1]: class A:
   ...:     def __init__(self):
   ...:         print ("init")
   ...:
   ...:     def __call__(self):
   ...:         print ( "call")
   ...:
   ...:
In [2]: a = A()
init
In [3]: a()   called as function call
call
```

the __init__ method is used when the class is called to initialize the instance,

the __call__ method is called when the instance is called

Overloading

OVERLOAD of Operators

Operator	Method
+	__add__(self, other)
-	__sub__(self, other)
*	__mul__(self, other)
/	__truediv__(self, other)
%	__mod__(self, other)
<	__lt__(self, other)
<=	__le__(self, other)
==	__eq__(self, other)
!=	__ne__(self, other)
>	__gt__(self, other)
>=	__ge__(self, other)

OVERLOAD
of custom function

```
class Student:
    def hello(self, name=None):
        if name is not None:
            print('Hey ' + name)
        else:
            print('Hey ')
# Creating a class instance
std = Student()
# Call the method
std.hello()
# Call the method and pass a parameter
std.hello('Nicholas')
```

OVERLOAD
of Builtin function

```
class Purchase:
    def __init__(self, basket, buyer):
        self.basket = list(basket)
        self.buyer = buyer
    def __len__(self):
        return len(self.basket)
purchase = Purchase(['pen', 'book', 'pencil'], 'Python')
print(len(purchase))
```

Esempio di overloading

```
class Dog:
    # Class Attribute
    species = 'mammal'
    # Initializer / Instance Attributes
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def __gt__(self,b):
        return self.age > b.age

a = Dog('Qui',6)
b = Dog('Quo',5)
c = Dog('Qua',4)

K = max (a,b,c)
```

Method objects

- Called immediately:

```
x.f()
```

- can be referenced:

```
xf = x.f
while 1:
   print xf()
```

```
    def __init__(self):
        Item.ct=Item.ct+1
        print("Item.ct --> ", Item.ct)
    def f(self):
        print("Item.ct --> ",Item.ct)
        return Item.ct

a=Item()
b=Item()
print(a.f())
print(b.f())
```

- object is passed as first argument of function → 'self'
  - x.f() is equivalent to MyClass.f(x)

Notes on classes

- Data attributes override method attributes with the same name
- no real hiding → not usable to implement pure abstract data types
- clients (users) of an object can add data attributes
- first argument of method usually called self
  - '`self`' has **no** special meaning (cf. Java)

Another example

- bag.py

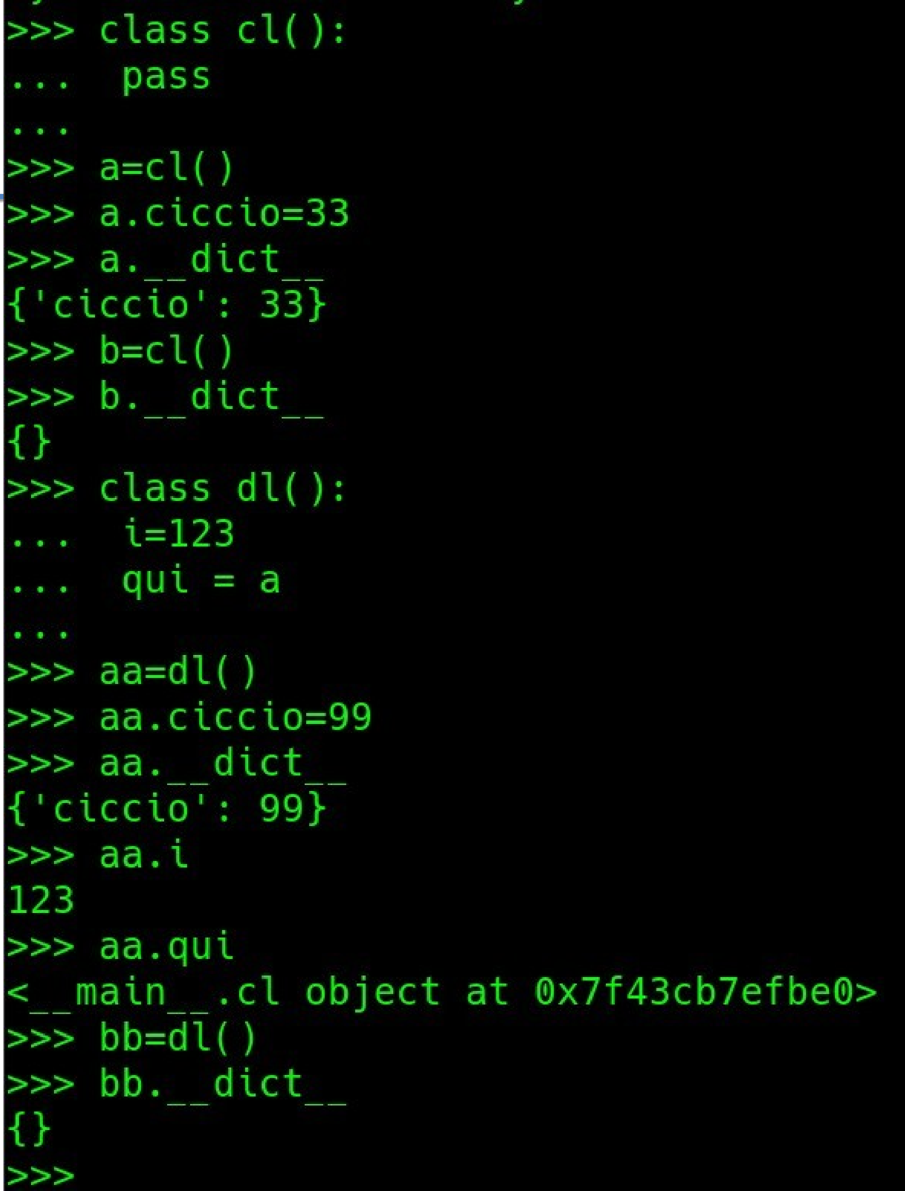
```
 classe Persone():
       Nome = “”
       Cognome = “”
 class Bag:
   def __init__(self):
     self.data = []
     self.persona = Persone()
   def add(self, x):
     self.data.append(x)
   def addtwice(self,x):
     self.add(x)
     self.add(x)
```

- invoke:

```
>>> from bag import *
>>> l = Bag()
>>> l.add('first')
>>> l.add('second')
>>> l.data
['first', 'second']
```

Class and instance attribute

```
class Dog:
    # Class Attribute
    species = 'mammal'
    # Initializer / Instance Attributes
    def __init__(self, name, age):
        self.name = name
        self.age = age
        if self.name == 'Jack':
           self.bastard = True
```



```
>>> class cl():
...  pass
...
>>> a=cl()
>>> a.ciccio=33
>>> a.__dict__
{'ciccio': 33}
>>> b=cl()
>>> b.__dict__
{}
>>> class dl():
...  i=123
...  qui = a
...
>>> aa=dl()
>>> aa.ciccio=99
>>> aa.__dict__
{'ciccio': 99}
>>> aa.i
123
>>> aa.qui
<__main__.cl object at 0x7f43cb7efbe0>
>>> bb=dl()
>>> bb.__dict__
{}
>>>
```

Attenzione alle istanze

```
>>> class Dog:
...     pass
...
>>> Dog()
<__main__.Dog object at
0x1004ccc50>
>>> Dog()
<__main__.Dog object at
0x1004ccc90>
>>> a = Dog()
>>> b = Dog()
>>> a == b
False
```

```
# coding: utf-8
class cl():
    a = []
    def __init__(self,k):
        self.b = k

m = cl(3)
s = cl(4)
m.a
m.a.append(99)
m.a
s.a
m.a = [1,2,3]
s.a
m.__dict__
s.__dict__
get_ipython().run_line_magic('save', '11-
22 classexemple.py ')
```

Variabili di Istanza e variabili di Classe

- Nella nostra classe Dog
  - le variabili di classe definiscono proprietà comuni a tutte le istanze non le devo specificare ogni volta e ci posso tramite
  - la classe → Dog.species
  - o tramite l'istanza → fuffi.species = 'cats'
  - notare che quando proviamo ad **accedere** all'attributo di un'Oggetto, Python controlla anzitutto se l'Istanza contiene quell'attributo, e se non è presente allora va a cercarlo come Variabile di Classe
  - se **variamo** il valore la nostra Istanza si crea la sua versione personalizzata della Variabile, senza influenzare gli oggetti restanti.

```
class Dog:
    # Class Attribute
    species = 'mammal'
    # Initializer / Instance Attributes
    def __init__(self, name, age):
        self.name = name
        self.age = age
        if self.name == 'Jack':
            self.bastard = True



fuffi = Dog('Fuffi', 6)
```

Variabile di classe
condivisa da tutte
le istanze della classe

self → riferimento
all'istanza
variabili di istanza

Variabili di istanza e di classe

- Vogliamo una variabile che rappresenta il numero totale di cani nel negozio di cani
  - incrementiamo il numero di cani_tot ogni volta che inizializziamo un cane che non viene affidato in giornata

```
class Dog:
    cani_tot = 0
    species = 'mammal'
    # Initializer / Instance Attributes
    def __init__(self, name, age, dogshop=True):
        self.name = name
        self.age = age
        if dogshop:
           Dog.cani_tot += 1
        if self.name == 'Jack':
           self.bastard = True
```

Inheritance

```
class DerivedClassName(BaseClassName)
   <statement-1>
   ...
   <statement-N>
```

- search class attribute, descending chain of base classes
- may override methods in the base class
- call directly via `BaseClassName.method`

Multiple inheritance

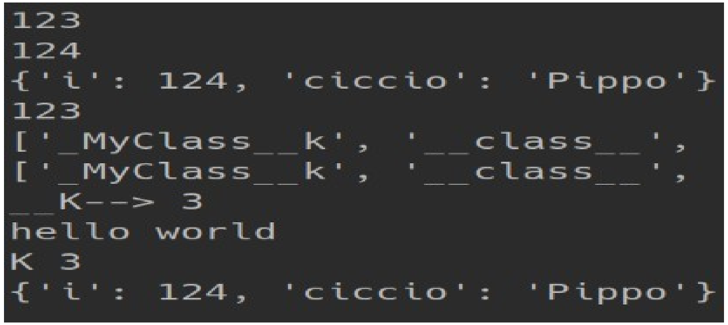
```
class DerivedClass(Base1,Base2,Base3):
    <statement>
```

- depth-first, left-to-right
- problem: class derived from two classes with a common base class

Private variables

- No real support, but textual replacement (name mangling)
- `__var` is replaced by `_classname__var`
- prevents only accidental modification, not true protection

```
i = 555
class MyClass:
  # "A simple example class"
  i = 123
  __k=3

  def f(self,a):
    i ='234'
    print("__K-->",MyClass.__k)
    return 'hello world'

a = MyClass()
print(a.i)
a.i = 124
print(a.i)
a.ciccio = "Pippo"
print(a.__dict__)
print(MyClass.i)
print(dir(a))
print(dir(MyClass))
print(a.f(5555))
print("K",a._MyClass__k)
print(a.__dict__)
```



```
123
124
{'i': 124, 'ciccio': 'Pippo'}
123
['_MyClass__k', '__class__',
['_MyClass__k', '__class__',
__K--> 3
hello world
K 3
{'i': 124, 'ciccio': 'Pippo'}
```

Subclassing

```
class Pets:
    ptot = 0
    def __init__(self, name, age, tipo, dogshop=True):
        self.name = name
        self.age = age
        self.tipo = tipo
        if dogshop:
           Pets.ptot += 1

class Gatti(Pets):
   pass



class Cani(Pets):
  pass

furia = Gatti('furia',0,'gatto')
ciclope = Cani('ciclope',1,'cane')
```

Python è andato a cercare il metodo costruttore __init__ all'interno delle due classi che abbiamo istanziato, e non avendolo trovato è andato a prenderselo da Pets

Possiamo istanziare sia Gatti che Cani
senza aver scritto niente perche ereditano da Pets
anche init

```
>> help(Gatti)
```

Subclassing

- ma è ovvio che i gatti sono gatti ed i cani cani, creiamo una versione personalizzata del metodo init per questi (per le sottoclassi)

```
class Gatti(Pets):
    def __init__(self,nome,eta,razza):
        super().__init__(nome,eta,'gatto')
        self.razza = razza

fur=Gatti('fur',0,'bastard')
>>> fur.__dict__    → {'name': 'fur', 'age': 0, 'tipo': 'gatto', 'razza': 'bastard'}
>>> fur.ptot  → 2
>>> pur=Gatti('pur',0,'bastard')
>>> pur.ptot  → 3
```

- funzione super() nome, età, tipo gestiti dal metodo __init__ della Classe genitore
- razza dalla sottoclasse

subclassing

canigatti.py

```
# coding: utf-8
class Pets:
    ptot = 0
    def __init__(self, name, age, tipo, dogshop=True):
        self.name = name
        self.age = age
        self.tipo = tipo
        if dogshop:
           Pets.ptot += 1

class Gatti(Pets):
   pass
class Cani(Pets):
  pass

class Gatti(Pets):
     def __init__(self,name,age,razza):
          super().__init__(name,age,'gatto')
          self.razza = razza

fur=Gatti('fur',0,'bastard')
def sk(self):
    return f'Il {self.tipo} {self.name} ha {self.age} anni'

Pets.sk =sk

print(fur.sk())

class cani(Pets):
     pedigree = True
     def __init__(self,name,age,razza):
         super().__init__(name,age,'cane')
         self.razza = razza
     def sk(self):
         r =  self.razza + ' certificata' if self.pedigree else ''
         return super().sk() + f' ed è di razza {r}'

joe = cani('joe',4,'jack_russel')
```

Appendice object oriented

Metodi di classe

- Metodi – > funzioni interne alle classi, primo parametro self, riferimento alla istanza
- Vogliamo implementare un metodo che ci consenta di istanziare oggetti in situazioni particolari.
- Nel nostro negozio di animali riceviamo da una struttura municipale gli animali identificati con un codice a barre (in pratica una stringA), che dovremmo accasare e vogliamo evitare di inserirli nel nostro sistema manualmente
- Modificheremo l'init della classe in maniera da accettare anche questo input
  - il metodo sarà legato alla classe e non all'istanza
  - cls e non più self
  - usiamo il decoratore @classmethod

Metodi di classe

- Il decoratore ci permette di passare come argomento la classe e non l'istanza
- chiameremo l'alternativa from_bar (i campi sono separati da spazio)
- chi la lettura passa però solo gli elementi di Pets senza le specificità di cani e gatti

canigatti2.py

Metodi di classe

- Sono utili quando è qualora si voglia un Metodo che cambia comportamento in base alla sottoclasse che lo sta richiamando.
-
  ```
  @classmethod
  def  bbb(cls):
    if (cls.__name__ == 'Gatti') :
        return 'Sono un Gatto'
    else:
        return 'Sono un cane'
  ```

Metodi di classe

- Sono anche utili quando si vuole modificare una proprietà della classe genitore
  - per esempio potremmo voler inizializzare/resettare la variabile di classe ptot a qualche valore

Static method

- A static method is also a method which is bound to the class and not the object of the class.
- A static method can’t access or modify class state.
- It is present in a class because it makes sense for the method to be present in class.

```
class C(object):
    @staticmethod
    def fun(arg1, arg2, ...):
        ...
returns: a static method for function
fun.
```

Class method vs Static Method

- A class method takes cls as first parameter while a static method needs no specific parameters.
- A class method can access or modify class state while a static method can’t access or modify it.
- In general, static methods know nothing about class state. They are utility type methods that take some parameters and work upon those parameters. On the other hand class methods must have class as parameter.
- We use @classmethod decorator in python to create a class method and we use @staticmethod decorator to create a static method in python.
- We generally use class method to create factory methods. Factory methods return class object (similar to a constructor) for different use cases.
- We generally use static methods to create utility functions (bah !)

An idea to create factory class with class method

```
class Pizza:
    def __init__(self, ingredients):
        self.ingredients = ingredients
    def __repr__(self):
        return f'Pizza({self.ingredients!r})'   # to give some useful representation of the object
    @classmethod
    def margherita(cls):
        return cls(['mozzarella', 'pomodoro'])
    @classmethod
    def prosciutto(cls):
        return cls(['mozzarella', 'pomodoro', 'prosciutto'])

Pizza.margherita()
Pizza(['mozzarella', 'pomodoro'])

Pizza.prosciutto()
Pizza(['mozzarella', 'pomodoro', 'prosciutto'])
```

They all use the same __init__ constructor internally and simply provide a shortcut for remembering all of the various ingredients.

The property function

- Syntax: property(fget, fset, fdel, doc)
- Parameters:
    - fget() – used to get the value of attribute
    - fset() – used to set the value of attribute
    - fdel() – used to delete the attribute value
    - doc() – string that contains the documentation (docstring) for the attribute
- Return: Returns a property attribute from the given getter, setter and deleter.

Property example – ritorna le proprietà di una classe

```
class Alphabet:
    def __init__(self, value):
        self._value = value

    # getting the values
    def getValue(self):
        print('Getting value')
        return self._value

    # setting the values
    def setValue(self, value):
        print('Setting value to ' + value)
        self._value = value

    # deleting the values
    def delValue(self):
        print('Deleting value')
        del self._value

    value = property(getValue, setValue, delValue, )
```

```
# passing the value
x = Alphabet('GeeksforGeeks')
print(x.value)

x.value = 'GfG'

del x.value
```

By using property() method, we can modify our class and implement the value constraint without any change required to the client code. So that the implementation is backward compatible.

https://www.programiz.com/python-programming/property

Closures

- A function defined inside another function is called a nested function.
- Nested functions can access variables of the enclosing scope.
  - Remind decorators!!

```
def print_msg(msg):
# This is the outer enclosing function

    def printer():
# This is the nested function
        print(msg)

    printer()

# We execute the function
# Output: Hello
print_msg("Hello")
```

Closure2

```
def print_msg(msg):
# This is the outer enclosing function

    def printer():
# This is the nested function
        print(msg)

    return printer  # this got changed

# Now let's try calling this function.
# Output: Hello

another = print_msg("Hello")
another()

#Output
Hello
```

- The print_msg() function was called with the string "Hello" and the returned function was bound to the name another.
- On calling another(), the message was still remembered although we had already finished executing the print_msg() function.
- **Attaching some data ("Hello") to the code is called closure in Python.**
- This value in the enclosing scope is remembered even when the variable goes out of scope or the function itself is removed from the current namespace.

When use closures

- Closures can avoid the use of global values and provides some form of data hiding. **It can also provide an object oriented solution to the problem**.
- When there are few methods (one method in most cases) to be implemented in a class, closures can provide an alternate and more elegant solutions. But when the number of attributes and methods get larger, better implement a class.
- Here is a simple example where a closure might be more preferable than defining a class and making objects

```
def make_multiplier_of(n):
    def multiplier(x):
        return x * n
    return multiplier

# Multiplier of 3
times3 = make_multiplier_of(3)

# Multiplier of 5
times5 = make_multiplier_of(5)

# Output: 27
print(times3(9))

# Output: 15
print(times5(3))

# Output: 30
print(times5(times3(2)))
```

Singleton

- Stated the OO/Non OO Python approach we have discuss before …
- A singleton is a way to provide one and only one object of a particular type
    - for others a way to have a single set of state data for all objects

```
class _Singleton:
    _instance = None

    def hello(self):
        print("Hello!")


def Singleton():
    if _Singleton._instance is None:
        _Singleton._instance =
_Singleton()
    return _Singleton._instance



s1 = Singleton()
s2 = Singleton()

print(s1)
print(s2)
assert s1 is s2
```

Singleton

- Various way to implement
  - the simpler → a module singleton.py
  - a base class
    - We control the object creation by delegation to a single instance of **a private nested inner class**:
    - so we put things out of the hands of the programmer!

`__new__` method is similar to the `__init__` method, but if both exist, `__new__` method executes first.

In the base class object, `__new__` is defined as a static method and needs to pass a parameter cls.

cls represent the classe that need to be instantiated, and this parameter is provided automatically by python parser at instantiation time.

(newexample.py)

Singleton

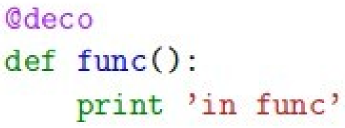
- inner class is named with a double underscore, it is "private" so the user cannot directly access it.
- The inner class contains all the methods that you would normally put in the class if it weren't going to be a singleton, and then it is wrapped in the outer class which controls creation by using its constructor.
- The first time you create an OnlyOne, it initializes instance, but after that it just ignores you.
- Access comes through delegation, using the __getattr__() method to redirect calls to the single instance.
- You can see from the output that even though it appears that multiple objects have been created, the same __OnlyOne object is used for both.
    - The instances of OnlyOne are distinct but they all proxy to the same __OnlyOne object.

```
class OnlyOne(object):
    class __OnlyOne:
        def __init__(self):
            self.val = None
        def __str__(self):
            return repr(self) + self.val
    instance = None
    def __new__(cls): # __new__ always a classmethod
        if not OnlyOne.instance:
            OnlyOne.instance = OnlyOne.__OnlyOne()
        return OnlyOne.instance
    def __getattr__(self, name):
        return getattr(self.instance, name)
    def __setattr__(self, name):
        return setattr(self.instance, name)

x = OnlyOne()
x.val = 'qui'
print(x)
y = OnlyOne()
y.val = 'quo'
print(y)
z = OnlyOne()
z.val = 'qua'
print(repr(x))
print(repr(y))
print(repr(z))
```

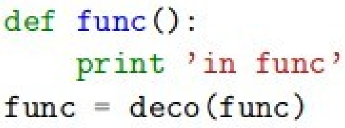
```
<__main__.OnlyOne.__OnlyOne object at 0x7f8fde509390>qui
<__main__.OnlyOne.__OnlyOne object at 0x7f8fde509390>quo
<__main__.OnlyOne.__OnlyOne object at 0x7f8fde509390>qua
<__main__.OnlyOne.__OnlyOne object at 0x7f8fde509390>qua
<__main__.OnlyOne.__OnlyOne object at 0x7f8fde509390>qua
```

Appendice: Decorators

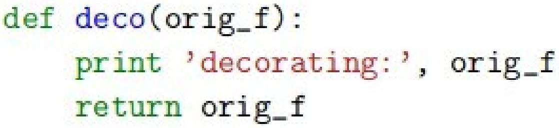
- Add functionality to existing functions and classes
- Decorators are functions that wrap other functions or classes
- passing of a function object through a filter + syntax
  - Can work on classes or functions
  - can be written as classes or functions … nothing new under the sun ;)
  - just cleaner



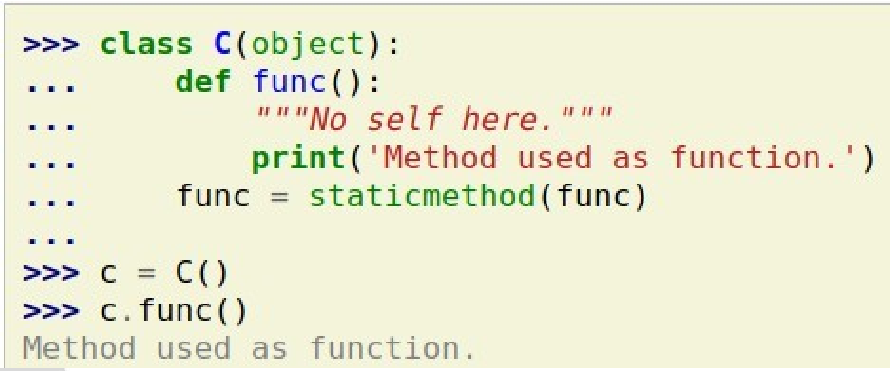
```
@deco
def func():
    print ’in func’
```



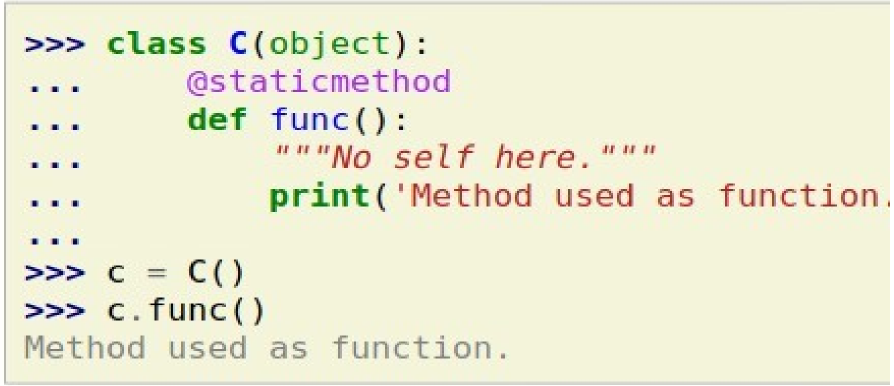
```
def func():
    print ’in func’
func = deco(func)
```



```
def deco(orig_f):
    print ’decorating:’, orig_f
    return orig_f
```



```
>>> class C(object):
...     def func():
...         """No self here."""
...         print('Method used as function.')
...     func = staticmethod(func)
...
>>> c = C()
>>> c.func()
Method used as function.
```



```
>>> class C(object):
...     @staticmethod
...     def func():
...         """No self here."""
...         print('Method used as function.
...
>>> c = C()
>>> c.func()
Method used as function.
```

Decorators 2

- Functions are objects; they can be referenced to, passed to a variable and returned from other functions as well.
- Functions can be defined inside another function and can also be passed as argument to another function.
- Decorators are very powerful and useful tool in Python
  - allows programmers to modify the behavior of function or class. Decorators allow us to wrap another function in order to extend the behavior of wrapped function, without permanently modifying it.
- In Decorators, functions are taken as the argument into another function and then called inside the wrapper function.

Decorators 3 Syntax

```
@gfg_decorator
def hello_decorator():
    print("Gfg")
```

```
def hello_decorator():
    print("Gfg")

hello_decorator = 
gfg_decorator(hello_decorator)'''
```

gfg_decorator is a callable function, will add some code on the top of some another callable function, hello_decorator function and return the wrapper function.

Decorators 4

```
# defining a decorator
def hello_decorator(func):

    # inner1 is a Wrapper function in
    # which the argument is called

    # inner function can access the outer local
    # functions like in this case "func"
    def inner1():
        print("Hello, this is before function execution")

        # calling the actual function now
        # inside the wrapper function.
        func()

        print("This is after function execution")

    return inner1

# defining a function, to be called inside wrapper
def function_to_be_used():
    print("This is inside the function !!")

# passing 'function_to_be_used' inside the
# decorator to control its behavior
function_to_be_used = hello_decorator(function_to_be_used)

# calling the function
function_to_be_used()
```

Output

```
Hello, this is before function execution
This is inside the function !!
This is after function execution
```

decorators 5

prende il valore di ritorno
e lo mette maiuscolo

```
def uppercase_decorator(function):
    def wrapper():
        func = function()
        make_uppercase = func.upper()
        return make_uppercase
    return wrapper

def say_hi():
    return 'hello there'
print(say_hi())

@uppercase_decorator
def say_hi():
    return 'hello there'


print('Solo uppercase',say_hi())


def split_string(function):
    def wrapper():
        func = function()
        splitted_string = func.split()
        return splitted_string


    return wrapper


@split_string
@uppercase_decorator
def say_hi():
    return 'hello there'


print('Upppercase e split', say_hi())
```

~ C structs

- Empty class definition:

```
class Employee:
  pass

john = Employee()
john.name = 'John Doe'
john.dept = 'CS'
john.salary = 1000
```

Esercizi §

- Using the Dog class, instantiate three new dogs, each with a different age. Then write a function called, get_biggest_dog(), that takes any number of dogs (*args) and returns the oldest one. Then output the age of the oldest dog like so:
- *The oldest dog **name** is 7 years old.*

```
class Dog:
    # Class Attribute
    species = 'mammal'
    # Initializer / Instance Attributes
    def __init__(self, name, age):
        self.name = name
        self.age = age
```

Eccezioni

Eccezioni

- An exception is an event, which occurs during the execution of a program that disrupts the normal flow of the program's instructions.
- When a Python script encounters a situation that it cannot cope with, it raises an exception.
- An exception is a Python object that represents an error.
- When a Python script raises an exception, it must either handle the exception immediately otherwise it terminates and quits.

Exceptions

- syntax (parsing) errors

```
while 1 :
    print 'Hello World'
File "<stdin>", line 1
  while 1 print ( 'Hello World')
              ^
SyntaxError: invalid syntax
```

- exceptions
  - run-time errors
  - e.g., `ZeroDivisionError, NameError, TypeError`

Handling exceptions

```
while 1:
  try:
    x = int(input("Please enter a number: "))
    pass
  except ValueError as msg:
    print(msg, "Not a valid number")
  except Exception as msg1:
    print(msg1,’Errore diverso da value error”)
   a =“Istruzione dopo”
```

- First, execute `try` clause
- if no exception, skip `except` clause
- if exception, skip rest of `try` clause and use except clause
- if no matching exception, attempt outer `try` statement

Handling exceptions

- **try.py**

```
import sys  # python try.py arg1 arg2 arg2     arg0=try.py
for arg in sys.argv[1:]:
    try:
        f = open(arg)
    except IOError as msg:
        print (f'cannot open {arg} → error is {msg}')
    else:
      print (arg, 'lines:', len(f.readlines()))
      f.close()
```

- e.g., as python try.py *.py

Eccezioni formato generale

```
try:
   fh = open("testfile", "w")
   fh.write("This is my test file for exception
handling!!")
except IOError:
   print ("Error: can 't find file or read data")
else:
   print ("Written content in the file successfully")
   fh.close()
finally:
   print(“testfile processed”)
```

try:

  You do your operations here;

  ....

except ExceptionI:

  If there is ExceptionI, then execute this block.

except ExceptionII:

  If there is ExceptionII, then execute this block.

  ....

else:

  If there is **no** exception then execute this block.

finally:

  clean up viene eseguito alla fine del blocco
sia che ci siano state eccezioni che no

```
dbg=True

def f():
   try:
      print('boh')
      if dbg:
         raise ValueError
   except IOError as msg:
      print('Catched IOERROR')
   else:
      print("E' andata bene")
   finally:          #Nel caso in cui l'eccezione sia gestita o meno
      print('Finito!')
   print('Sono dopo')   #Solo nel caso in cui l'eccezione sia gestita

try:
  f()
except Exception as msg:
    print('Presa Fuori!!!!', msg)
```

EXCEPTION NAME	DESCRIPTION
Exception	Base class for all exceptions
StopIteration	Raised when the next() method of an iterator does not point to any object.
SystemExit	Raised by the sys.exit() function.
StandardError	Base class for all built-in exceptions except StopIteration and SystemExit.
ArithmeticError	Base class for all errors that occur for numeric calculation.
OverflowError	Raised when a calculation exceeds maximum limit for a numeric type.
FloatingPointError	Raised when a floating point calculation fails.
ZeroDivisonError	Raised when division or modulo by zero takes place for all numeric types.
AssertionError	Raised in case of failure of the Assert statement.
AttributeError	Raised in case of failure of attribute reference or assignment.
EOFError	Raised when there is no input from either the raw_input() or input() function and the end of file is reached.
ImportError	Raised when an import statement fails.
KeyboardInterrupt	Raised when the user interrupts program execution, usually by pressing Ctrl+c.
LookupError	Base class for all lookup errors.
IndexError **KeyError**	Raised when an index is not found in a sequence. Raised when the specified key is not found in the dictionary.
NameError	Raised when an identifier is not found in the local or global namespace.
UnboundLocalError **EnvironmentError**	Raised when trying to access a local variable in a function or method but no value has been assigned to it. Base class for all exceptions that occur outside the Python environment.
IOError **IOError**	Raised when an input/ output operation fails, such as the print statement or the open() function when trying to open a file that does not exist. Raised for operating system-related errors.
SyntaxError **IndentationError**	Raised when there is an error in Python syntax. Raised when indentation is not specified properly.
SystemError	Raised when the interpreter finds an internal problem, but when this error is encountered the Python interpreter does not exit.
SystemExit	Raised when Python interpreter is quit by using the sys.exit() function. If not handled in the code, causes the interpreter to exit.
ValueError	Raised when the built-in function for a data type has the valid type of arguments, but the arguments have invalid values specified.

Raise

- Solleva una eccezione specifica
  - comando raise seguito dall'eccezione:
  - raise IndexError('IndexError generato con raise')
- passare il controllo di un'eccezione sollevata da altri

```
def generaIndexError(propaga):
    try:
        #genero l'eccezione
        raise IndexError('index error')
    except IndexError:#intercetto l'eccezione localmente
        if not propaga:
            print("raccolgo localmente ma non propago l'eccezione")
        else:
            print("raccolgo localmente e propago l'eccezione")
            raise #propago l'eccezione
```

Extend exception class

```
class HostNotFound(Exception):
     def __init__( self, host ):
         self.host = host
         Exception.__init__(self, f'Host Not Found exception: missing {self.host}')

‘’’
------------------------------------------------------------
Esempio di uso
------------------------------------------------------------
‘’’
try:
         raise HostNotFound("taoriver.net")
     except HostNotFound as exc:
         # Handle exception.
         print (exc)  # -> 'Host Not Found exception: missing taoriver.net'
         print (exc.host)  # -> 'taoriver.net'
```

Trap

```
def main():
    pass

######################
#   MAIN PROGRAM
######################
if __name__ == "__main__":
   try :
      main()
   except: Exception as msg:
      pass
```

Asserzioni

- An assertion is a sanity-check that you can turn on or turn off when you are done with your testing of the program.

```
assert Expression[, Arguments]
```

```
#!/usr/bin/python
def KelvinToFahrenheit(Temperature):
 assert (Temperature >= 0),"Colder than absolute  zero!"
 return ((Temperature-273)*1.8)+32

print (KelvinToFahrenheit(273))
print (int(KelvinToFahrenheit(505.78)))
print (KelvinToFahrenheit(-5))
32.0
451
Traceback (most recent call last):
File "test.py", line 9, in
print KelvinToFahrenheit(-5)
File "test.py", line 4, in KelvinToFahrenheit
assert (Temperature >= 0),"Colder than absolute zero!"
AssertionError: Colder than absolute zero!
```

Logging

- **To print log messages to the screen, copy and paste this code:**

```
import logging
logging.basicConfig(level=logging.DEBUG, format='%(asctime)s - %(levelname)s - %(message)s')
logging.debug('This is a log message.')
logging.disable(logging.CRITICAL)
```

- **To write log messages to a file, you can copy and paste this code (the only difference is in bold)**:

```
import logging
logging.basicConfig(filename='log_filename.txt', level=logging.DEBUG, format='%(asctime)s - %(levelname)s - %(message)s')
logging.debug('This is a log message.')
```

  - Later runs of the program will append to the end of the log file, rather than overwrite the file.

- **To log messages to a file AND printed to the screen, copy and paste the following:**

```
import logging
logger = logging.getLogger()
logger.setLevel(logging.DEBUG)

formatter = logging.Formatter('%(asctime)s - %(levelname)s - %(message)s')

fh = logging.FileHandler('log_filename.txt')
fh.setLevel(logging.DEBUG)      call sets the minimum log level of messages it actually logs.
fh.setFormatter(formatter)
logger.addHandler(fh)

ch = logging.StreamHandler()
ch.setLevel(logging.DEBUG)
ch.setFormatter(formatter)
logger.addHandler(ch)
```

- logger.debug('This is a test log message.')

Cenni di programmazione funzionale

- statement di controllo del flusso (´if´, ´elif´, ´else´, ´assert´, ´try´, ´except´, ´finally´, ´for´, ´break´, ´continue´, ´while´, ´def´)
  - richiede import functools
  - + [1,2] if [(cond), se_vero, se_falso) ….
  - possono essere tutti gestiti in stile funzionale usando le funzioni e gli operatori di programmazione funzionale.

Cenni di programmazione funzionale

- `filter(`*`function, sequence`*`)`

  ```
  def f(x): return x%2 != 0 and x%3 == 0
  filter(f, range(2,25))
  ```

- `map(`*`function, sequence)`*
  - call function for each item
  - return list of return values
- `reduce(`*`function, sequence`*`)    - nella functools --`
  - return a single value
  - call binary function on the first two items
  - then on the result and next item
  - iterate

I/O Base

FILE IO

```
open(file, mode='r', buffering=-1, encoding=None,  errors=None, newline=None, closefd=True, opener=None)¶
```

```
>>> f = open(“names.txt")
>>> f.readline()
'Yaqin\n'
```

QUICK

```
>>> lst= [ x for x in open("text.txt","r").readlines() ]
>>> lst
['Chen Lin\n', 'clin@brandeis.edu\n', 'Volen 110\n', 'Office Hour: Thurs. 3-
    5\n', '\n', 'Yaqin Yang\n', 'yaqin@brandeis.edu\n', 'Volen 110\n',
    'Offiche Hour: Tues. 3-5\n']
```

Ignore the header?

```
for (i,line) in enumerate(open(‘text.txt’,"r").readlines()):
        if i == 0: continue
        print (line)
```

FILE IO Write

```
f.write(‘ciccio \n’)
print(“ciccio”,file=f)
```

```
input_file = open(“in.txt")
output_file = open(“out.txt", "w",[buffering],newline=None)    '\n' | '\r\n'
for line in input_file:
        output_file.write(line)
```

If the buffering value is set to 0, no buffering takes place. If the buffering value is 1, line buffering is performed while accessing a file. If you specify the buffering value as an integer greater than 1, then buffering action is performed with the indicated buffer size. If negative, the buffer size is the system default(default behavior).

```
“w(+)” = “write mode”
“a” = “append mode”
“wb” = “write in binary”
“r(+)” = “read mode” (default) e pos iniz
“rb” = “read in binary”
“U” = “read files with Unix
“a+” = append & reading e mi pos. fine
or Windows line endings”
```

File

- Lettura
  - unica riga → f.read()
  - in una lista di stringhe → readline**s**()
  - una riga alla volta → readline()
  - blocchi binario read(n)
- Context manager
  - with open(...) as ...
    - sostituisce f = open() con with open() as f:
    - consente di omettere la chiusura esplicita
  -
  - f= open()
  - ...
  - f.close

```
with open() as f:
    op1
    op2

non devo chiudere, appena esco dal blocco
viene fatto automaticamente
```

....

File

- object = open(file_name [, access_mode][, buffering])

```
#!/usr/bin/python
# Open a file
fo = open("foo.txt", "r+")
str = fo.read(10);
print "Read String is : ", str
# Check current position
position = fo.tell();
print "Current file position : ", position
# Reposition pointer at the beginning once again
position = fo.seek(0, 0);
str = fo.read(10);
print "Again read String is : ", str
# Close opend file
fo.close()
```

```
fileObject.seek(offset[, whence])
  0 inizio
  1 da dove sono
  2 dalla fine
```

Metodi base

- #!/usr/bin/python
- str = input("Enter your input: ");
- print "Received input is : ", str
-
- #!/usr/bin/python
- # Open a file
- fo = open("foo.txt", "wb")
- print ("Name of the file: ", fo.name)
- print ("Closed or not : ", fo.closed)
- print ("Opening mode : ", fo.mode)
- print ("Softspace flag : ", fo.softspac)

CSV Methods

```
name,department,birthday month
John Smith,Accounting,November
Erica Meyers,IT,March
```

```
import csv
with open('employee_birthday.txt') as csv_file:
    csv_reader = csv.reader(csv_file, delimiter=',')
    line_count = 0
    for row in csv_reader:
        if line_count == 0:
            print(f'Column names are {", ".join(row)}')
            line_count += 1
        else:
            print(f'\t{row[0]} works in the {row[1]} department, and was born in {row[2]}.')
            line_count += 1
    print(f'Processed {line_count} lines.')
```

Parameters

- **delimiter** specifies the character used to separate each field. The default is the comma (',').
- **quotechar** specifies the character used to surround fields that contain the delimiter character. The default is a double quote (' " ').
- **escapechar** specifies the character used to escape the delimiter character, in case quotes aren't used. The default is no escape character.

```
#Read as Dictionary
import csv
with open('employee_birthday.txt', mode='r') as csv_file:
    csv_reader = csv.DictReader(csv_file)
    line_count = 0
    for row in csv_reader:
        if line_count == 0:
            print(f'Column names are {", ".join(row)}')
            line_count += 1
        print(f'\t{row["name"]} works in the \
{row["department"]} department, and was born in \
{row["birthday month"]}.')
        line_count += 1
    print(f'Processed {line_count} lines.')
```

```
#Write CSV
import csv
with open('employee_file.csv', mode='w') as employee_file:
    employee_writer = csv.writer(employee_file,\
delimiter=',', quotechar='"', quoting=csv.QUOTE_MINIMAL)
    employee_writer.writerow(['John ', 'Acct', 'Nov'])
    employee_writer.writerow(['Erica ', 'IT', 'Mar'])

#with dict
import csv
with open('employee_file2.csv', mode='w') as csv_file:
    fieldnames = ['emp_name', 'dept', 'birth_month']
    writer = csv.DictWriter(csv_file,fieldnames=fieldnames)
    writer.writeheader()
    writer.writerow({'emp_name': 'John', 'dept':'Acct',
'birth_month': 'Nov'})
    writer.writerow({'emp_name': 'Erica', 'dept':'IT',
'birth_month': 'Mar'})
```

Riferimenti per CSV

- https://docs.python.org/3/library/csv.html

L'oggetto OS

- #!/usr/bin/python
- import os
- os.remove("text2.txt")
- os.rename("text2.txt", “tes3.txt”)
- os.mkdir("newdir")
- os.chdir("newdir")
- …..

```
from pathlib import Path
directory = Path("/etc")
filepath = directory / "test_file.txt"
if filepath.exists():
    stuff
```

Espressioni regolari

- https://www.w3schools.com/python/python_regex.asp
- https://docs.python.org/3/howto/regex.html#regex-howto
- https://pythex.org/

Moduli e Packages

Moduli

- Modules are files containing Python definitions and statements (ex. *name*.py)
- A module’s definitions can be imported into other modules by using “import *name*”
- Sometimes installation name ad import name are not the same
  - use help('modules')
- The module’s name is available as a global variable value
- To access a module’s functions, type *“name.function()”*

Moduli (2)

- Modules can contain executable statements along with function definitions
- Each module has its own private symbol table used as the global symbol table by all functions in the module
- Modules can import other modules
- Each module is imported once per interpreter session
  - *reload(name)*
- Can import names from a module into the importing module’s symbol table
  - *from mod import m1, m2 (*or **)*
  - *m1()*

Moduli (3)

- When a Python program starts it only has access to a basic functions and classes.

  (“int”, “dict”, “len”, “sum”, “range”, ...)

- “Modules” contain additional functionality.
- Use “import” to tell Python to load a module.

```
>>> import math
>>> import nltk
```

- collection of functions and variables, typically in scripts
- definitions can be imported
- file name is module name + .py
- e.g., create module fibo.py

```
def fib(n): # write Fib. series up to n
  ...
def fib2(n): # return Fib. series up to n
```

Moduli (5)

- function definition + executable statements
- executed only when module is imported
- modules have private symbol tables
  - e.g. define a namespace
- avoids name clash for global variables
- accessible as *module.globalname*
- can import into name space:

```
>>> from fibo import fib, fib2
>>> fib(500)
```

- can import all names defined by module:

```
>>> from fibo import *
```

Executing Modules

- *python name.py <arguments>*
  - *sys.argv → lista che contiene parametri passati*
    - *len(sys.argv) quanti ne ho passati*
    - *sys.argv[0] è il nome del programma python che ho lanciato*
    - *sys.argv[1] è il primo parametro e cosi via*
  - Runs code as if it was imported
  - Il setting *__name__ == "__main__" se testato*
    - *Se vero il file è sto invocato come script*
    - *Se falso il file è stato importato come libreria*

Module Search Path

- The interpreter searches for a file named *name.py*
  - Current directory given by variable *sys.path*
  - List of directories specified by **PYTHONPATH**
  - Default path
    - **(in UNIX may be - *.:/usr/local/lib/python*)**
- Script being run should not have the same name as a standard module or an error will occur when the module is imported

Moduli logica di ricerca

1) nella cartella dove risiede il programma principale (main.py nello schema sopra)
2) nelle cartelle specificate nella variabile di ambiente PYTHONPATH (se impostata)
3) nelle cartelle dove risiedono le librerie standard (dipendenti dalla particolare installazione, ad es C:\Python32\Lib)
4) nelle cartelle specificate in eventuali file con estensione pth (se presenti, una cartella per riga, utile ad esempio, se si hanno installate più versioni dell'interprete)

Packages

- Organizzano i moduli in modo strutturato.
  - I moduli possono essere strutturati in cartelle (packages)
  - le cartelle possono essere importate come se fossero moduli con la possibilita' di accedere a tutti i moduli che esse contengono.
- Creare un package:
  - la cartella deve essere situata in uno dei 'luoghi' dove l'interprete ricerca i moduli
  - deve contenere un file '__init__.py che viene eseguito alla prima importazione del package e che puo' essere anche vuoto (non più obbligatorio)
  - i moduli all'interno delle cartelle sono accessibili con la sintassi nomepackage.nomemodulo mediante gli statement di importazione usati con i moduli semplici
  - i packages possono essere annidati

Packages

- “dotted module names” (ex. a.b)
  - Submodule b in package a
- Saves authors of multi-module packages from worrying about each other’s module names
- Python searches through sys.path directories for the package subdirectory
- Users of the package can import individual modules from the package
- Ways to import submodules
  - import sound.effects.echo
  - from sound.effects import echo
- Submodules must be referenced by full name
- An ImportError exception is raised when the package cannot be found

Importing * From a Package

- * does not import all submodules from a package
- Ensures that the package has been imported, only importing the names of the submodules defined in the package
  - import sound.effects.echo
  - import sound.effects.surround
  - from sound.effects import *

Module search path

- current directory
- list of directories specified in PYTHONPATH environment variable (Win PYTHON_PATH)
  - export PYTHONPATH in shell adding you preferences as root dir for your libraries
  - insert in the root dir(s) __init__.py (and in all first level packages)
- uses installation-default if not defined, e.g., .:/usr/local/lib/python
- uses sys.path
  - you can also add path from program sys.path.append(os.path.abspath('../../'))

```
>>> import sys
>>> sys.path
['', 'C:\\PROGRA~1\\Python2.2', 'C:\\Program Files\\Python2.2\\DLLs', 'C:\\Program Files\\Python2.2\\
   lib', 'C:\\Program Files\\Python2.2\\lib\\lib-tk', 'C:\\Program Files\\Python2.2', 'C:\\Program
   Files\\Python2.2\\lib\\site-packages']
```

Intra-package References

- Submodules can refer to each other
  - *Surround* might use *echo* module
  - *import echo* also loads *surround* module
- *import* statement first looks in the containing package before looking in the standard module search path
- Absolute imports refer to submodules of sibling packages
  - *sound.filters.vocoder* uses *echo* module

    from sound.effects import echo
- Can write explicit relative imports
  - *from . import echo*
  - *from .. import formats*
  - *from ..filters import equalizer*

Trucchetti

- -*O* flag generates optimized code and stores it in *.pyo* files
  - Only removes *assert* statements
  - *.pyc* files are ignored and *.py* files are compiled to optimized bytecode
- Passing two –*OO* flags
  - Can result in malfunctioning programs
  - _*doc*_ strings are removed
- Same speed when read from *.pyc, .pyo,* or *.py* files, *.pyo* and *.pyc* files are loaded faster
- Startup time of a script can be reduced by moving its code to a module and importing the module
- Can have a *.pyc* or *.pyo* file without having a *.py* file for the same module
- Module *compileall* creates *.pyc* or *.pyo* files for all modules in a directory

Module listing

- use dir() for each module

```
>>> dir(fibo)
['__name__', 'fib', 'fib2']
>>> dir(sys)
['__displayhook__', '__doc__', '__excepthook__', '__name__', '__stderr__', '__st
din__', '__stdout__', '_getframe', 'argv', 'builtin_module_names', 'byteorder',
'copyright', 'displayhook', 'dllhandle', 'exc_info', 'exc_type', 'excepthook', '
exec_prefix', 'executable', 'exit', 'getdefaultencoding', 'getrecursionlimit', '
getrefcount', 'hexversion', 'last_type', 'last_value', 'maxint', 'maxunicode', '
modules', 'path', 'platform', 'prefix', 'ps1', 'ps2', 'setcheckinterval', 'setpr
ofile', 'setrecursionlimit', 'settrace', 'stderr', 'stdin', 'stdout', 'version',
 'version_info', 'warnoptions', 'winver']
```

- help(“sys.platform”)

The dir() Function

- Used to find the names a module defines and returns a sorted list of strings
  - *>>> import mod*

    >>> dir(mod)

    [‘_name_’, ‘m1’, ‘m2’]
- Without arguments, it lists the names currently defined (variables, modules, functions, etc)
- Does not list names of built-in functions and variables
  - Use __*builtin*__to view all built-in functions and variables

PIP

Installing a package

```
$ pip install simplejson
[... progress report ...]
Successfully installed simplejson
```

Upgrading a package

```
$ pip install --upgrade simplejson
[... progress report ...]
Successfully installed simplejson
```

Removing a package

```
$ pip uninstall simplejson
Uninstalling simplejson:
  /home/me/env/lib/python2.7/site-packages/simplejson
  /home/me/env/lib/python2.7/site-packages/simplejson-2.2.1-py2.7.egg-info
Proceed (y/n)? y
  Successfully uninstalled simplejson
```

Searching a package

```
#Search PyPI for packages
$ pip search "query"
```

Checking status of a package

```
# To get info about an installed package, including its location and files:
pip show ProjectName
```

Standard modules

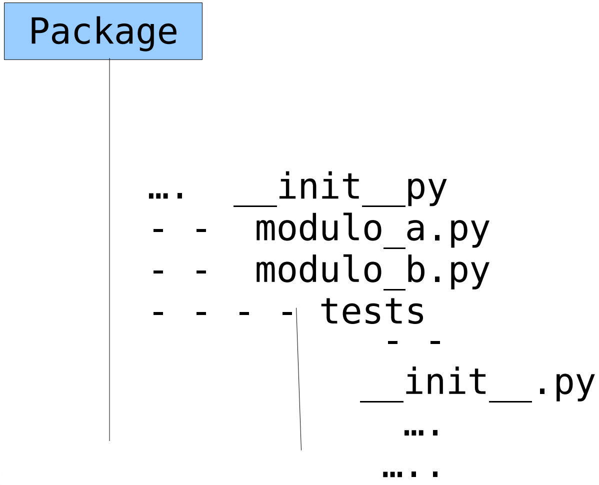
- system-dependent list
- always sys module

```
>>> import sys
>>> sys.p1
'>>> '
>>> sys.p2
'... '
>>> sys.path.append('/some/directory')
```

Organizzazione del codice

(una lista di directory per python è uno spazio dei nomi per gli import per evitare complicazioni nelle sovrapposizioni ignora come spazi di nomi quelii che non contengono __init__.py) nota*

- Packages
  - In Python, un pacchetto è una directory importabile (con __init__.py) contenente i file di origine (cioè i moduli).
  - Non è un pacchetto del sistema operativo
    - ma può essere implementato come un pacchetto del sistema operativo



```
Package

  ….  __init__py
  - -  modulo_a.py
  - -  modulo_b.py
  - - - - tests
            - -
          __init__.py
           ….
          …..
```

```
il file __init__.py presente nella directory fa si che
python tratti al directory stessa come contenente
pacchetti
può essere:
  • un file vuoto
  • contenere codice di inizializzazione per il
     package
  • impostare la variabile __all__ *nota
```

Compiled Python files

- python -m py_compile fileA.py fileB.py fileC.py …
  - include byte-compiled version of module if there exists fibo.pyc in same directory as fibo.py
  - only if creation time of fibo.pyc matches fibo.py
  - automatically write compiled file, if possible
  - platform independent
  - **doesn't run any faster, but *loads* faster**
  - can have only .pyc file → hide source

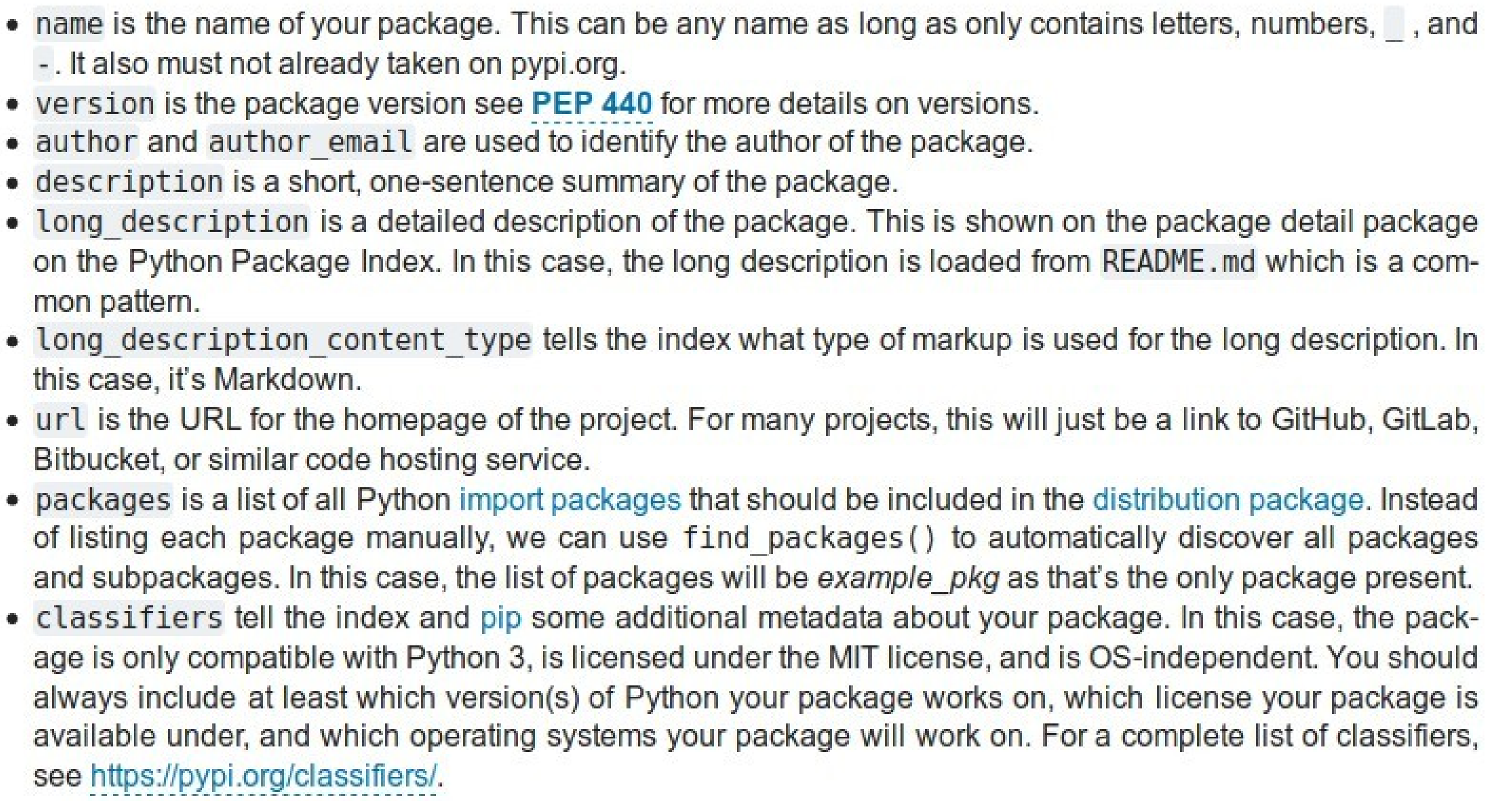
Deploy methods

- Ce ne sono molti → il piu comune è utilizzare setup.py
- File in python di direttive per il setup del Vs. package racconta a setuptools come comportarsi con il Vs. package via metadata
- Presume una struttura simile

```
some_root_dir/
|-- README
|-- setup.py
|-- example_pkg
|   |-- __init__.py
|   |-- useful_1.py
|   |-- useful_2.py
|-- tests
|-- |-- __init__.py
|-- |-- runall.py
|-- |-- test0.py
```

```
import os
from setuptools import setup
# Utility function to read the README file.
# Used for the long_description.  It's nice, because now 1) we have a top level
# README file and 2) it's easier to type in the README file than to put a raw
# string in below ...
def read(fname):
    return open(os.path.join(os.path.dirname(__file__), fname)).read()
setup(
    name = "an_example_pypi_project",
    version = "0.0.4",
    author = "Andrew Carter",
    author_email = "andrewjcarter@gmail.com",
    description = ("An demonstration of how to create, document, and publish "
                                   "to the cheese shop a5 pypi.org."),
    license = "BSD",
    keywords = "example documentation tutorial",
    url = "http://packages.python.org/an_example_pypi_project",
    packages=['an_example_pypi_project', 'tests'],
    long_description=read('README'),
    classifiers=[
        "Development Status :: 3 - Alpha",
        "Topic :: Utilities",
        "License :: OSI Approved :: BSD License",
    ],
)
```

Le proprietà



- `name` is the name of your package. This can be any name as long as only contains letters, numbers, `_` , and `-`. It also must not already taken on pypi.org.
- `version` is the package version see PEP 440 for more details on versions.
- `author` and `author_email` are used to identify the author of the package.
- `description` is a short, one-sentence summary of the package.
- `long_description` is a detailed description of the package. This is shown on the package detail package on the Python Package Index. In this case, the long description is loaded from `README.md` which is a common pattern.
- `long_description_content_type` tells the index what type of markup is used for the long description. In this case, it's Markdown.
- `url` is the URL for the homepage of the project. For many projects, this will just be a link to GitHub, GitLab, Bitbucket, or similar code hosting service.
- `packages` is a list of all Python import packages that should be included in the distribution package. Instead of listing each package manually, we can use `find_packages()` to automatically discover all packages and subpackages. In this case, the list of packages will be *example_pkg* as that's the only package present.
- `classifiers` tell the index and pip some additional metadata about your package. In this case, the package is only compatible with Python 3, is licensed under the MIT license, and is OS-independent. You should always include at least which version(s) of Python your package works on, which license your package is available under, and which operating systems your package will work on. For a complete list of classifiers, see https://pypi.org/classifiers/.

Setup.py basic usage – $ python setup.py <some_command> <options>

- $ python setup.py –help-commands
  - Step 1 python setup.py sdist # impacchetta il codice in un singolo file di archivio creando una **dist** subdir e creando un tar.tgz o zip

Include :

- Tutti i python sorgente
- Tutti i file C menzionati in ext_modules o libreria
- Scripts identificati dallo script options
- Quansiasi cosa che assomigli a test e.g. test/test.py
- Top level files named README.*, setup.py setup.cfg
- Tutti i file elencati in un eventuale MANIFEST.in file

L’ecosistema di packaging

- Package
  - Una semplice directory con un __init__.py file dentro
  - Cio crea un package che può essere importato usando import
- Ok per piccoli progetti → working directory come package location
- Distutils ci concede di installare packages in PYTHONPATH
  - PYTHONPATH che dovrebbe essere ma non è equivalente a sys.path in codice è una lista di posti dove cercare per I Python Packages
    - In realtà PYTHONPATH gestisce gli extra path
    - python -c 'import sys; print(sys.path)'
- Python central service per contributing packages.
  - The Python Package Index (PyPI).

L’ecosistema di packaging

- Setuptools
  - Introduce easy_install e il package setuptools che puo essere importato nello script setup.py ed in pkg resources
- Wheels
  - python setup.py bdist_wheel – python setup install
    - Se i vostri pacchetti non sono su PyPi bisogna copiare a mano i pacchetti nel Wheel folder che il comando stesso crea

Esempio

```
from setuptools import setup, find_packages

setup(name='funniest',
      version='0.1',
      description='The funniest joke in the world',
      long_description='Really, the funniest around.',
      classifiers=[
        'Development Status :: 3 - Alpha',
        'License :: OSI Approved :: MIT License',
        'Programming Language :: Python :: 2.7',
        'Topic :: Text Processing :: Linguistic',
      ],
      keywords='funniest joke comedy flying circus',
      url='http://github.com/storborg/funniest',
      author='Flying Circus',
      author_email='flyingcircus@example.com',
      license='MIT',
      packages=find_packages(),
      install_requires=[
          'markdown',
      ],
      include_package_data=True,
      zip_safe=False)
```

```
call conda activate [my_env]
python my_script.py
call conda deactivate
….
```

```
call c:\...myenv\bin\activate
python my_script.py
call deactivate
...
```

```
source ..myenv\bin\activate
python my_script.py
deactivate
```

https://docs.python.org/3.7/distutils/setupscript.html

```
cmd "/c activate [my_env] && python my_script.py && deactivate"
```

setup tools

- python setup.py <some_command> <options>
- python setup.py --help-commands

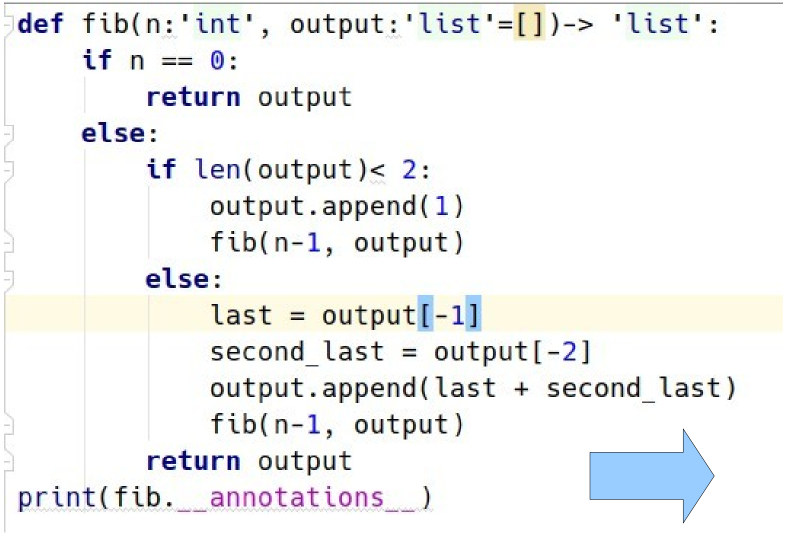
```
Standard commands:
  build             build everything needed to install
  build_py          "build" pure Python modules (copy to build directory)
  build_ext         build C/C++ extensions (compile/link to build directory)
  build_clib        build C/C++ libraries used by Python extensions
  build_scripts     "build" scripts (copy and fixup #! line)
  clean             clean up temporary files from 'build' command
  install           install everything from build directory
  install_lib       install all Python modules (extensions and pure Python)
  install_headers   install C/C++ header files
  install_scripts   install scripts (Python or otherwise)
  install_data      install data files
  sdist             create a source distribution (tarball, zip file, etc.)
  register          register the distribution with the Python package index
  bdist             create a built (binary) distribution
  bdist_dumb        create a "dumb" built distribution
  bdist_rpm         create an RPM distribution
  bdist_wininst     create an executable installer for MS Windows
  upload            upload binary package to PyPI
```

Dipendenze

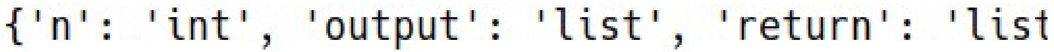
- pip freeze > requirements.txt. (tutti)
- pipreqs /your_project/path   (solo quelli di progetto da import)
  - pipdeptree (check)
- **Offline**
  - pip download -r requirements.txt   Scarica i requirements
  - pip install --no-index --find-links /path/to/download/dir/ -r requirements.txt

Annotations

- Annotations can be used to collect information about the type of the parameters and the return type of the function to keep track of the type change occurring in the function.
  - **[def foo(a:”int”, b:”float”=5.0) -> ”int”]**



```
def fib(n:'int', output:'list'=[])-> 'list':
    if n == 0:
        return output
    else:
        if len(output)< 2:
            output.append(1)
            fib(n-1, output)
        else:
            last = output[-1]
            second_last = output[-2]
            output.append(last + second_last)
            fib(n-1, output)
        return output
print(fib.__annotations__)
```



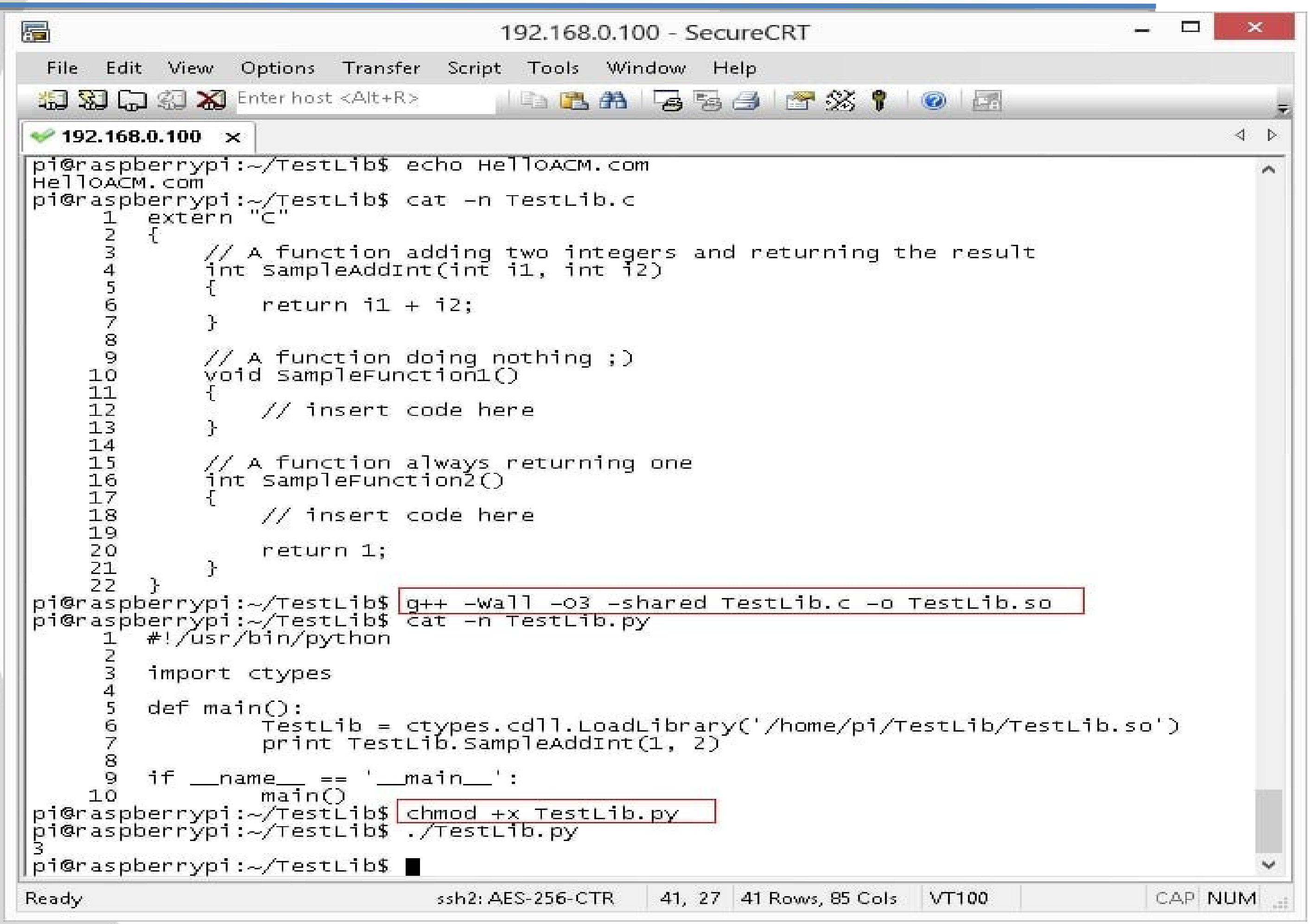
```
{'n': 'int', 'output': 'list', 'return': 'list
```

Appendice : Type hints > 3.5

- Types Any, Union, Tuple, Callable, TypeVar, and Generic.

```
def greeting(name: str) -> str:
    return 'Hello ' + name
```

Integrare C e Python



```
pi@raspberrypi:~/TestLib$ echo HelloACM.com
HelloACM.com
pi@raspberrypi:~/TestLib$ cat -n TestLib.c
     1  extern "C"
     2  {
     3      // A function adding two integers and returning the result
     4      int SampleAddInt(int i1, int i2)
     5      {
     6          return i1 + i2;
     7      }
     8
     9      // A function doing nothing ;)
    10      void SampleFunction1()
    11      {
    12          // insert code here
    13      }
    14
    15      // A function always returning one
    16      int SampleFunction2()
    17      {
    18          // insert code here
    19
    20          return 1;
    21      }
    22  }
pi@raspberrypi:~/TestLib$ g++ -Wall -O3 -shared TestLib.c -o TestLib.so
pi@raspberrypi:~/TestLib$ cat -n TestLib.py
     1  #!/usr/bin/python
     2
     3  import ctypes
     4
     5  def main():
     6          TestLib = ctypes.cdll.LoadLibrary('/home/pi/TestLib/TestLib.so')
     7          print TestLib.SampleAddInt(1, 2)
     8
     9  if __name__ == '__main__':
    10          main()
pi@raspberrypi:~/TestLib$ chmod +x TestLib.py
pi@raspberrypi:~/TestLib$ ./TestLib.py
3
pi@raspberrypi:~/TestLib$
```

Integrare C e Python

```
#!/usr/bin/python
import ctypes
def main():
        TestLib = 
ctypes.cdll.LoadLibrary('/home/pi/TestLib/TestLib.so')
        print (TestLib.SampleAddInt(1, 2))
if __name__ == '__main__':
        main()
```

I siti

- PyPI - the Python Package Index
  - https://pypi.python.org/pypi
- Python
  - https://www.python.org/
  - http://www.python.it/
- Documentazione
  - https://docs.python.org/3/
- Learned Python
  - http://learnpythonthehardway.org/book/

Fonti sorgenti e ringraziamenti a …

- https://github.com/jakevdp/PythonDataScienceHandbook.git
- **https://github.com/ehmatthes/intro_programming.git**
- https://github.com/jdwittenauer/ipython-notebooks.git
- https://github.com/rajathkmp/Python-Lectures.git
- https://github.com/jrjohansson/scientific-python-lectures.git
- https://bitbucket.org/hrojas/learn-pandas.git
- https://github.com/mmmayo13/scikit-learn-beginners-tutorials.git
- https://github.com/mmmayo13/scikit-learn-classifiers.git
- https://github.com/jakevdp/sklearn_tutorial.git

Discussion